diploma thesis

Adaptive Building Intelligence
A multi-Agent approach

Ueli Rutishauser
<urut@easc.ch>

Alain Schäfer
<alani@easc.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
8640 Rapperswil, Switzerland

Institute of Neuroinformatics
University and ETH Zurich
Winterthurstrasse 190
8057 Zurich, Switzerland

December 10, 2002

Typeset by LaTeX

Abstract

This diploma thesis explores methods for making a building intelligent. A building is perceived as being intelligent if it is able to learn from experience. We demonstrate a advanced multi-agent architecture for controlling a building. The multi-agent system is connected to the sensors and effectors via a dedicated fieldbus network (LonWorks). Based on this data the system takes decisions about the actions it wants to execute. Decisions are taken by a fuzzy inferencing engine on basis of a block of fuzzy rules. The multi-agent system utilizes a generic structure information system to abstract the complicated relationships between sensors, effectors and static structures (like rooms).

We then extend this work with an anytime learning algorithm that dynamically learns the rulebase used by the fuzzy inferencing process. The reinforcement signals for the learning algorithm are automatically generated from the input data acquired from the sensors of the building. The learning process is completely unsupervised. There are no special methods for the user to provide feedback.

We furthermore demonstrate the feasibility of the approach by deploying the system in a real-world environment consisting of a number of different rooms.

An other issue issue for making a building intelligent is personalization. We demonstrate with a prototype what is already possible and what not using the technologies available as of today. Our prototype uses the Bluetooth technology to track and identify individual persons moving freely in a building.

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

Contents

Preface **xiii**

I Introduction **1**

1 Introduction **3**

1.1 Overview 3

1.2 Content 4

1.3 Related work 4

2 Motivation **6**

II Architecture **8**

3 System Architecture **10**

3.1 Basic principles 10

3.2 Connectivity 10

3.3 Generic Structure information 11

4 Software Architecture **13**

4.1 Overview 13

4.2 Agents 13

4.3 Implementation requirements 14

5 Multi-agent system **15**

5.1 Overview 15

5.2 Agents 15

5.3 FIPA and JAS 16

5.3.1 Overview 16

5.3.2 The JAS platform 17
5.3.3 Agent Naming Service 18
5.3.4 Agent Directory Service 18
5.3.5 Message Transport System 18
5.3.6 Lifecycle Service 18
5.3.7 Usage scenarios 19
5.3.8 Package 19
5.4 Messaging 19
5.4.1 Interest based, asynchronous messaging 19
5.4.2 Topics 20
5.4.3 Messages 20

III Design and Implementation 21

6 Design and Implementation 23
6.1 Programming language and runtime environment 23
6.2 MAS framework 23
6.3 3rd party frameworks 23
6.4 Packages 24

7 Simulation 25
7.1 Motivation 25
7.2 Design and Implementation 25

8 Learning a maximal structure fuzzy rule base with an anytime learning algorithm 27
8.1 Overview 27
8.1.1 Features of the algorithm 27
8.1.2 Related Work 27
8.2 Notation 28
8.3 The algorithm 29
8.3.1 Definitions 29
8.3.2 Operators 29
8.3.3 Algorithm 30
8.4 Motivation 30
8.5 Examples 31

8.6 Implementation 31

9 Anytime learning and fuzzy inferencing applied to intelligent buildings 36

9.1 Overview 36

9.2 Model and learning units 36

9.3 Decision Making 37

9.4 Design of membership functions 38

9.4.1 Radiation 39

9.4.2 Temperature 39

9.4.3 Illumination 39

9.4.4 Daytime 39

9.5 Structure of rulebase 40

9.5.1 Fuzzy OR operator 40

9.5.2 OR to AND conversion 41

9.6 Implementation 41

9.6.1 Dynamic creation of rules 42

9.7 Related work 42

10 Agents 45

10.1 Messaging 45

10.2 Bus Agent 47

10.2.1 Functionality 47

10.2.2 Messages 47

10.2.3 Implementation 47

10.3 Structure Agent 49

10.3.1 Functionality 49

10.3.2 Messages 49

10.3.3 Implementation 49

10.4 Simulator Agents 51

10.4.1 BusSim - Simulation of BusAgent 51

10.4.2 Simulator Agent - Control of simulation time 54

10.5 Control Agent 56

10.5.1 Functionality 56

10.5.2 Messages 56

10.5.3 Implementation 56
10.6 Distribution Agent 61
10.6.1 Functionality 61
10.6.2 Messages 61
10.6.3 Implementation 62
10.7 BuildingViewer Agent 63
10.7.1 Functionality 63
10.7.2 Messages 64
10.7.3 Implementation 64
10.8 Person Locator Agent 66
10.8.1 Functionality 66
10.8.2 Messages 66
10.8.3 Implementation 66
10.9 Virtual Person Agent 67
10.9.1 Functionality 67
10.9.2 Messages 67
10.9.3 Implementation 67

11 Personalization 68

11.1 Person Tracking – Bluetooth 68
11.1.1 Mobile Device Program 68
11.1.2 Server side 69

12 System Configuration 71

12.1 Envrionment Variables 71
12.2 Config Files 71

IV Results and future work 73

13 Results 75

13.1 Evaluation 75
13.2 Architecture 75
13.2.1 Messaging 76
13.2.2 Simulation 76
13.2.3 JAS 77

13.3 Learning and inferencing . . . 77
13.4 Personalization . . . 78

14 Future **79**
14.1 Learning and inferencing . . . 79
14.2 Structure information . . . 79
14.3 Personalization . . . 80

V Appendix, Glossary and Bibliography **81**

A Fuzzy Logic Primer **83**
A.1 Fuzzy Logic primer . . . 83
A.1.1 Definitions . . . 83
A.1.2 Membership functions . . . 83
A.1.3 Fuzzy inferencing . . . 85
A.1.4 Fuzzy logic controllers . . . 85
A.1.5 Fuzzification process . . . 86
A.1.6 Defuzzification process . . . 86
A.1.7 Rule based inferencing . . . 87

B Agent Building and Learning Environment **89**
B.1 Agents . . . 89
B.1.1 JAS . . . 89
B.2 ARL . . . 90
B.2.1 Structure of ARL . . . 90
B.3 Fuzzy Inferencing . . . 91

C Configuration **93**
C.1 BusSim Schema File . . . 93
C.2 Virtual Person schema file and example . . . 94

Glossary **97**

List of Figures

3.1 A floor of a building structured into rooms . . . 10

3.2 System architecture . . . 11

3.3 Variable Mapping in BusAgent . . . 12

3.4 One floor of an imaginative building . . . 12

4.1 Different types of agents used . . . 13

5.1 FIPA abstract architecture specification and it's relation to JAS (from the FIPA specification) 16

5.2 JAS Platform services as specified in JSR-87 (except for lifecycle service) . . . 17

5.3 Agents running inside the JAS framework . . . 17

5.4 FIPA Agent directory (Register) . . . 18

5.5 FIPA Agent directory (Query) . . . 18

5.6 Agents declare there interests to the distribution agent . . . 20

7.1 The Simulation Subsystem . . . 26

8.1 UML activity diagram for the learning algorithm. All numbers refer to the respective steps (0 to 9) of the description of the algorithm. . . . 33

8.2 The state space covered by the rule $((\{x, y, z\}, \{h, j\}), y_{j2})$ . . . 34

8.3 The state space covered by the rule $((\{y, z\}, \{h, j\}), y_{j2})$ . . . 34

8.4 UML class diagram of the learning algorithm implementation . . . 35

9.1 Relationship between Model and Learning Unit . . . 37

9.2 Structure of a single room (example) . . . 37

9.3 An FLC and it's inputs and outputs . . . 38

9.4 Membership function of radiation . . . 39

9.5 Graphical version of the membership function radiation . . . 39

9.6 Membership function of Temperature . . . 39

9.7 Graphical version of the membership function temperature . . . 40

9.8 Membership function of illumination 40
9.9 Graphical version of the membership function illumination 41
9.10 Membership function of DayTime 41
9.11 Graphical version of the membership function daytime 42
9.12 The fuzzy set normal of the variable Daylightindoor 42
9.13 Fuzzy hedge not applied to the fuzzy set normal of the variable Daylightindoor 43
9.14 Fuzzy hedges not positively slightly applied to the fuzzy set normal of the variable Daylightindoor 43
9.15 UML class diagram of the package *aha.rules* 44

10.1 UML class diagram of the bus agent 48
10.2 UML class diagram of the structure agent 50
10.3 Relationship between Variables/Connections/Mappings 51
10.4 UML class diagram of the simulation agents (simulation,bussim, artificial persons) 55
10.5 UML class diagram of the control agent (Agent itself, part 1) 58
10.6 UML class diagram of the control agent (Elements, part 2) 59
10.7 UML class diagram of the control agent (Learning, part 3) 60
10.8 UML class diagram of the distribution agent 62
10.9 Simulation of BuildingViewer Agent 63
10.10UML class diagram of the BuildingView agent 65
10.11UML class diagram of the locator agent 66
10.12UML class diagram of the locator agent 67

11.1 Architecture of our Bluetooth prototype 69
11.2 Screenshot of main window 69
11.3 Screenshot of access point choice 70

A.1 Triangular membership function 84
A.2 Trapezoidal membership function 84
A.3 Sigmoidal membership function 84
A.4 Gauss distribution as a membership function 85
A.5 Basic layout of a non-fuzzy controller 85
A.6 Basic layout of a fuzzy controller 86
A.7 Fuzzification Process 86
A.8 Defuzzification with center-of-area/gravity 87

B.1 ARL structure . . . 90

B.2 ARL definition of a fuzzy variable . . . 90

B.3 Visualization of the membership function DayTime . . . 91

B.4 ARL definition of a categorical variable . . . 91

B.5 ARL definition of the input and output variables . . . 92

B.6 ARL rules . . . 92

Listings

10.1 example config structure agent 49
10.2 Extract a config file for a simulation 52
10.3 example config for a variable filter 52
10.4 example config for an update action 53
10.5 example reference to timed events 53
10.6 example for the list of timed events 53
C.1 Grammar for describing a simulated sensor network 93
C.2 Grammar for virtual persons 94
C.3 Example of a config file for virtual persons 95
C.4 Grammar for id to variable name mapping, BusAgent 96

List of Tables

10.1 Table of all messages 46

Preface

By Prof. Dr. Joseph M. Joller, University of Applied Sciences Rapperswil, Switzerland

After the “Semesterarbeit”, this diploma thesis is a second step towards more adaptive, intelligent and multi-agent based buildings. A quick look at the literature shows, how advanced the concepts presented in the diploma thesis are and this should help us, to position the work relative to similar projects (Essex, MIT, . . .).

The project gave us several new views on old problems:

- The question of learning (online, real-time versus history / database based [more information mining]);
- The question of personal identification: how can we locate a person using standard devices (e.g. using Bluetooth) and / or how can this information be used to navigate a person to a target point.
- The question of identification of structures: how can devices find out, to which structure they belong, which structure they define (in simpler terms: what makes up a room, a floor, a building, from an abstract point of view)?

The diploma contains contributions to the solution of all three of the above questions.

Typically a software engineer would try to find an answer to the third questions by using “biology inspired” informatics:

- how can a cell know, in which environment it is living and growing?
  Software engineering suggests a strong cohesion und weak coupling of objects. Each object knows, how it can be combined with other objects, what is allowed and what is not.
  Biologists would probably give a simple answer: the genes know, what combinations proved to be stable and which combinations survived.
- A different answer would we get, if we asked a neuroscientist. He would (probably) argue about information processing in the brain and come up with a solution, similar to the one that was implemented in the diploma thesis (structure agent).

This very simple example gives some hints, that future work on adaptive multi-agent systems may have significant impacts an the way we are structuring complex software systems.

I look forward to see more results. . .

Part I

Introduction

Chapter 1

Introduction

1.1 Overview

In this paper we describe our approach for making a building intelligent. To achieve this we use a learning algorithm specifically developed for this purpose. Our approach is to learn maximal structure fuzzy rules from data acquired from sensors and to adapt this rules driven by a punishment and reward scheme. We prove that our approach is suitable by deploying it within a whole floor of a commercial office building.

We regard learning from non-explicit feedback as the most important issue in making a building intelligent. An intelligent building, from a computer science point of view, should take decisions about the building such that the user of the building doesn't have to make (and execute) them himself. The most important feature of such a learning algorithm is that it is non-supervised. A user of a building shouldn't notice that the building he is in is actually intelligent which means that there are no special input devices with which the system driving the building can interact with the user. The only tools for interaction are the sensors and effectors present in an ordinary building: presence detectors, weather sensors, switches, lights and blinds. All feedback for the learning algorithm needs to be taken from these sensors.

Users and the preferences of these users constantly change. The consequence of this is that an intelligent building has to learn continually (*anytime learning*). Something learnt about the behavior of some user may be valid at the present time but it may become invalid in the future because the preferences of the user changed. There is no training phase and production phase in such a system. The system has to learn anytime in it's lifetime.

Our view of an intelligent building is very similar to our view of robots. We regard an intelligent building as a robot. The only difference between an ordinary robot and an intelligent building is that real people are present inside an intelligent building. This creates all sorts of new challenges. Within the context of an intelligent building we deal with real environments with real people that are used to do real work. An intelligent building is an environment where there can be no compromises regarding stability and efficiency. An autonomous system driving an intelligent building needs to take all decisions in near-realtime because a noticeable delay between sensor data changing and actions happening won't be accepted by a user of a building in the long-term.

We tackle this challenges by using a layered multi-agent system. Decisions are taken by many different agents. Every agent is responsible for a small subset of the whole state space. This emphasizes localized decision making which makes the system much faster. The same principle is applied to the learning. There are different so called learning units. Each of these learning units learn about a small subset of the whole state space. This guarantees much faster convergence of the learning.

To judge the quality of such a system requires running the system for a considerable long period of time within a real building. Because this would slow down the development process of such a system considerably we developed a simulator of a building which simulates the behavior of a building and it's users. This includes the influence of external factors like weather that constantly change. This simulator enables us to test a specific system and it's parameters much faster because it is capable of running many times faster than real time.

1.2 Content

This document is structured into the five parts Introduction (part I), Architecture (part II), Design and Implementation (part III), Results (part IV) and Appendix, Glossary and Bibliography (part V).

The first part, Introduction (part I), gives a general introduction to the topic. It consists of the chapters Motivation (2) and Introduction (1). Also part of the introduction are an extensive list of papers that report about related work.

Architecture, which consists of the chapters system architecture (3), software architecture (4) and multi agent system (5), gives a detailed introduction to the underlying architectural principles used within ABI. Especially important to mention here is the introduction into the multi-agent architecture in chapter 5. This chapter describes the relationship between the multi-agent and the JAS/FIPA architecture specifications. A very important architectural decision is mentioned in section section 3.3. This section gives an explanation of how we coped with the difficult problem of managing relationships between static and dynamic elements of a structure (eg a building). The same principle is also used to manage the sensor/effector relationships.

Details about the implementation of the system are given in part III. Design decisions made for the overall system are justified in chapter 6 and design decisions regarding the simulation are justified in chapter 7. The two chapters 8 and 9 explain the core of the adaptive system, the learning algorithm and how it is used to make the system capable of adapt itself to the needs of the users. Within chapter 10 there is a separate section for every agent that is part of ABI. For every agent there is a detailed specification of it's functionality, which messages it sends and processes and a explanation of the classes and objects used for implementing it.

Section 11 looks at the issue of personalization and means of how persons can be identified and tracked inside a building.

Chapters 13 and 14 evaluate what has been achieved and what not within the context of the project and what could be done in the future. It also lists potential topics for future research projects.

For readers not familiar with fuzzy logic there is a fuzzy logic primer in Appendix A and for readers not familiar with the ABLE framework and ARL there is a introduction in appendix B.

1.3 Related work

The basic ideas and principles of *Adaptive Building Intelligence (ABI)* were first developed during the projects *ADA – An artificial organism* ([EBB$^+$02]) and *Adaptive Home Automation (AHA)* ([RS02]), both of which were conducted at the Institute of Neuroinformatics in Zurich,Switzerland (which belongs to the ETH). *Adaptive Building Intelligence* is built on top of the architecture developed of the preceding *Adaptive Home Automation (AHA)* project. Whereas the focus of AHA was on establishing a multi-agent based infrastructure to make decisions on basis of a static fuzzy logic rulebase, the primary focus of *Adaptive Building Intelligence* is to make the building adaptive through online learning.

There is a variety of relevant projects being conducted at other places. [Coe97] and [Coe98] describe approaches where the focus is to make a single room intelligent through the usage of special sensors and effectors like cameras or robotic arms. In [Coe98] the distinct focus is on how to make a single room intelligent by detecting where people are located in the room (person tracking), by detecting what this people are doing (pointing) and by interact with people in the room over speech recognition (input) and speech synthesis (output). In [Coe97] the authors show how they use different agents to make a single room intelligent by using the same principles as shown in [Coe98]. [Bro97] gives a good introduction to the MIT Intelligent room project in which's context [Coe97] and [Coe98] evolved. In general one can say that the MIT Intelligent room project's focus is on the human-machine-interface rather than learning the behavior of the people itself.

[HCCC02] shows a approach that is perhaps most similar to our work of all papers known to the authors of this paper. They describe a system where different sensors of a room are connected via a sensor network (LonWorks) to a single embedded agent that is located physically in a room. This agent uses the information acquired to learn fuzzy logic rules with a genetic algorithm learning paradigm. There learning procedure requires explicit feedback of the user. Related to [HCCC02] is [VC00] where the authors describe an architecture consisting of embedded agents that is used to build intelligent systems. Here our work is different: We use a learning algorithm which doesn't require any explicit feedback by the users at all. All feedback is acquired by means that the user doesn't notice or by actions the user would also execute if there were no learning system present (eg switching on the light).

An other interesting perspective is presented in "Artificial Decision Making Under Uncertainty in Intelligent Buildings" ([BDY99]). This paper focuses on the decision making under uncertainty itself. It is partly very similar to our project as it is also using a multi agent system (MAS) architecture and LonWorks for connecting to sensors and actors.

Ada ([EBB$^+$02]), a project done at the Neuroinformatics Institute of the Swiss Federal Institute of Technology and the University of Zurich, regards a room as a artificial organism. Ada has an emotional state just like every other organism. She expresses here internal states to visitors and interacts with visitors. Input sensors are acoustic/speech recognition, vision processing and touch (floor). The floor of Ada is divided into a large number of tiles. Every of these tiles is a autonomous system that provides input to the overall system and gives feedback from the system. Every of these tiles measures the weight of the person standing on it. These weights are important inputs to the system. Furthermore every of these tiles can emit light of an arbitrary color. Output sensors are video, audio/voice and light (light fingers and tiles). The goal of Ada is to dynamically change its overall functionality and quality through an active dialog with visitors.

Chapter 2

Motivation

Within this chapter we motivate why it is worth trying to make a building intelligent.

Our experience with similar projects like [EBB$^+$02] and [RS02] showed that it is possible to use results of research in neuromorphic systems, traditional and non-traditional artificial intelligence and computer science to build intelligent rooms and buildings. Whereas these projects have been hugely successful they still lack one of the most important abilities of something considered intelligent: *learning*. The next big step forward is taking one of these existing systems and make them really intelligent by incorporating various forms of learning.

An environment like an intelligent building is a particularly interesting and challenging environment to apply learning. Buildings are an environment which is used by real people to do real work. It is not some sort of playground or specifically designed example which are typically used to verify new algorithms and architectures. An environment used by real people everyday poses many additional challenges which makes applying algorithms in such an environment even more interesting. Results obtained from research in such an environment are likely to be much more substantial then results obtained from a specifically designed simulated environment.

As new technologies like Bluetooth or Wireless LAN become available new things like tracking individuals inside a building become feasible. Using Bluetooth for tracking and identifying people inside a building is one of the possibilities to enable a building to behave in a personalized way. In this paper we explore various possibilities using Bluetooth and demonstrate its capabilities and limits with a prototype.

A building is a very complex system. From a computational point of view a building behaves completely non-deterministic. Additional complexity is added because the environment of a building is largely non-episodic which means that actions execute by some agents (for example a person) my trigger the execution of other actions by other persons. [Wei99] shows that such an environment is best controlled by a multi-agent system. [Wei99] calls this approach *Distributed Artificial Intelligence (DAI)*. Different agents don't communicate directly with each other about there goals and actions they take. The only communication between agents that occurs is indirect through changes in the environment (*stimergy*).

If one looks at the characteristics of an intelligent building it first appears that such an environment is very unique in itself. But thinking about a building as a kind of robot reveals many similarities. Many problems in research about intelligent buildings has it's similarities in robotic research. A good example of this is a robot moving freely in a room. Such a robot also has to deal with a completely non-deterministic environment as it can't influence what happens around it in any way. It can just react to it. Because of this we regard an intelligent building as a robot. The only difference between a traditional robot and an intelligent building is that in an intelligent people there are people living and working whereas for a traditional robot people are working and living around the robot.

Embodied artificial intelligence emphasizes the importance of a real world environment for intelligence. Only a real environment can reveal how intelligent a system really is. As already shown in [RSDJ02] a building is a suitable environment for embodiment. The use of indirect communication between agents through the environment (stimergy) also depends fully on the availability of an environment which can be modified by effectors and which can be sensed by sensors.

Part II

Architecture

Chapter 3

System Architecture

3.1 Basic principles

The input and output devices of the system are sensors and effectors which can be accessed over a sensor network. Figure 3.1 shows an example of a hardware architecture. All sensors and effectors of all rooms are connected to the same network. This network is, with the help of a gateway, accessible from an IP based network.

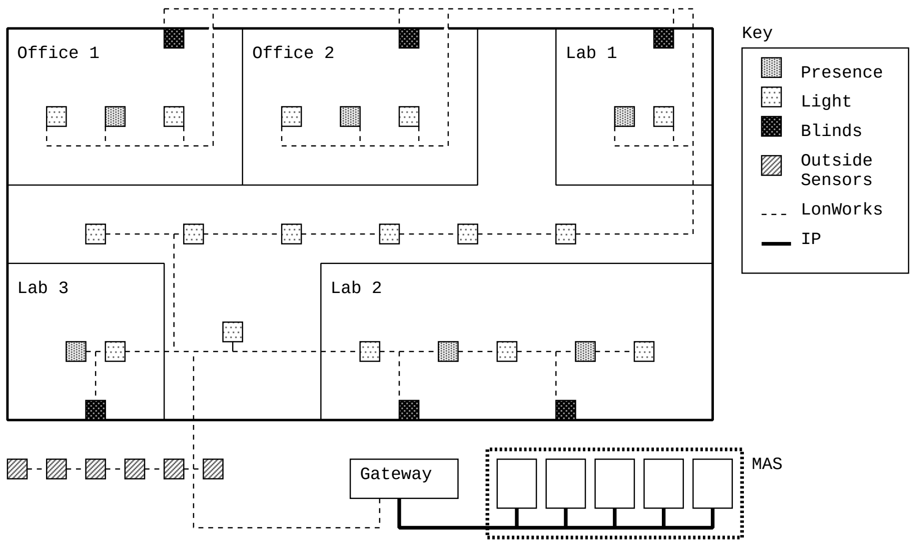


Figure 3.1: A floor of a building structured into rooms

3.2 Connectivity

The overall ABI system is a combination of two completely different network architectures: field bus and a computer network (like Ethernet is). A modern office building is typically wired with these two bus systems. ABI requires access to both, the field bus and the computer network of a building. The field bus (in our case LonWorks) is used to communicate with the sensors and actors of the building automation system whereas the computer network is used to communicate between the different agents of the system.

The two networks are connected to each other with the help of a gateway that is connected to both networks. The gateway forwards data between the computer network and the field bus such that all software parts of

ABI can run on the computer network without any direct access to the field bus.

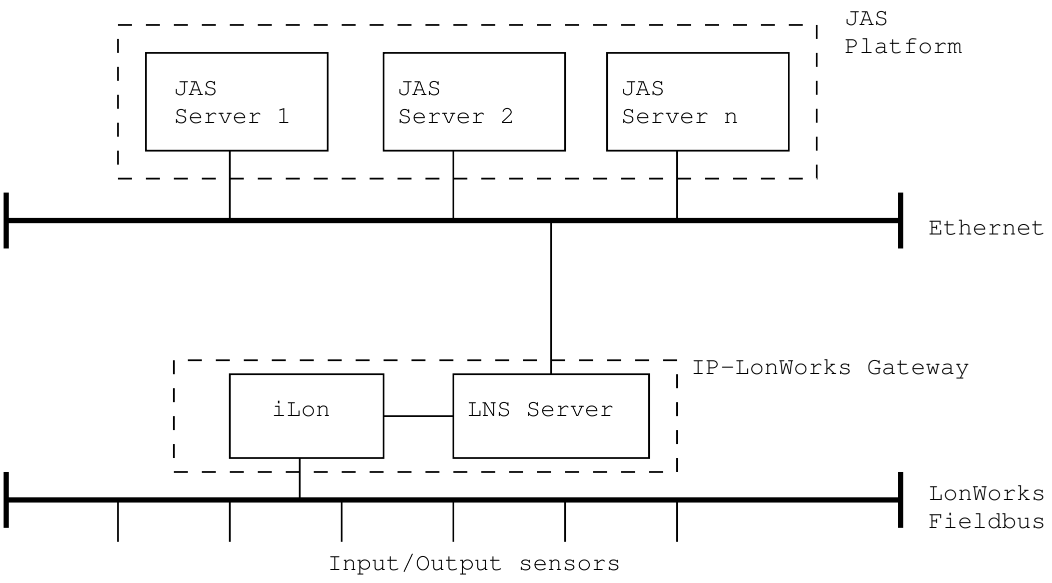


Figure 3.2: System architecture

Figure 3.2 shows the system architecture of ABI. ABI isn't dependent on any specific kind of field bus system, the LonWorks field bus standard is just used as one possibility in the figure. LNS and iLon are LonWorks specific products that act as an IP-LonWorks gateway. Communication between iLon (which is the real LonWorks-IP gateway) and the LNS server (which is just a software) is already done over IP, so the only hardware device that requires direct access to the LonWork bus is the iLon.

Please refer to the chapter Bus abstraction in [RS02] for details about the generic bus interface that abstracts the complexity of different bus systems from the users.

3.3 Generic Structure information

In the predecessor project AHA we ended up with agents which clearly depended on knowledge of the building they were running in. The agents depended on the naming convention used, when the field bus was installed. This was due to the BusAgent exporting the variable names to it's clients. As an example the concept room was a group of variables which all had the same variable name prefix.

This system was also pretty limited since it was only able to control one light switch connected to two lights and one blind switch connected to two blind controllers. The relationships between these switches and controllers were hard coded or discovered through the usage of regular expressions on the variable names and thus also relying upon knowledge of the variable naming scheme.

A truly adaptive system should not depend upon such knowledge. It should easily adapt to different buildings. It should not use the naming system to discover which sensor has a relation to some effectors.

A first step in avoiding this is to not export the real variable names. BusAgent, the one responsible to encapsulate access to the field bus, should map from real variables names to artificial variable ids and back. That way the variables information lies only in its value and not in its name (variableId) anymore. For an example see Figure 3.3.

Although this solves the problem of dependency on a certain naming schema, the agents don't know anything about the structure of the building and are not able to obtain any knowledge about that. The most simple solution to this is to introduce another agent (structure Information agent). This agent would be responsible for providing informations about the sensors and effectors and their structure in the building.

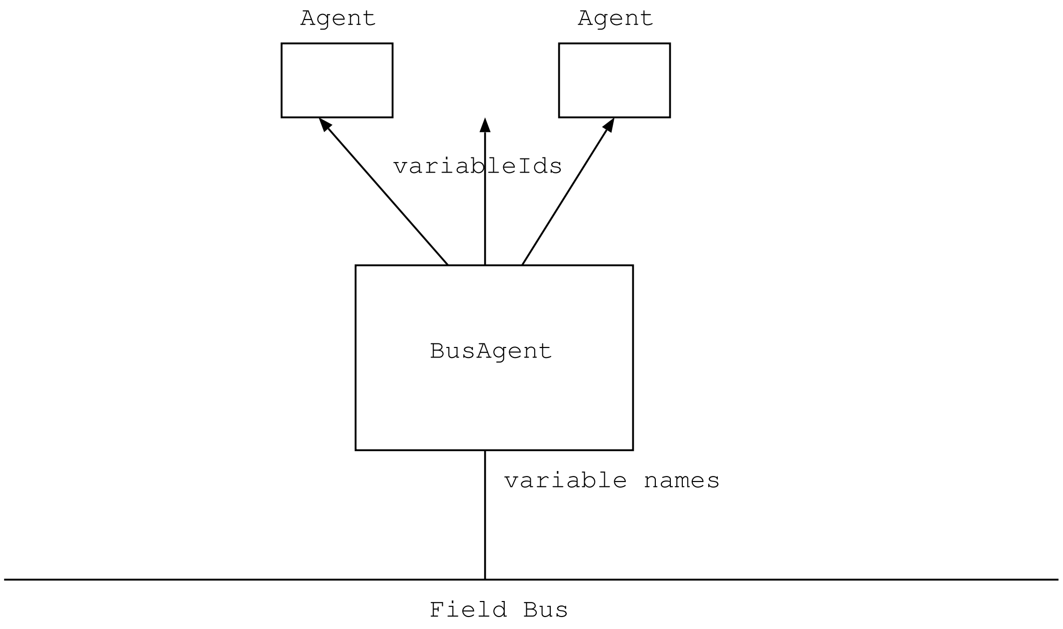


Figure 3.3: Variable Mapping in BusAgent

If we could provide this information in a most general way, our system would be able to adapt to total different environments. Maybe even environments which we can not imagine today. A very general way of providing this information is by introducing the concept of clusters. With a cluster being just a group of variables and other clusters. One can easily describe a building and its field bus by using this notion. For an example see figure 3.4.

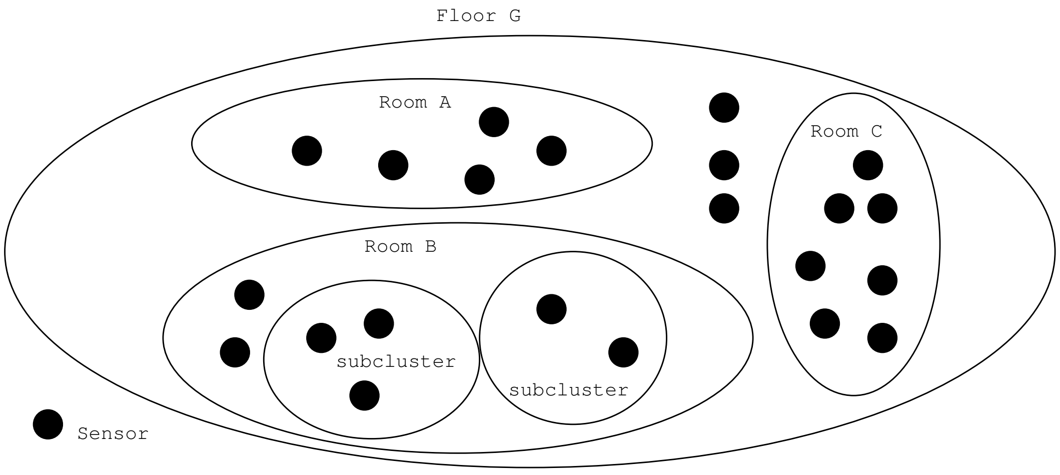


Figure 3.4: One floor of an imaginative building

So far we did not talk about how this structure agent will discover the knowledge about the building. A simple but limited way is to read this information from a configuration file. In a first iteration such an approach is acceptable. Because the structure agent provides a generic interface, it can later easily be exchanged with an agent which discovers this information in a more sophisticated way.

Chapter 4

Software Architecture

4.1 Overview

The system is realized as a collection of agents which together form a multi agent system (MAS). Every agent is responsible for one specific task and offers this task as a service to the other agents. Agents that require functionality they don't offer themself collaborate with other agents that offer the required capability to achieve their aims. There is no central agent that acts as a kind of coordinator, all agents act independently of each other and pursue their specific goals.

The software architecture of the system doesn't make any specific assumptions about the actual environment where the agents are running. The whole system could either run on a single machine or distributed across multiple machines.

4.2 Agents

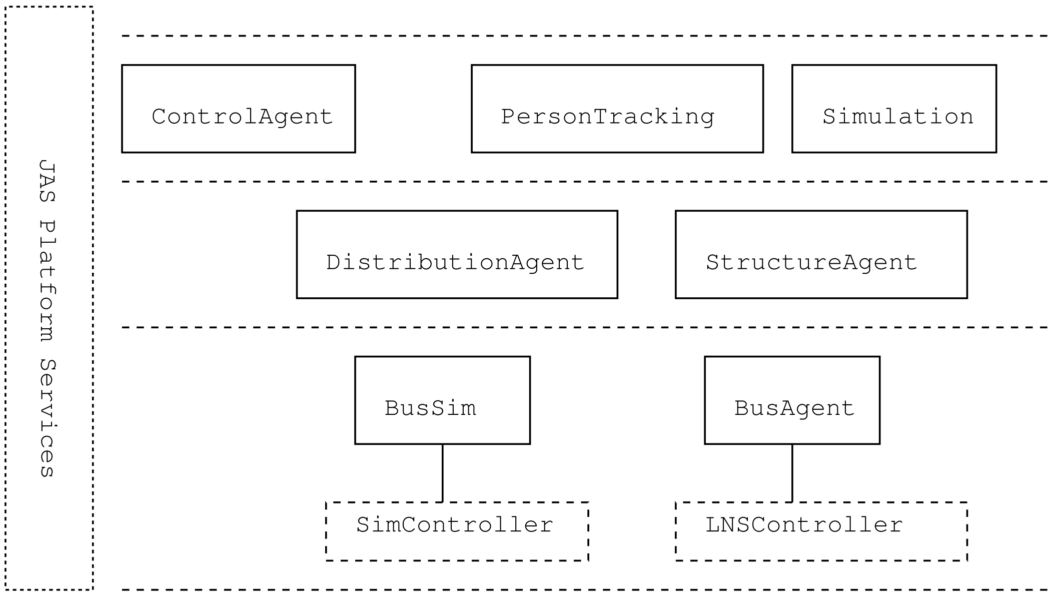


Figure 4.1: Different types of agents used

The system is composed of the following agents:

1. Simulation: Simulates a building in case no real (physical) fieldbus is present.
2. BusSim: Simulates a physical fieldbus system. Has the same external interface as the real BusAgent.
3. BusAgent: Generic interface to the physical bus system (for example a LON network).
4. DistributionAgent: Collects interests and distributes messages asynchronously according to this interests.
5. StructureAgent: Defines the structure of the sensor/effector relationships. Reads this structure information from XML files.
6. ControlAgent: Controls a single room consisting of multiple clusters. Learns and adapts rules. Takes decisions.
7. PersonTracking: Interface to Bluetooth. Tracks and identifies persons. Notifies other agents when someone enters or leaves a room.
8. VirtualPerson: Simulates an artificial person in a simulated environment

Every agent belongs to one of the three domains of functionality *Bus Interface*, *Agent Services* and *system specific functionality*. Agents are associated to the the different domains as following:

1. Bus Interface: BusSim, BusAgent
2. Agent Services: DistributionAgent, StructureAgent
3. System specific functionality: ControlAgent, PersonTracking, Simulation, Virtual persons

Only the domain *system specific functionality* contains agents which are specific to building intelligence. All other agents in the system provide basic infrastructure services like messaging or hardware bus interface for other agents.

4.3 Implementation requirements

As ABI is the successor of AHA ([RS02]), we are using the same architecture as in the former AHA project. All implementation requirements like evaluation of a MAS or the programming language are the same as in AHA.

See [RS02], chapter software architecture, for these requirements which are thus also valid for ABI.

Chapter 5

Multi-agent system

5.1 Overview

The general system architecture approach used within the ABI framework is that of a multi-agent system. A multi-agent system is structured into several independent and autonomous agents. Each of these agents pursues it's own goals independently and cooperates with other agents if necessary. Agents can communicate with each other in two different ways: direct (1:1) or indirect (1:n). Both approaches are used within ABI. Details of the interagent communication process can be found in section 5.4.

This chapter is structured as following. Firstly, the concept of an agent in principle is introduced. Following this an introduction to the principles of JAS, on which basis ABI is constructed, is given. Afterwards the messaging architecture used for interagent communication is explained.

5.2 Agents

First and most important is the concept of an agent. What exactly is a agent? There is no agreement on a formal definition of a agent yet but the following definition is generally accepted:
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 ([WJ94]).

A more informal definition of an agent would be that a agent is a piece of software that is continually reevaluating the input it gets from it's environment to determine the output it should send back (action) to the system. A agent is generally non-determinating which means that it doesn't end at any point of time.

An agent, as defined above, that is only reacting to its environment can not be called intelligent. What are the differences between an agent and an intelligent agent? An intelligent agent is an agent that has the following characteristics (([WJ94]):

- *reactivity*: intelligent agents are able to perceive their environment, and respond in a timely fashion to changes that occur in it in order to satisfy their design objectives
- *pro-activeness*: intelligent agents exhibit goal-directed behavior by *taking the initiative* in order to satisfy their design objectives
- *social ability*: intelligent agents are capable of interacting with other agents (and possibly humans) in order to satisfy their design objectives.

An agent can by no means be restricted to be some part of software, as an agent can also be a person or a piece of hardware (embedded agent). This is especially the case for intelligent buildings where there are all three sorts of agents present: persons, software agents and hardware agents.

5.3 FIPA and JAS

5.3.1 Overview

JAS (JSR-87), the java agent specification, is an emerging industry standard currently being developed under the java community process (JCP). It is, as of late 2002, available as a public review draft ([FLoA02]). There are already several JAS compliant multi-agent toolkits available from various vendors. JAS specifies an architecture for agents which run concurrently on physically distributed systems. Such agents require a possibility to discover each other and to communicate with each other in a standardized way. JAS provides such a common platform for agents.

JAS is essentially a reification of the FIPA abstract agent architecture (FIPA Standard 00001, [FIP02]). This makes JAS based systems FIPA (Foundation for Intelligent Physical Agents) compliant. JAS offers a set of objects and service interfaces that support agent deployment and operation. It doesn't specify any implementation specific issues and can as such be implemented with any technology, be it a java based one or a proprietary one. The JAS specification includes a reference implementation as well as a compatibility test suit. The reference implementation can be used during development as a replacement for an other JAS implementation.

The FIPA abstract architecture specification ([FIP02] defines a set of abstract concepts like *agent*, *agent-directory-service* and *agent-communication-language (ACL)* but it doesn't specify concrete, programming language and platform dependent, interfaces for such a system. JAS is one concrete implementation of the FIPA abstract architecture specification that defines concrete classes and interfaces.

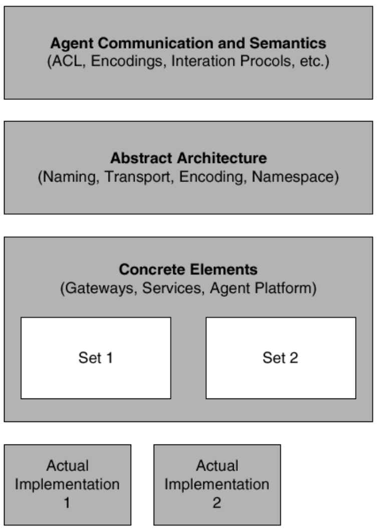

Agent Communication and Semantics
(ACL, Encodings, Interation Procols, etc.)
Abstract Architecture
(Naming, Transport, Encoding, Namespace)
Concrete Elements
(Gateways, Services, Agent Platform)
Set 1
Set 2
Actual Implementation 1
Actual Implementation 2

Figure 5.1: FIPA abstract architecture specification and it's relation to JAS (from the FIPA specification)

It is important to notice that the FIPA abstract architecture specification is not backwards compatible with FIPA97, FIPA98 or FIPA2000. Thus JAS is only compliant with the FIPA abstract architecture specification but not with any of the previous FIPA specifications which had a much narrower scope.

Figure 5.1 shows the relationship between the FIPA abstract architecture, the JAS specification and ABLE. The layers *concrete elements* and *actual implementation* are covered by ABLE and the JAS reference implementation whereas the *Agent communication and semantics* are implemented in ABI itself.

5.3.2 The JAS platform

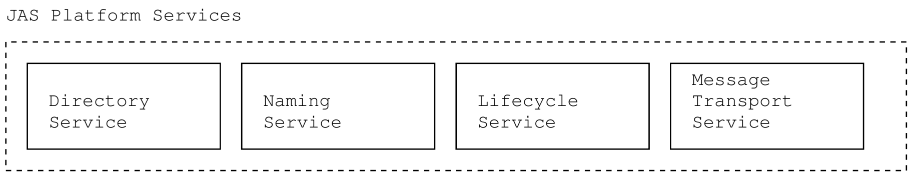


Figure 5.2: JAS Platform services as specified in JSR-87 (except for lifecycle service)

JAS offers a very high-level interface to agents that take advantage of it. It offers agents the complete infrastructure to communicate with each other, manage instances of each other and to distribute a multi-agent system across multiple systems. For the user of a JAS system (an agent) JAS looks like one homogen platform – a user of the JAS platform doesn't have to care about where physically an agent is actually running. JAS hides this complexity from the agent. This enables the agent, or better the developer of the agent, to focus on it's core competences and not on minor technical issues like code migration or dynamic discovery of other agents.

JAS, as proposed in [FLoA02] (public review draft) consists of the three building blocks (5.2):

1. Agent Naming Service
2. Agent Directory Service
3. Message Transport System

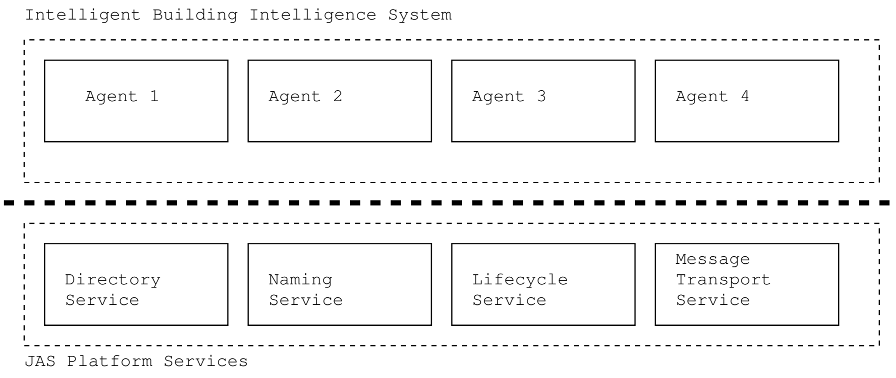


Figure 5.3: Agents running inside the JAS framework

Every of these services is part of the JAS platform. All these services can be used by any agent running within the JAS framework (Figure 5.3). Agents don't need to worry about where these services are actually running in an JAS system. The JAS platform automatically takes care of all these issues. The internals and implementation details of the JAS platform are completely transparent to the agents. The only thing in common between the agents and the JAS platform are the standardized JAS interfaces as specified.

Figure 5.2 shows a 4'rth service, the lifecycle service. The lifecycle service is not part of the official JAS specification draft but absolutely necessary to use a JAS system. Because the lifecycle service isn't standardized yet every vendor of a multi-agent system delivers it's own implementation of a lifecycle service so far. The lifecycle service is not part of the FIPA Abstract architecture specification and because of this it is also not part of JAS. FIPA argues that lifecycle operations are to implementation specific and can thus not be abstracted enough to become part of the FIPA abstract architecture specification.

To distinguish agents in the JAS platfrom every agent is assigned a GUID by the JAS platform during instanciation. All references to an agent in JAS ultimately consist of these global unique ID.

5.3.3 Agent Naming Service

The agent naming service is very similar to other naming services like conventional DNS or the RMI registry. The agent naming service maintains a database of all available types of agents in a system. This database can be queried by users of the agent naming service with different properties like display-name or classname. As a result the agent naming service returns a unique reference to the type of agent queried that allows it to later on instanciate such an agent with the help of the lifecycle service.

5.3.4 Agent Directory Service

The directory service manages instances of currently running agents. As such it can be compared to other directory services like LDAP. The directory service allows it to dynamically find out which other agents are currently active in the system and how to communicate to them.

The two essential operations that the agent directory service offers are registration of an agent instance (Figure 5.4) and query (Figure 5.5).

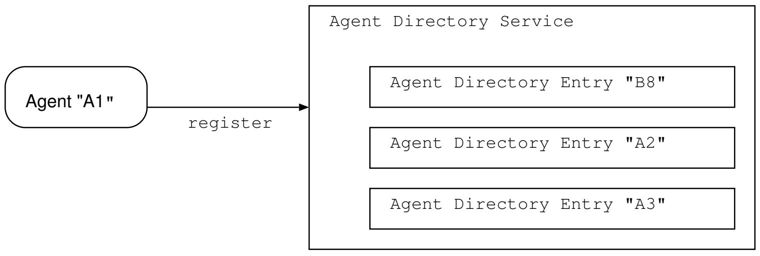

Agent Directory Service
Agent "A1"
register
Agent Directory Entry "B8"
Agent Directory Entry "A2"
Agent Directory Entry "A3"

Figure 5.4: FIPA Agent directory (Register)

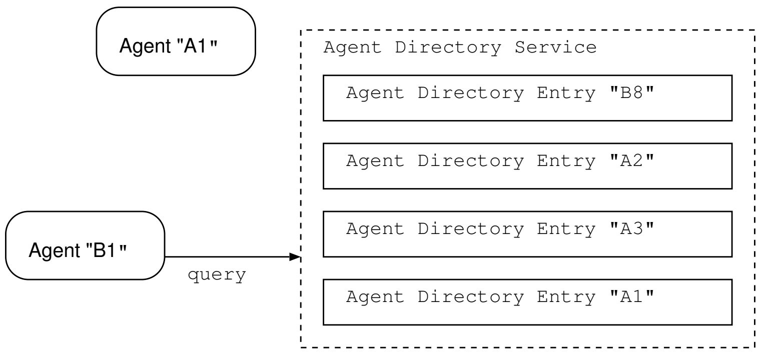

Agent "A1"
Agent Directory Service
Agent Directory Entry "B8"
Agent Directory Entry "A2"
Agent "B1"
query
Agent Directory Entry "A3"
Agent Directory Entry "A1"

Figure 5.5: FIPA Agent directory (Query)

5.3.5 Message Transport System

The message transport system is the core of JAS. It defines a common way on how agents can exchange messages with each other. It defines a standard agent communication language (ACL) that can encapsulate arbitrary types of data as payload. The ACL specified by the message transport system is FIPA compliant.

5.3.6 Lifecycle Service

The lifecycle service is responsible for actually creating new instances of agents and destroying them. It takes commands from other agents to create or destroy other agents on demand. Typical lifecycle management operations are *stop agent*, *start agent*, *freeze*, *suspend*, *unfreeze* and *restart*.

The lifecycle service is not part of JAS ([FLoA02]) or FIPA ([FIP02]) and thus implementation specific. We are using the ABLE ([ABL] implementation.

The lifecycle service offers uniform, single-point, access to the JAS platform. The caller of a lifecycle operation doesn't have to know or specify where (physically) an agent is running.

5.3.7 Usage scenarios

A typical usage scenario of JAS is:

1. Start up the JAS platform on all servers participating in the multi-agent system. This also automatically starts up all 4 JAS standard services: agent naming service, agent directory service, message transport system and lifecycle service.
2. A client (or an agent) connects to JAS and uses the agent naming service to discover the agents available (Figure 5.5) and to register itself (Figure 5.4). After selecting the agents suitable it uses the lifecycle services to create the necessary instances of those agents.
3. Agents are communicating with each other with the help of the message transport system. Agents discover each other through usage of the directory service and start/stop each other agents by using the lifecycle service.

5.3.8 Package

JAS resides in the official namespace *javax.agent.* This package only contains interfaces and classes which are part of the specification. Any specific implementation resides in an other package (for example *org.jagent*).

5.4 Messaging

Interagent communication in big multi-agent systems with many participating agents is a big challenge. As soon as there are more then just a few agents participating in the system the message flow between different agent starts to get very complex. Every agents is pursuing it's own goals and is thus interested in different types of messages. Some agents care more about a particular message than do others. But how does a agent which wants to send out a message know which other agents it should send the message to? To all of them or just to a few selected ones? This requirements and questions clearly show the need for a messaging middleware to which a sending agent can delegate the task of distributing messages in a multi-agent system.

An additional problem faced by interagent communication is speed. If there are many agents to which a message needs to be sent to the sending agent requires more time to send out the message. During this time it is occupied and thus can't pursue the actions it was originally designed to for. Communication lines between agents may be of poor quality which further slows down message exchange or some agents might even become unreachable for some time. This additional obstacles are best met by asynchronous message delivery.

5.4.1 Interest based, asynchronous messaging

Our approach to deal with the complexity of communication in a multi-agent system is to delegate the communication itself to an agent whose distinct task it is to distribute messages according to interests. The distribution agent receives messages from other agents with instructions on how to deliver them. It than takes over the message and delivers it to all other agents interested in this type of message. The distribution agent is an agent which can deliver hundreds of messages concurrently. It achieves this by using a large threadpool which enables it to assign every message for every destination an own thread. This architecture makes message delivery much faster and more reliable.

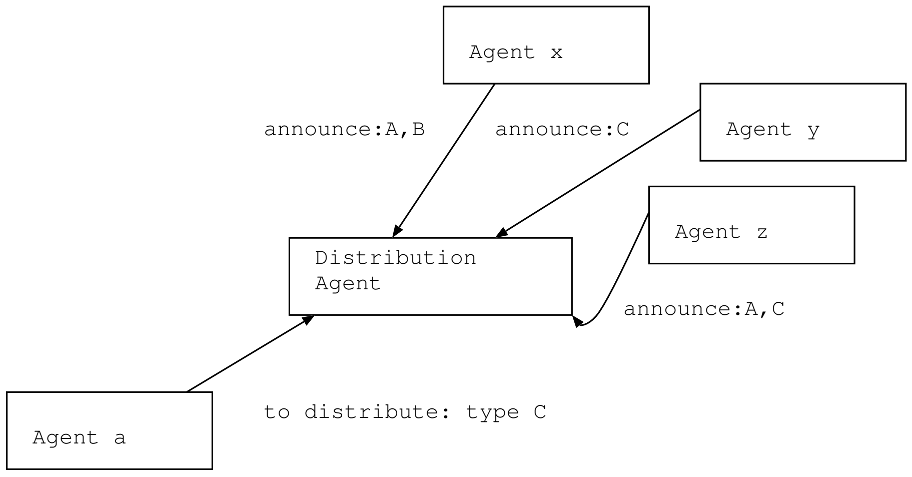


Figure 5.6: Agents declare there interests to the distribution agent

To be able to fulfill its task the distribution agent needs to know about the interests of other agents. Every agent that has an interest in some data needs to announce this interest towards the distribution agent (Figure 5.6). The distribution agent collects this announcements and distributes messages according to this collected interests.

Figure 5.6 shows an example of this principle. Agent x is interested in messages of type A and B, agent y in messages of type C and agent z is interested in messages of type A and C. Now agent a sends a message of type C to the distributor. The distributor takes this message and re-distributes it to agent y and z.

5.4.2 Topics

The distribution agent manages interests of other agents in terms of topics. A topic is an identification of some type of message. All *register interest* and *unregister interest* messages relate to a certain topic. The distribution agent does not now the available topics in advance. The first time a new topic is registered it automatically creates this topic. If there are no interested agents for a topic the distribution agent destroys the topic.

There are different types of topics. Every topic is identified by a unique string and this string also tells of which type a topic is. In case a topic represents a network variable that is managed by the *BusAgent* the distribution agent notifies the *BusAgent* everytime a topic representing a network variable is created (*topic create*) or destroyed (*topic destroy*). When the *BusAgent* (see) gets a notification about a new topic it automatically subscribes to the corresponding variable on the network and starts forwarding incoming updates for the variable to the distribution agent.

5.4.3 Messages

Messages exchanged between agents are all encapsulated within JAS/FIPA compliant ACL transport message envelops and can thus be forwarded between any JAS/FIPA transport system. Every type of message is identified by an ID that is unique system wide. See section 10.1 for a detailed description of the content of every type of message.

Part III

Design and Implementation

Chapter 6

Design and Implementation

This chapter outlines the technologies used to build ABI.

ABI is a continuation of the former AHA project ([RS02]) and thus many design decisions made during the implementation of AHA do have a strong influence on the implementation of ABI. This is the reason why there are many factors, like the programming language or the agent toolkit used, which aren't justified here. Details about there justification can be found in the documentation of [RS02].

6.1 Programming language and runtime environment

The programming language of choice for ABI is Java, version 1.4.0.

6.2 MAS framework

As MAS framework ABLE ([ABL]) (version 1.4.1) and the JAS reference implementation ([FLoA02]) is used. ABLE offers a JAS compliant MAS framework. Details about ABLE are given in appendix B.

6.3 3rd party frameworks

Besides of the MAS framework some other 3rd party libraries are used:

- JFreeChart ([JFR]): Library that provides basic charting capabilities
- JCommon ([JCO]): collection of useful classes; Required by JFreeChart.
- LNS Java SDK ([LNS]): Java API for accessing the LNS server. This API is a product provided by Echelon (LonWorks).
- Log4J ([L4J]): Logging framework, manages debug/warning/error output, distributed logging
- ANTLR ([ANT]): general purpose parser generator. We use it to parse ARL files
- CASTOR ([CAS]): Castor is an open source data binding framework for Java.

All of these tools and libraries are provided under a opensource license and are thus freely available.

6.4 Packages

The source is structured into the following packages:

- aha.control: Control Agent
- aha.bus: Bus Agent (generic, field bus system independent parts)
- aha.bus.lonworks: LonWorks specific implementation
- aha.framework: Utility classes, Supporting classes
- aha.middleware: Common message format definitions
- aha.messaging: Distribution agent, common definition of messages
- aha.sim : Simulator, User Interface for the system
- aha.sim.person : artificial persons for the simulation
- aha.sim.structure : Generic structure information
- aha.rules: fuzzy logic rules (in ARL)
- aha.rules.learning : learning algorithm for learning fuzzy logic rules

Chapter 7

Simulation

7.1 Motivation

During the preceding project AHA we saw the need to develop our system in a simulated environment. It became difficult to develop new ideas, because they had all to be tested on the real building control system. As soon as more than one person works on such a project the building control system becomes a critical resource. As we are developing in the same building where our system is also running, we have the further problem that our tests do interfere with the running system.

We thus needed an environment which gives our group of agents the same interface (sensors, actors, devices) as they have when running in the real environment.

Another advantage of a simulated environment is the ability to "accelerate" time. It would be possible to simulate a whole day in for ex. 1 hour. The behavior of different building control systems over a longer period could thus be observed in a short period.

A major drawback of observing and testing such systems in a simulation lies in the Etymology of the word to simulate. Latin : simulare which comes from similis - like but not exactly the same. A simulation is always only an approximation to reality. Adaptive and self-learning systems could thus develop completely different behaviors in a simulation that they would in reality.

A simulation normally doesn't need any graphical user interface. For convenience reasons we decided to write a GUI. Since our simulation should provide the same interface like the real system this GUI could also be used to interact with the real system.

7.2 Design and Implementation

The most important part which was needed for a simulated replacement is the BusAgent which in turn uses LNSController to query and update the field bus. This subsystem would be responsible to imitate the behavior of all the sensors and effectors. It would have to simulate no only all the values sensors and effectors have during the simulation, but also all the relationships between sensors and effectors. The design of this subsystem affects how accurate our simulation would bee.

Another part of the simulation is presence detection and person location. As described in chapter 11, we try to track person locations and learn their behavior via Bluetooth devices. In the simulation we need virtual persons walking around the floor. Thats what the Agent aha.sim.persons.PersonLocator is responsible for.

Finally one needs a controller for the whole simulation. This subsystem would control the simulation time, start and stop agents for the simulation life cycle.

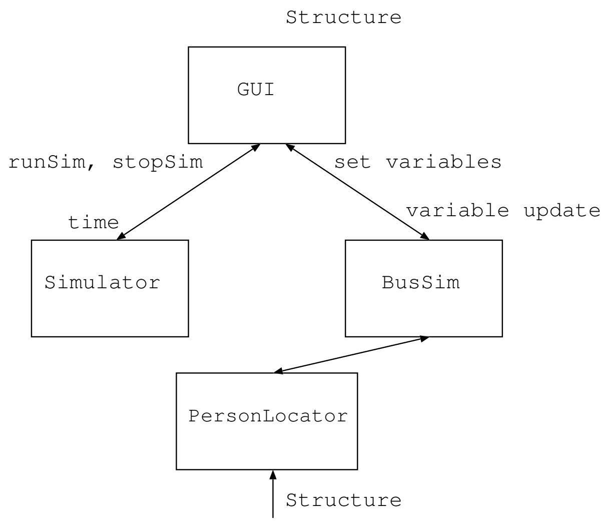


Figure 7.1: The Simulation Subsystem

To develop such a discrete simulation we faced the choice of either using an existing simulation toolkit or develop our own. With the decision of our implementation language being Java the choice was already limited. The toolkits which were left were such as Java Simulation Package [JSP] or JToops [Rin] or JSim [JSI].

But we decided not to use such a preexisting toolkit. The main reason was that we expected to spend at least as much time to learn to toolkit as we would need to implement the small part which we actually needed. By doing our own implementation we could also integrate the simulation much better with our whole system, using ABLE as an example.

Chapter 8

Learning a maximal structure fuzzy rule base with an anytime learning algorithm

8.1 Overview

This chapter is about the learning algorithm used for making a building intelligent. It is an online learning algorithm which continuously updates the rulebase of several fuzzy logic controllers. It continually adapts itself to the changing conditions of the environment through a implicit punish/reward feedback mechanism. The inputs to the learning process are real valued variables acquired from sensors, the output of the learning algorithm is a model consisting of a number of fuzzy rules. These fuzzy rules are used by a fuzzy logic controller to take decisions. Feedback acquired from the environment is continuously used by the learning process to adapt the fuzzy logic rules.

The algorithm presented here learns rules for a multiple input single output (MISO) fuzzy logic controller. The control system for the adaptive building intelligence system, however, is a multiple input multiple output (MIMO) system. We will show in the next chapter how we apply the MISO learning algorithm to a system that is MIMO.

8.1.1 Features of the algorithm

- online / anytime
- punish and reward type of feedback
- learns maximal structure fuzzy rules
- adapts itself to changing conditions – what might have been true in the past doesn't necessarily have to be true in the future

8.1.2 Related Work

The development of this algorithm has been inspired by many different sources. The most important sources of inspiration were the papers [CCSZ21], [SZ96], [Bon97] and [Bon96a].

There are a few offline algorithms for learning maximal structure fuzzy rules from a fixed size training set, for example: [CCSZ21], [CZ16], [CZ65]. These algorithms are distinct offline processes: first there is learning from some training set and second the learned rules are applied to other data. There is no feedback process that changes the once learned rules according to the changing conditions of the environment.

Other approaches for learning fuzzy logic rules include algorithms that are based on the ideas of evolutionary computing. See for example [Bon96b].

8.2 Notation

We largely use the same notation as [Ful00], [CCSZ21] and [CZ16].

An introduction to fuzzy logic and fuzzy inferencing is given in Appendix A.

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: $\mathcal{L}(X_0) = \{HN, MN, SN, Z, SP, MP, HP\}$ (which is a very typical generic fuzzy variable, see [Ful00] , section 1.5.3). 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 $\mathcal{R}$ (equation 8.1). r specifies the number of rules of the ruleset (equation 8.2). It isn't necessary to specify the output variable as this is a MISO system where there is only one output.

$$\mathcal{R} = \{R_1, R_2, \ldots, R_r\} \tag{8.1}$$

$$r = |\mathcal{R}| \tag{8.2}$$

A rule (equation 8.3) 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: $y_j \in \mathcal{L}(Y)$.

$$R_i = ((E_0 \subset \mathcal{L}(X_0), E_1 \subset \mathcal{L}(X_1), \ldots, E_r \subset \mathcal{L}(X_{N-1})), y_j) \tag{8.3}$$

N_i (equation 8.4) denotes the number of fuzzy input variables of rule R_i.

$$N_i = |\{X_0, X_1, \ldots, X_{N-1}\}| \tag{8.4}$$

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

8.3 The algorithm

Two sets of rules are used by the algorithm: The trainingSet $\mathcal{R}_T$ and the set of definitive rules $\mathcal{R}_{DEF}$. All definitive rules together form the model on basis of which a fuzzy logic controller takes decisions.

The trainingSet consists of fuzzy rules that were directly constructed from external input received. A entry of the trainingSet is constructed by fuzzifying the real-valued input for every input and output variable of the fuzzy logic controller. Such an entry is very simple as there is only one condition for every fuzzy input variable. An example of a training set entry would be $((\{A\}, \{G\}), y_0)$.

There are typically many more entries in the set of training entries than in the set of definitive rules. This is due to the nature of the algorithm whose aim it is to deduct maximal structure rules from the known training set entries.

8.3.1 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.*

Definition 2 (real subset) *A set A is a real subset of B if $(A \subset B)$ and $(|A| < |B|)$ is true.*

8.3.2 Operators

The algorithm uses the operators **subsumes**, **antecedentSubsume**, **amplify** and **amplifyIsPossible**.

Definition 3 (subsume) *The rule R_i subsumes in the 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 $E_{ik} \subset E_{jk}$. N is the number of fuzzy variables. subsume(R_i, R_j) is true if the above conditions hold and $y_i = y_j$.*

Definition 4 (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.*

Definition 5 (amplify) *amplify(T) computes a collection of rules which are generalizations of T whereas $T \in \mathcal{R}_T$. Every entry of the collection contains an additional condition for one fuzzy variable. The amplification is done incrementally: Every time an additional label from $\mathcal{L}_i$ is added to E_i for the lowest possible i for which $|E_i| < |\mathcal{L}(X_i)|$. The output variable and value is not influenced by the amplification process.*

Definition 6 (amplifyIsPossible) *An amplification R_A is possible if there are no trainingSet entries $T \in \mathcal{R}_T$ for which antecedentSubsume(T, R_A) is true and for which $y_i \neq y_j$ is true.*

8.3.3 Algorithm

The following procedure is execute every time the system receives new input from the system (which can be a punishment or a reward):

Step 0 Wait for new input.

Step 1 Transform the real-valued sample into an trainingSet entry (fuzzification). This new trainingSet entry becomes rule R_T. R_T has a set of antecedent conditions E_{Ti} for every fuzzy input variable X_i.

Step 2 Test for every rule R_D that is part of the definitive ruleset ($R_D \in \mathcal{R}_{DEF}$) whether antecedentSubsume(R_T, R_D) is true.
If yes, assign R_a ($R_a \in \mathcal{R}_{DEF}$) to the rule R_D for which antecedentSubsume(R_T,R_D) was true and goto step 3.
If no, goto step 8.

Step 3 Test if subsume(R_T,R_a) is true. If yes, the input sample was a reward. Stop and go to Step 0. If subsume(R_T,R_a) is false there is an invalid rule in $\mathcal{R}_{DEF}$ (environment changed). At this point the output value of the sample is different from the output value of a rule in the definitive ruleset ($y_s \neq y_t$). Goto stop 4.

Step 4 If all $E_{Ti} \subset E_{ai}$ ($i = 0..N$) and a minimum of one of these subsets is a real subset ($|E_{Ti}| < |E_{ai}|$) the rule can be re-used. Goto step 5 in this case. If not, goto Step 6.

Step 5 Assign $R_m = R_a$. Remove R_a from the set of definitiveRules $\mathcal{R}_{DEF}$.
Reassign every E_{mi} as following: $E_{mi} = E_{ai} - E_{Ti}$ (remove all conditions $E_{Ti} \subset E_{ai}$ from E_{ai}). Add R_m to the definitive ruleset $\mathcal{R}_{DEF}$ and goto step 7.

Step 6 Remove R_a from the set of definitive Rules $\mathcal{R}_{DEF}$ and goto step 7.

Step 7 Test if there are entries in the trainingSet $\mathcal{R}_{R_T}$ which became invalid because of the removal of R_a. All trainingSet entries T_j for which antecedentSubsume(T_j, R_a) is true but antecedentSubsume(T_j, R_m) is false are removed. Goto step 8.

Step 8 Add the new sample R_T to the trainingSet $\mathcal{R}_T$. Goto step 9.

Step 9 Amplify R_T and assign the resulting sorted set of amplifications to $\mathcal{R}_A$. R_{amp} is the rule currently considered ($R_{amp} \in \mathcal{R}_A$). Sort $\mathcal{R}_A$ such that the least general rule $R \in \mathcal{R}_A$ is the first element of $\mathcal{R}_A$ and the most general rule $R \in \mathcal{R}_A$ is the last element of $\mathcal{R}_A$.

Step 10 Assign R_{amp} to the least general rule in $\mathcal{R}_A$ for which amyplifyIsPossible(R_{amp}) has not been called yet. Goto Step 11.

Step 11 If amplifyIsPossible(R_{amp}) is true, goto Step 10. If false, add the last R_{amp} for which amplifyIsPossible(R_{amp}) was true to the set of definitive rules $\mathcal{R}_{DEF}$. Goto step 0.

Figure 8.1 illustrates the algorithm.

8.4 Motivation

The basic idea of the algorithm is to cover as much of the input space as possible with the fewest possible rules. In case of n fuzzy input variables the input space is a n Dimensional with a finite number of elements in every dimension (every fuzzy set of every fuzzy variable). The principle of the fewest possible rules can also be state as aiming for the most general rules.

To illustrate the principle idea a fuzzy ruleset with the two fuzzy input variables A and B is assumed. $\mathcal{L}(A) = \{x, y, z\}$ and $\mathcal{L}(B) = \{g, h, j\}$. Figure 8.2 shows the state space covered by the rule $((\{x, y, z\}, \{h, j\}), y_{j2})$ illustrated as an area in 2-dimensional space.

In case the additional trainingset entry $((x, h), y_{j1})$ is added to the trainingSet the state space covered by the existing rule as shown in figure 8.2 needs to be reduced. Figure 8.3 shows the state space covered after the modification of the rule (which made the rule less general).

8.5 Examples

This section contains a very simple example of how rules are learned from data. Initial assumptions are:

- Fuzzy input variables are A and B
- Fuzzy sets of these variables are $\mathcal{L}(A) = \{x, y, z\}$, $\mathcal{L}(B) = \{g, h, j\}$
- Definitive Ruleset $\mathcal{R}_{DEF}$ is empty
- Output variable is Z, $\mathcal{L}(Z) = \{z_1, z_2\}$

Now a trainingset entry $((x, g), z_1)$ is added to the trainingset. As a result of the learning algorithm the rule $((\{x, y, z\}, \{g, h, j\}), z_1)$ is added to the set of definitive rules. Note that this is the most general rule possible. This is the case because there is just one trainingSet entry available which allows it to construct a maximally general rule because there will never be any conflict between output values.

Assume the next trainingset entry (added incrementally when available) is $((z, g), z_1)$. This doesn't change anything because it is already covered by the rule in the definitive ruleset that was added in the last step.

Ensuing this an entry $((z, h), z_2)$ becomes available. This does have to change something in the definitive ruleset because as this entry generates a conflict with the rules already in there. The result of the learning algorithm removes the rule $((\{x, y, z\}, \{g, h, j\}), z_1)$ and adds it again as $((\{x, y\}, \{g, j\}), z_1)$. In addition it adds the new rule $((\{x, y, z\}, \{g, h\}), z_2)$.

The definitive ruleset $\mathcal{R}_{DEF}$ now contains the two rules:

- $((\{x, y\}, \{g, j\}), z_1)$
- $((\{x, y, z\}, \{g, h\}), z_2)$

8.6 Implementation

Figure 8.4 shows the structure of the implementation of the learning algorithm.

The implementation (which is in package *aha.rules.learning*) is structured into classes as following:

- *FuzzyLearner*: The main class that does the learning. It is responsible for loading the state from persistent storage and saving state to persistent storage.
- *DefinitiveRuleSet*: Set that holds all definitive rules and offers operations that affect all rules in the set. Implements the operator subsumes.
- *DefinitiveRuleSetIterator*: Iterator for DefinitiveRuleSet.
- *FuzzifiedRule*: Represents one single rule in a fuzzified form. Implements the operators amplify, isAntecedentSubset, isAntecedentRealSubset and subsumes.
- *FuzzifiedRuleFactory*: Supporting Methods for creating FuzzifiedRule instances from different sources.
- *TrainingSet*: Set of all entries in the trainingset. Implements the operator isAmplificationPossible.

- *TrainingSetEntry*: A single trainingset entry.
- *TrainingSetIterator*: Iterator for TrainingSet.
- *Model*: Model is part of the package aha.rules and explained elsewhere (section 9.6).

A agent that is using the learning algorithm as implemented in *aha.rules.learning* only uses the methods exported in *FuzzyLearner*. This class offers all methods necessary to incrementally add new data samples. *FuzzyLearner* decides afterwards whether the data sample provided is a punishment or a reward and takes the necessary actions to adapt the ruleset. *FuzzyLearner* also takes care of the actual Model that instanciates the rules learnt. It continuously updates the model with the updated rulebase so that the Model can be used without interruption, even during the time learning takes place. See section 9.6 for a detailed description of the internals of the Model and it's supporting classes for dynamic rule creation and modification.

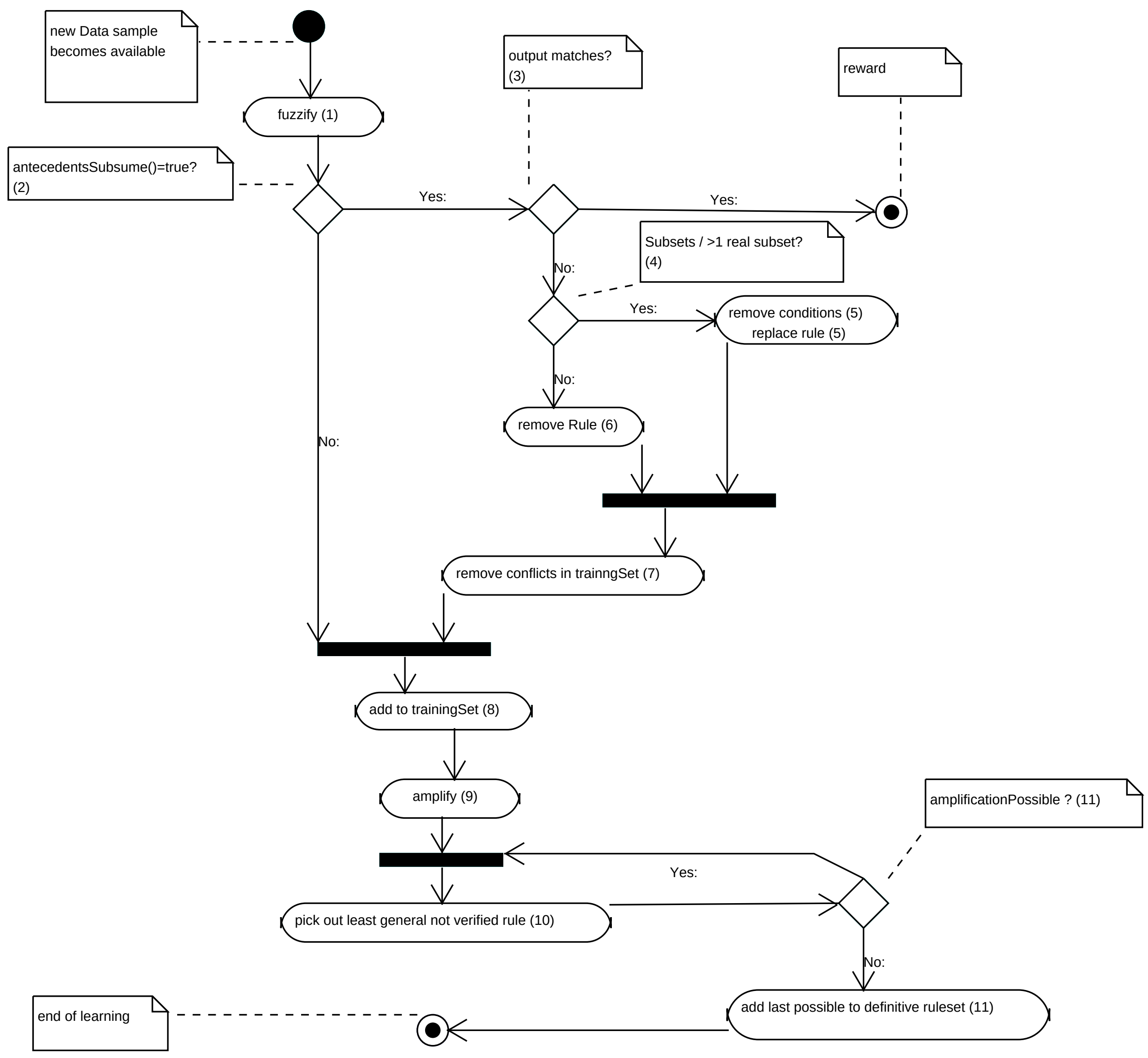

new Data sample becomes available
fuzzify (1)
antecedentsSubsume()=true? (2)
Yes:
output matches? (3)
Yes:
reward
Subsets / >1 real subset? (4)
No:
Yes:
remove conditions (5)
replace rule (5)
No:
remove Rule (6)
No:
remove conflicts in trainngSet (7)
add to trainingSet (8)
amplify (9)
amplificationPossible ? (11)
Yes:
pick out least general not verified rule (10)
No:
end of learning
add last possible to definitive ruleset (11)

Figure 8.1: UML activity diagram for the learning algorithm. All numbers refer to the respective steps (0 to 9) of the description of the algorithm.

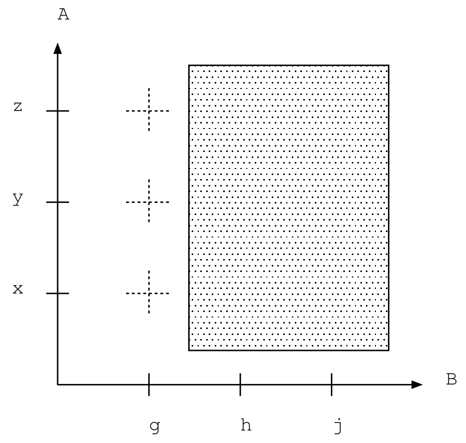


Figure 8.2: The state space covered by the rule $((\{x, y, z\}, \{h, j\}), y_{j2})$

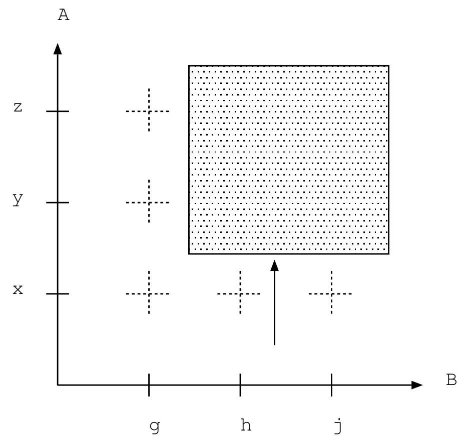


Figure 8.3: The state space covered by the rule $((\{y, z\}, \{h, j\}), y_{j2})$

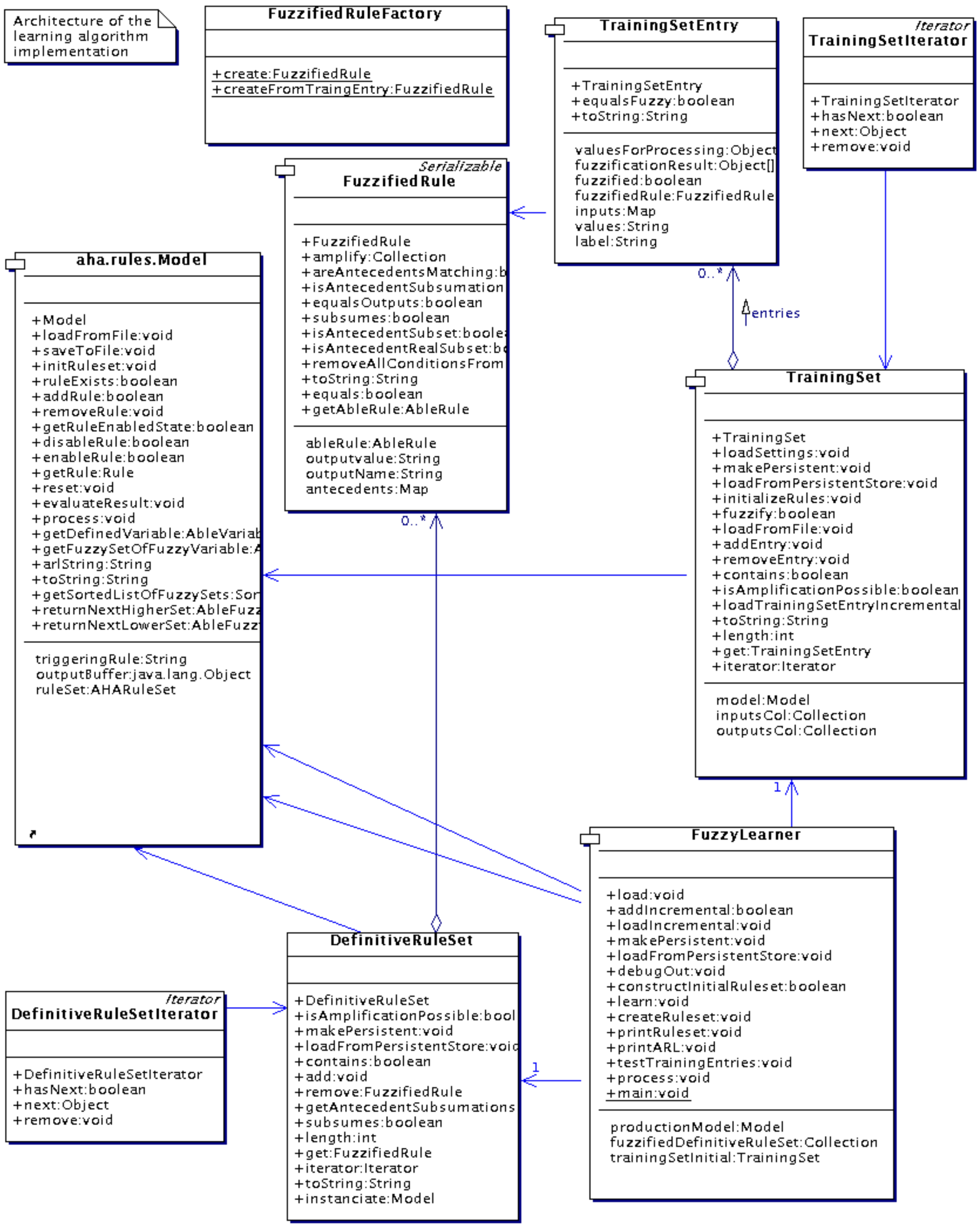


Figure 8.4: UML class diagram of the learning algorithm implementation

Chapter 9

Anytime learning and fuzzy inferencing applied to intelligent buildings

9.1 Overview

This chapter is about the actual decision making on basis of the rulebase learnt and constructed by the learning algorithm. The basic fuzzy logic principles used throughout this chapter are explained in detail in appendix A. Appendix B gives details about ABLE and ARL. Especially familiarity with ARL is useful to understand the content of this chapter.

9.2 Model and learning units

The abstract concept underlying all decision making is the *Model*. A model is a internal representation on which basis all decisions are taken and all learning takes place. Each model consists of a set of rules which are modified according to feedback from the environment. Every model is independent of all other models in the system which implies that every model has exclusive access to some output value (effector).

All learning takes place on the level of learning units. Every *learning unit* has an associated model. Model and learning units are thus tightly integrated structures. They take decisions about the same output value(s) and base there decisions on the same input values. They, however, have different responsibilities. The model is responsible to take decisions on basis of existing rules whereas the learning unit is responsible for providing these rules to the model. The learning unit constantly adapts the rulebase of the model. This architecture is a two-layer inferencing process because there is a process that uses a model (learning unit) to take decisions about the rulebase of an other model (FLC).

Figure 9.1 shows the relationship between a model that takes decisions about a room and a learning unit. The learning unit (b) takes decisions about the rulebase of the model (a). For the learning the learning unit (b) uses the learning algorithm as well as a rulebase (c). The rulebase used for the learning process (c) is fixed and is only required for the fuzzification of the data samples that the learning unit gets as reinforcement signal. The learning unit fuzzifies this data samples and generates punishments/rewards from it. The model that takes decisions about the environment uses the rulebase (d) as a basis which gets constantly updated by the learning unit (b).

The learning algorithm as explained in chapter 8 is only capable of learning a rulebase for multiple input single output (MISO) fuzzy logic controllers. Because of this ABI is structured such that every output value (effector) is controlled by one learning unit and by one model.

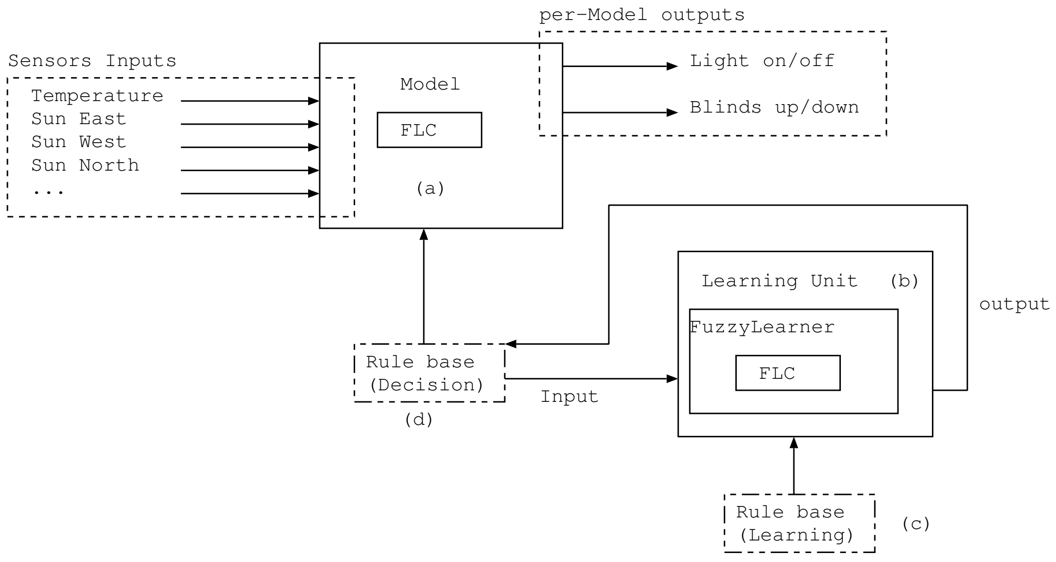


Figure 9.1: Relationship between Model and Learning Unit

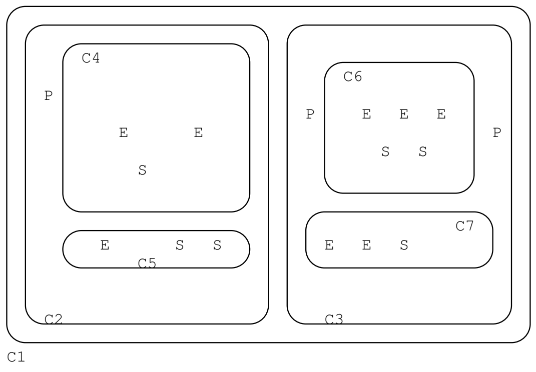


Figure 9.2: Structure of a single room (example)

Figure 9.2 shows how learning units are associated to clusters in a room. Cluster C1 consists of the two clusters C2 and C3. C2 contains the two clusters C4 and C5 which each are associated to a learning unit. C4 contains two effectors and one sensors and C5 contains two sensors and one effector. Because the clusters C4 and C5 each have there own learning unit they also have there own rulebase on basis of which decisions about the effectors in the clusters C4 and C5 are taken.

9.3 Decision Making

Decisions in AHA are taken by a fuzzy logic inferencing engine on basis of a rulebase. Every *Model*, which represents a number of rules, acts as an independent Fuzzy logic controller (FLC). This fuzzy logic controller takes care of the fuzzification, the inferencing itself and the defuzzification. See section A.1.3 for details.

Figure 9.3 shows an example of a possible FLC with input and output values. The actual input values and the output value of an FLC are dynamic. Which FLC gets associated with which input and output values is determined by the structure information delivered by the structure agent.

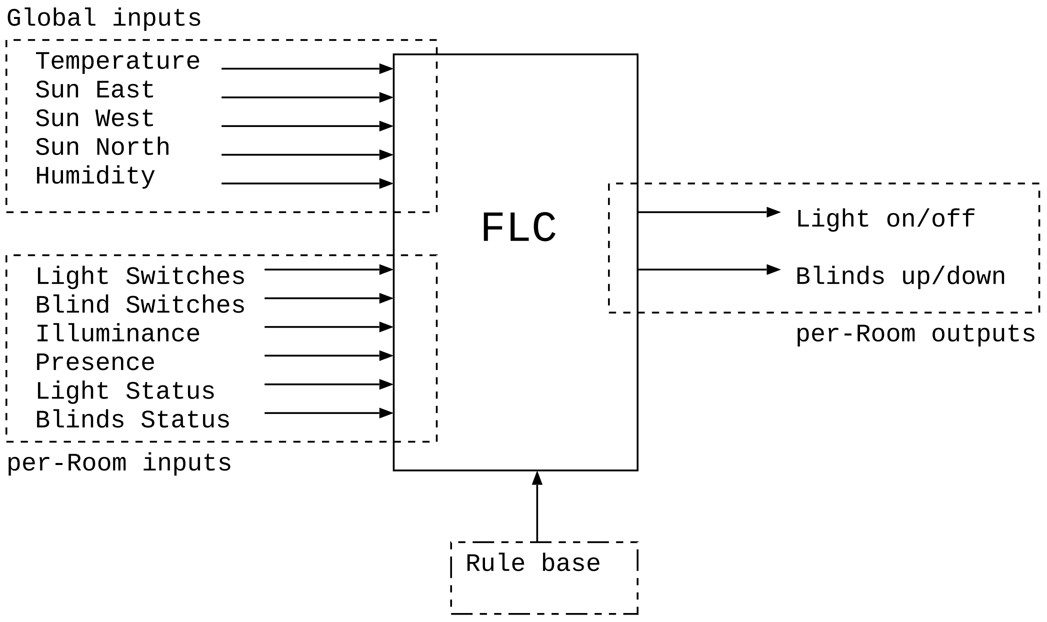


Figure 9.3: An FLC and it's inputs and outputs

9.4 Design of membership functions

The input values of the system that need to be fuzzified are:

- DayLight (indoor)
- Presence (Movement)
- Light status
- Blind status
- Humidity
- Temperature
- Illumination
- Radiation
- Time

For every of these input variables there is a fuzzy membership function associated with a number of fuzzy sets. These definitions of membership functions are static and pre-defined. The system doesn't change them during runtime. Membership functions are formally defined in ARL ([ARL]).

In the following sections some of the membership function definitions are given in ARL and graphical. For the definition of the other ARL files see the arl source files.

What is important about membership functions that are used for automatic learning processes is that they are defined for the whole domain of the fuzzy variable. To assure this the membership function must assign at least one fuzzy set a truth value that is greater than the alpha cutoff value. We use a alpha cutoff value of 0.2 which means that every fuzzy variable V must satisfy the condition $V(x, y) > 0.2$ for every x in the domain of the fuzzy variable for at least on y whereas y is a fuzzy set.

9.4.1 Radiation

Figure 9.4 shows the ARL definition of the fuzzy variable *Radiation*. Figure 9.5 shows the graph of the fuzzy variable *Radiation*.

```
Fuzzy RadiationEast = new Fuzzy(0.0 , 10000.0)  {
  Sigmoid    A_completlyDark = new Sigmoid  (0.0, 10.0, 20.0, ARL.Down);
  Shoulder  E_lotsOfSun = new Shoulder (3000.0, 5000.0, ARL.Right);
  Triangle   D_sun = new Triangle (900.0, 2000.0, 3500.0);
  Triangle   B_littleLight = new Triangle (15.0, 50.0, 250.0);
  Triangle   C_light = new Triangle (200.0, 600.0, 1400.0);
 };
```

Figure 9.4: Membership function of radiation

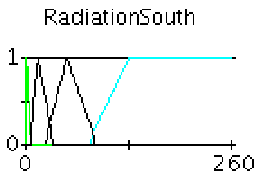


Figure 9.5: Graphical version of the membership function radiation

9.4.2 Temperature

Figure 9.6 shows the ARL definition of the fuzzy variable *Temperature*. Figure 9.7 shows the graph of the fuzzy variable *Temperature*.

```
Fuzzy Temperature = new Fuzzy(-30.0 , 100.0)  {
  Shoulder  E_hot = new Shoulder (28.0, 100.0, ARL.Right);
  Triangle   D_veryWarm = new Triangle (18.0, 25.0, 39.0);
  Shoulder  A_veryCold = new Shoulder (-30.0, 0.0, ARL.Left);
  Triangle   C_warm = new Triangle (8.0, 15.0, 22.0);
  Triangle   B_cold = new Triangle (-5.0, 5.0, 14.0);
 };
```

Figure 9.6: Membership function of Temperature

9.4.3 Illumination

Figure 9.8 shows the ARL definition of the fuzzy variable *Illumination*. Figure 9.9 shows the graph of the fuzzy variable *Illumination*.

9.4.4 Daytime

Figure 9.10 shows the ARL definition of the fuzzy variable *Daytime*. Figure 9.11 shows the graph of the fuzzy variable *Daytime*.

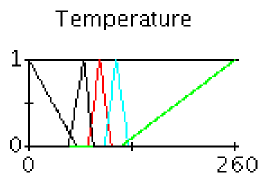


Figure 9.7: Graphical version of the membership function temperature

```
Fuzzy Illumination1 = new Fuzzy(0.0 , 10000.0)  {
    Sigmoid    A_completlyDark = new Sigmoid  (0.0, 10.0, 20.0, ARL.Down);
    Shoulder  E_lotsOfSun = new Shoulder (3000.0, 5000.0, ARL.Right);
    Triangle   D_sun = new Triangle (900.0, 2000.0, 3500.0);
    Triangle   B_littleLight = new Triangle (15.0, 50.0, 250.0);
    Triangle   C_light = new Triangle (200.0, 600.0, 1400.0);
   };
```

Figure 9.8: Membership function of illumination

9.5 Structure of rulebase

The rulebase for the different models in ABI contain three parts:

1. Definition of fuzzy variables
2. Static rules
3. Dynamic rules

The static rule part consists of a block of rules which are static. This rules are used for the fuzzification of the data samples by the learning process. The labels of static rules start with the prefix *SUPPORT_*. The other block of rules is the block of rules that are continuously adapted by the learning process. These rules can be removed and modified and the learning process can add new ones to this block of rules. The labels of dynamic rules start with the prefix *RULE_*.

9.5.1 Fuzzy OR operator

Because the fuzzy inferencing engine of ABLE doesn't support the fuzzy OR operator the learned rules (which use fuzzy OR and AND) can't be used without converting them into an equal set of conditions combined with fuzzy AND.

Converting a list of conditions that are combined with OR into a list of conditions that are combined with AND is no problem in boolean logic but not in fuzzy logic. Figure 9.12 and Figure 9.13 illustrates the problem. Figure 9.13 shows the fuzzy hedge *not* applied to the fuzzy set normal of the variable *DayLightIndoor*. Because the slope of falling edge is not steep enough the *not normal* fuzzy set of *DayLightIndoor* is also true in most of the range where the fuzzy set *normal* is also true. This constructs a situation where there are states where both states, *normal* and *not normal* are true.

To construct a fuzzy set that works as a complement to an other fuzzy set the rising and falling edges of the not function need to be much steeper. Figure 9.14 illustrates the function after applying the three hedges *not*, *positively* and *slightly*.

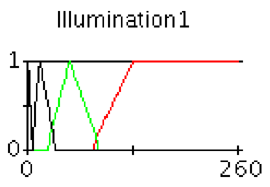


Figure 9.9: Graphical version of the membership function illumination

```
Fuzzy DayTime = new Fuzzy(0.0 , 86400.0)  {
  Linear    F_night2 = new Linear   (75600.0, 86400.0, ARL.Up);
  Linear    A_night1 = new Linear   (0.0, 18000.0, ARL.Down);
  Triangle   E_evening = new Triangle (57600.0, 68400.0, 79200.0);
  Triangle   B_morning = new Triangle (14400.0, 25200.0, 36000.0);
  Triangle   C_midday = new Triangle (28800.0, 43200.0, 57600.0);
  Triangle   D_afternoon = new Triangle (43200.0, 54000.0, 64800.0);
 };
```

Figure 9.10: Membership function of DayTime

9.5.2 OR to AND conversion

The structure of the fuzzified rules that result from the learning process is $((\{a, b, c\}, \{x, y, z\}), o_1)$ as described in chapter 8. The sets E_i, $\{a, b, c\}$ and $\{x, y, z\}$ in the above example, are meant to be translated to rules as *X is a or b or c*.

Because a fuzzy or operator isn't available the inverse set $E_{iinv} = \mathcal{L}(X) - E_i$ for every variable X is taken and connected with *and not positively slightly*. Applied to the example above (under the assumption that $\mathcal{L}(X_1) = \{a, b, c, d, e\}$) this would result in the rule *X is not positively slightly d and not positively slightly e*.

9.6 Implementation

Figure 9.15 shows the UML class diagram of the package *aha.rules* which contains all code related to the underlying Model of a FLC as well as all code related to the dynamic creation of rules.

The implementation (which is in package *aha.rules*) is structured into classes as following:

- *Model*: Represents a generic model, consisting of a set of rules and methods for modifying, adding and removing rules.
- *ControlModel*: Specific Model for learning that inherits from Model. Implements building specific behavior.
- *Rule*: Represents a single rule. For this it is associated with an instance of AbleRule.
- *RuleInfo*: Encapsulates additional attributes of a rule.
- *AHARuleSet*: A set of rules (consisting of several blocks of rules) that inherits from AbleRuleSet. It's only purpose is to get access to the protected attributes of AbleRuleSet.
- *AHARuleBlock*: Inherits from AHARuleBlock. Supports the dynamic removal and addition of rules from/to AbleRuleBlock instances.
- *DecisionHistory*: Class that manages a history of all decisions taken.

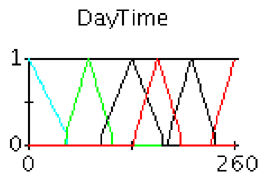


Figure 9.11: Graphical version of the membership function daytime

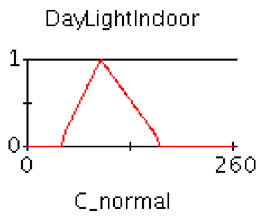


Figure 9.12: The fuzzy set normal of the variable Daylightindoor

- *DecisionHistoryEntry*: Represents a single DecisionHistory entry. A decision history entry stores all input values that were valid at the point of time a decision was taken.

9.6.1 Dynamic creation of rules

New rules produced by the learning process are first created as instances of *FuzzifiedRule*. *FuzzifiedRule* has a method *getAbleRule* that constructs a *Rule* instance out of a *FuzzifiedRule*. This method uses various support routines to dynamically create a new rule that can be inserted into an AbleRuleBlock.

9.7 Related work

In [Sto00] (Layered reinforcement learning) a similar approach is used but with reinforcement learning algorithms but the multi-layered learning principle used is the same (output of one learning unit is input to an other).

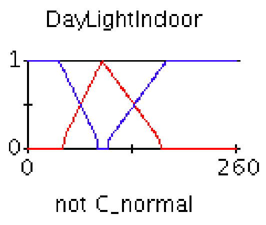


Figure 9.13: Fuzzy hedge not applied to the fuzzy set normal of the variable Daylightindoor

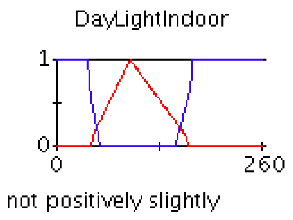


Figure 9.14: Fuzzy hedges not positively slightly applied to the fuzzy set normal of the variable Daylightindoor

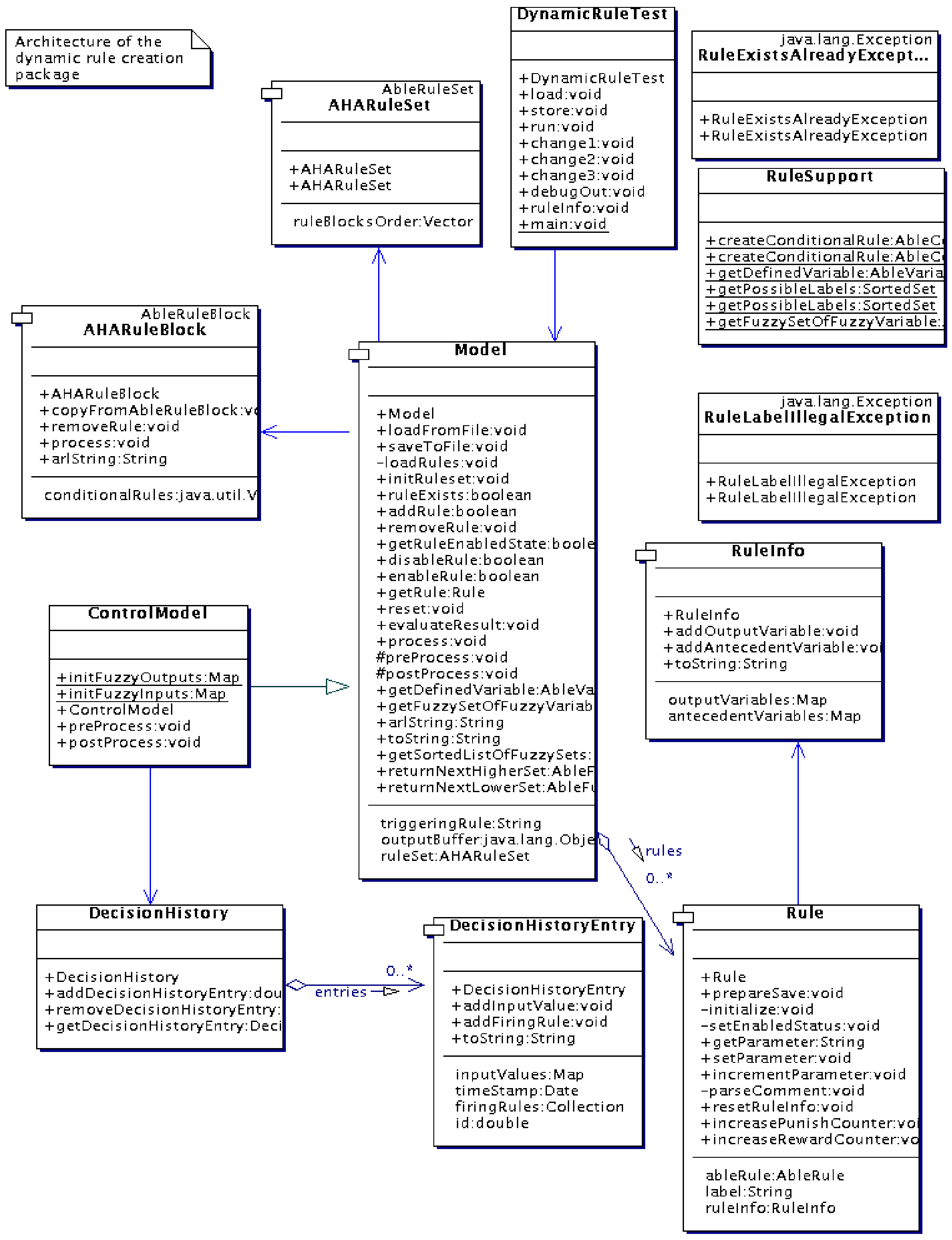


Figure 9.15: UML class diagram of the package *aha.rules*

Chapter 10

Agents

ABI consists of a number of agents as shown in figure 4.1. Every agent is communicating with other agents with asynchronous messaging. There are no synchronous method calls between agents.

In this chapter every agent is described in detail. This includes a specification of all messages sent out and all messages received by every agent.

Every agent inherits directly from the class *AHAAgent* which implements functionality which is common to all agents. *AHAAgent* indirectly implements *AbleJASAgent* which makes every AHAAgent also a JAS compliant agent.

10.1 Messaging

Table 10.1 lists all messages and their parameters. The only content of messages are key/value pairs which are the parameters. There is no other payload. Content is identified by a string (*aha.msg.**). The identification of a message is a system wide unique ID.

ID	Description	Sent to	Parameters: Name	Parameters: Type
SUBSCRIBE_INTEREST	Subscribe to topic	distribution agent	aha.msg.topic	String
			aha.msg.sender	AgentDescription
DESUBSCRIBE_INTEREST	desubscribe from topic	distribution agent	aha.msg.topic	String
			aha.msg.sender	AgentDescription
DISTRIBUTE_MESSAGE	Request delivery to topic	distribution agent	aha.msg.topic	String
			aha.msg.receiver	AgentDescription
			aha.msg.type	String
			aha.msg.content	Object
			aha.msg.delay	Integer
TIMED_EVENT	Schedule message for later delivery	distribution agent	aha.msg.seconds	Integer
			aha.msg.sender	AgentDescription
			aha.msg.locator	Locator
			aha.msg.receiver	AgentDescription
			aha.msg.msg	AHAMessage
SET_PROPERTY	Update value (modify)	bus agent	any	any
GET_PROPERTY	Request value	bus agent	aha.msg.property	String
			aha.msg.sender	AgentDescription
VARIABLE_UPDATE	New value available	arbitrary	any	any
TOPIC_CREATE	New topic created by subscriber	arbitrary	aha.msg.topic	String
TOPIC_REMOVE	Last subscriber unsubscribed	arbitrary	aha.msg.topic	String
GET_ELEMENTS	Request structure information	structure agent	aha.msg.clusterId	Integer
			aha.msg.elementType	Integer
			aha.msg.sender	AgentDescription
GET_ELEMENTS_RESULTS	Structure information	arbitrary	aha.msg.clusterId	Integer
			aha.msg.elementType	Integer
			aha.msg.elements	Collection
PERSON_ENTERED	Person entered a room	arbitrary	aha.msg.clusterId	Integer
			aha.msg.personId	Integer
PERSON_LEFT	Person left a room	arbitrary	aha.msg.clusterId	Integer
			aha.msg.personId	Integer
ANNOUNCE_RULE	Human-readable rulebase	arbitrary	aha.msg.clusterID	Integer
			aha.msg.type	String
			aha.msg.arlRuleBase	String
			aha.msg. fuzzifiedRuleBase	Collection

Table 10.1: Table of all messages

10.2 Bus Agent

10.2.1 Functionality

The bus agent provides a way of accessing the field bus of a building. The bus agent acts as a proxy between the other agents and the bus abstraction implementation itself. The current BusAgent is an improved version of the one used in the predecessor project AHA. The added functionality was the change from RMI to message based communication and the encapsulation of variable names behind ids.

10.2.2 Messages

BusAgent receives the following messages :

- TOPIC_CREATE
- TOPIC_REMOVE
- GET_PROPERTY
- SET_PROPERTY

10.2.3 Implementation

Bus agent instantiates a concrete implementation of the bus abstraction layer. It reads from the config file which implementation to instantiate.

All common tasks between BusAgent and BusSim have been extracted into the common class BusFrontend. As an example although the messages are received by BusAgent, it is actually BusFrontend who processes them.

For the encapsulation of variable names BusAgent needs a config file which specifies how to map between ids and name. The grammar for this file is shown in listing C.4

The implementation (which is in package *aha.bus*) is structured into classes as following:

- *BusAgent*: interfaces to the actually used implementation of the BusController interface.
- *BusFronted*: does all the message handling for BusAgent
- *BusController*: interface which must be implemented for every field bus type.

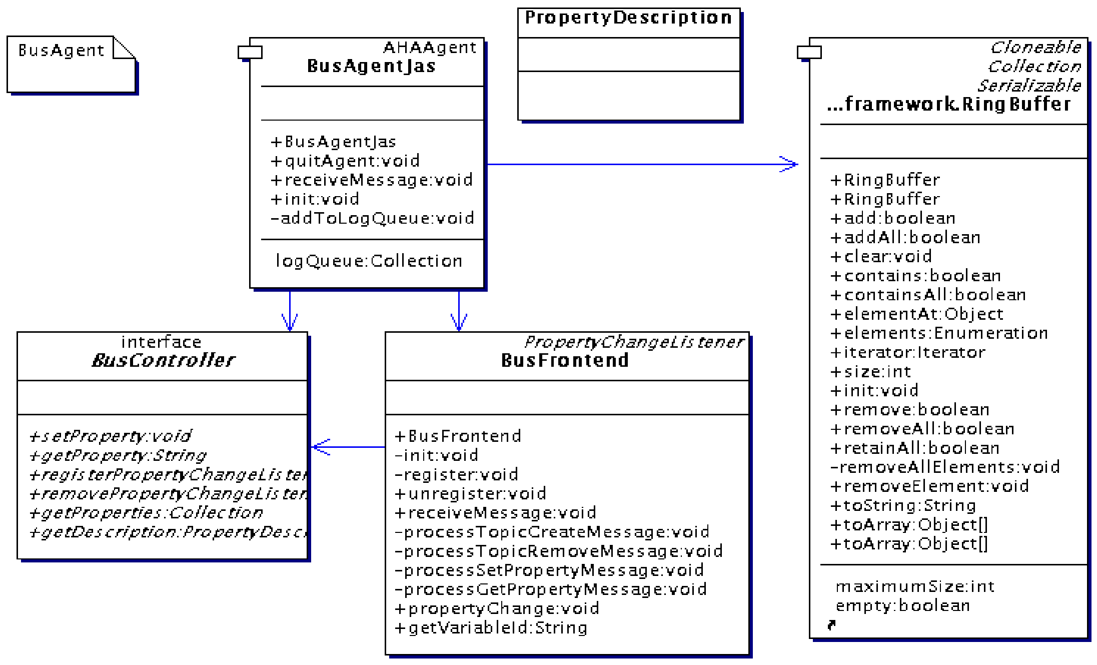


Figure 10.1: UML class diagram of the bus agent

10.3 Structure Agent

10.3.1 Functionality

This agent is responsible to deliver information about the structure of the building and its sensors and effectors to its clients. See 3.3 for an elaboration on the reasons for creating such a structure information.

10.3.2 Messages

Structure Agent only listens to GET_ELEMENTS messages. It recieves them from other clients wanting information about the structure of the building, field bus. In response to these messages it sends out GET_ELEMENTS_RESULT messages, which contain a collection of the matching ElementInfo classes.

The GET_ELEMENTS message contains the following fields :

- aha.messaging.clusterId = Java.lang.Integer
- aha.messaging.elementType = Java.lang.Integer
- aha.messaging.getRecursiveElements Java.lang.Boolean

elementType specifies which kind of element that you want. Examples are : light, blinds, blindSwitch, etc. The elementTypes are specified in the class aha.structure.ElementType.

clusterId specifies the clusterId in which Structure Agent should search for elements. I f null is passed it simply search in a list of all elements, if 0 is passed it only searches the top level elements/clusters.

getRecursiveElements specifies if Structure Agent should search recursively. A clusterId of 0 and recursive true would thus return as a clusterId of null.

10.3.3 Implementation

As described in 3.3 this implementation makes no dynamic discovery of structure information. It reads all the data from an XML file. See refstructureGrammar for the grammar of this file. 10.1 is an example of this file. On can clearly see the two concepts cluster and element.

Listing 10.1: example config structure agent

```
<ahaStruct>
        <cluster id="1" displayName="4154_" >
                <cluster id="2" displayName="4154_" >
                        <cluster id="2000" displayName="4154_">
                        <element id="3" type="light" effector="true" sensor="true"
                            feedback="false"/>
                        <element id="4" type="light" effector="true" sensor="true"
                            feedback="false"/>
                        <element id="5" type="lightSwitch" effector="false" sensor="
                            true" feedback="true"/>
                        <element id="6" type="lightSwitch" effector="false" sensor="
                            true" feedback="true"/>
                        <element id="15" type="light" effector="true" sensor="true"
                            feedback="false"/>
                        <element id="16" type="light" effector="true" sensor="true"
                            feedback="false"/>
                        <element id="17" type="lightSwitch" effector="false" sensor
                            ="true" feedback="true"/>
                        <element id="18" type="lightSwitch" effector="false" sensor
                            ="true" feedback="true"/>
                        <element id="19" type="lightSwitch" effector="false" sensor
                            ="true" feedback="true"/>
                        <element id="20" type="lightSwitch" effector="false" sensor
                            ="true" feedback="true"/>
                        </cluster>
```

Altough it is not visible in the configuration file Structure Agent treats elements and cluster as almost the same. For clusters it returns also ElementInfo instances. It manages all these elements in two collections, one contains only clusters and the other contains all ElementInfos. The first one is used to find clusters and their sub clusters.

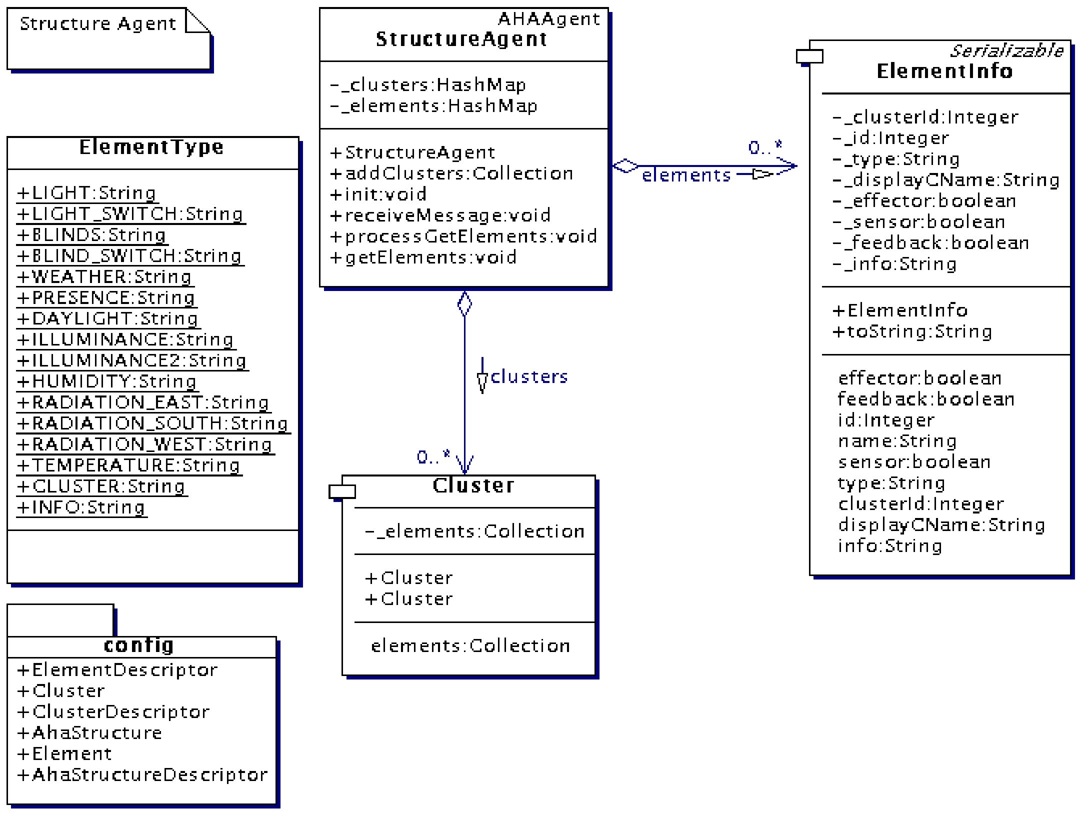


Figure 10.2: UML class diagram of the structure agent

The implementation (which is in package *aha.structure*) is structured into classes as following:

- *StructureAgent*: Agent itself, does all the management of the elements
- *ElementType*: contains constants for all the element types we use
- *ElementInfo*: value object contains all the information we have about an element
- *Cluster*: used to manage cluster - sub cluster relationships

10.4 Simulator Agents

10.4.1 BusSim - Simulation of BusAgent

Functionality

The most important part which was in need for a simulated replacement was the BusAgent which in turn used LNSController to query and update the building network. This Agent is responsible to imitate the behavior of all the sensors and effectors. It has to simulate not only all values sensor have during the simulation but also all the relationships between sensor and effectors. The design of this subsystem greatly affects how accurate our simulation will be.

Messages

As BusSim should be a replacement for BusAgent, it should logically also receive and send the same messages as BusAgent does. These messages are : setproperty and property update. See chapter 10.2 for more details about these messages.

Implementation

As mentioned before the design of this agent greatly affects the precision of the simulation. A simple approach for such a simulation is to see the building network as a group of variables and certain connections between different variables. An example for such a connection would be between a light switch and the actual light. Whenever the light switch's state changes, the light's state should also change. When and how this state changes can mostly be described by some simple rules.

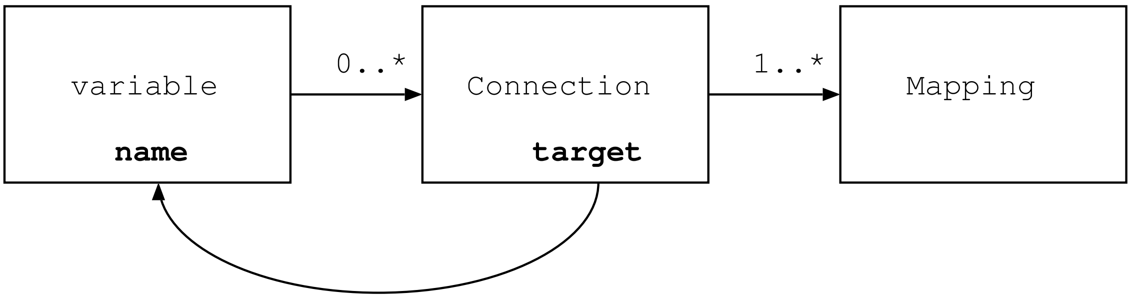


Figure 10.3: Relationship between Variables/Connections/Mappings

We actually went this way and first designed an XML-Grammar to describe such a simulation see listing C.1. This grammar consist of 5 constructs :

- ahaSim
  Top level construct, contains several devices
- Device
  A simple construct to group variables which belong to the same device, is child of ahaSim
- Variable
  A variable with the following attributes : name, id, default and comment. Name being the internal name for the simulation and id being the numerical id which is also known to the clients of BusSim. See chapter 10.3
- Connection
  Connections show which other variables are affected by changes to a certain variable. Connections are

always children of variables. The attribute target points to the target variable, the internal name is used. To enhance the power and expressiveness of the configuration language regular expressions can be used for the names of connection targets.

- Mapping
  The Mappings describe how a connections target should be affected. See more below.

Mappings are the most complex of these constructs. Thus they deserve their own paragraph. A connection may always have more but at least one mapping. See figure 10.3 for an illustration of the relationship between these concepts. It's attributes can be divided in two groups. The condition and action attributes. The attributes belonging to the first group are cmp and value. These two attributes specify when that mapping, resp. it's action should executed. The attribute value is a simple string which is somehow compared to the actual value of the mappings parent variable.

Listing 10.2: Extract a config file for a simulation

```
<device  room="4154_" name="Y355G-4154_-UPL_LT_C">
        <variable name="Y355G-4154_-UPL_LT_C.nvoSetting[2].2" id="19" default="
            SET_OFF 0.0 0.00">
                <connection target=".*-4154_-PHI_LRC_C.nvi0102irSetting.14">
                        <mapping cmp="contains" value="SET_OFF" action="set">
                                SET_OFF 0.0 0.00
                        </mapping>
                        <mapping cmp="contains" value="SET_ON" action="set">
                                SET_ON 0.0 0.00
                        </mapping>
                </connection>
        </variable>
        <variable name="Y355G-4154_-UPL_LT_C.nvoSetting[3].3" id="20" default="
            SET_OFF 0.0 0.00">
                <connection target=".*-4154_-PHI_LRC_C.nvi0102irSetting.15">
                        <mapping cmp="contains" value="SET_OFF" action="set">
                                SET_OFF 0.0 0.00
                        </mapping>
                        <mapping cmp="contains" value="SET_ON" action="set">
                                SET_ON 0.0 0.00
                        </mapping>
                </connection>
        </variable>
</device>
```

The other two attributes specify what should be done whenever the variable value changes and this mapping is responsible.

The actions are more or less self-explanatory. But two of the supported actions are more complicated and shall be explained here. These two actions are needed to simulate the behavior of the daylight sensor. Its behavior is a not very simple since it also involves the state of other variables.

filter : This action adds a new filter to a variable. A filter amplifies or lessens another variables value. It consists of a source variable and the filters value.The filter concept is used for every update of a variable. Whenever the variable needs to be updated the filter is applied. The source variables value is loaded, multiplied with the filter value and stored in the target variable.

Listing 10.3: example config for a variable filter

```
<variable name="Y455G-4154_-MS_STA_G29-30.nviSettingLcl.12" id="24" default="SET_OFF
      0.0 0.00">
        <connection target=".*-4154_-HTS_ECO.nvoDaylightLux.5">
                <mapping cmp="contains" value="SET_OFF" action="filter">
                        Y355G-41_-GRELLOW.nvoG91___080MB1.30,0.4
                </mapping>
                <mapping cmp="contains" value="SET_ON" action="filter">
                        Y355G-41_-GRELLOW.nvoG91___080MB1.30,-0.4
                </mapping>
        </connection>
</variable>
```

Listing 10.3 shows an example how a filter is configured. Here we have a filter which adds or subtracts 0.4 to the daylights filter filter value whenever the blinds are down or up. This reflects the fact that it is darker when the blinds are down. It is also possible that several variables can change another variables filter value.

update Update is used to make a variable with filters update its value. This action should be used whenever the source of a variables filter changes its value.

Listing 10.4: example config for an update action

```
<variable name="Y355G-41_-GRELLOW.nvoG91___080MB1.30" id="115" default="7">
        <connection target=".*-HTS_ECO.*\\.nvoDaylightLux.5">
                <mapping cmp="" value="" action="update">
                </mapping>
        </connection>
</variable>
```

Listing 10.4 shows an example of update. Whenever the exterior illumination changes, all the daylight sensors in the building should be updated. This is also a nice example for the usage of regular expressions. The Pattern .*-HTS_ECO.*\.nvoDaylightLux.5 matches to all daylight sensors in the building. We thus have a connection from GRELLOW to every daylight sensor in the building. Now always when illuminance changes all the daylight sensors are updated.

timed events
The building we were working with provided us with some exterior sensors like temperature, illumination, humidity etc. These are sensors which just change all over the day. We simulate these by simple repeating values we have sampled form a real system. In the top-level tag of the simulation config file one can specify a file containing a list of sampled data, see listing 10.5. Upon start of the simulation this data is sent to Simulator Agent who sends according SET_PROPERTY messages at the right time.

Listing 10.5: example reference to timed events

```
<ahaSim timedEventsFile="examples/weatherdata.20020616" startTime
    ="2002-06-16 00:00:00">
...
</ahaSim>
```

Listing 10.6: example for the list of timed events

```
Y355G-41_-GRELLOW.nvoG91___080MB1.30,2002-06-16 00:00:22,7
Y355G-41_-GRELLOW.nvoG91___080MB1.30,2002-06-16 04:42:35,9
Y355G-41_-GRELLOW.nvoG91___080MB1.30,2002-06-16 05:14:31,31
Y355G-41_-GRELLOW.nvoG91___080MB1.30,2002-06-16 05:27:10,53
```

The implementation (which is in package *aha.sim*) is structured into classes as following:

- *BusSim*: receives messages and manages the variable values
- *Connection*: models the connection between variables
- *Mapping*: models the mappings (triggers and actions)

10.4.2 Simulator Agent - Control of simulation time

Functionality

Simulator Agent is responsible for the control of the whole simulation. It start, stops and pauses the simulation. It controls the simulation time and manages the timed events.

Messages

The following messages received by Simulator Agent control the simulation :

- SIMULATION_RUN switches the simulation between running and paused forth and back
- SIMULATION_STOP stops the simulation completely, another message SIMULATION_RUN is needed to restart the simulation
- SIMULATION_SET_DELAY sets the simulation speed, specifies how many milliseconds the simulator agent should sleep before advancing to the next simulation second

The message TIME_EVENT adds another timed event to the pending event queue. The message contains another message which is sent to its designated receiver when the specified amount of seconds have passed.

Simulator Agent sends the message TIME, which is the actual simulation time, every 10 simulation seconds. This message is sent to the topic time. Agents interested in the actual time should thus subscribe for that topic instead of reading the local system time.

Implementation

The simulation of time is achieved by using Able's TimerEvent mechanism. Able calls the method processTimerEvent every n seconds. Where n is specified calling the method setDelay. This call is done whenever Simulator Agent receives the SIMULATION_SET_DELAY message.

Simulator Agent has an ordered map which stores all the timed events. These timed events are stored in instances of the class WakeUpEvent. Every time when the simulated time is advanced Simulator Agent checks if there are any expired WakeUpEvents and sends them to their receiver.

The implementation (which is in package *aha.sim*) is structured into classes as following:

- *SimulatorImpl*: The Simulator Agent, the postfix Impl is a historical leftover from when we still used RMI to control Simulator Agent
- *WakeUpEvent*: stores a timed event and knows how to send it

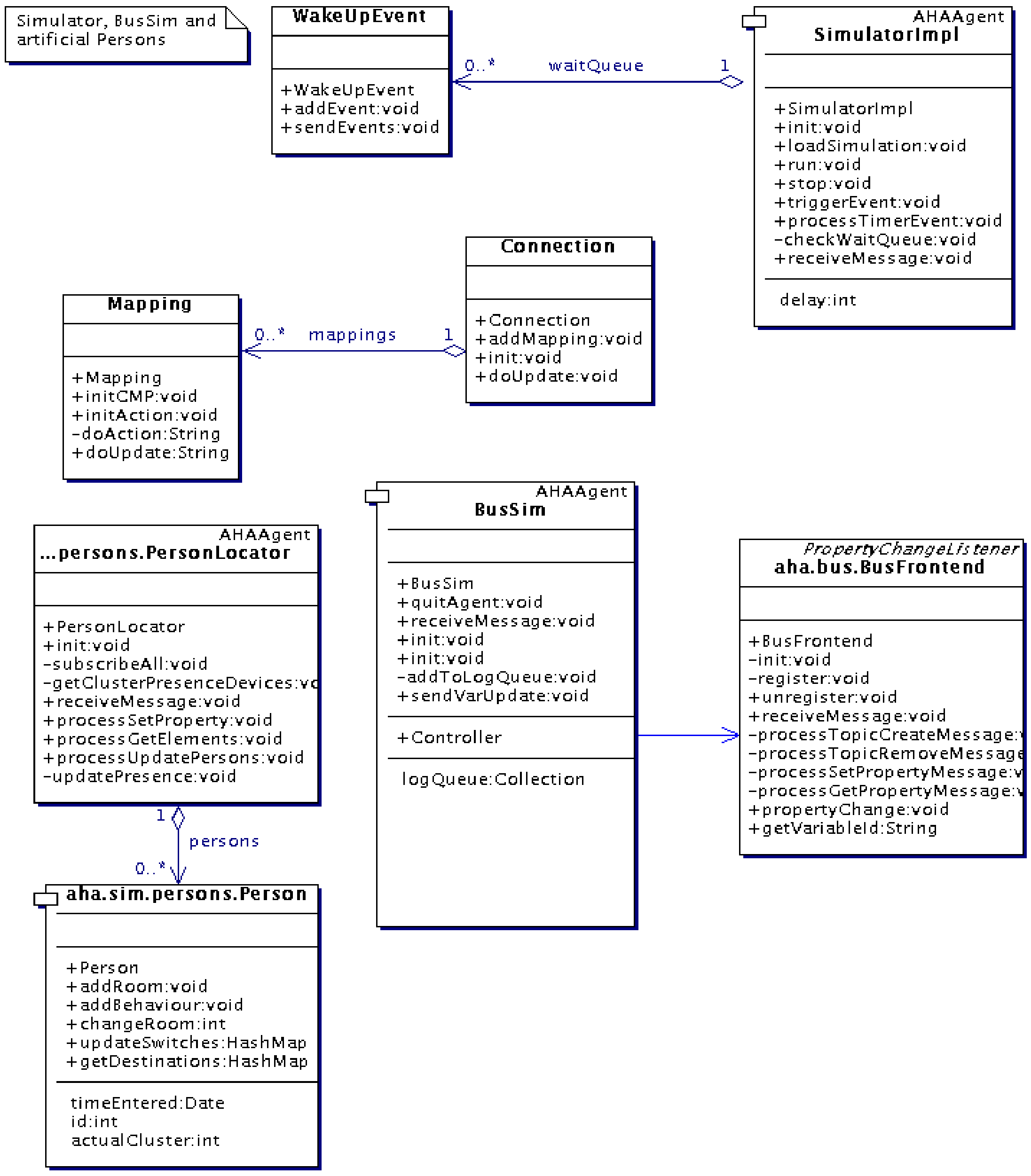


Figure 10.4: UML class diagram of the simulation agents (simulation,bussim, artificial persons)

10.5 Control Agent

10.5.1 Functionality

The control agent is responsible for controlling a single room (which is represented by a cluster in the structure information as communicated by the structure agent). It controls all effectors of a single room and declares it's interest in all sensory input data values available for the room it controls. Depending on the number of subclusters of one room it instanciates a number of learning units and models on which basis it generates punishment and reward reinforcement signals for the learning processes. It uses the dynamically generated rules generated by the learning process for taking decisions.

All structure information used by the control agent is request by it from the structure agent. A room is typically divided into a number of light clusters and a number of blind clusters (see Figure 9.2 for an example). A light cluster contains all light switches and light control units which are connected together (eg the same switch turnes on several lights) whereas a blind cluster contains all blind controllers and blind switches which are connected together. This subclusters are the basic units on basis of which rulebases are learnt. Additionally, at least one presence detector is assigned to every subcluster.

The specific structure of such a subcluster is different for every room and is managed by the structure agent.

10.5.2 Messages

The control agent sends the following types of messages:

- ANNOUNCE_RULE: ARL string of all rules currently used for processing. Is sent out every time the rulebase changes.
- GET_ELEMENTS: last subscription for a topic was removed, no one is interested in this topic anymore.
- SET_PROPERTY: sent to bus agent to interact with the environment (effectors).
- GET_PROPERTY: sent to bus agent to request a value of variable currently not available.
- GET_ELEMENTS: sent to structure agent to dynamically discover structure information.
- SUBSCRIBE_INTEREST: to declare a new interest to the distribution agent
- DESUBSCRIBE_INTEREST: to declare that the control agent is not interested in a topic anymore.

The distribution agent accepts this types of messages:

- VARIABLE_UPDATE: Update of a value for a variable.
- GET_ELEMENTS_RESULT: Answer to structure information request message.

10.5.3 Implementation

Figure 10.5 shows the UML class diagram of the control agent itself, figure 10.6 the diagram of all class representing the structure and relationships of a room and figure 10.7 all classes related to learning.

The implementation (which is in package *aha.control*) is structured into classes as following:

- *Room*: Represents a whole room, consisting of a recursive structure of subclusters and elements as obtained from the structure agent.

- *RoomControlAgent*: The control agent itself. Receives messages and forwards them to the respective instances for processing.
- *ControlSupport*: Various static support methods for converting data (like time).
- *SubCluster*: Subcluster, consisting of a number of learning units and a number of presence detectors.
- *Presence*: Determines for every subcluster the presence status (yes/no). Runs in it's own thread.
- *PresenceMovement*: ElementGroup. Manages a number of presence movement input values (aquired from presence detector). Processes incoming messages that contain data updates for presence movement variables.
- *PresenceLight*: ElementGroup. Manages a number of presence light values (aquired from presence detectors). Takes average/max values in case there is more then one presence detector assigned to this group. Processes incoming messages that contain data updates for presence light variables.
- *Weather*: Group of all weather variables. Processes incoming messages that contain data updates for weather variables.
- *LightSwitches*: Group of light switches. Forwards incoming messages to the respective reinforcement generator for generating punishments.
- *BlindSwitches*: Group of blind switches. Forwards incoming messages to the respective reinforcement generator for generating punishments.
- *InputGroup*: Group of input variables (sensors).
- *ElementGroup*: Group of any variables (input and output).
- *OutputGroup*: Group of output variables (effectors).
- *Blinds*: Group of blinds (output). Supports methods like *allUp* and *allDown*.
- *Lights*: Group of lights (output). Supports methods like *allOf* and *allOn*.
- *Inputs*: Static definitions of various constants valid for the whole control agent.
- *Learner*: Runs in it's own threads. Collects punishments and rewards from the reinforcement generators and runs the incremental learning. Updates the rulebase.
- *LearningUnit*: Generic learning unit. Represents a single model and a single rulebase.
- *LearningUnitBlinds*: Blinds specific learning unit. Takes decisions about blinds and executes them.
- *LearningUnitLight*: Lights specific learning unit. Takes decisions about lights and executes them.
- *ReinforcementGeneratorLight*: Runs in it's own thread. Evaluates goal function and generates punishments and rewards.
- *ReinforcementGeneratorBlinds*: Runs in it's own thread. Evaluates goal function and generates punishments and rewards.

As shown in Figure 10.6 every elementtype of a building (light, lightswitch, etc.) is represented by a single class that inherits from ElementGroup. Every kind of elementgroup manages all sensors/effectors of the same type.

The present detector is a special case: It's raw values are managed by the ElementGroup subclasses *Presence-Movement* and *PresenceLight* but there is an additional class *Presence*. Methods of this class are constantly running in an own thread. This thread uses the raw values of the presence detectors to infere whether there is a person present or not. This information is not directly available from the presence detector as the presence detector is only delivery a value when there is a change of movement and not when there is no movement at all.

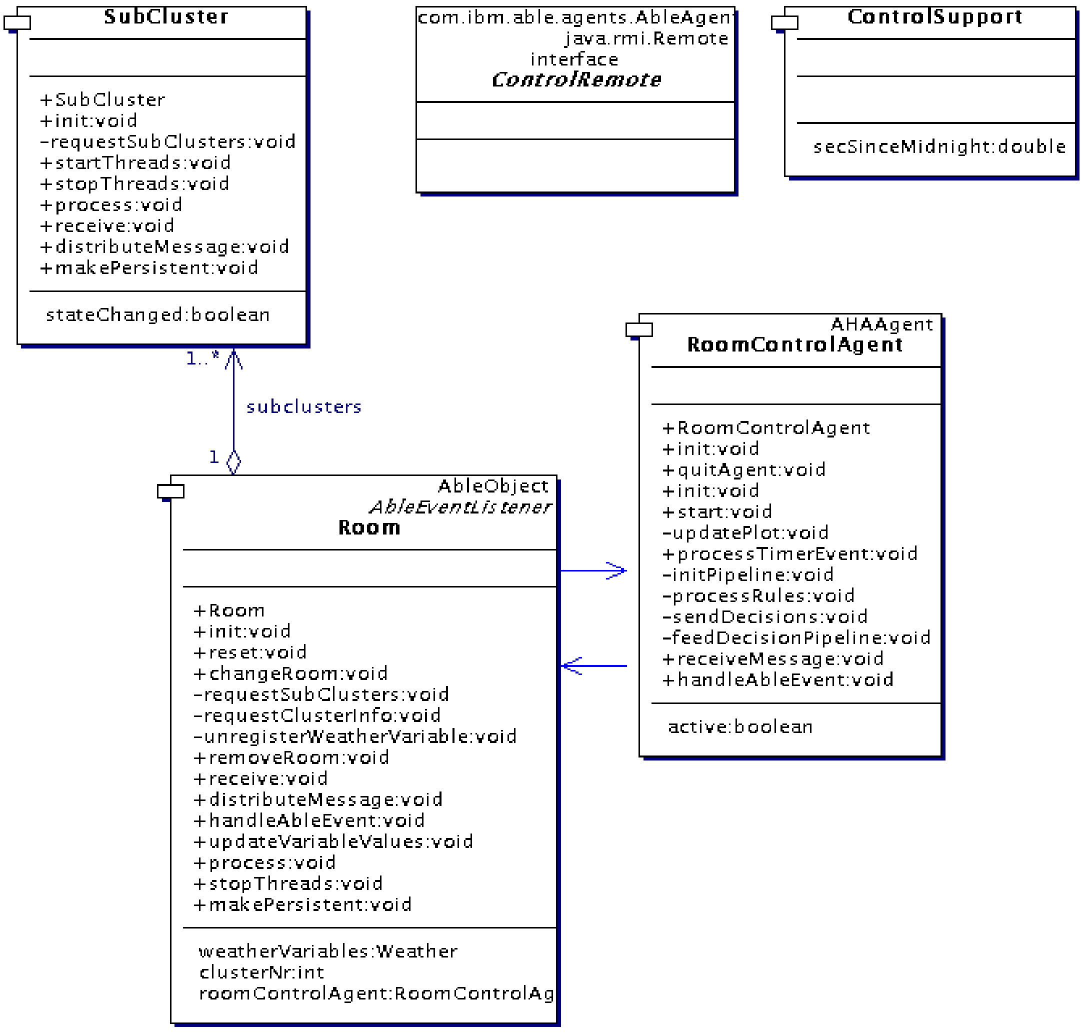


Figure 10.5: UML class diagram of the control agent (Agent itself, part 1)

Threads of either type *ReinforcementGeneratorLight* or *ReinforcementGeneratorBlinds* are running for every learning unit. The reinforcement generator threads have fix coded goal functions that are constantly evaluated. In case a certain goal isn't met anymore (like light needs to be off when there is noone present) a punishment is sent to the learning unit to which the particular effector is connected.

For every learning unit there is also a Thread of type Learner. This thread deals with incoming rewards and punishments and feeds them to the learning process and adapts the underlying rulebase. The learning happens in it's own thread to make sure that learning and decision making can happen at the same time.

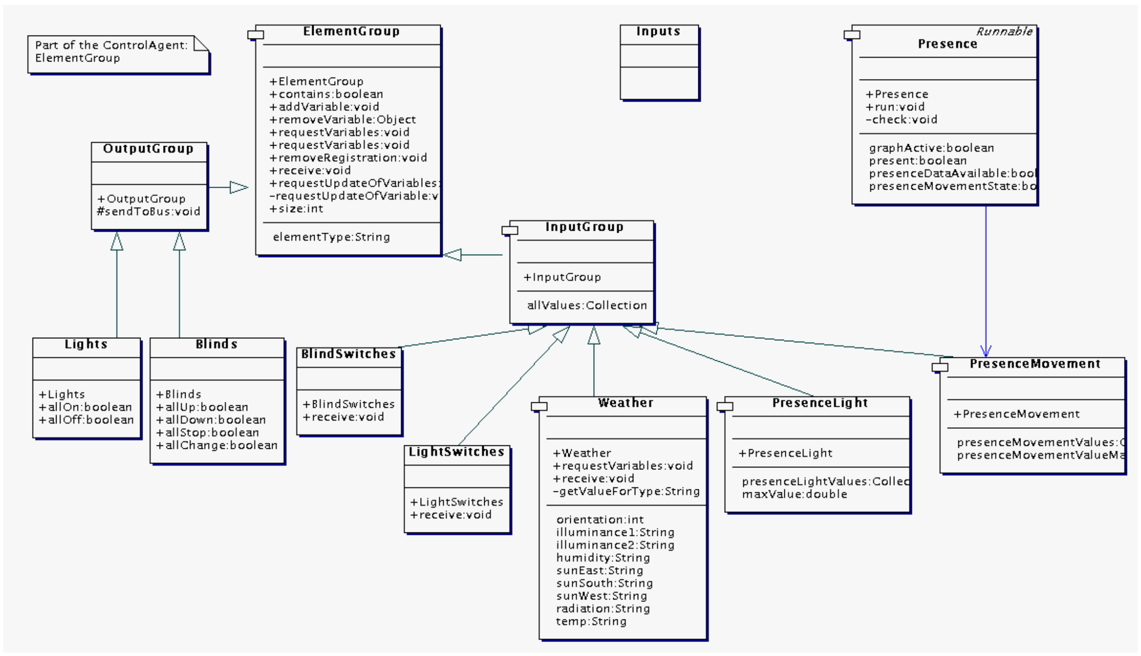


Figure 10.6: UML class diagram of the control agent (Elements, part 2)

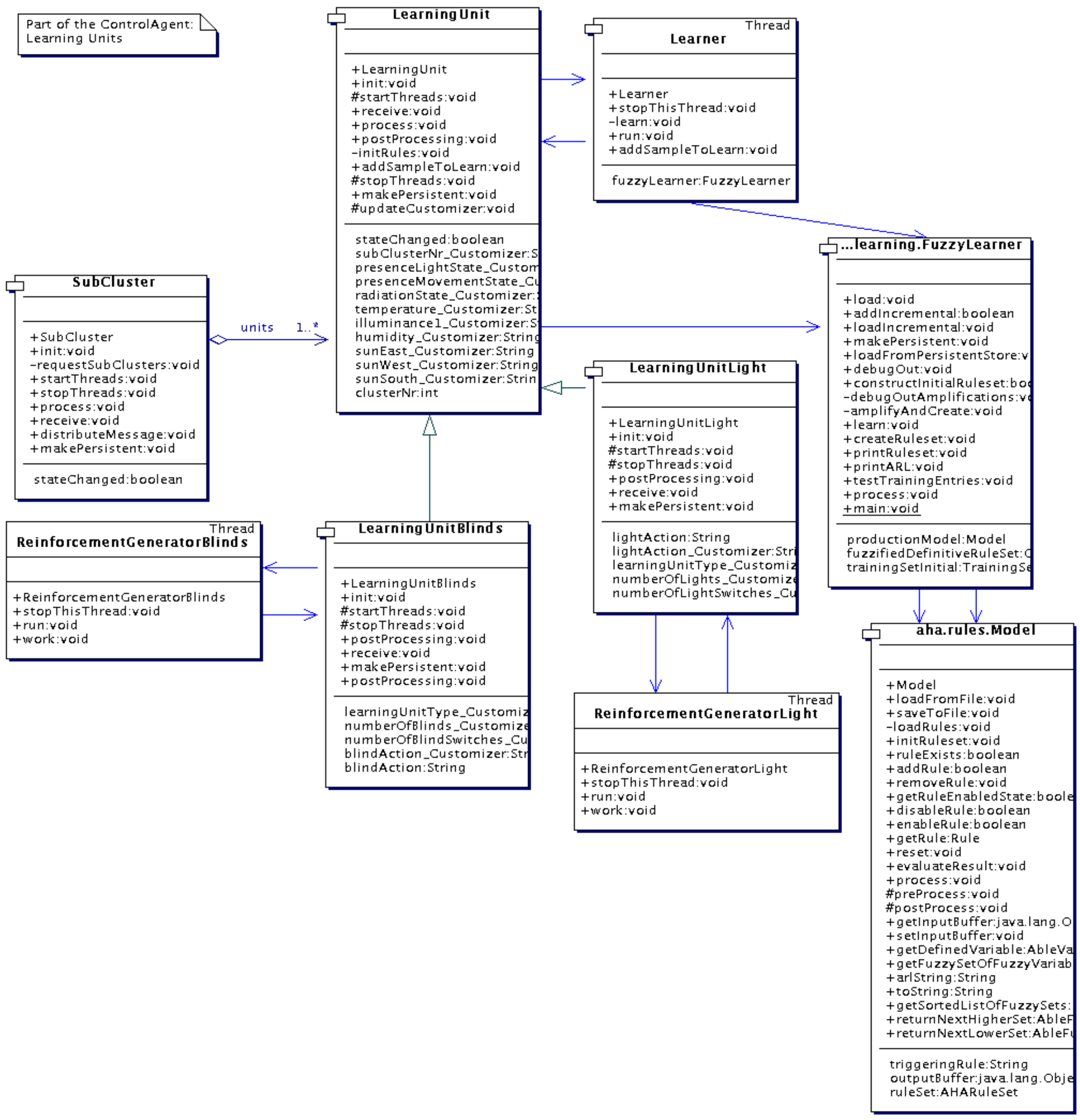


Figure 10.7: UML class diagram of the control agent (Learning, part 3)

10.6 Distribution Agent

10.6.1 Functionality

The distribution agent is responsible for distributing arbitrary messages to other agents. It distributes this messages according to the declared interests of all other agents present in the MAS. All agents that are interested in some kind of message declare this interest towards the distribution agent which stores this interests in a internal data structure. An agent that wants to distribute a message according to an interest just sends this message together with the type of interest to the distribution agent and the distribution agent than takes over the job of actually distributing the message.

Message distribution in a multi-agent environment is complex because of a number of reasons:

1. *Interests*: Every agent is interested in different kinds of messages. Interests of agents constantly change.
2. *Communication*: Because agents could possibly be distributed physically, some agents my become unreachable because of communication failures. Communication between agents could be prone to failure or be slow.
3. *no unregister*:When an agent gets stopped for some reason it usually doesn't care about interests it still has registered with other agents. Such invalid subscriptions for non-existing agents need to be removed automatically.

The distribution agent uses a large pool of threads to distribute many messages in parallel. Every message for every destination is assigned to it's own thread so that even the same message is sent out in parallel.

The subscription list (which agent is interested in which messages) is constantly updated by the distribution agent. In case an agent with an active subscription can't be reached for whatever reason the distribution agent automatically unsubscribes it.

Every type of message possible in the system is identified by a system wide unique identifier. Interests are managed in terms of this identifiers (which agent is interested in which identifier). Every message in the system is marked with this identifier so every agent nows at every time what type of message it just received/sent.

There are different types of type of messages (interests). The type of message is included in the message type name with a prefix that precedes every message type name.

The special prefix *var* denotes a message that is being generated externally by the sensor network. Every time the first agent declares an interest for such a variable towards the distribution agent, the distribution agent notifies the bus agent. The bus agent in turn subscribes to this message with the sensor network and forwards incoming updates directly to the distribution agent for later distribution to all interested agents.

Whenever the last agent de-subscribes to a type of message the distribution agent notifies the bus agent, so that the bus agent can de-subscribe the interest with the sensor network as well.

A type of message is called a *Topic* in the following definition of messages.

The distribution agent is a service agent. In a JAS platform there is usually just one instance of the distribution agent running.

10.6.2 Messages

The distribution agent sends the following types of messages:

- TOPIC_CREATE: first subscription for a topic, at least one agent is from now on interested in this topic.

- TOPIC_REMOVE: last subscription for a topic was removed, no one is interested in this topic anymore.
- arbitrary types of messages: Messages that the distribution agent distributes as substitute for other agents.

The distribution agent accepts this types of messages:

- DISTRIBUTE_MESSAGE: tells the distribution agent to distribute the encapsulated message according to it's interests list.

10.6.3 Implementation

Figure 10.8 shows the most important classes and their relationships.

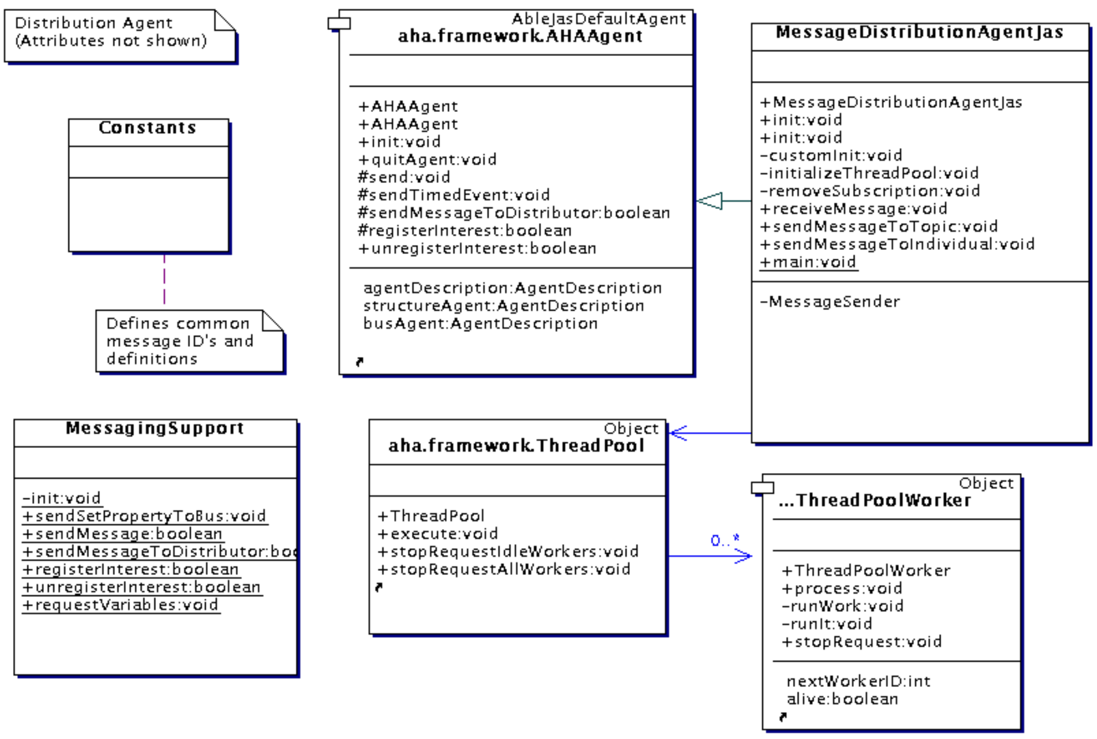


Figure 10.8: UML class diagram of the distribution agent

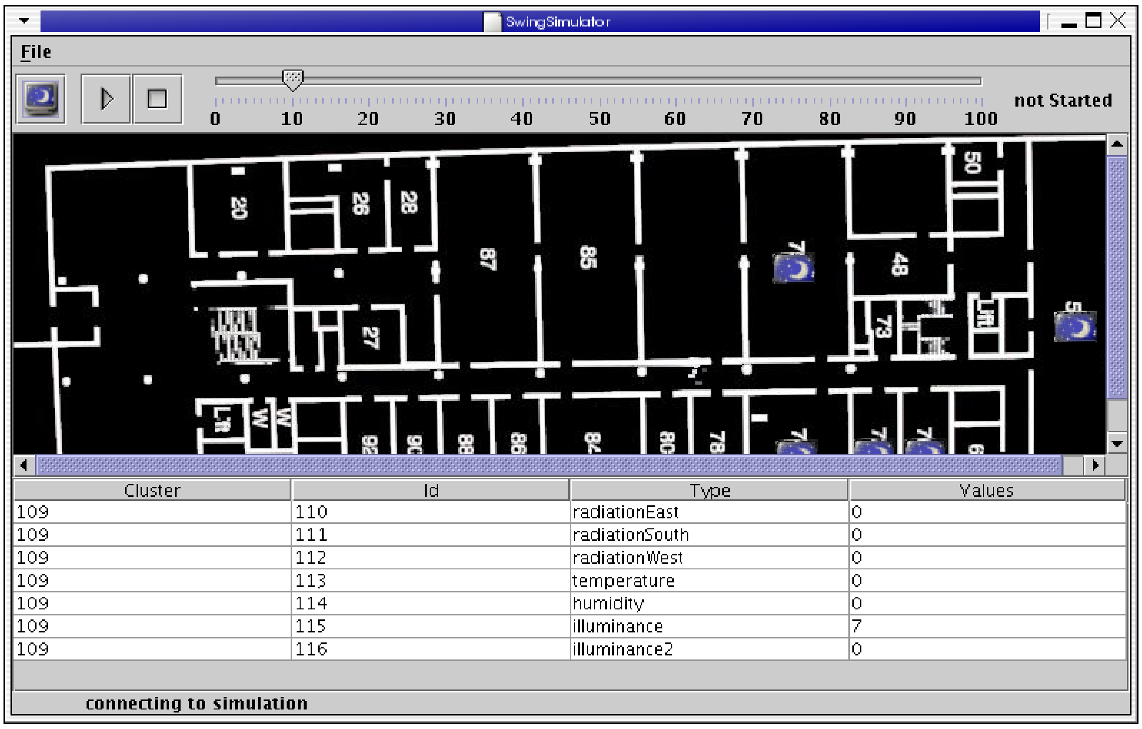
The implementation (which is in package *aha.messaging*) is structured into classes as following:

- *ThreadPool*: Threadpool that manages the asynchronous message delivery.
- *ThreadPoolWorker*: A single, runnable, class which task it is to distribute a single message to a single destination. After doing this job it stops.
- *MessageDistributionAgent*: The message distribution agent that manages the threadpool and the interests mapping list.
- *MessagingSupport*: A support class that offers various static methods for supporting message delivery for other agents.
- *Constants*: static constants for all system wide unique message id's and topic names.

10.7 BuildingViewer Agent

10.7.1 Functionality

This Agent visualizes what is happening in a building. But it allows one also to control a simulation if one is running. It is meant to let one to interact with our system in an visually more appealing way than just a command line interface.



Cluster	Id	Type	Values
109	110	radiationEast	0
109	111	radiationSouth	0
109	112	radiationWest	0
109	113	temperature	0
109	114	humidity	0
109	115	illuminance	7
109	116	illuminance2	0

connecting to simulation

Figure 10.9: Simulation of BuildingViewer Agent

The building viewer UI mainly consits of the following elements : toolbar, floor view, weather data and the statusbar.

Toolbar

Except of the first one, all controls in the toolbar allow us to control the simulation. These controlls are disabled when there is no use for them.

- moon : closes the building viewer
- play : starts / pauses the simulation.
- stop : stops the simulation
- slider : controlls the speed of simulation time
- time : shows the actual time in the simulation

Floorview

The Floorview gives an overview of the actual state of a whole floor/building. It uses 3 different icons to illustrate the state of a room. Light bulb, Moon and gnome. A light bulb means one or more lights in that room are switched on. A Moon means the blinds are down and a gnome shows that at least one person is present in the room. A double click on a room in the Floorview opens a window displaying all variables of that room.

WeatherData

Below the Floorview, BuildingViewer shows a list of variables and their values. These variables are external weather data delivered from the Grellow device.

10.7.2 Messages

Since BuildingViewer is an important interface to the whole abi system, one would expect it to listen and send many different messages. Actually it communicates only with 4 other agents. These are :

- SimulatorAgent for the simulation control
- StructureAgent to find the blinds, lights, weatherData etc.
- BusSim to be notified when to observed variables change
- PersonLocatorAgent to be notified when person enter/leave

The received messages are :

- GET_ELEMENTS_RESULT
- VARIABLE_UPDATE
- PERSON_ENTERED
- PERSON_LEFT
- TIME

And the messages sent are :

- SIMULATION_RUN
- SIMULATION_STOP
- SIMULATION_SET_DELAY
- SET_PROPERTY
- GET_PROPERTY
- GET_ELEMENTS

10.7.3 Implementation

BuildingViewer is implemented using the Java Swing toolkit. It reads an image displaying the floor, resp. all the floors of a building. It reads a configuration file which specifies the coordinates of the different rooms in the image. The image is displayed using the component FloorView which inherits from JPanel. Since FloorView shows the status of the lights and blinds in every room, BuildingViewer needs to subscribe for variable changes of all lights and blinds in the building.

To display the variables in a room, the component RoomView, which is also a JPanel, is used. RoomView contains a JTable to display the variables of a room. It uses VariableTableModel as the DataModel. While a RoomView is displayed BuildingViewer is subscribed to VARIABLE_UPDATES for all variables in that room and unsubscribes as soon as this RoomView is closed.

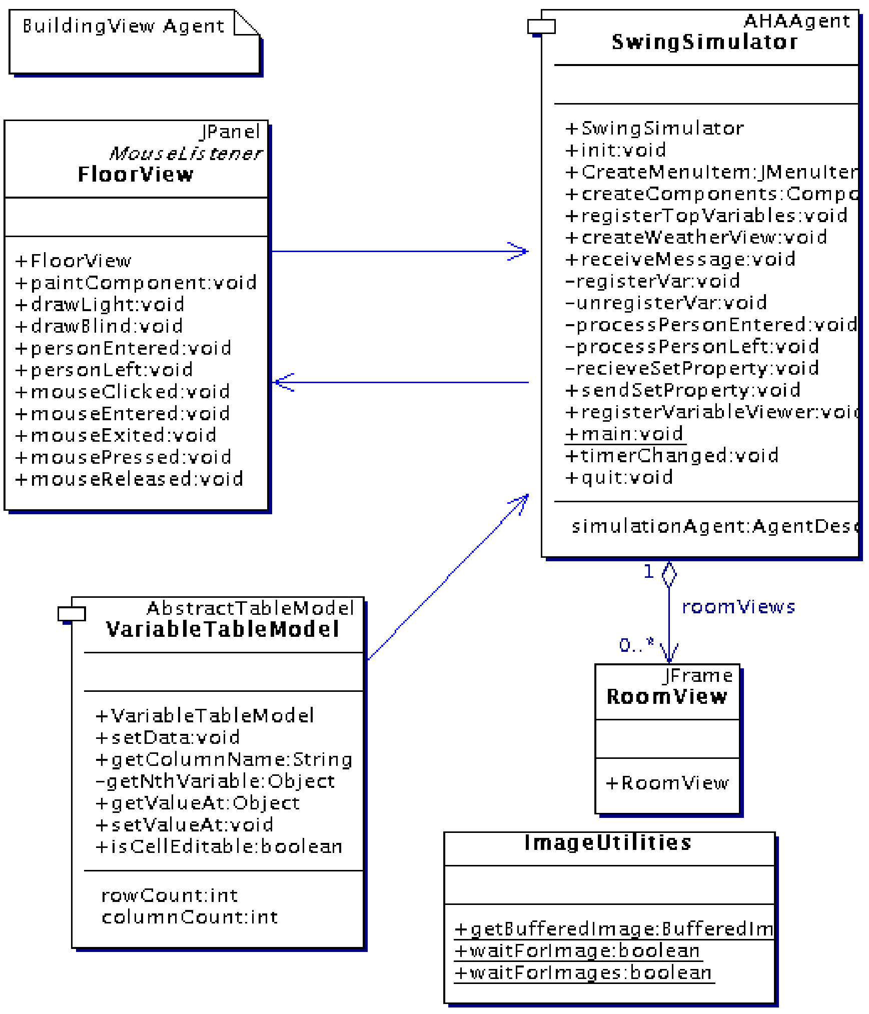


Figure 10.10: UML class diagram of the BuildingView agent

The implementation (which is in package *aha.sim*) is structured into classes as following:

- *SwingSimulator*: The BuildingViewer, class name is a historical leftover
- *FloorView*: JPanel based component to display the building
- *RoomView*: Component to display a rooms variables
- *VariableTableModel*: Data model to hold a rooms variable data

10.8 Person Locator Agent

10.8.1 Functionality

This agents informs about person locations in the buildings different rooms. It is the implementation used for the Bluetooth person localization further described in chapter 11.1

10.8.2 Messages

This agent does only listen to GET_ELEMENTS_RESULT. It uses these messages to respond to GET_ELEMENTS to discover the top level clusters for the rooms.

PersonLocator sends the messages PERSON_ENTERED and PERSON_LEFT to inform other agents when persons have left or entered a certain room resp. cluster.

10.8.3 Implementation

Since this is a pretty simple agent it consist only of one class for the agent itself. The HttpServlet WebLocator is described in chapter 11.1.2. The agent reads two property files device2person and ip2room, which map device ids to persons and the Bluetooth access points IP to the room. The room is a string which is used to look up the corresponding cluster Id, which is discovered by GET_ELEMENTS and GET_ELEMENTS_RESULT, issued when the agent is initialized.

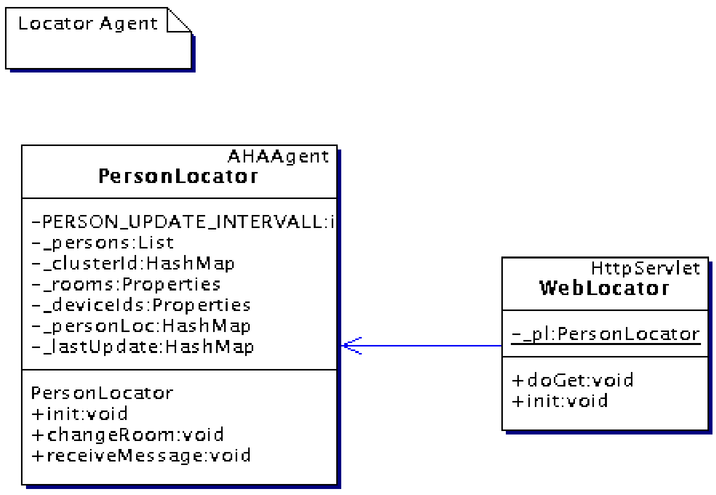


Figure 10.11: UML class diagram of the locator agent

10.9 Virtual Person Agent

10.9.1 Functionality

In a simulated building environment we are not able to use PersonLocater since there are no persons with their Bluetooth enabled PDAs walking around a simulated building. This agent is responsible to simulate virtual persons.

10.9.2 Messages

This agent sends the messages PERSON_ENTERED and PERSON_LEFT to inform other agents when persons have left or entered a certain room resp. cluster. It also simulates their behaviour, like switching on the light when it is too dark or pulling up the blinds.

10.9.3 Implementation

Virtual Person Agent reads a configuration file which describes the persons it should simulate. The Persons are simulated following a probablistic approach. This file uses the grammar described in listing C.2 an example can be seen in listing C.3.

An instance of Person is created for every person found in the config file. This class is regularly asked to change the room. It uses the roullet wheel selection algorithm to determine which room to visit. Every person also has a start and end time, these are the times when they start or stop working, thus they leave the building.

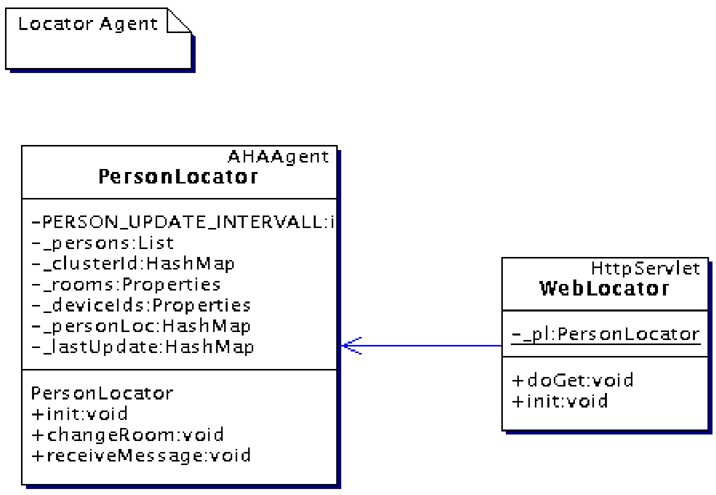


Figure 10.12: UML class diagram of the locator agent

Chapter 11

Personalization

11.1 Person Tracking – Bluetooth

Based on the fact that devices enabled for Bluetooth are coming to mass market more and more, using Bluetooth seems to be a reasonable approach. According to many different sources [ZDN], [INS], citeBBC a big majority of working people in western europe posseses a mobile phone. More and more these mobile phones are equipped with Bluetooth chips. It is thus reasonable to that we'll be able to detect presence of persons in a room over their personal Bluetooth device (Mobile Phone, PDA, etc.).

When trying to locate people over Bluetooth several problems arise. Some of these are:

- no roaming between base stations. see [blua] and [blub]
- overlapping Bluetooth clouds
- increased energy consumption when using Bluetooth extensively

We didn't deal with this problems. We are of the opinion that they will be solved in the future be solved in the future by developers of Bluetooth hardware. Instead we want to implement a prototype using a Palm handheld and several Bluetooth LAN access points. See figure 11.1.

The Prototype consist of two parts :

- Palm Bluetooth browser and web client
- Java Servlet and PersonLocator Agent

The Program on the mobile device discovers all reachable Bluetooth LAN Access Points, chooses one, connects to it and sends a HTTP GET to the Servlet. The Servlet passes the information on to the PersonLocator Agent, who informs all other Agents using the MessageDistribution mechanism described in 10.1.

11.1.1 Mobile Device Program

The client on the mobile device is a simple application which needs some user interaction. A more sophisticated application could be developed which would work with any user interaction.

After starting the application an panel with a button is presented to the user. See figure 11.2. With a click on this button the Bluetooth device discovery is started. The Palm device searches for all LAN Access Points

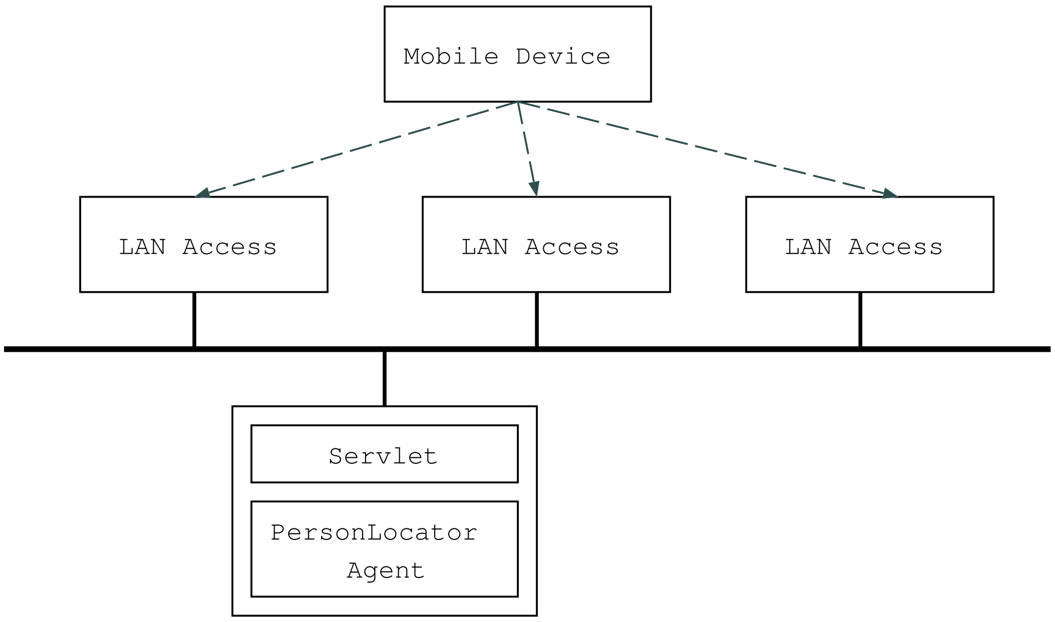
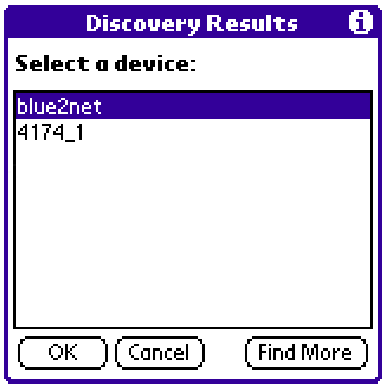


Figure 11.1: Architecture of our Bluetooth prototype



Select a device:

blue2net
4174_1

Figure 11.2: Screenshot of main window

in its neighborhood. A list of all devices found is presented to the user. See figure 11.3. After choosing one of the devices a network connection is built up and a HTTP GET request to the person localization servlet is issued. Part of this request is the Palm's RoomId which uniqueliy identifies the Palm device.

After this the application should be closed and the Network connection disconnected.

Please see the source code in CVS (src/palm/Src/INetLow.c) on how this prototype is realized. It is written in C and uses several Palm Libraries. The Bluetooth library is used for the device discovery. To issue the HTTP request we use Palms INetLib. This library does also the automatic connection to the Access Point.

11.1.2 Server side

The server side consists of a servlet running in a Tomcat 4.1 application server (but there are not limitations concerning an other application server). This servlet instanciates the PersonLocator and passes roomId, personId and RemoteIP to it whenever it recieves a request.

The remoteIP is a simple way to know which room a person is presently visiting. Since we have an access point in every room and these access points are doing NAT, we can use the access points IP as a hint to

Figure 11.3: Screenshot of access point choice

from which room this request is comming. The client thus only has to pass the deviceId.

Chapter 12

System Configuration

Over time the configuration of ABI got more and more complex. Here we shall give an overview on the environement variables and the config files with appropriate pointers to chapters with a more detailed description.

12.1 Envrionment Variables

The follwing two Environment variables must all be set for the correct startup of any ABI process :

- AHA_HOME path to the aha home directory
- ABLE_HOME path to the able home directory

12.2 Config Files

ABI relies on the presence of one main configuration file, .ahaConfig. This file is read by the global configuration class aha.Config. The files entries are mostly pointers to the location of the agents configuration files.

Listing 12.1: Extract a config file for a simulation

```
#Global JAS configuration
JASKernelServerIP=squatter.alani.ch
JASKernelServerPort=55551

#Bus Agent
BusController=aha.bus.lonworks.LNSController
busVariableMapping=config/INITestLonworks.xml

#LNS Controller
lonBusIP=130.60.71.231
lonBusPort=2540

#Adaptive Control Agent
ruleDir=/src/aha/rules/ruleFiles
learningStateStorage=/state

#BusSim Agent
simConfig=config/INITestSimulation.xml
```

```
#Structure Agent
structureConfig=config/INITestStructure.xml

#Virtual Person Locator
personConfig=TwoPersons.xml

#Bluetooth Person locator
device2person=/config/device2person
ip2room=/config/ip2room
```

variable	Description	Details
busVariableMapping	Mapping between real variable names and artificial ids	Chapter 10.2
learningStateStorage	Directory where ControllAgent stores it's memory (learning state)	Chapter 10.5
simConfig	Configuration of the field bus simulation	Chapter 10.4.1
structureConfig	Structure information about the building	Chapter 10.3
personConfig	Virtual Persons simulated in the building	Chapter 10.9
device2person	Mapping from Palm device id to person id	Chapter 10.8
busVariableMapping	Mapping between real variable names and artificial ids	Chapter 10.2

Part IV

Results and future work

Chapter 13

Results

The evaluation of the project is distributed among two different chapters. This chapter evaluates what has been achieved during the present project so far whereas the next chapter (chapter 14) evaluates the project in a broader context. It also gives an idea of where research in intelligent buildings may possibly head to in the future.

13.1 Evaluation

ABI proves that it is possible to make a building intelligent. We deployed ABI in an a commercial building which is used daily. The system is capable of adapting itself continuously to the users of the building. ABI not only proves that it is possible to develop and deploy such a system but it also shows that such a system isn't computationally expensive. Actual computation required to control a building is minimal. This is due to the highly distributed architecture that emphasizes localized decision making and control.

13.2 Architecture

All software developed was focused on the architectural decision to regard the system as a multi-agent system. This proved to be feasible. The multi-agent system enabled us to develop many agents. Each of these agents is very specialized and thus small and focused. This greatly sped up development speed and efficiency and it also increases the systems stability. The system can even fulfill it's task (as least partially) if some of the agents aren't working anymore because of a system failure or communication problems.

Although over the whole development phase the application domain stayed the same we decided not to assume any structure or relationships between sensors and effectors during development. We wanted to abstract all these data such that no agent needs to make any assumptions about it's environment. Our solutions to tackle this requirement is the structure agent. The structure agent is the only agent that knows about the structure of the specific environment. This includes all sensor/effector relationships as well as the relationship between static structures (rooms, floors) and dynamic structures (people, sensors, effectors). This architecture proved to be a very good starting point. All agents request their structure information from this central structure information point. If the system needs to be deployed in some new environment or the existing environment changes (for example some new sensors get added) there is only one place where this new information needs to be incorporated (the structure agent). As of today the structure agent uses static configuration files (XML files) to get it's information. In the future this agent could be enhanced such that it discovers it's structure information and all sensor/effector relationship itself with some self organization algorithm.

An other very successful architectural decision was the decision to abstract the actual fieldbus (sensor network) from the rest of the system. There is no agent that knows anything about the fieldbus except for the bus agent. The bus agent converts all messages from the fieldbus into ordinary JAS messages and forwards them to the distribution agent for distribution to all the other agents. Writing values works the other way round: An agent that wants to send some value to an effector just sends a variable update to the bus agent and the bus agent converts this request to the required special format of the fieldbus. This architecture also allows it to use the same system for different kinds of fieldbus types (like lonworks or EIB). The only agent that needs to be changed in this case is the busagent.

13.2.1 Messaging

In contrast to the former projects [RS02] and [EBB+02] all communication between agents in ABI is done with messages. There are no synchronous method calls between agents whatsoever. This has advantages and disadvantages. The major advantage is that it makes the agents completely decoupled. There is no dependency between the internal structure of the agents whatsoever. The only common thing between agents is the definition of the messages (identification and structure of payload). An other advantage is that a system that is only coupled by (asynchronous) messaging is much more stable and resistant to failure. Agents can easily be re-started, shutdown or restarted during runtime without influencing other agents. Partial communication problems between agents only affect the agents affected by the problem and not the whole system.

The major disadvantage is that development of such agents is very different from traditional software development and thus more complex. In such agents there is no consistent execution flow as in agents that make synchronous system calls to each other. The program flow is consistently interrupted (send out message) and continued at an other point in the program where incoming messages are processed (receive answer to message). All kinds of difficult concurrency issues arise due to the highly parallel and multi-threaded nature of such agents.

Despite the drawbacks the decision to use asynchronous messaging for interagent communication proved to be a very good one. Once one gets accustomed to the different way of developing such a system it actually becomes second nature and one can develop just as one would develop a agent that uses synchronous method calls. The many advantages of asynchronous messaging clearly outweight the disadvantages. Such a system is completely decoupled and very resistant to failure and changes during runtime.

An additional property of our system is the interest based messaging. To tackle the problem that every agent in such a system is just interested in receiving a small subset of all available type of messages we introduced an additional agent that takes over the responsibility to distribute messages according to interests. Every agent that wants to receive messages declares it's interests towards this agent and every agent that wants to send out a message delegates this task to the distribution agent. This architecture minimizes the communication and processing overhead for agents considerably.

13.2.2 Simulation

The simulation was the first task we started to develop. We thought it was a requirement to develop and test the adaptive control agents. First we also thought that it will be a quick job to develop this simulation. And surprisingly it was one! A first prototype was quickly developed. The requirements for the simulation were changing all the time. And because our time frame was very limited we sometimes didn't do enough refactoring, thus the software design of the simulation might look a bit awkward.

A few behaviors of the field bus were really difficult to simulate. Especially the daylight sensors behavior turned out to be difficult to approximate, because it does not only depend on state of the blind variables in the same room, but it is also a non linear function of the exterior illumination.

We think we have achieved a working and close enough approximation of a real building. And it turned out to be really useful for developing the adaptive agents. Many problems we would have faced later when developing the adaptive agents turned up during development of the simulation and the UI and thus were not a problem anymore for the later development. It allowed us thus to concentrate more on the adaptive part of the agents.

13.2.3 JAS

We decided to build our multi-agent system on top of a JAS ([FLoA02]) compliant platform (ABLE, see [ABL]). The standard services offered by a JAS platform (agent directory, agent naming, message transport system) proved to be very useful for the development of a multi-agent system. It allows to develop such a system on a much more abstract level because one doesn't have to deal with problems like how to discover other agents or how to send a message to an other agent. The JAS specification, however, doesn't include any standard for lifecycle management. This is due to reasons explained in chapter 5. Some kind of lifecycle management service is absolutely necessary to build a multi-agent system. We used the proprietary implementation of ABLE ([ABL]) for our system.

What the JAS specification, and thus the JAS compliant implementation of ABLE as well, doesn't provide is asynchronous messaging. This needs to be implemented by the multi-agent system builder itself. We implemented this by using a large threadpool inside of the distribution agent.

JAS, and the JAS compliant implementation ABLE, was very helpful to develop ABI. It freed us of the need to develop standard services like the lifecycle service or an agent directory service.

13.3 Learning and inferencing

The learning algorithm described uses very simple principles of fuzzy logic inferencing and set theory to automatically learn a maximal structure fuzzy rule base. Although the algorithm is built on the basis of very simple principles it's performance is excellent which once more proves the powerfulness of simple algorithms.

The algorithm is specifically designed to run in an continuously changing environment. It is an anytime learning algorithm that is a capable of changing it's believes according to the changing environments. Believes held to be true don't necessarily have to be true in the future. The algorithm is capable of detecting such changes and eliminates conflicts within the rulebase that arise from such changes.

Learning in terms of learning units allows it to learn a great number of very limited and small rulebases. The parallel learning of many such small rulebases ensures minimal computational complexity and fast convergence at the same time. The design decision to split up the whole state space into many small units to emphasize localized decision making and control proved to be justified; It made the system must leaner in terms of required computation.

The emphasize on non-explicit user interaction proved to be very successful. Users don't wish to interact with a building explicitly. They just want the building to adapt to the preferences of the users as expressed with their actions. Our learning algorithm takes the input values available from the sensors and automatically generates punishment/reward reinforcement signals for the learning algorithm.

Fuzzy logic and fuzzy inferencing as the basic mechanisms for decision making allows it to deal with imprecise input data and noise. In our specific case the use of fuzzy input variables also allowed it to greatly reduce the dimensionality of the state space of the system. Instead of a real valued input variable there is just a fuzzy variable that can be set to a finite number (and typically very small) number of fuzzy states accompanied by a truth value.

Learning in terms of fuzzy sets and variables allows it to deal with a problem on a much higher level of abstraction than real valued input would allow it. Learned fuzzy logic rules formulate knowledge of the system on a very high level that also holds truth for imprecise inputs and noise in the system.

We believe that the use of fuzzy logic rules was one of the most important decisions taken. Learning in such a system without using fuzzy logic would have been much more complex if not impossible.

13.4 Personalization

The development of a prototype for tracking persons via Bluetooth did not work as smooth as we expected. At a first look Bluetooth looks rellay suitable for that task. After a closer look at how Bluetooth works and what can be done with the stack we were not sure about that any more.

The prototype we developed (see chapter 11) allows the tracking of persons via Bluetooth. But it has several limitations. First it requires manual iteraction from the user. A better solution should work fully automatic. Two steps now require manual interaction. The start of inquiry and the choice of which access point to connect to. The first one could easily be automated. A new inquiry could be done whenever the connection goes down, or every n minutes. It just needs to be considered that the Bluetooth inquiry process has a high power consumption. The choice of which access point to connect to could be decided depending on the quality of the signal. Although such a function to get the signal quality exists in the the Bluetooth standard (see [bluc]) we could not figure out how to do that with the Palm SDK.

Then we faced also some issues with the development language. Programming in C again after having been programming in Java was a bit difficult. And the documentation for the Palm SDK was sometimes really sparse (especially the documentation of the Bluetooth SDK part).

Chapter 14

Future

This chapter deals with possible future extensions of ABI and the future of research in building intelligence in general.

14.1 Learning and inferencing

The learning algorithm in use could be enhanced in various ways. We think that the current algorithm could be a good basis to future expand it's capabilities. Features that could be introduced or improved are:

- Policy transfer between learning units
- improved algorithm by using more complex amplification method (powersets and not linear expansion of sets)
- learn individual preferences
- make learning probabilistic such that a ruleset only has to be consistent to a certain degree (not like now whereas there is always 100% consistency)
- count rewards/punishments and act only after a certain threshold of number of punishments/rewards. Don't react immediately (as it is now) to every reward/punishment.
- optimize implementation of the algorithm by using a fast java math library to replace all operations carried out on sets.

What would also be a very interesting and challenging task is to make the membership functions of the different fuzzy sets stochastic in itself. Adding a stochastic term to the definition of a fuzzy set would make the representation of fuzzy concepts like *cold* or *warm* more realistic as there borders would be really fuzzy (stochastic) and different every time. A good starting point would be to try gaussian noise.

14.2 Structure information

In the current system the structure agent is the single point where all information about the structure of the system originates. It retrieves this information form human-edited XML files. This agent could in a future release be replaced by a much more enhanced agent that uses self-organization algorithms to dynamically learn about the structure (and especially the sensor/effector relationships). This would be particularly interesting in very complex building environments that change frequently like flexible office space that is already in use in modern commercial buildings. Learning the structure rather then statically define it would add an other layer of learning to the system which would make it even more flexible and adaptive to change.

14.3 Personalization

Exploring the full potential of actually personalizing all learning and behavior inside an intelligent building is the next big step forward. As we showed with experiments with our prototypes (chapter 11.1), it is theoretically and practically possible to track and identify people with the help of Bluetooth. To make this really useable in a real-world environment specialized Bluetooth hardware needs to be developed. This special *bluetooth presence detectors* would have to be directly integrated into the fieldbus network where all other sensors of the building are connected to. One example would be a *bluetooth presence detector* device that can directly be connected to a lonworks fieldbus network. Such a *bluetooth presence detector* would have to deliver very highlevel information to it's users such that all complicated work like link state quality evaluation of Bluetooth links would be done directly in hardware.

As soon as such a technology is available all learning and control could be tight to persons. This would greatly enhance the perceived intelligence of a building.

Part V

Appendix, Glossary and Bibliography

Appendix A

Fuzzy Logic Primer

A.1 Fuzzy Logic primer

Proofs for the following definitions are in [Ful00].

A.1.1 Definitions

We always assume that X denotes a non-fuzzy set and A denotes the corresponding fuzzy set.

The Membership Function (A.1) is a function that assigns every $t \in X$ a value $0 \leq x \leq 1$.

$$\mu_A : X \rightarrow [0,1] \tag{A.1}$$

Equation A.2 shows a frequently used short form for the membership function.

$$A(x) = \mu_A(x) \tag{A.2}$$

The support (A.3) of a fuzzy set A is the set of all $x \in X$ such that $A(x) > 0$.

$$supp(A) = \{x \in X | A(x) > 0\} \tag{A.3}$$

The α cut (Equation A.4) of a Fuzzy set A defines a subset of X (non fuzzy).

$$[A]^{\{\alpha\}} = \begin{cases} \overline{\{t \in X | A(t) \geq \alpha\}} & \text{if } \alpha > 0 \\ supp(A) & \text{if } \alpha = 0 \end{cases} \tag{A.4}$$

A.1.2 Membership functions

The following equations assume that $\alpha > 0$ is the left width and $\beta > 0$ is the right width with $\alpha \leq \beta$.

Triangular fuzzy number (Equation A.5 and Figure A.1). Shortform: $A = (a, \alpha, \beta)$.

$$A(t) = \begin{cases} 1 - (a-t)/\alpha & \text{if } a - \alpha \leq t \leq a \\ 1 - (t-a)/\beta & \text{if } a \leq t \leq a + \beta \\ 0 & \text{otherwise} \end{cases} \tag{A.5}$$

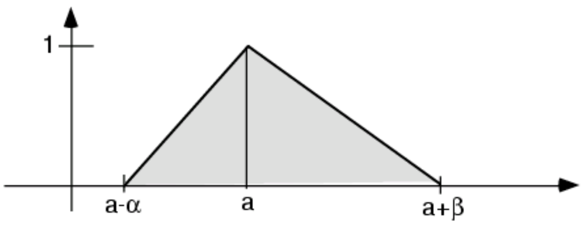


Figure A.1: Triangular membership function

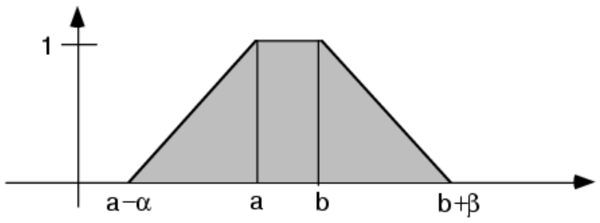


Figure A.2: Trapezoidal membership function

Trapezoidal fuzzy number (Equation A.6 and Figure A.2). Shortform: $A = (a, b, \alpha, \beta)$

$$A(t) = \begin{cases} 1 - (a-t)/\alpha & \text{if } a - \alpha \leq t \leq a \\ 1 & \text{if } a \leq t \leq b \\ 1 - (t-b)/\beta & \text{if } a \leq t \leq b + \beta \\ 0 & \text{otherwise} \end{cases} \tag{A.6}$$

LR-representation (Equation A.7) of fuzzy numbers, which is a generalized form of the trapezoidal representation. Shortform: $A = (a, b, \alpha, \beta)_{LR}$

$$A(t) = \begin{cases} L((a-t)/\alpha) & \text{if } t \in [a - \alpha, a] \\ 1 & \text{if } t \in [a, b] \\ R((t-b)/\beta) & \text{if } t \in [b, b + \beta] \\ 0 & \text{otherwise} \end{cases} \tag{A.7}$$

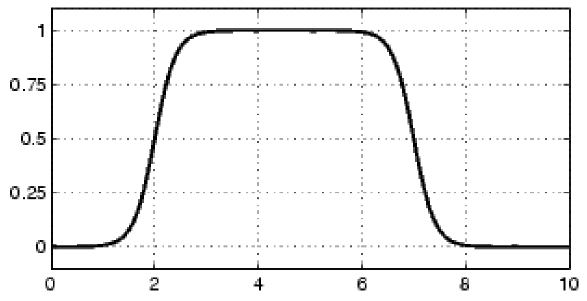


Figure A.3: Sigmoidal membership function

Sigmoidal fuzzy number/variable (Equation A.8 and Figure A.3).

$$f(x) = \frac{1}{1 + e^{-x}} \tag{A.8}$$

Gaussian membership function (Gaussian fuzzy number). In equation A.9 ρ is the standard deviation and μ is the mean of the distribution (Figure A.4).

$$A(x) = \frac{1}{\rho\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\rho})^2} \tag{A.9}$$

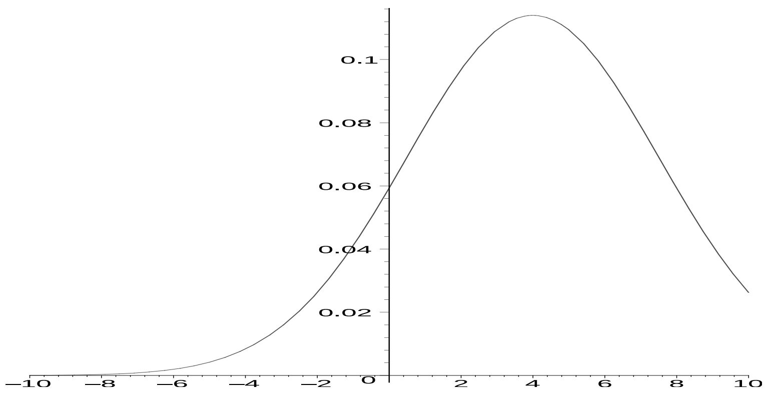


Figure A.4: Gauss distribution as a membership function

Other membership functions that aren't explained here in detail are shoulder, linear, segments and beta function. Consult [Ful00] for details about these functions.

A.1.3 Fuzzy inferencing

The basic inferencing process involves the following steps:

1. fuzzification of the input variables
2. rule evaluation
3. aggregation of the rule outputs
4. defuzzification

This specific inferencing process is called Mamdani-style fuzzy inferencing process.

A.1.4 Fuzzy logic controllers

Control theory looks at complex systems from a black box perspective. The system to be controlled is idealized as a simple black box with an input and an output. A controller is the part of the system (Figure A.5) which is responsible to adjust the input to the system according to the output of it. This is made in an attempt to achieve the desired output.

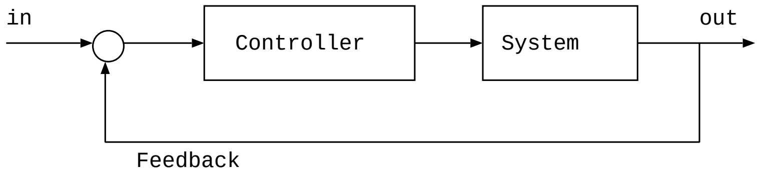


Figure A.5: Basic layout of a non-fuzzy controller

Fuzzy logic control is built on the same principles but uses a different technique to decide what the output of the controller should be. All decisions of the controller are made on the basis of fuzzy logic rules which describe the relationship between fuzzy variables. A fuzzy logic controller (FLC) can only work with fuzzy input and output values. Because of this the input values must be converted from crisp numbers to fuzzy variables and the output values must be converted from fuzzy variables to crisp numbers. This two processes are called fuzzification and defuzzification. An enhanced version of the original control system is shown in Figure A.6.

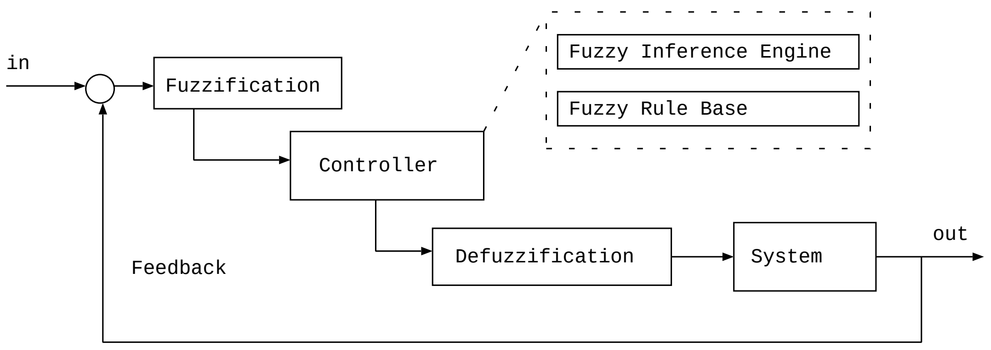


Figure A.6: Basic layout of a fuzzy controller

A FLC consists of two different parts: the inferencing engine and the fuzzy rule base. The fuzzy rule base is a number of fuzzy logic rules which are used by the inferencing engine to reach a conclusion from the input presented.

FLC's are classified as either multi-input-multi-output (MIMO), multi-input-single-output (MISO) or single-input-single-output (SISO) fuzzy systems (see [Ful00], page 67).

A.1.5 Fuzzification process

Fuzzification is the process that uses the membership functions to convert a crisp value to a fuzzy variable. A fuzzy variable associates a membership value/strength for every fuzzy variable to a crisp value: $\mu_A : X \to [0,1]$.

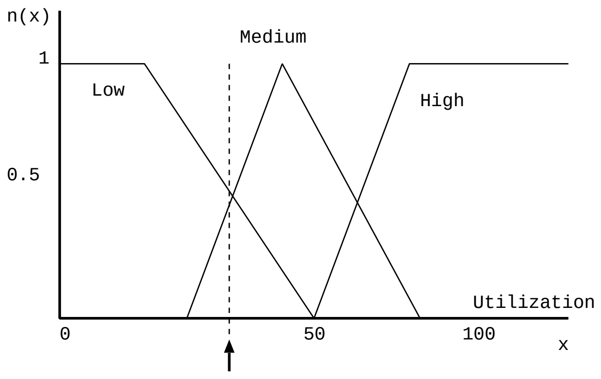


Figure A.7: Fuzzification Process

Figure A.7 shows an example of the fuzzification process for three fuzzy variables. If, for example, the crisp input value is 37.5 the values of the three fuzzy variables would be Low = 0.5, Medium = 0.5 and High = 0.0.

A.1.6 Defuzzification process

The end result, after all rules have been evaluated and aggregated (combined to one fuzzy set), is a single fuzzy set that represents the decision taken. To apply the output of the fuzzy inferencing a crisp value is required. Defuzzification is the process that is converting a single fuzzy set back into a crisp number. Defuzzification is deferred till the last possible point of time. This is because defuzzification is not for free. All the information that is contained in a fuzzy set is reduced to one crisp number which contains obviously less information then the fuzzy set contains. This is a considerable loss of information.

There are many defuzzification algorithms. Some of the most important ones are:

- Center of Area / Center of Gravity
- Center of Sums, Center of Largest Area
- First of Maxima
- Middle of Maxima
- Max Criterion
- Height defuzzification

Only center of gravity is explained in more detail here since this is the only method used in AHA. For details about the other methods see [Ful00].

$$z_0 = \frac{\int \mu(x)dx}{\int xdx} \tag{A.10}$$

Center of gravity (Equation A.10) calculates the center of mass if the area of the fuzzy set is regarded as a physical area. Calculation of the center of gravity is usually not possible in continuous space. Thus, Equation A.11 is used in practice to calculate the center of gravity of a fuzzy set that has a discrete membership function (which is usually the case).

$$z_0 = \frac{\sum x\mu(x)}{\sum \mu(x)} \tag{A.11}$$

z_0 is the crisp result of the fuzzy inferencing process (Figure A.8).

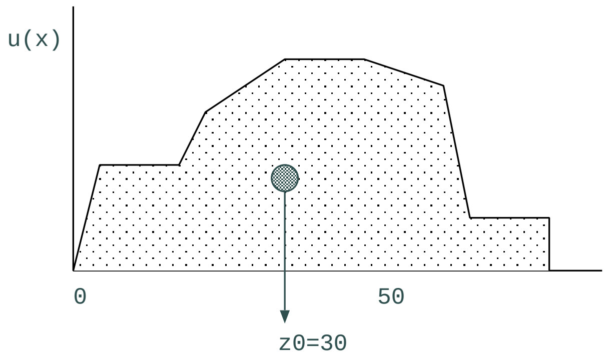


Figure A.8: Defuzzification with center-of-area/gravity

A.1.7 Rule based inferencing

The basis for decisions taken by a fuzzy logic controller are linguistic description rules. These rules are expert knowledge that needs to be set up by a human domain expert. The rules are of the form:

R_1 : if x is A_1 and y is B_1 then z is C_1
R_2 : if x is A_2 and y is B_2 then z is C_2
R_3 : if x is A_3 and y is B_3 then z is C_3
R_n : if x is A_n and y is B_n then z is C_n

R_n is the identification of the rule, x and y are input variables and z is the output (control) variable. A_i, B_i and C_i are linguistic variables. x, y and z are fuzzy variables.

To obtain the output z_i for a particular set of input values all rules are evaluated in parallel. This involves the following steps:

1. Find the firing level of each of the rules
2. Find the output of each of the rules
3. Aggregate the individual rule outputs to obtain the overall system output

Appendix B

Agent Building and Learning Environment

ABLE, short for Agent Building and Learning Environment ([ABL]), is an extensive Java framework and library for developing multi agent systems that utilize machine learning and reasoning. It is a research project of the IBM T.J. Watson Research Center and offered by IBM on alphaworks.

ABLE was used extensively during the development of ABI and the former project AHA ([RS02]) and it is thus necessary for the reader of this documentation to understand the basic principles of ABLE. In the following sections a short introduction is given to the parts of ABLE that are used within the ABI framework. Parts of ABLE which aren't used within the ABI project aren't mentioned here. For more information about ABLE consult [BSP$^+$02] or the ABLE documentation ([ABL] and [ARL]). A more advanced introduction to ABLE and it's functionality and principles is given in the book [PB01].

B.1 Agents

ABLE offers an extensive framework for developing multi-agent systems. It provides generic base classes for agents that can later on be used as JAS compliant agents. These classes provide all necessary features an agent developer needs. This includes lifecycle support, registration in the agent directory, intra-and interagent communication and message processing.

Every agent that uses ABLE and is JAS compliant implements the interface AbleJASAgent. AbleJAS-DefaultAgent is a default implementation of AbleJASAgent. All agents in ABI indirectly inherit from AbleJASDefaultAgent. AbleJASDefaultAgent already provides the facilities necessary to communicate with the JAS platform.

B.1.1 JAS

ABLE is fully JAS (Java Agent Specification, also called JSR-87) compliant as described in section 5.3. This includes an implementation of the Agent Directory Service, the Agent Naming Service and the Message Transport System. ABLE uses the official JAS reference implementation. In addition to the standardized JAS platform services ABLE offers lifecycle management facilities. This additional platform service allows it to delegate start,stop and reset agent operations to the JAS platform itself. This reduces the management required for a multi-agent system considerably because all the lifecycle operations are carried out by JAS regardless on which physical machine the agent actually resides.

B.2 ARL

ARL, short for ABLE Rule Language, is a Java like language for specifying rules and variables for inferencing. Within the context of ABI only the fuzzy parts of ARL are used which includes the specification of fuzzy membership functions and of if-then rules with fuzzy operators.

B.2.1 Structure of ARL

An ARL file represents a single ruleset consisting of one or more ruleblocks. The general structure is shown in B.1. Only the parts of ARL used in ABI are mentioned in the following section, see [ARL] for a complete documentation. All examples given in the following section are extracted from one of the rulesets used for this project.

```
ruleset <name> {

variables {
<Variable Declartions>+
}

inputs { <variableName>* }
outputs { <variableName>* }

void process() using <Inference Engine Type> {  //ruleblock 1
<rule>+
}

}
```

Figure B.1: ARL structure

Every input variable specified needs to be fuzzified (in case it is not already of a categorical type). The relationship between a fuzzy variable and a real valued variable is described by membership functions which are also part of an ARL definition. Membership functions are composed of a number of fuzzy sets of different forms (Linear, Triangle, Sigmoidal, Gaussian, etc). Every of these fuzzy sets is associated with a human-readable label (for example A_night1 in example B.2) with which it is later on referenced (from within the ruleblocks). A fuzzy variable is always based on a continuous range of real numbers, for example the range $[0.0, 86400..0]$ in figure B.2. Figure B.3 shows the membership function shown in figure B.2 as a graph.

```
    Fuzzy DayTime = new Fuzzy(0.0 , 86400.0)  {
      Linear    F_night2 = new Linear   (75600.0, 86400.0, ARL.Up);
      Linear    A_night1 = new Linear   (0.0, 18000.0, ARL.Down);
      Triangle   B_morning = new Triangle (14400.0, 25200.0, 36000.0);
      Triangle   E_evening = new Triangle (57600.0, 68400.0, 79200.0);
      Triangle   C_midday = new Triangle (28800.0, 43200.0, 57600.0);
      Triangle   D_afternoon = new Triangle (43200.0, 54000.0, 64800.0);
     };
```

Figure B.2: ARL definition of a fuzzy variable

The second variable type used is the categorical variable. A categorical variable consists of a finite set of valid values. All other values are invalid. Categorical variables don't need to be fuzzified. Figure B.4 shows an example of a definition of a categorical variable.

All input and output values of the inferencing process are real-valued (either a real number or a string in case of a categorical variable). Figure B.5 shows an example of the definition of the input and output variables.

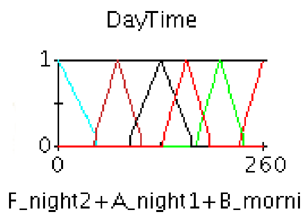


Figure B.3: Visualization of the membership function DayTime

```
Categorical TemperatureFuzzyState = {"A_veryCold", "B_cold", "C_warm", "D_veryWarm", "E_hot"};
```

Figure B.4: ARL definition of a categorical variable

The rules itself are structured within blocks. Every block of rules (called ruleblock) has a unique name and contains a number of rules. Every of this rules is identified by a unique label (rule label, rule name). Figure B.6 shows an example of two rules. The first rule (called Support_Presence2) is a rule that uses a conditional variable in the antecedent of the if clause and the second rule (called Support_DayLightIndoorFuzzyStateA_dark) uses a fuzzy variable in the antecedent of the clause. *DayLightIndoor* is the fuzzy variable and *A_dark* is a label that identifies a fuzzy set that is part of the definition of the fuzzy variable *DayLightIndoor*.

ABLE comes with a graphical editor for editing, verifying and visualizing ARL rule files. It automatically verifies ARL rulefiles which allows it to test ARL rulefiles before actually integrating them into an application.

B.3 Fuzzy Inferencing

ABLE offers a fuzzy inferencing engine which is capable of using a if-then rulebase and a set of definitions of fuzzy variables to do fuzzy inferencing including the preceding fuzzification and the ensuing defuzzification. The rulebase and all necessary definitions for the fuzzification and defuzzification process are specified with ARL (ABLE Rule language, [ARL]).This ARL definitions are used by the fuzzy inferencing engine together with input values to make an inference.

The fuzzy inferencing engine supports the operators *is* (for example "if A is X“), *and* ("if A is X and B is Y“) as well as various fuzzy hedges (for example *not*, *slightly*, *above*, *below*).

Unfortunately the fuzzy operator *or* is not supported by ABLE. *or* is absolutely necessary if one wants to formulate complex rules (especially rules that are automatically learned and processes). The workaround for this is to use a combination of fuzzy hedges. See section 9.5.1 for details.

```
inputs{Temperature, DayTime, Presence, DayLightIndoor,
RadiationEast, RadiationWest, RadiationSouth, Illumination1};

outputs{firedRules, notUsed, LightAction, BlindAction,
TemperatureFuzzyState, DayTimeFuzzyState, DayLightIndoorFuzzyState,
RadiationEastFuzzyState, notUsed, notUsed, Illumination1FuzzyState,
PresenceState};
```

Figure B.5: ARL definition of the input and output variables

```
Support_Presence2:
      if (Presence == 0.0)
      then  PresenceState = 0.0;

Support_DayLightIndoorFuzzyStateA_dark:
      if (DayLightIndoor is A_dark)
      then  DayLightIndoorFuzzyState = "A_dark";
```

Figure B.6: ARL rules

Appendix C

Configuration

C.1 BusSim Schema File

Listing C.1: Grammar for describing a simulated sensor network

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://castor.exolab.org/Test/Invoice">

   <xsd:element name="ahaSim">
      <xsd:complexType>
         <xsd:sequence>
                <xsd:element ref="device" maxOccurs="unbounded" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attributeGroup ref="AhaSimAttributes"/>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="device">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="variable" minOccurs="0" maxOccurs
                                    ="unbounded"/>
                </xsd:sequence>
                <xsd:attributeGroup ref="DeviceAttributes"/>
        </xsd:complexType>
        </xsd:element>

   <xsd:element name="variable">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="connection" minOccurs="0"
                                    maxOccurs="unbounded"/>
                </xsd:sequence>
                <xsd:attributeGroup ref="VariableAttributes"/>
        </xsd:complexType>
        </xsd:element>

   <xsd:element name="connection">
                <xsd:complexType>
                        <xsd:sequence>
                                <xsd:element ref="mapping" minOccurs="1" maxOccurs="
                                    unbounded"/>
                        </xsd:sequence>
                <xsd:attributeGroup ref="ConnectionAttributes"/>
        </xsd:complexType>
        </xsd:element>
    <xsd:element name="mapping">
                <xsd:complexType>
                        <xsd:simpleContent>
                                <xsd:extension base="xsd:string">
                                        <xsd:attribute name="cmp" type="xsd:string
                                            "/>
                                        <xsd:attribute name="value" type="xsd:string
                                            "/>
                                        <xsd:attribute name="action" type="xsd:
                                            string"/>
                                </xsd:extension>
                        </xsd:simpleContent>
                </xsd:complexType>
   </xsd:element>

   <xsd:attributeGroup name="DeviceAttributes">
        <xsd:attribute name="room" type="xsd:string" use="required"/>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
        <xsd:attribute name="comment" type="xsd:string" use="optional"/>
   </xsd:attributeGroup>

   <xsd:attributeGroup name="VariableAttributes">
        <xsd:attribute name="name" type="xsd:string" use="required"/>
        <xsd:attribute name="id" type="xsd:integer" use="required"/>
        <xsd:attribute name="default" type="xsd:string" use="required"/>
        <xsd:attribute name="comment" type="xsd:string" use="optional"/>
   </xsd:attributeGroup>

   <xsd:attributeGroup name="ConnectionAttributes">
        <xsd:attribute name="target" type="xsd:string" use="required"/>
   </xsd:attributeGroup>

   <xsd:attributeGroup name="AhaSimAttributes">
        <xsd:attribute name="timedEventsFile" type="xsd:string" use="optional"/>
        <xsd:attribute name="startTime" type="xsd:string" use="optional"/>
   </xsd:attributeGroup>

</xsd:schema>
```

C.2 Virtual Person schema file and example

Listing C.2: Grammar for virtual persons

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://castor.exolab.org/Test/Invoice">

   <xsd:element name="ahaPerson">
      <xsd:complexType>
         <xsd:sequence>
                <xsd:element ref="person" maxOccurs="unbounded" minOccurs="0"/>
        </xsd:sequence>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="person">
                <xsd:complexType>
```

```
                        <xsd:sequence>
                                <xsd:element ref="room" minOccurs="1" maxOccurs="
                                    unbounded"/>
                                <xsd:element ref="behaviour" minOccurs="0" maxOccurs
                                    ="unbounded"/>
                </xsd:sequence>
                <xsd:attributeGroup ref="PersonAttributes"/>
        </xsd:complexType>
        </xsd:element>

   <xsd:element name="room">
                <xsd:complexType>
                <xsd:attributeGroup ref="RoomAttributes"/>
        </xsd:complexType>
   </xsd:element>

   <xsd:element name="behaviour">
                <xsd:complexType>
                        <xsd:simpleContent>
                                <xsd:extension base="xsd:string">
                                <xsd:attribute name="source" type="xsd:string" use="
                                    required"/>
                                <xsd:attribute name="cmp" type="xsd:string" use="
                                    required"/>
                                <xsd:attribute name="value" type="xsd:string" use="
                                    required"/>
                                <xsd:attribute name="dest" type="xsd:string" use="
                                    required"/>
                                </xsd:extension>
                        </xsd:simpleContent>
        </xsd:complexType>
        </xsd:element>

   <xsd:attributeGroup name="PersonAttributes">
        <xsd:attribute name="id" type="xsd:integer" use="required"/>
        <xsd:attribute name="startTime" type="xsd:time" use="required"/>
        <xsd:attribute name="endTime" type="xsd:time" use="required"/>
        <xsd:attribute name="variance" type="xsd:double" use="required"/>
   </xsd:attributeGroup>

   <xsd:attributeGroup name="RoomAttributes">
        <xsd:attribute name="main" type="xsd:boolean" use="required"/>
        <xsd:attribute name="id" type="xsd:integer" use="required"/>
        <xsd:attribute name="probability" type="xsd:integer" use="required"/>
   </xsd:attributeGroup>
</xsd:schema>
```

Listing C.3: Example of a config file for virtual persons

```
<ahaPerson>
        <person id="1" startTime="00:30:00-00:00" endTime="17:30:00-00:00" variance
            ="0.75">
                <room id="3" probability="50" main="false"/>
                <room id="1" probability="20" main="false"/>
                <room id="8" probability="20" main="false"/>
                <room id="7" probability="10" main="false"/>
                <behaviour source="daylight" cmp="lower" value="5000" dest="blinds">
                        SET_ON 0.0 0.00
                </behaviour>
                <behaviour source="daylight" cmp="lower" value="3000" dest="
                    lightSwitch">
                        SET_ON 0.0 0.00
                </behaviour>
                <behaviour source="daylight" cmp="higher" value="5000" dest="
                    lightSwitch">
                        SET_OFF 0.0 0.00
                </behaviour>
                <behaviour source="daylight" cmp="higher" value="7000" dest="blinds
                    ">
                        SET_OFF 0.0 0.00
                </behaviour>
        </person>
        <person id="2" startTime="01:30:00-00:00" endTime="17:30:00-00:00" variance
            ="0.75">
                <room id="3" probability="60" main="false"/>
                <room id="1" probability="10" main="false"/>
                <room id="8" probability="30" main="false"/>
        </person>
</ahaPerson>
```

Listing C.4: Grammar for id to variable name mapping, BusAgent

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://castor.exolab.org/Test/Invoice">

   <xsd:element name="ahaBus">
      <xsd:complexType>
         <xsd:sequence>
                <xsd:element ref="variable" maxOccurs="unbounded" minOccurs="0"/>
        </xsd:sequence>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="variable">
                <xsd:complexType>
                <xsd:attributeGroup ref="variableAttributes"/>
        </xsd:complexType>
        </xsd:element>

   <xsd:attributeGroup name="variableAttributes">
        <xsd:attribute name="id" type="xsd:integer" use="required"/>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
   </xsd:attributeGroup>
</xsd:schema>
```

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 ([WJ94]).
AHA	Adaptive Home Automation
AI	Artificial intelligence
ARL	ABLE rule language
DAI	Distributed artificial intelligence
Defuzzification	The process of converting a fuzzy set to a crisp number (FLC output)
DGC	Distributed garbage collection
EIB	European installation bus
ETH	Swiss Federal Institute of Technology
FIPA	Foundation for Intelligent Physical Agents
FLC	Fuzzy logic controller
Fuzzyification	The process of converting a crisp number to a fuzzy set
IB	Intelligent Building
IE	Intelligent Environment
INI	Institute of Neuroinformatics, University and ETH Zurich, Switzerland
JAS	Java Agent Specification
JCP	Java community process
JSR-87	Java Change Request Nr. 87, the official name of the JAS specification during development/review.
LNS	Lon Network Server
LonWorks	Field bus network protocol / standard
MAS	Multi agent system
MIMO	Multi-input-multi-output (classification for a FLC)
MISO	Multi-input-single-output (classification for a FLC)
ML	Machine learning
SISO	single-input-single-output (classification for a FLC)
UNIZH	University of zurich

Bibliography

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

[ANT] Antlr, another tool for language recognition. http://wwww.antlr.org.

[ARL] IBM alphaworks. *ABLE Team, Able Rule Language (ARL) Documentation.*

[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.

[blua] Bluetooth roaming proposals. http://www.hpl.hp.com/personal/Jean%5FTourrilhes/Papers/apr-jt.pdf.

[blub] Roaming for bluetooth. http://users.pandora.be/ElJeffe/Roaming%20for%20Bluetooth.pdf.

[bluc] Bs. http://www.bluetooth.org/specifications.htm.

[Bon96a] Andrea Bonarini. Delayed Reinforcement, Fuzzy Q-Learning and Fuzzy Logic Controllers. In F. Herrera and J. L. Verdegay, editors, *Genetic Algorithms and Soft Computing, (Studies in Fuzziness, 8)*, pages 447–466, Berlin, D, 1996. Physica-Verlag.

[Bon96b] 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.

[Bon97] A. Bonarini. Anytime learning and adaptation of structured fuzzy behaviors. *Adaptive Behavior Journal*, Vol 5(Nr. 3-4), 1997.

[Bro97] R. Brooks. The intelligent room project. In *Proceedings of the second International Cognitive Technology Conference (ICT'97), Aizu, Japan*, 1997.

[BSP$^+$02] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao. Able: A toolkit for building multiagent autonomic systems. *IBM Systems Journal, Applications of Artificial Intelligence*, Vol 41(Nr.3):350–371, 2002.

[CAS] Castor. http://castor.exolab.org.

[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.

[Coe97] Michael H. Coen. Building brains for rooms: Designing distributed software agents. In *Proceedings of the Ninth Innovative Applications of Artificial Intelligence Conference.* AAAI Press, 1997.

[Coe98] Michael H. Coen. Design principles for intelligent environments. In *Proceedings of the 1998 National Conference on Artificial Intelligence.* AAAI Press, 1998.

[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.

[CZ65] Juan Luis Castro and Jose Manuel Zurita. A generic atms. *International Journal of Approximate Reasoning*, 14:259–280, 1996/5.

[EBB$^+$02] Kynan Eng, Andreas Baebler, Ulysses Bernardet, Mark Blanchard, Adam Briska, Joerg Conradt, Marcio Costa, Tobi Delbrueck, Rodney J Douglas, Klaus Hepp, David Klein, Jonatas Manzolli, Matti Mintz, Thomas Netter, Fabian Roth, Ueli Rutishauser, Klaus Wassermann, Adrian Whatley, Aaron Wittmann, Reto Wyss, and Paul F M J Verschure. Ada: Constructing a synthetic organism. In *Proceedings of IROS 2002, IEEE/RSJ International Conference on Intelligent Robots and Systems, Lausanne, Switzerland*. Institute of Neuroinformatics ETH/University of Zurich, 2002.

[FIP02] Fipa abstract architecture specification (xc00001k). Technical report, Foundation For Intelligent Physical Agents, Geneva, Switzerland, 2002.

[FLoA02] Inc. Fujitsu Laboratories of America. Jas agent services (jsr-87) specification. http://www.java-agent.org, 2002.

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

[HCCC02] 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*, In Press, Uncorrected Proof, 2002.

[INS] Mobile phone outlook takes a bite from bluetooth forecast. http://www.instat.com/newmk.asp?ID=213.

[JCO] Jcommon, general java library with supporting classes for various purposes. http://www.object-refinery.com/jcommon/index.html.

[JFR] Jfreechart, a library for producing dynamic plots out of java. http://www.object-refinery.com/jfreechart/index.html.

[JSI] Jsim a java-based simulation and animation environment. http://chief.cs.uga.edu/ jam/jsim/.

[JSP] Java simulation package. http://www.cs.utah.edu/ gback/process/.

[L4J] Log4j logging framework. http://jakarta.apache.org/ant.

[LNS] Lns hmi developer's kit for the java platform. http://www.echelon.com/Products/lns/lnsJava.htm.

[PB01] Joseph P.Bigus and Jennifer Bigus. *Constructing Intelligent Agents using Java*. John Wiley Sons, Inc., New York, 2001. ISBN : 0-471-39601-X.

[Rin] Andreas Rinkel. Java tools for object oriented process simulation (jtoops). http://rinkel.ita.hsr.ch/jtoops/index.html.

[RS02] Ueli Rutishauser and Alain 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.

[RSDJ02] Ueli Rutishauser, Alain Schaefer, Rodney Douglas, and Josef Joller. Adaptive building intelligence, a multi-agent approach. In *Proceedings of Embodied AI Workshop, Special issue on the design principles. Zurich, Switzerland*. Artificial Intelligence Laboratory, Department of Information Technology, University of Zurich, Switzerland, 2002.

[Sto00] Peter Stone. *Layered Learning in Multiagent Systems*. MIT Press, Cambridge,Massachusetts, 2000. ISBN : 0-262-19438-4.

[SZ96] Jurgen Schmidhuber and Jieyu Zhao. Multi-agent learning with the success-story algorithm. In *ECAI Workshop LDAIS / ICMAS Workshop LIOME*, pages 82–93, 1996.

[VC00] Graham Clarke Victor Callaghan. A soft-computing dai architecture for intelligent buildings. Technical report, Department of Computer Science, University of Essex and Department of Computer Science, University of Hull, 2000. http://cswww.essex.ac.uk/intelligent-buildings/publications/springerverlag.pdf.

[Wei99] Gerhard Weiss, editor. *MULTIAGENT SYSTEMS, A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, Massachusetts, 1999. ISBN : 0-262-23203-0.

[WJ94] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. HTTP://www.doc.mmu.ac.uk/STAFF/mike/ker95/ker95-html.h (Hypertext version of Knowledge Engineering Review paper), 1994.

[ZDN] Uk passes mobile adoption milestone. http://news.zdnet.co.uk/story/0,,s2079994,00.html.