MASTER THESIS

Coupling Media Façades and Building Automation

Submitted at the Faculty of Electrical Engineering and Information Technology,
Vienna University of Technology
in partial fulfillment of the requirements for the degree of
Master of Science

under supervision of

O. Univ. Prof. Dipl.-Ing. Dr.techn. Dietmar Dietrich
and
Ass. Dipl.-Ing. Thomas Rausch

Institute number: 384
Institute of Computer Technology

by

Bernhard Breinbauer
Student ID 0025121
Schwendt 7, 4775 Taufkirchen

Vienna 27. Nov. 2007 _____

Abstract

Information technology is radically changing our daily life and the world we live in. Media façades are a remarkable good example for this, because they are literally changing the appearance of buildings and therefore our surroundings. However media façades have not yet tapped their full potential as they are self-contained systems and have only limited capabilities to interact with their environment. This shortcoming may be overcome by opening media façades to other systems.

Building automation systems gather lots of information on the state of a building. The information can be valuable for media façades. Furthermore building automation may support media façades in showing visual effects on the façade of a building by either dampening the background room lighting or by producing visual effects itself.

A problem caused by coupling building automation and media façades are the different semantics and data structures used in each system. This master thesis attempts to solve the problem by implementing a system which enables communication between building automation and media façades by translating information into native data structures of the respective domain.

For transforming building automation data into a form that is usable for media façades the *Artificial Recognition System* is used. The Artificial Recognition System comprises a *symbolization* technology, which condensates huge amounts of sensory input data of small informational value into a smaller number of data points with high informational value. The data produced by the symbolization process contains more inherent information and is therefore of more value to media façades than single building automation data points.

Furthermore the implemented system enables media façades to control aspects of the building automation system by utilizing an interface which is native in the media façade domain. The interface is based on *pixels* which are mapped to actuators (eg. sunblinds and room lighting) of the building automation system. The pixels of a bitmap provided to the interface are applied to the corresponding actuators and produce the according effect on the façade of the building.

To test and demonstrate the implemented system parts of a building automation system and parts of a media façade have been integrated into a prototype system. The prototype comprises a representation of a building façade with windows, room lighting, sunblinds and a media façade. The representation of the building façade is used to demonstrate the cooperation of building automation and media façade.

Kurzfassung

Informationstechnologie verändert stetig unsere Lebensweise und die Welt in der wir leben. Diese Veränderung ist am Beispiel Medienfassaden sehr gut zu beobachten, da Medienfassaden das Aussehen von Gebäuden und damit unserer Umgebung verändern. Doch Medienfassaden haben ihr Potential zur Veränderung unserer Umwelt noch nicht ausgeschöpft, da sie in sich geschlossene Systeme sind und deshalb nur eingeschränkt mit der Umwelt interagieren und kommunizieren köennen.

Gebäudeautomation sammelt Daten über den Zustand eines Gebäudes. Diese Daten können auch für Medienfassaden von Interesse sein. Weiters kann Gebäudeautomation die Effekte der Medienfassade unterstützen indem entweder störende Raumbeleuchtung abgedunkelt wird oder indem visuelle Effekte mit der Gebäudeautomation erzeugt werden.

Gebäudeautomation und Medienfassaden können nicht direkten Datenaustausch betreiben, da die jeweiligen Datenstrukturen und Semantiken zu unterschiedlich sind. Diese Diplomarbeit implementiert ein System, das als Vermittler zwischen Medienfassade und Gebäudeautomation agiert indem es Information von einer Domäne in Datenstrukturen der anderen übersetzt.

Für die Aufbereitung von Gebäudeautomationsdaten in eine für Medienfassaden nutzbare Form wird das *Artificial Recognition System* verwendet. Das Artificial Recognition System verfügt über eine *Symbolisierung* genannte Technologie, die große Mengen von Sensordaten mit relativ wenig Informationsgehalt in eine kleinere Menge an Daten mit größerem Informationsgehalt verwandelt. Die durch Symbolisierung produzierten Ausgangsdaten beinhalten mehr inhärente Information und sind daher für Medienfassaden von mehr Wert als einzelne Sensordaten.

Weiters ermöglicht das implementierte System Medienfassaden auf Teile des Gebäudeautomationssystems Einfluss zu nehmen. Der Medienfassade wird dabei eine, in der Domäne von Medienfassaden gebräuchliche, pixelbasierte Schnittstelle zur Verfügung gestellt. Dazu werden Aktuatoren des Gebäudeautomationssystems Pixelkoordinaten zugeordnet. Die implementierte Schnittstelle nimmt von der Medienfassade Bitmaps entgegen. Die Werte der Bitmappixel werden auf die Aktuatoren im Gebäudeautomationssystem angewendet und damit die entsprechenden Effekte auf der Fassade des Gebäudes erzeugt.

Für Demonstrationszwecke und zum Testen der entwickelten Komponenten wurden Teile eines Gebäudeautomationssystems und Teile eines Medienfassadensystems in einen Prototypen integriert. Der Prototyp enthält die Darstellung einer Gebäudefassade mit Fenster, Raumbeleuchtung, Sonnenblenden und einer Medienfassade. Die Darstellung der Gebäudefassade wird zur Demonstration der Interaktionen zwischen Gebäudeautomation und Medienfassade verwendet.

Acknowledgments

My studies would have never been completed without the support of many individuals.
First I would like to thank my parents, who supported me in my decision to study and backed me up since the beginning. Also I am very grateful to my grandparents, my sisters and my brother for the warmth and strength they shared with me.
I would like to thank my girlfriend Irmgard, who always patched me up when my motivation and mood went down.

For support in completing this work I am very grateful to Thomas Rausch for his great supervising and that he suggested this particular topic to me. Also I would like to thank Wolfgang Burgstaller for the discussions while planning and building the demonstration suitcase. Thanks also to the people involved in the *Mediafacade.net* project for the great and enjoyable cooperation.

Contents

Abbreviations

API	Application Programming Interface
ARS	Artificial Recognition System
ARS-PC	Artificial Recognition System – Perception
ARS-PA	Artificial Recognition System – Psychoanalysis
ASHRAE	American Society of Heating, Refrigerating and Air-Conditioning Engineers
ASN.1	Abstract Syntax Notation number One
AXIS	Apache eXtensible Interaction System
DIN	Deutsches Institut für Normung
GUI	Graphical User Interface
ICT	Institute of Computer Technology
IDL	Interface Description Language
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISO	International Organization for Organization
JAX-WS	Java API for XML – Web Services
HTML	Hypertext Markup Language
HTTP	Hyper Text Transport Protocol
HVAC	Heating, Ventilation and Air Conditioning
LAN	Local Area Network
LED	Light Emitting Diode
LCD	Liquid Crystal Display
M2M	Machine-to-Machine
NTP	Network Time Protocol
NV	Network Variable
NVC	Constant Data Point
NVE	External Data Point
NVL	Local Data Point
OASIS	Organization for the Advancement of Structured Information Standards

oBIX	Open Building Information Xchange
PoE	Power over Ethernet
RFC	Request for Comments
RPC	Remote Procedure Call
SMTP	Simple Mail Transfer Protocol
SNVT	Standard Network Variable Types
SOAP	Simple Object Access Protocol
UDDI	Universal Description Discovery and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WSDL	Web Service Description Language
XML	eXtensible Markup Language

Chapter 1

Introduction

Modern architecture attempts to revolutionize today's concept of "buildings". New building materials [GTL94] and advances in information technology [KNS05] allow architects to design and create the spaces, where we live, work and meet in, in ways not possible before. Amongst other technologies, media façades fuel this architectural revolution by allowing architects and artists to change the way buildings are perceived in the public.

In the past, changing the appearance of a building after its completion was nearly impossible and tied to huge costs. With media façades, this got remarkably easier: the appearance of a building can be altered very fast and with low costs after the installation of the media façade. Most of today's media façades are utilized on office buildings for companies to increase their prestige in the public. This is achieved by showing, amongst other things, entertaining projections, commercials or allowing artists to utilize the media façade.

The content displayed by the media façade is mostly created off-line, meaning that images, videos and light patterns are designed remotely by artistically skilled people and not generated by the media façade system itself. The beforehand created content is uploaded to the media façade control system, which will deliver the content to the façade in a system specific sequence. Apart from some exceptional projects like project *Touch*[1] at the *Dexia Towers* in Brussels, where by-passers can influence the content of the media façade, the sequence of the displayed content does not react to events in the environment of the media façade. The definition of environment used here, includes the surrounding of the building, the building itself and the inside of the building. It is desirable to provide means to allow interaction of the media façade control system with the environment. For such interaction the media façade has to be "aware" of events and situations in its environment. Information on situations occurring inside of the buildings is today already available in building automation systems.

Building automation is a fast growing and evolving technical discipline which enables monitoring of environmental data. Based on this data, building automation operates the controls of the environment it is responsible for, thus allowing efficient use of resources and ensuring comfort of the occupants. As building automation is utilized in nearly every modern office building and its functionality is constantly growing [KNS05], more and more applications will be interested in reusing the environmental data gathered by building automation. Media façades are such an application.

[1] `http://www.dexia-towers.com/index_e.php?file=dtb_2006_touch`

Interconnecting building automation and media façades is usually not possible in an easy and straight forward way. Building automation systems are encapsulated systems which are only recently developing interfaces to the gathered data [PSKD06]. Furthermore building automation operates on low level data, which has not much meaning for other systems. Also manufacturers of media façades have only limited knowledge on building automation technology and its possibilities. Therefore an interface which allows interaction of these two worlds is needed.

This master thesis is part of the project *Mediafacade.net*[2], which is a consortium of companies and research facilities, which aims to develop standardized state-of-the-art technologies for media façades. The *Institute of Computer Technology* (ICT) is member of the Mediafacade.net consortium and is responsible for researching possibilities, how integration of building automation and media façades can be accomplished.

1.1 Motivation and Background

As described above the goal of this master thesis is to build a bridge between building automation and media façades. The motivation for this attempt is three-fold:

1. The displayed content of media façades is predefined and the dynamic appearance of the media façade therefore limited. This limits the usage scenarios of media façades. By providing environmental data to the media façade the limitation can be weakened. Furthermore the interconnection is interesting from the architectural point of view. Buildings are designed to integrate into its surrounding (but in a interventional way – the surrounding is also altered by buildings). After the designing phase this integration can not be changed anymore. But by enabling environment-aware media façades interaction of building and surrounding is possible and anticipated. Such facilities could lead to interesting artistic and architectural experiments regarding building-environment-interaction.

2. Building automation is on the verge of a new era. The expected exponential growth of data points [PP05] will lead to new services and functionality. To provide this new services to external systems new interfaces are being developed. These interfaces provide access to the data available in building automation systems and therefore allow external systems to retrieve information on the status of the building. The retrieved information allows systems to become aware of its environment.

3. The *Artificial Recognition System*[3] (ARS) [PP05] is currently in development to aid in managing the vast amount of data points in building automation systems, which are expected in the near future. ARS aims to implement a technical model of the human brain. The implemented system will be able to perceive and understand its environment and to interact autonomously. As ARS is meant to deal with great amounts of sensory input data, an "information condensing process" has been developed which transforms huge amounts of data with small informational value into high value information. The information condensing technology is the reason why ARS is relevant for this thesis.

[2] `http://www.mediafacade.net/`
[3] `http://ars.ict.tuwien.ac.at/`

While building automation operates on low value data, media façades control systems are not interested in such data, but they are interested in data which describe situations in the surrounding. Therefore a "translation system" is required which helps in the communication between building automation and media façades and ARS may be used as such a system.

The following sections provide some background information on each of the three main topics (building automation, cognitive systems and media façades) of this master thesis.

1.1.1 Building Automation

Smart dust, as described in [HSW+00], advances in reduced power consumption [MGH05] and wireless networks [Cal03] in combination with advances in building automation technology [PM03] are some of the reasons, why the growth of the number of nodes in building automation systems is expected to be exponential. New approaches to ensure functional, manageable and economical building automation systems are required and developed [PP05].

The growth of nodes – especially sensor nodes – will allow to retrieve much more information about the environment than today and will allow to build a much more comprehensive view of the environment the building automation system is operating in. In the future applications outside the building automation domain will get interested in this comprehensive view of the environment. In recent years this interest in building automation data already led to the production of "gateway nodes", which allow communication between building automation systems and other computer systems.

In this master thesis such a gateway node will be used to implement a system, which acts as interface between building automation and media façade. A detailed description of the gateway node and its utilization can be found in Chapter 3.

1.1.2 Cognitive Systems

Project ARS attempts to implement models of the human brain developed by scientific disciplines like neuroscience and psychoanalysis into a technical system [Pra06]. The domains of these two disciplines – while working on the same "material" – are rather different. Neuroscience deals with the low level components of the brain and in which ways they interact to eventually accumulate to intelligent beings. Psychoanalysis on the other hand tries to understand the human mind as whole. Drives, desires, emotions and decision making is amongst other things the domain of psychoanalysis. ARS adopts the two views of neuroscience and psychoanalysis by following a two-sided approach:

- ARS-PC (Perception) is a bottom up approach to perception. It transforms low level data into high level information by means of *symbolization* [Rus03]. Symbolization refers to the process of condensing data into high level information (called *symbols*). For example: If the floor in a room is equipped with tactile sensors a sequence of messages "tactile sensor number x activated" can be condensed into the information "someone went through the room", which contains much more inherent information than the sequence of data.

- ARS-PA (PsychoAnalysis) is a top down approach that allows systems to make independent decisions, by implementing Sigmund Freud's Ego-Superego-Id model into a technical system. Based on emotions and drives it allows systems to evaluate situations and work autonomously towards specific goals.

In this thesis ARS will be evaluated if it is applicable for the envisioned task: It should act as the "translation system" described above, to provide situation describing, high value information to the media façade control system. In Chapter 4 the utilization of ARS in this master thesis is discussed.

1.1.3 Media Façades

Media façades are light installations on publicly viewable façades of buildings. They use LED technology to show patterns, images or even short movies to the viewers. Popular examples for already deployed media façades are the *Kunsthaus*[4] in Graz and the *Uniqa Tower*[5] in Vienna, which is shown in Figure 1.1. More examples of existing media façades can be found on the website of the *media architecture group*[6].

The used LEDs are able to produce multiples of colors which allows much more sophisticated effects than approaches with monochrome LEDs. Typically the LEDs are mounted on the façade of the building in lines. Depending on the architecture of the building this lines may be vertical or horizontal. These "lines of light" in vertical form are visible in Figure 1.1. The distance between the LEDs depends on architectural design of the building and aimed resolution. From some distance the individual LEDs appear – depending on the overall resolution – as a fairly homogeneous display and the viewer is able to recognize the displayed images.

The content, which will be displayed on a media façade, has to be designed and prepared beforehand by the users of media façades. Users are typically artists or media designers. The control system of the media façade processes the content and activates the respective LEDs. The sequence of displayed content is either statically defined when adding new content or may be randomly chosen to generate a more dynamic appearance. The control system has no notion of the meaning of the content nor does it have a notion of the surrounding environment and the state the building is in. This lack of information may lead to wrong behavior in exceptional situations. For expample, if the building is on fire and the fire brigade tries to save people by fetching them from the windows, the still operating media façade might dazzle the fire fighters and therefore hinder the saving of people. In normal situations the media façade could act supportingly to the purpose of the building if it has information on the environment, for instance on a multi-storey car park a media façade could indicate in which levels there are still free parking lots by highlighting this levels.

As shown above providing media façades with information about the environment extends the usage scenarios from simple, but artistic projection of images to more sophisticated applications. But it also allows to improve the artistic usage. Media designers may assign designated content to environment scenarios and therefore allow the media façade to appear interacting with its environment.

[4] `http://www.bix.at`
[5] `http://tower.uniqa.at`
[6] `http://www.mediaarchitecture.org`

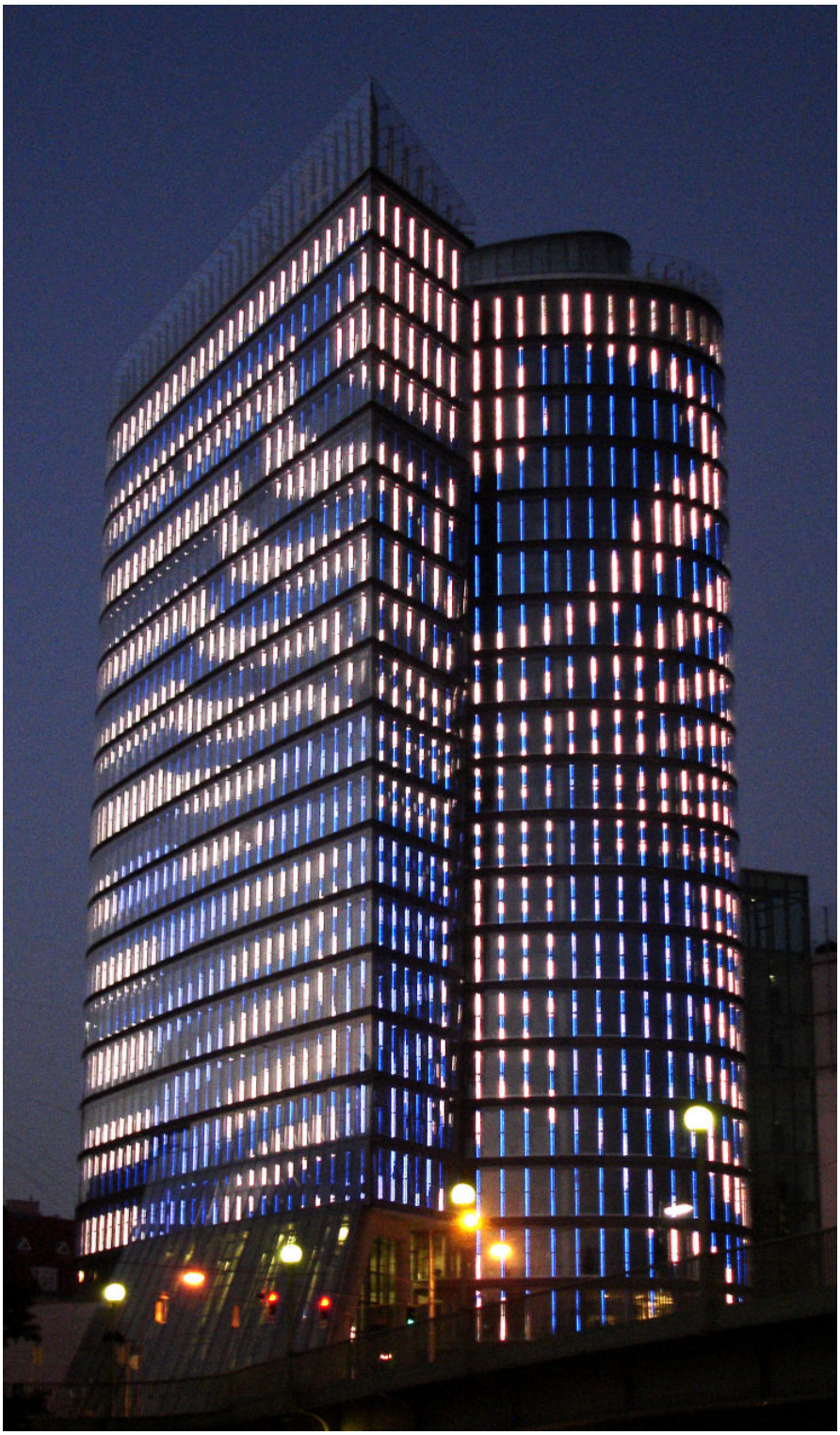

Figure 1.1: Vienna: Uniqa Tower media façade. This media façade is structured in vertical lines, which are clearly visible in the picture.

Furthermore it would be possible to grant the media façade a way to control parts of the systems integrated into building automation. This would enable the media façade to emphasize the shown effects moving the sunblinds or turning on and off the room lighting.

1.2 Goals

The goal of this master thesis is to implement a system which acts as an interface between media façades and building automation systems. Data will be retrieved from a building automation installation and transformed into high value information. Based on the gathered information, the media façade will be able to decide which content it has to show. Additionally the media façade will be granted a way to control parts of the building automation system. Figure 1.2 shows a conceptual overview of the dataflow in the system.

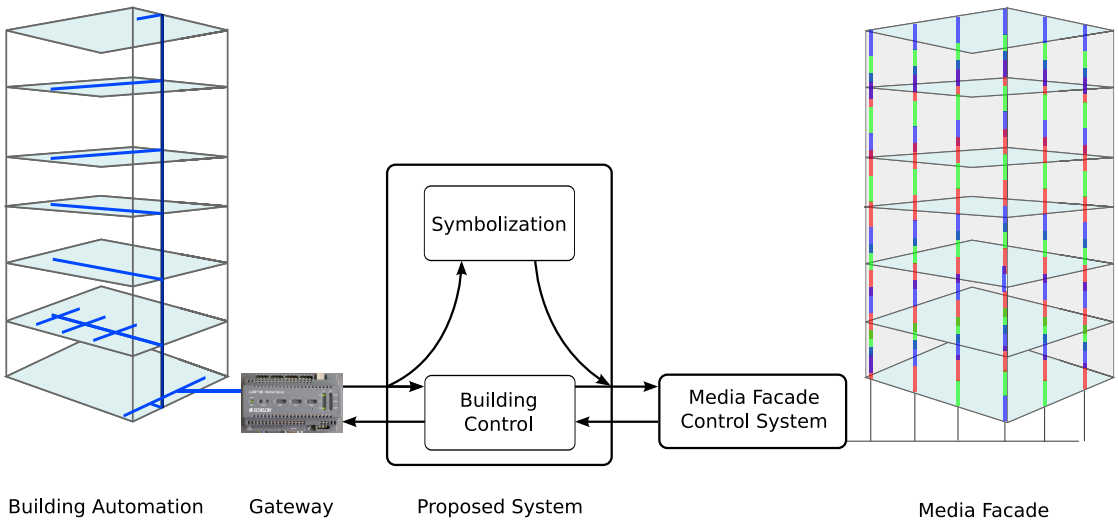

Figure 1.2: Overview of the dataflow in the proposed system. *Building automation* data is gathered and forwarded to the *media façade*. Optionally the data can be transformed into high value information by the means of *symbolization*. In the other direction the *media façade control system* is provided an interface to control building automation systems.

The system collects building automation data which comprises a huge amount of low-level information such as “light on”, “switch pressed” and “temperature 20°C”. Such low-level data has to be transformed into high-value information. ARS will be evaluated and eventually used for this task. The high value information is then passed to the media façade which can use it for decision making. A dataflow channel starting at the media façade ending at the building automation (see Figure 1.2) enables the media façade to control building automation systems. Such a control channel can be used for emphasizing the displayed images as described above.

To provide valuable information to media façades some scenarios in which the data can be utilized has to be identified and investigated. For each of these scenarios the requirements of media façades have to be evaluated and the proposed system has to be designed to fulfill these requirements.

The result of the work will be a prototype, which shows the interaction of media façade systems and building automation in two ways, as shown in Figure 1.2.

1. An interface will be developed which allows media façades to control aspects of building automation systems. This functionality will be integrated into a demonstration suit case. The suit case will comprise a small number of building automation nodes and a model of a media façade. The combination of building automation and media façade into one suit case will allow to present the interaction of media façade and building automation at trade shows.

2. Ways to utilize ARS for extracting information from building automation data will be evaluated. If ARS proofs to be capable of this task, a system will be developed which passes building automation data to ARS and provides the aggregated information to the media façade system.

Chapter 2

Media Façades

The term *media façade* refers to building façades which are in some way able to show content on its surface. Usually the term is used to refer to light installations and projections of images, but is not constricted to this sort of façades. For example mechanical systems mounted on the surface of a building are also referred to as media façades. An example for such a mechanical system is the *Aegis Hyposurface*[1].

However this master thesis will focus on light installations as they are more widely used than other systems and because the partners in the *Mediafacade.net*[2] project have experience with LED (*Light Emitting Diode*) based media façades.

2.1 Existing Media Façades

Before discussing the desired cooperation of media façades and building automation some existing media façade installations are introduced and some characteristics of media façades are described. One of the first media façade installations done in Austria is *BIX*[3] at the *Kunsthaus Graz*. The outer hull of the Kunsthaus consists of acrylic glass[4]. On the east side of the Kunsthaus 930 fluorescent lamps with a ringlike form are mounted behind the outer hull. The fluorescent lamps produce only white light, therefore the display is monochrome. The brightness of the lamps can be varied continuously. The façade is able to show images and even animations and movies with a frame rate of 20 frames per second.

BIX is a rather untypical example for a media façade because it uses fluorescent lamps to produce visual effects. In the recent future LED technology has become the most used technology for media façades, because of ever decreasing prices of semiconductor technologies, the possibility to tightly pack LEDs in packages and the ability to produce bright and multicolor light. An example for a media façade with LED technology is the *Uniqa Tower*[5] in Vienna, which has already been shown in Figure 1.1.

[1] `http://www.sial.rmit.edu.au/Projects/Aegis_Hyposurface.php`
[2] `http://www.mediafacade.net/`
[3] `http://www.bix.at/`
[4] Better known as Plexiglas
[5] `http://tower.uniqa.at`

The hull of the Uniqa Tower consists of two layers of glass. Between the layers LED modules are mounted. The LED modules are about four square centimeters of size and contain four multicolor LEDs. 40.000 LED modules are mounted on the whole façade of the tower, which is about 7.000 square meters of size. The used LEDs are not bright enough to run at daylight, therefore the façade is activated every day at dusk. The resolution of the Uniqa Tower media façade is rather low, therefore it is mostly used to show animations on it. Very rarely videos and images have been shown on the façade.

The above mentioned media façades BIX and Uniqa Tower are encapsulated systems which comprise only management interfaces to define the projected content but no interface for data input about the environment. As stated in Chapter 1, architects and media designers are interested in media façades which are "interacting" with the environment. Experiments and implementation of such interaction scenarios already exist, one of them will be described below.

From December 2006 to January 2007 the group *LAb[au]*[6] used the *Dexia Towers*[7] in Brussels to experiment with possible interaction scenarios between people and buildings. For the project, named *Touch*[8], a computer station was placed at the plaza at the bottom of the tower. The computer station comprised a touch screen, which was able to react on multiple events at once and which was big enough to be operated by more than one user at once. By-passers were able and allowed to use the touch screen. The actions done on the touch screen by the users influenced the media façade of the Dexia Tower.

As can be seen by the example of the Dexia Tower there is interest to use new technologies like media façades to alter the public appearance of buildings. The Touch project does not really interact with its environment but with users of the touchscreen. But also interaction scenarios which are directly connected to the environment are of interest for architects and media façades. Information on the environment can be gathered and utilized to produce a more interactive appearance of the media façade. As source for environmental data building automation can be used because building automation systems do gather data on the building to control the building automation related systems inside the building. The next section discusses the benefits of information exchange between media façades and building automation.

2.2 Media Façades and Building Automation

Today building automation systems and media façades are separated systems, even though they could benefit from information exchange. As described above architects are interested in changing the public appearance of buildings depending on situations in the environment. But not only interaction also communication of the building with the outside is desired. A repeatedly used slogan for such a communication is *"to communicate the inside of a building to the outside"*. Doubtlessly that statement is rather imprecise and fuzzy. Luckily the goal of this master thesis is not to do communication between inside and outside of buildings, but to provide a technical foundation for architects and media designers to experiment, work on

[6] `http://www.lab-au.com/`
[7] `http://www.dexia-towers.com/`
[8] `http://www.dexia-towers.com/index_e.php?file=dtb_2006_touch`

and implement their visions. Therefore the main focus is finding and implementing ways how media façades can exchange information and therefore interact with each other.

Building automation gathers information on the current conditions in the inside of buildings. The information is used to control actuators of the incorporated systems like heating, air conditioning and room lighting. Ever since the beginnings of building automation the deployed sensors have become more and more sophisticated. For example in the last years sensors which can detect if rooms are occupied are deployed regularly in modern buildings. Sophisticated sensor systems will be used more and more and the number of applied sensors will grow drastically in the near future [PP05].

Therefore building automation is a quite capable candidate to act as source of environmental data. Chapter 3 discusses how data can be retrieved from building automation systems, while Chapter 4 introduces concepts how building automation data can be transformed into a format suitable for media façades. This section will continue with some scenarios where information exchange between building automation systems and media façades would be beneficial.

One example for benefits if building automation and media façade would be able to exchange data has already been presented in Chapter 1. In critical situations like fire alarm and building evacuation the media façade should be deactivated automatically. The media façade could even deactivate itself if it would be notified about such critical situations.

Media designers are also interested in cooperation of media façades and building automation systems. An example where cooperation would be beneficial for media façades is shown in Figure 2.1. It shows the Uniqa Tower with activated media façade. But because the room lighting is switched on the effects of the media façade are barely visible.

Turning off the room lighting in the whole building is no option, because the building should still be inhabitable, even if the media façade is activated. A solution to prevent interfering room lighting would be to allow the media façade some control over sunblinds, which may be mounted at the windows. If the media façade is activated, the sunblinds of the windows where the room lighting is turned on, can be lowered. Therefore the room lighting is no longer visible from the outside and interference of the room lighting with the effects of the media façades is prevented.

Another possibility to utilize building automation for supporting the effects produced by a media façade is to use the systems incorporated into the building automation system to produce effects. For example the room lighting could be used to produce visual effects. Something similar has been done for the project *Blinkenlights*[9]. The windows of a vacant building were used as public display. While for project Blinkenlights the inside of the building have been heavily modified, modern building automation systems would allow similar effects with the already deployed technology. In a similar way building automation can be used to emphasize the effects of the media façade. For example room lighting or sunblinds can be used to produce a "wave effect" by altering the lights/sunblind positions sequentially in specific pattern.

Aside from producing visual effects for the outside of the building, building automation technologies can also be used to gather information about the status of the inside of the building. Building automation data which could be interesting for media façades and data which is available from building automation systems is discussed in the next section.

[9] `http://www.blinkenlights.de/`

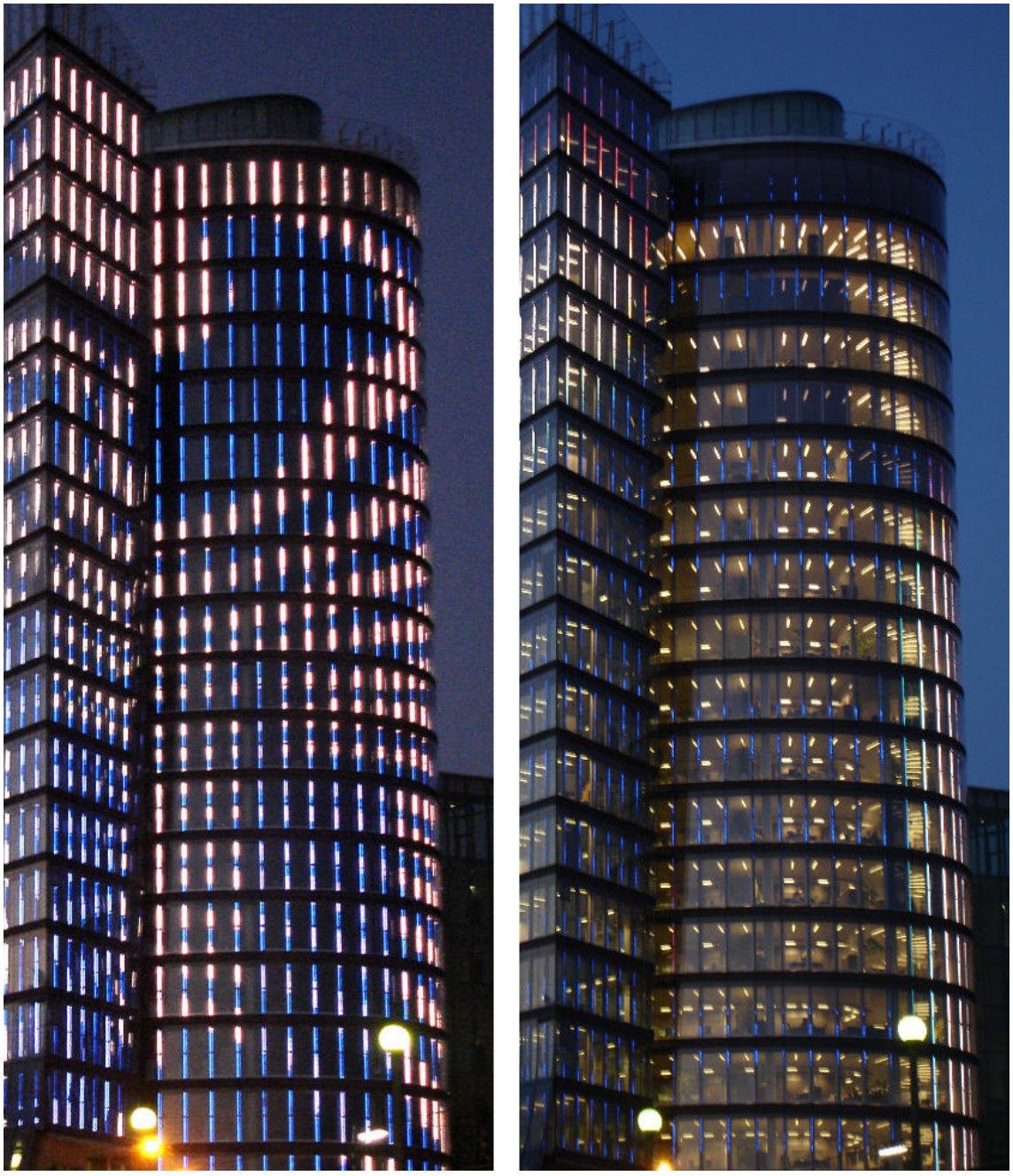

Figure 2.1: Room lighting interfering with media façade projections. In the left picture the effects shown by the media façade are clearly visible. In the right picture the effects are barely visible because the room lighting of the building is interfering heavily.

2.2.1 Usage of Building Automation Data

As already discussed above media designers would like to utilize media façades for communication with the outside of the building. One example for such a communication scenario is the Touch project, where by-passers were able to influence the projected content of the media façade. In this master thesis ways to exchange data between media façades and building automation systems had to be found. This section introduces scenarios and building automation data which could be valuable for media façades.

As already described above modern building automation systems often comprise sensors to detect if a room is occupied. Inside the building automation system the occupied/unoccupied information may be used to turn on or off the lights of a room. For media façades the number of occupied rooms in relation to unoccupied rooms might be interesting. Along with the occupied data, information about the location of the rooms might be valuable as media façades could concentrate its effects around rooms which are occupied or unoccupied.

In some buildings the systems which allow and observe access to the building are integrated into the building automation system. Some generic access information might be of interest for media façades. For example the number of people in particular sections of the building. Another example would be a shopping mall which comprises a system to measure the flow of customers inside the building. The media façade might use this information in its projections.

Of practical interest for media façades may be information on the current weather conditions. For example information on the position of the sun and how bright the sun shines – which includes information on cloudiness – can be used to adapt the brightness of the LEDs of the media façade. Another possibility would be to show effects only on the parts of the building which are not directly exposed to the sun light or to alter the projected content if it starts to rain.

As shown in the examples above there are a lot of scenarios – practical or artistical – where data exchange between media façades and building automation is beneficial. Chapters 3 and 4 discuss possibilities how data exchange between media façades and building automation can be implemented.

2.2.2 Technical Design of a Media Façade

This section will introduce the technical design of a media façade system at the example of *BLIP*[10] products. BLIP is a partner in the Mediafacade.net project.

Big media façade systems are typically hierarchically structured. Figure 2.2 shows a typical hierarchical structure for media façades as it can be found in [BLI07]. The LEDs of the media façade are interconnected via a bus for data transmission and wires for power supply. Every LED node comprises a chip, which implements the transmission protocol and controls brightness and color of the LED. A picture of such interconnected LEDs is shown in Figure 6.7.

In Figure 2.2 an *Image Generator* is connected to communication bus and power supply wires. The Image Generator provides power and sends commands over the communication bus. The Image Generator is the contolling instance on the communication bus, the LED nodes simply react to command issued by the Image Generator.

[10] `http://www.blipcreative.com/`

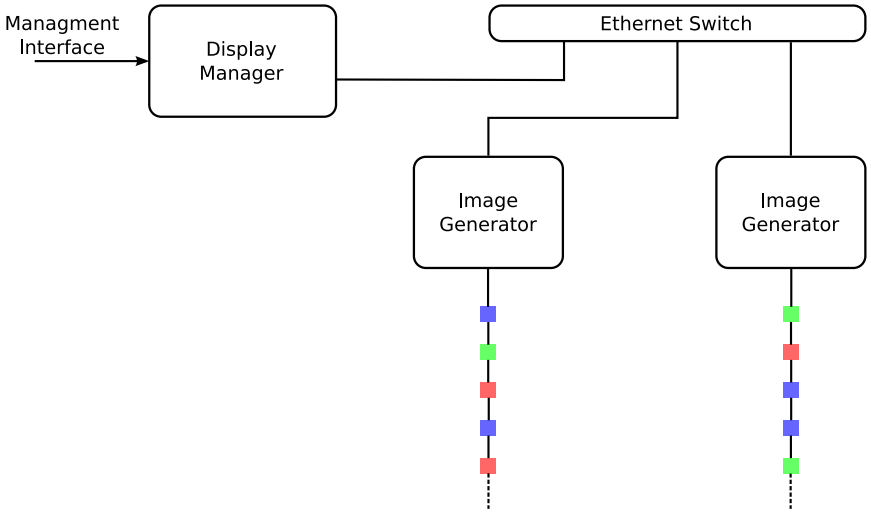

Figure 2.2: Hierarchical structure of a media façade control system. The LEDs are connected to Image Generators which provide power and control the LEDs. A Display Manager can be used synchronize the actions taken by individual Image Generators.

If more LEDs than one Image Generator can manage are required a *Display Manager* can be used. It can be connected to various Image Generators via an Ethernet [IEE95] switch. The Display Manager's responsibility is to synchronize the effects generated by individual Image Generators.

Chapter 3

Interfacing Building Automation Systems

In its beginnings building automation comprised simple tasks like room temperature and lighting control. Such simple control tasks are typically composed of three components: a sensor, an actuator and a control unit. Figure 3.1 shows schematically a control application with these three components. Every control task needs a reference value, on which it tries to stabilize its working point [Wei98, pp. 9–11]. The reference value is also shown in the figure.

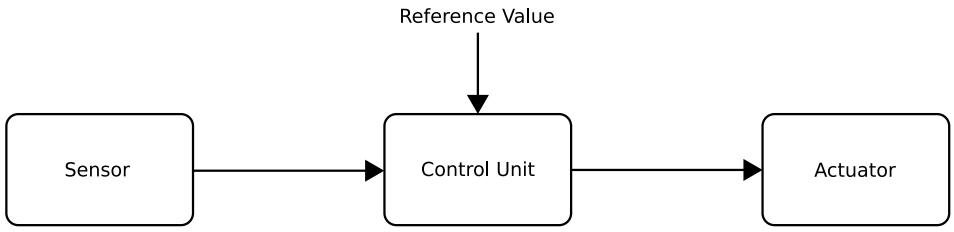

Figure 3.1: Control application comprising *sensor*, *actuator* and *control unit*. The control unit manipulates the actuator to ensure the *sensor value* equals the *reference value*.

An example of a simple control loop in building automation is room temperature control. The actuator is a device at the heater which enables the control of the dissipated heat. A temperature sensor – typically located in some distance of the heaters – enables the monitoring of the room temperature. The control unit reads the temperature from the sensor and manipulates the actuator to achieve constant temperature values in the room[1].

Simple control tasks have to be done everywhere in a modern building. Moving the sunblinds according to the position of the sun, switching on the light and supplying fresh, cool air. In the past the tasks were separated from each other, every functionality was implemented by another manufacturer, the systems were not designed to interact [Bus97].

[1] Usually the temperature sensor and the control unit are integrated into one device.

Integration came along with the development of field buses. Field buses are specialized buses for short distances and small amounts of data. Because of this constraints the buses and nodes are relatively cheap and can be deployed on a large scale. Control systems started to use field buses, firstly separated from each other. Obviously the desire to utilize only one bus per building rose because of economical and operational advantages, such as reduced wiring costs and reduced maintenance work when using only one common bus.

Field buses specialized for building automation requirements were developed. Examples of this buses are: KNX [KA04], BACnet [ASH04] and LonWorks [CEA02]. Every sensor, every actuator and every monitoring and control unit in a building are connected to the deployed bus. The various control tasks operate on the same communication channel and therefore can cooperate and exchange information. For instance the temperature control system can take in account the position of the sun and adjust the heaters respectively the air condition accordingly. This integration process of different control systems is nearly completed today. Modern buildings utilize one dedicated bus for the building automation system.

While building automation evolved from simple, separated control tasks to an autonomous, interconnected and complex system, other network technologies advanced at least at the same rate. Ethernet [IEE95] evolved to the de facto standard for Local Area Networks (LAN) and the internet grew [XP03] to an colossal accumulation of information, services and junk[2]. Obviously the idea rose to allow some sort of interconnection between building automation networks and the other information technology networks deployed in a building (typically Ethernet). As data transmissions on the building automation network can not be simply mapped on Ethernet data transmissions a system is required which provides an interface to the building automation data. Most of these system use known and proven internet technologies, like web services, to provide such an interface [Ehr04].

Nowadays nodes, which act as gateway between the building automation system and the local PC network, are available for every building automation bus system. Most of them comprise a web server to allow easy supervision, manipulation and management of the building automation control functions. Every PC with a web browser can be used to perform these tasks. While this is nice for human operators it is insufficient for interaction with other technical systems. Therefore the gateway nodes provide additional interfaces which are designed for data exchange with technical systems. Such communication is usually called *Machine-to-Machine* (M2M). Most of the gateway nodes implement a *web service* to allow external applications to access the data available in the building automation system. Today these web services are technology dependent, which means they differ depending on which building automation bus is used. But there are already standards in the works which attempts to solve this diversity.

3.1 Web Services

Web services are defined by the *World Wide Web Consortium*[3] (W3C) as: "*A Web service is a software system designed to support interoperable Machine-to-Machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using*

[2]The presence of junk on the internet may be obvious, but for detailed description and an estimation of the size of the problem see [Atk03] and [RW07].

[3]`http://www.w3.org/`

SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards" [W3C04].

Therefore web services are an approach to M2M communication. The services expose interfaces which utilize Web-related standards to provide access to underlying systems and data in a platform independent manner. Platform independency is assured by using a XML-based protocols.

3.1.1 Concept

Three participants are involved in the process of using – usually called consuming – a web service. A service provider which provides an interface (web service) to the outside world. A service consumer which attempts to consume a web service and a service broker which comprise a registry of known web services and can therefore help the service consumer to find the service provider. Figure 3.2 shows the three participants and the involved communication flows.

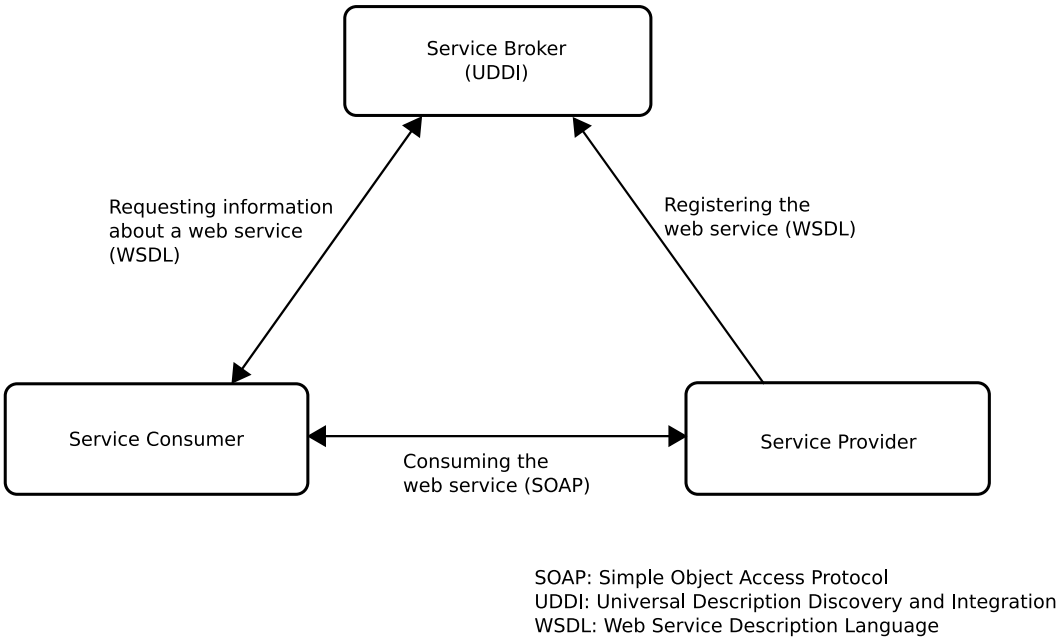

Figure 3.2: Web service roles and communication flows. The *service provider* registers its WSDL service description at the *service broker*. The *service consumer* queries the service broker for available service providers and consumes the web services via exchange of SOAP messages.

A typical web service invocation scenario starts with the service provider registering its services at the service broker. If a service consumer is searching for a specific web service and does not know where it is located, it can request that information from the service broker. If the service broker finds the requested web service in its registry it delivers the information to the service consumer. After receiving this information the consumer can start to use the web service. In this scenario three technology standards are utilized:

- *Universal Description Discovery and Integration* (UDDI) [OAS04] for service discovery.
- *Web Services Description Language* (WSDL) [W3C07b] for service description. [4]
- *Simple Object Access Protocol* (SOAP) [W3C07a] for service consumption.

Revisiting the scenario above the service provider sends a machine readable description of the interfaces of its web services to the service broker. It is important to provide a machine readable interface description because web services are designed for Machine-to-Machine communications. Therefore the participating systems have to "grasp" the capabilities and interfaces of each other.

In the information technology domain *Interface Definition Languages* (IDL) are utilized to describe interfaces in a machine readable format. For web services such an IDL has been developed and is called WSDL. WSDL is XML-based and comprise the name, the location and additional attributes of its associated web service. Most importantly the WSDL file describes the functions of the web service and the respective parameters of the functions. When an application consumes a web service it invokes one or more of the functions described in the WSDL file of the web service.

As the WSDL file of a web service thoroughly describes the web service, this WSDL file is sent by the service provider to the service broker. The service broker integrates the WSDL in its registry. The structure of such an registry is described in the UDDI standard to ensure compatibility between different implementations of consumers, providers and brokers.

Again revisiting the above scenario the service consumer may need to use a web service in some point of time. It may be that the service consumer does not know where the requested web service is located. In this case the service consumer can query the registry of the service broker. If a matching web service is known to the service broker the associated WSDL file is transmitted to the service consumer. From the received WSDL file the consumer can extract the location and the exact specification of the interface of the service provider. With this information the service consumer is able to invoke the web service functions.

A web service can be described as a bunch of functions which can be invoked from a remote system. It therefore shows analogies to *Remote Procedure Calls* (RPC) [TS01, pp. 68–79]. The difference is that the exchange of data happens via XML files, so called SOAP messages. The utilization of XML ensures loose coupling between the involved applications and platform independency.

The process of invoking remote functions of web services, as described above, is called consuming a web service. Figure 3.3 shows a simple example of a web service consumption. The service provider comprise a web service with only one function named `getTime()`. This function delivers the current time as return value. Parts of the according WSDL file is shown in Listing 3.1.

Data exchange with web services is done SOAP. The SOAP specification is a recommendation [W3C07a] of the *World Wide Web Consortium* (W3C)[5] and is a widely deployed Machine-to-Machine communication technology.

[4] WSDL is usually pronounced "wiz-dull".
[5] `http://www.w3.org/`

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="TimeProvider"
...
             <element name="getTime">
             </element>

             <element name="TimeResponse">
                <complexType>
                   <sequence>
                      <element name="Time" type="xsd:time"/>
                   </sequence>
                </complexType>
             </element>
...
</wsdl:definitions>
```

Listing 3.1: Part of a simple WSDL file. The WSDL describes a web service which provides the function `getTime()` to retrieve the current time. The current time is encapsulated in a `TimeResponse` SOAP message.

The web service is invoked by exchanging SOAP messages. A SOAP message is a XML document containing an envelope tag, which comprise a body section and optionally a header section. The structure of a SOAP message is shown in Listing 3.2.

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope>
   ...
   <soap:Header>
     ...
   </soap:Header>
   <soap:Body>
     ...
   </soap:Body>
</soap:Envelope>
```

Listing 3.2: Structure of a SOAP message. The message consists of an *envelope*, which comprise a *body* and optionally a *header*. The actual data is enclosed in the body section.

In the envelope XML namespaces can be defined which will be valid for the whole SOAP message. The header section may provide some meta-information about type and location of the service. The body section contains the actual user data. The process of consuming the time web service presented above is shown in Figure 3.3. The figure also includes the involved SOAP request and response messages.

When a client wants to invoke a webservice it constructs a SOAP message, which specifies the function it wants to invoke and the parameters with which the function has to be invoked. After construction of the SOAP message the client transmits the message to the service provider. Usually HTTP (*Hyper Text Transfer Protocol*) [BLFF96] is used as transport protocol but other protocols, like IP (*Internet Protocol*) [Pos81a], are also possible. The web service provider receives the SOAP message, parses it and afterwards executes the associated *local* function. Emphasize is on local function, because after all a web service is just a publicly available interface to applications located on the service provider.

The return value of the local function is stored in another SOAP message and sent back to the service consumer. The consumer parses the SOAP response message, extracts the needed

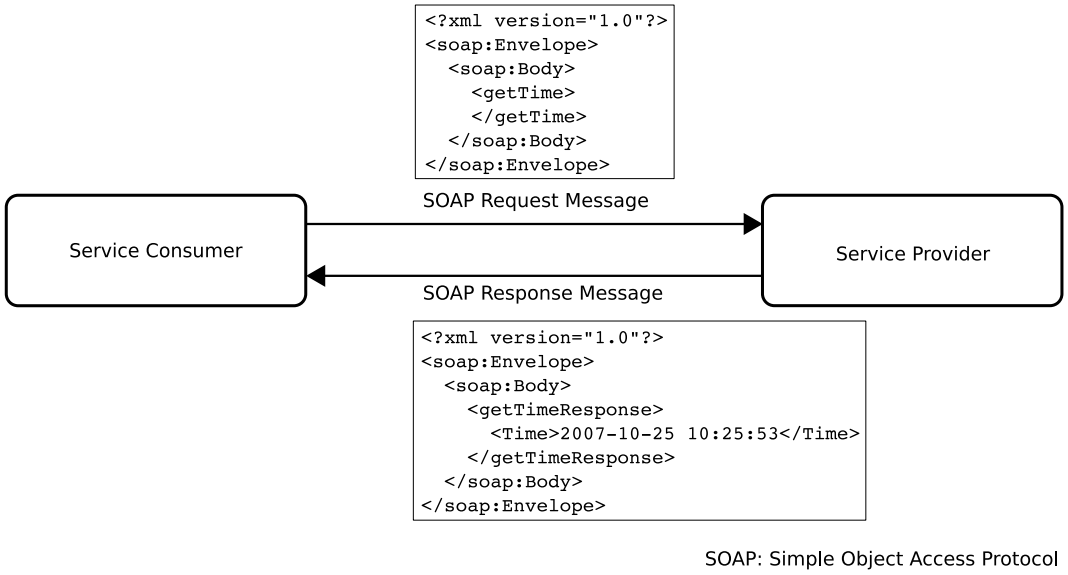

Figure 3.3: Consummation of a web service. The *service consumer* sends a *request message*, which asks for the result of the `getTime()` function, to the *service provider*. The service provider invokes the requested function and transmits the result in a *response message* back to the service consumer.

values and can continue to process the data locally. [6]

Web services are designed for Machine-to-Machine interaction, therefore applications have to be able to process and understand the interface of the web service. In web services WSDL is used to ensure this capabilities. Furthermore the machine readability allows to create tools which support the developers creating service providers and service consumers. These tools can parse a WSDL file and generate source code which eases the otherwise tedious task of creating and parsing SOAP messages. The WSDL transformation tools are typically part of a SOAP stack distribution, see Section 3.5 for a comparison of various available SOAP stacks.

If the WSDL transformation tool generates source code in an object oriented programming language, the SOAP messages are usually mapped to helper classes. In the above time web service example the request SOAP message would be mapped to a class with no member variables, because the time function does not have any parameters. The response SOAP message would be mapped to a class with a member variable of a type which represents a point in time. Which type would be used depends on the used programming language and if the language library comprise a class which can represent time (in Java the class `Calendar` in package `java.util` could be used).

To ease the invocation of web service functions the WSDL transformation tools typically also

[6] Note that there are two additional styles of web service invocation: *XML Remote Procedure Call* (XML-RPC) and *RESTful web services*. But as they are not fundamentally different and the SOAP style is used in this work, these two styles are not further mentioned.

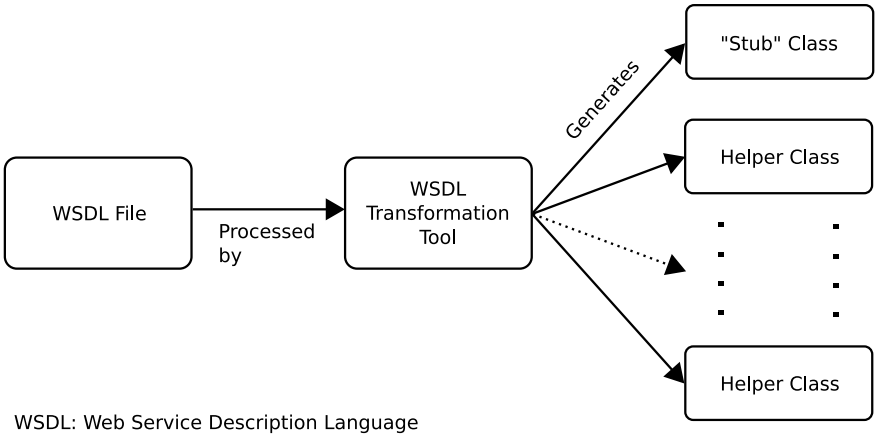

Figure 3.4: Schematic work flow of a *WSDL transformation tool*. The WSDL transformation tool parses a *WSDL file* and generates every SOAP message described in the WSDL file a *helper class*. Additionally a *stub class* is generated which eases the invocation of the web service.

generate a "stub" class. The stub represents the web service as a whole and has member functions, which map to the functions of the web service. Figure 3.4 shows the process of generating classes in an object oriented language classes by parsing a WSDL file.

If using a WSDL transformation tool the invocation of a web service boils down to: Instantiating an object of a class which represents the designated SOAP message. The members of the object (which map to the parameters of the function) have to be altered as needed. The actual web service function is invoked by calling a member function of the stub class. This member function needs as parameter the previously prepared object. The return value is again an object, whose corresponding class represents the response SOAP message. A stripped down Java code of this procedure is shown in Listing 3.3.

```
TimeProvider stub = new TimeProvider();

TimeResponse response = stub.getTime();

System.out.println(response.time);
```

Listing 3.3: Consuming a web service in Java utilizing a SOAP stack. The class `TimeProvider` is a stub class generated by a WSDL transformation tool. The member functions of TimeProvider are mapped to the functions of the associated web service. Invoking a member function actually starts the consummation of the web service.

The transformation of objects into XML styled text – typically called serialization – the transmission of the SOAP messages, receiving the response SOAP message, parsing of XML and the conversion into objects again is handled by the generated classes and the SOAP stack the WSDL transformation tool belongs to. The developer can use the instantiated objects and does not have to care about XML files and other underlying technologies.

The overhead generated by web services is rather large because of serialization and parsing of XML files. Nevertheless they are more and more deployed on embedded systems

too [MSMV06]. Also web services are used in building automation systems on gateway nodes to provide an interface to other information technology systems. Such an gateway node is discussed in more detail in Section 3.5. Reasons for the adoption of web services are the platform independency, the decoupling of applications and – last but not least – the hype which happened around web services in the last years. Furthermore the utilization of XML allows to represent characteristics of building automation like hierarchies and interaction between nodes.

3.1.2 Open Building Information Exchange

As mentioned in the previous section web services have grown to be the favored interface for building automation systems in the last years. Although the handling, transmission and parsing of XML requires some amount of computing power and bandwidth, the openness and independency of XML outweighs the disadvantages. For nearly every building automation technology nodes exist, which provide a web service (and typically additional functionality like a web browser interface). These web services allow to retrieve data from the nodes present in the building automation system.

A problem with existing nodes is that the provided web services are incompatible to each other. Some of the web services even just map the messages of the building automation bus to XML. This approach leads to technology and platform dependent web services which contradicts the purpose of web services. Example of such a web service is BACnet/WS [ASH06] which is even an ANSI[7] (*American National Standards Institute*) standard, but is nevertheless only applicable to BACnet based building automation system. A second example is the web service provided by the i.LON 100 Internet Server[8]. Its web service allows to retrieve the value of *network variables* (the basic means of communication in a LonWorks based network), but this happens in a technology dependent way: The fields of a *network variable update message* are just wrapped into XML tags.

An approach to solve this kind of diversity is to define an open standard, which is applicable to all – and therefore independent from – building automation technologies. Such a proposed standard is *Open Building Information Xchange* (oBIX) [OAS06]. In the specification document oBIX is defined as: *"oBIX is designed to provide access to the embedded software systems which sense and control the world around us. Historically integrating to these systems required custom low level protocols, often custom physical network interfaces. But now the rapid increase in ubiquitous networking and the availability of powerful microprocessors for low cost embedded devices is weaving these systems into the very fabric of the Internet. Generically the term M2M for Machine-to-Machine describes the transformation occurring in this space because it opens a new chapter in the development of the Web – machines autonomously communicating with each other. The oBIX specification lays the groundwork building this M2M Web using standard, enterprise friendly technologies like XML, HTTP, and URIs."*

As stated above, the purpose of oBIX is to define a common way how to describe building automation data in XML. oBIX does not dictate a specialized interface for web services but basic XML elements which can be combined to represent real world data. Therefore oBIX has

[7] `http://www.ansi.org/`
[8] Product description at: `http://www.echelon.de/products/internet/ilon100.htm`

similarities to a programming language, it defines a grammar which can be used to describe more complex problems.

oBIX attempts to hide the technology dependent parts of the underlying system by mapping a foundation of building automation related concepts into XML. The basic concepts of oBIX are:

- *Objects* are XML elements. Therefore everything in an oBIX file is an object.
- *Points* are objects which represent a sensor or an actuator value.
- *Contracts* are objects which are used to define new types of objects.
- *Operations* represent functions which can be invoked.

Objects can – naturally as they are just XML elements – contain any number of sub-objects. This allows the representation of complex operations by combination of simpler sub-objects. In Listing 3.4 a time source is represented in oBIX syntax.

```
<obj name="timeSource1" href="http://firstFloor/timeSource">
  <abstime name="time" val="2007-10-25T10:25:53"/>
  <int name="calendarWeek" val="43"/>
</obj>
```

Listing 3.4: oBIX representation of a *time source*. The time source is named *timeSource1* and its location is defined by the *href* attribute. Two points, the actual time and the number of the calendar week, are part of the timeSource1 *object*.

The time source contains two points, the absolute time and the number of the calendar week. The types *abstime* and *int* – amongst others – are defined in the oBIX standard. Another fundamental part of oBIX is shown in the listing: URIs (*Uniform Resource Identifier*) [BLFM05]. An URI is a string used to identify, name and locate a resource. In oBIX URIs are identified by the keyword *href* and define the location from where the associated object can be retrieved. In Listing 3.4 the oBIX document for *timeSource1* object can be retrieved at the location specified by the URI in line one.

Contracts are objects which allow to define new types of objects. Contracts are used as templates for objects. Listing 3.5 shows the contract of an alarm source as defined in the oBIX standard. The alarm contract contains a `ref` object to reference the object from where the alarm originated. Also a point for the point in time when the alarm occured and a point to indicated if it is a critical alarm, which is true as default, are included.

Listing 3.5 also shows that the URIs, identified by the *href* attributes, do not necessarily have to be HTTP URIs. oBIX does not dictate which protocol should be used for exchanging documents, therefore the documents can be located in any place. The communicating parties just have to have a way to locate and retrieve the documents via the URI.

If an object wants to implement a contract it has to reference the contract via the *is* statement. Listing 3.6 shows the implementation of an alarm contract. The object implements the *source* and *timestamp* elements and adds another element which delivers an alarm message. The object does not define the *critical* element, therefore the standard value, as defined in the contract, is used.

```
<obj href="obix:Alarm">
  <ref name="source"/>
  <abstime name="timestamp"/>
  <bool name="critical" value="true"/>
</obj>
```

Listing 3.5: oBIX contract for an alarm. The alarm object contains objects to reference the *source* of the alarm, to define a point in time (*timestamp* when the alarm occured and to indicate if the alarm is *critical*.

```
<obj href="http://firstFloor/alarmSource" is="obix:Alarm">
  <ref name="source" val="http://firstFloor/alarmSource"/>
  <abstime name="timestamp" val="2007-10-25T10:45:53"/>
  <str name="message" val="Fire on the first floor!"/>
</obj>
```

Listing 3.6: oBIX object implementing the *alarm contract*. This object implements the alarm contract by using the *is* keyword. The implementation sets values of objects which do not have a default value in the alarm contract and adds an object for an alarm *message*.

As already seen in Listing 3.6 new objects can be created by simple adding of elements and by so called *inheriting* contracts. Inheritance is done via the *is* keyword. If an object references a contract via *is* it inherits all elements of that contract. Listing 3.7 shows how to build a contract for a alarm clock by inheriting a time source and an alarm contract.

```
<obj name="timeSourceContract" href="/contracts/timeSource">
  <abstime name="time"/>
  <int name="calendarWeek"/>
</obj>

<obj name="alarmClockContract" href="/contracts/alarmClock"
    is="/contracts/timeSource obix:Alarm">
  <ref name="source"/>
  <abstime name="timestamp"/>
  <str name="message" val="Time to get up!"/>
  <bool name="critical" val="false"/>
</obj>


<obj name="alarmClockImplementation" href="/firstFloor/alarmClock"
    is="/contracts/alarmClock">
  <ref name="source" val="http://firstFloor/alarmClock"/>
  <abstime name="timestamp" val="2007-10-25T10:55:00"/>
</obj>
```

Listing 3.7: oBIX *contract* and *implementation* of an *alarm clock*. First a *time source* contract is defined, which is inherited to create an alarm clock contract in the second step. The last step shows an example implementation of the alarm clock.

The last building block of oBIX is operations. In oBIX operations represent functions which can be invoked on objects. Operations must take one object as input parameter and return one object as result. Operations are defined by using the *op* keyword. Listing 3.8 shows the definition of a *snooze* operation, which could be used in the alarm clock described above. The operation takes an empty object (*Nil*) as input parameter and returns an integer indicating

for how many minutes the alarm clock will "snooze".

```
<op href="http://operations/snooze" in="obix:Nil" out="obix:int">
```

Listing 3.8: Definition of an oBIX operation. An operation is defined by the *op* keyword and every operation has to have an input parameter (*in*) and a return value (*out*).

In its current form oBIX only defines a way in which building automation data can be represented in XML elements. It does not hinder to do this in a vendor or technology dependent way. For example by defining contract attributes which can only be provided by nodes from one particular vendor or nodes of one particular technology. This issue is well known and therefore the oBIX technical committee will focus its work on standardizing contracts for commonly used nodes. Very similar to the work the *LonMark Organization*[9] has done in standardizing types and functionality of LonWorks building automation nodes.

3.2 Introduction to LonWorks

oBIX is a relatively new standard and has not been implemented in building automation systems yet. Therefore an available system which provides an interface to access building automation data had to be found. The i.LON 100 (see Section 3.3) is an embedded system which provides a web service to interface with LonWorks [CEA02] based building automation systems. Section 3.3 describes the functionality provided by the i.LON 100, before diving into the functionality, the following sections will give a short introduction into the concepts of the LonWorks automation bus.

The i.LON 100 and the LonWorks building automation bus were chosen because of the availability of accompanying tools, like *LonMaker*[10], the relative "cheapness" of LON components, the already available knowledge and expertise on LON technology at the *Institute of Computer Technology* (ICT) and because LON components are already deployed in the *Centre of Excellence for Fieldbus Systems*[11] at the ICT.

3.2.1 Network Variables

On the application layer communication in LonWorks bus systems is done by utilization of *Network Variables* (NV) [Die99, pp. 180–183]. Every application on LonWorks-enabled nodes can define a set of input and output network variables. A output network variable can be "connected" to multiple input network variables. Also a input variable can be "connected" to multiple output network variables. If an output variable on a node is altered (for example by an application or a sensor value update) every connected input variable, which may or may not be on other nodes, will be notified and updated. The applications comprising the input variables can react to changes and process them accordingly.

Establishing a connection between network variables is called *binding* of network variables. Bindings are established by system integrator tools. Such a tool is the LonMaker application

[9] http://www.lonmark.org/
[10] http://www.echelon.com/Products/networktools/lonmaker/
[11] http://www.ict.tuwien.ac.at/komzent/

which has been used in this master thesis. LonMaker can connect to a LonWorks bus system and present the available nodes on the bus in a schematic. Figure 3.5 shows a LonMaker schematic with two devices, represented by the green rectangles at the bottom of the drawing, connected to a common bus.

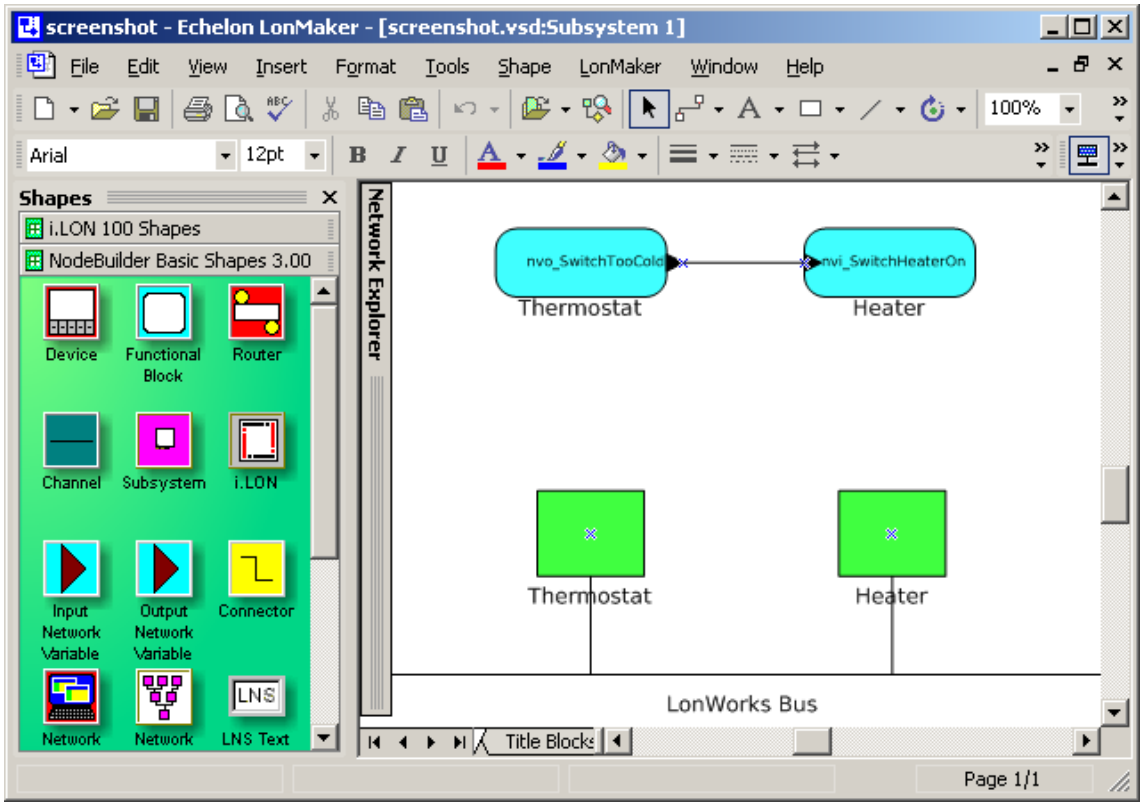

Figure 3.5: LonMaker schematic of a simple LonWorks bus with two nodes attached to it. The nodes are represented by the green rectangles at the bottom of the schematic. The turquoise rectangles with the rounded corners represent the functional blocks of the nodes.

3.2.2 Functional Blocks

Each node on the bus provide a specific functionality, which is presented as *Functional Block* in LonMaker. Two functional blocks, represented as turquoise rectangles with rounded corners, are shown in Figure 3.5. In the figure the functional blocks have been placed right above the associated devices, but this is not necessary at all. Functional blocks and devices can be placed anywhere in a LonMaker schematic.

Functional blocks comprise network variables. By binding output network variables to input network variables building automation functionality can be provided. For instance in a typical building every room comprises a thermostat node measuring the temperature in the room and a heater control node, which can switch on and off the heaters in the room. The thermostat

node probably[12] comprise a binary output network variable which indicates if the room is too cold or too hot. Likewise the heater control node probably comprise a binary input network variable which controls if the heater is switched on or off. In the LonMaker schematic the system integrator can *bind* the output network variable of the thermostat node to the input network variable of the heater control node as shown in Figure 3.5. The binding ensures that the heaters will be switched on if the thermostat node deems the room to be too cold and also that the heater will be switched off if the room reaches the desired temperature again.

The binding has to be done by special tools because the nodes on the LonWorks bus have to be informed about the binding. The system integrator tool informs the node which of its network variables is involved in a binding and the address and network variable of the other node which is involved in the binding. Therefore the node with the output network variable knows to which nodes it has to send network variable updates if the network variable has been changed. Also the node with the input network variable knows from which node it has to expect network variable updates.

Naturally not every network variable can establish a binding to another. It would make little sense to connect a output network variable of a thermostat node which carries the temperature as a floating point number[13] with a input network variable of a switching node, which comprises electrical-controlled switches to turn on and off eg. lights. The application reading the input variable on the switching node would expect a boolean value for either setting the switch on or off and would not be able to interpret a floating point value.

3.2.3 Functional Profiles

A binding between incompatible network variable as described above may lead to failure (of parts) of the building automation system. To avoid such failures system integrator tools only allow bindings between network variables which are compatible to each other. To restrict bindings to only compatible network variables, common types have to be defined. This has led to the standardization of network variable types. The standardization of LonWorks network variables is done by *LonMark International*[14]. It is an organization consisting of manufacturers, distributors, system integrators and end users of LonWorks based products. LonMark is producing a set of guidelines describing types of network variables[15] and types of *functional profiles*[16].

The network variables described in the LonMark guidelines are named *Standard Network Variable Types* (SNVT[17]) [Ech99, p. 4-4]. SNVTs have been created for a great variety of physical values and application scenarios [Die99, p. 272]. Examples are SNVT_lux (used to represent the sensor value of a light sensor), SNVT_temp (used to represent a temperature) and SNVT_switch (used to represent the state of a switch, eg. ON or OFF). In addition to network variable types also valid value ranges have been defined for SNVTs. For instance

[12] Note that this is a simplified example. Typical LonWorks-enabled thermostats provide much more functionality than a simple on and off network variable.

[13] Don't be confused. It is entirely possible and even likely that a thermostat node has (amongst others) a binary output network variable to control heaters and a output network variable to deliver the room temperature.

[14] `http://www.lonmark.org/`

[15] `http://types.lonmark.org`

[16] `http://www.lonmark.org/technical_resources/guidelines/functional_profiles.shtml`

[17] Typically pronounced "snivets"

network variables of type SNVT_switch can have the value *100.0 1* or *0.0 0*, representing the ON respectively the OFF state of the switch. As this network variables are standardized, they can be exchanged and processed by applications located on different nodes. Therefore ensuring the correct cooperation of applications on different and distributed LonWorks nodes.

To push the interoperability of LonWorks nodes even further *functional profiles* have been defined by LonMark. Functional profiles describe the interface of common automation applications. For instance functional profiles exists for light sensors, elevator control nodes, pump controllers, valve positioner and many more. The functional profiles specifications can be downloaded from the LonMark website.

Each of these functional profiles describe a minimum set of network variables a node with a specific function has to implement. The functional profiles have been designed to be interoperable. For instance the switch profile[18] comprise an output network variable of type SNVT_switch. The lamp actuator profile[19] comprise an input network variable also of type SNVT_switch. If a binding is established between these two network variables a user can switch on and off the light by pushing the switch. The *switch* and the *lamp actuator* functional profile as specified by LonMark are shown in Figure 3.6. The figure also shows that the node may implement more network variables than described in the associated profile.

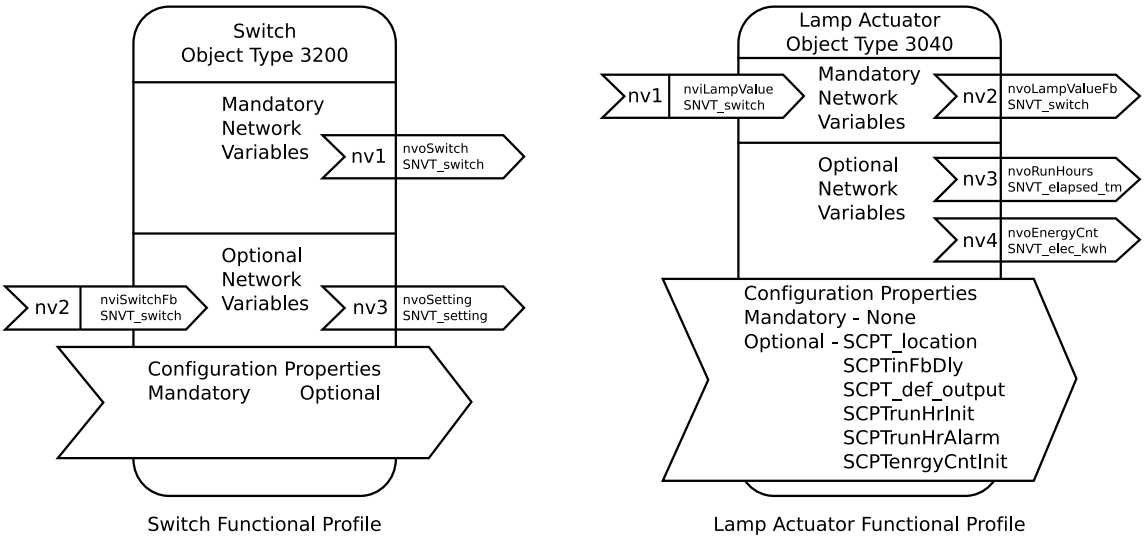

Figure 3.6: Switch and lamp actuator functional profile. Mandatory and optional network variables are listed with name and type.

Turning on and off a light by pushing a switch may not sound revolutionary, but some nodes are providing much more complex applications than the switch in the scenario above. Creating an easy way for these applications to cooperate – and this is exactly what network variables do – may indeed revolutionize the way buildings are built, managed and perceived.

[18] `http://www.lonmark.org/technical_resources/guidelines/docs/profiles/3200_10.PDF`
[19] `http://www.lonmark.org/technical_resources/guidelines/docs/profiles/3040_10.PDF`

3.3 i.LON 100 Internet Server

One goal of this master thesis is to develop an easy to use interface to building automation data. To accomplish this goal a way to interact with building automation systems had to be found. For LonWorks based building automation systems the i.LON 100 Internet Server[20] [Ech06b] provides a web service based interface, which allows to read and write building automation data.

3.3.1 Functionality

This section will give a brief overview of some of the functionality provided by the i.LON 100. The parts of functionality, which are used thoroughly in this work, will be explained in more detail in the following sections.

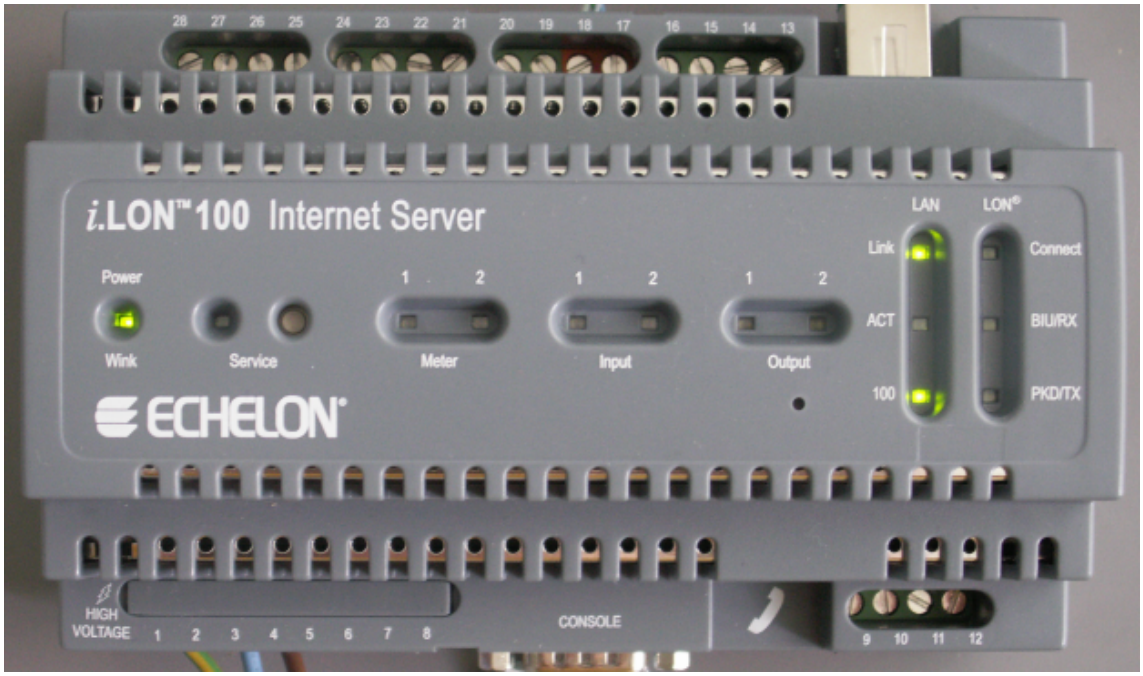

Figure 3.7: Picture of an i.LON 100.

The i.LON 100 is an embedded system which provides observation, maintenance and management functionality for building automation systems. Supported building automation buses are LonWorks [CEA02], Modbus [MI06] and M-Bus [EN05]. Non-building automation interfaces are an Ethernet port, an optional internal modem and the possibility to connect an external GSM/GPRS modem [Ech06a, pp. 4–22]. The latter two interfaces allow the i.LON 100 to initiate dial-up connections in cases where a always-online mode is not possible.

Picture 3.7 shows the forefront of an i.LON 100. On the left the power LED indicates the status of the i.LON 100. On the right side lights indicate status and activity on the attached

[20] Product description at: `http://www.echelon.de/products/internet/ilon100.htm`

LAN (*Local Area Network* – in this case Ethernet) and LON interfaces. At the top and the bottom of the picture the available connectors are visible.

Functionality provided by an i.LON 100 include [Ech06d, p. 3]:

- *Scheduling* allows the user to define tasks, which will be performed at particular date and time.
- *Clock Synchronization* is possible to ensure the scheduled tasks will be performed at the correct time. The *Network Time Protocol* (NTP) [Mil92] is used for clock synchronization.
- *Data Logging* records selected values for further analysis and trend evaluation
- *Alarming* enables the i.LON 100 to notify other systems when a predefined alarm condition occurs. The alarm can be signalized by either sending an E-Mail or a message on the LonWorks bus.
- A *Data Server* allows access to data regardless on which building automation bus the data is located.
- A *Web Service* allows to influence applications like alarming, data logging ... and enables remote applications to read and alter data available on the building automation bus by accessing the data server.

For administration of the provided functionality the i.LON 100 provides a web browser interface. The web browser interface can be used to set the IP (*Internet Protocol*) address, to define NTP (*Network Time Protocol*) servers for clock synchronization and SMTP (*Simple Mail Transfer Protocol*) [Kle01] servers for sending mail. Furthermore the management of scheduled tasks, data logs and alarm conditions is possible. The status of the i.LON 100 and the attached building automation bus can be observed. Optionally available is a routing functionality: Two or more i.LON 100 can use the attached Ethernet network as backbone to allow the exchange of LonWorks messages between not interconnected LonWorks network segments.

While the above mentioned features are mainly useful for system integrators the i.LON 100 additionally allows to upload custom web sites for the end user. These web sites enable the user to interact with the connected building automation nodes. For example it is possible to switch on and off the light or to change the temperature in some rooms. This enables the caretaker of the building to supervise and influence the automation system from an ordinary personal computer if a web browser is installed.

To make interaction between web browser and building automation system possible some proprietary technologies are used. The web sites have to be created with the *i.LON Vision*[21] tool, which is an extension to *Macromedia Contribute*[22], an application for web designers. In the web sites generated by the i.LON Vision tool proprietary HTML (*Hypertext Markup Language*) tags are included which enables the i.LON 100 web server to inform building automation nodes about relevant user interactions.

[21] `http://www.echelon.com/products/cis/`
[22] `http://www.adobe.com/products/contribute/`

Besides a web server the i.LON 100 also provides other meanings of access. A *File Transfer Protocol* (FTP) [PR85] server allows to upload and download files from the flash disk of the i.LON 100. A Telnet server provides a command line interface to the i.LON 100. From this command line a small set of maintenance programs can be invoked.

3.3.2 Data Points

As briefly mentioned in the last section the i.LON 100 comprise a *data server* which provides means to access building automation data. The data server acts as an abstraction layer between applications (Scheduling, Alarming, ...) and the supported building automation systems (LonWorks, Mod-Bus, M-Bus). Figure 3.8 shows this concept.

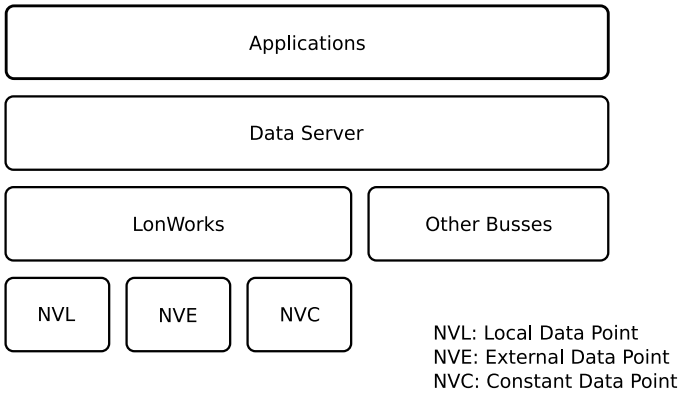

Figure 3.8: *Data server* acting as abstraction layer between *applications* and different data point types.

The abstraction of automation bus specific properties is done by representing data values on the bus by entities called *data points*. Applications only operate on data points regardless, on which bus the associated data is located. Well, at least that is the way the data server is promoted. Unfortunately the abstraction is not done thoroughly because an application which wants to read or alter a data point has to specify which type the data point is. Even worse for LonWorks data three data point types exist. If an application wants to access such a LonWorks based data point it has to know the type of the data point.

The types of LonWorks based data points are *local data points* (NVL), *external data points* (NVE) and *constant data points* (NVC). The difference of the data point type is how they correspond to network variables (NV) in the LonWorks network. Figure 3.9 shows the three different types of data points and how they correspond to network variables. Each of the LonWorks based types is discussed in a separate section below. Beside the LonWorks based data points there also exist *M-Bus data points* and *Modbus data points*, which are not used in this work and therefore not described in more detail.

Data points are representations of data available on an underlying building automation bus. Therefore every[23] data point has to be assigned to a data value on a connected automation bus. In LonWorks based systems such a data value is a network variable.

[23] NVC data points are the infamous exception of this rule. They are discussed in more detail in Section 3.3.2.3.

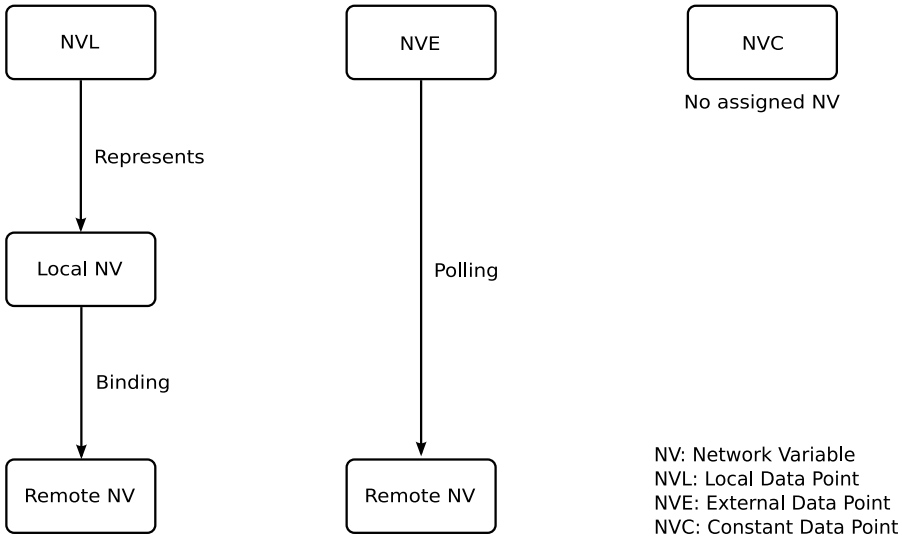

Figure 3.9: Comparison of *data point* types regarding the way *network variables* can be assigned to them. *NVC* has no assigned network variable. *NVE* retrieves the value of the assigned network variable by polling. *NVL* is a representation of a local network variable, which can be bound to other network variables on the LonWorks bus.

The creation of data points and the assignment to network variables has to be done by tools which are aware of the connected building automation buses. Also the tool has to know which nodes are located on the building automation bus and what network variable types the nodes comprise. The *i.LON 100 Configuration Utility* [Ech05] is such a tool and is delivered along side with every i.LON 100. It is a plug-in for LonMaker and can be launched by right clicking on the i.LON 100 device in a LonMaker schematic and choosing *Configure*. The Configuration Utility and the web browser interface of the i.LON 100 provide in some parts overlapping functionality. But while the web browser interface only provide functionality to view and configure data points, the Configuration Utility also provide means to create data points.

The procedures to create a new data points are different for every type of data point. In the sections below the three types (NVE, NVL, NVC) of LonWorks based data points and the procedures to create these types of data point are discussed in more detail.

3.3.2.1 External Data Points (NVE)

NVEs are data points whose assigned network variables are on remote nodes, as shown in Figure 3.9. These nodes have to be connected to the LonWorks bus and have to be reachable bus by the i.LON 100. If an application queries the value of a NVE data point, the i.LON 100 data server requests the associated network variable from the remote host and delivers the value as NVE to the application. If an application wants to change the value of a NVE data point, the data server will send a network variable update message to the remote node.

When utilizing NVE data points no binding is needed, which allows easy creation of NVE data points, as the only information needed for creation of a NVE data point is its assigned network variable. The network variable provides all information (type, input/output ...) to create the NVE. No further tasks, like binding, are needed afterwards.

The downside of the missing binding is that there is no guaranteed update of the NVE if the assigned network variable changes. Therefore the data server has to poll the assigned network variable in fixed time intervals. The i.LON 100 comprise two configuration properties influencing this polling intervals.

The first property is named *Poll Interval*, which is a property of every data point and can therefore be changed individually. The default value is 120 seconds. If a data point is not read or write to in this time span, the data server will trigger a query on the LonWorks bus to update the NVE data point to the current value of the assigned network variable. The value of the poll interval can be changed in the web browser interface of the i.LON 100 and in the i.LON 100 Configuration Utility.

The second property is named *Data Point Life Time* in the i.LON 100 Configuration Utility and *Max Age* in the web browser interface of the i.LON 100. It is a global configuration property of the i.LON 100 and applies therefore to all NVE data points. The default value is zero, which means the configuration property has no effect. If the value is altered to a value greater than zero the value will be interpreted as a time span in seconds by the i.LON 100. For example if the NVE life time is set to five and an application queries the value of a NVE data point, the data server compares the current time with a time stamp saved at the last access of the network variable assigned to the NVE data point. If the last access of the assigned network variable was within the last five seconds the data server will return the currently saved value of the NVE data point and will not query the assigned network variable. If the last access of the assigned network variable was not within the last five seconds, the data server will query the assigned network variable and will return the new value to the requesting application.

NVE data points can be created with the i.LON 100 Configuration Utility. For every available network variable in the LonMaker schematic a NVE data point can be created. Applications can use the NVE data points immediately after they have been created successfully. No further configuration tasks are necessary.

The *i.LON 100 e3 User's Guide* mentions another way [Ech06d, p. 69] to create NVE data points by modifying the configuration file of the i.LON 100 directly. The user guide refers to the *i.LON 100 e3 Programmer's Reference* [Ech06c] for detailed instructions, but the programmer's reference only addresses NVL and NVC data points. It does not contain any information regarding NVE data points, so the mentioned instructions to create NVE data points by modifying the configuration file is missing. However while analyzing the network traffic of the i.LON 100 Configuration Utility with a network analyzer[24] it was discovered that the Configuration Utility uses the web service of the i.LON 100 to create NVE data points. The name of the consumed web service function is `driverNVESet()`, which is regrettably not documented by Echelon. An attempt was started to use the `driverNVESet()` regardless the missing documentation by analyzing the protocol used by the i.LON 100 Configuration Utility to learn more about the correct usage of the `driverNVESet()` function.

[24]The used tool was Wireshark. It can be downloaded from `http://www.wireshark.org/`.

In the course of analyzing the protocol it was discovered that the Configuration Utility still uses the older version 1.1 of the i.LON 100 web service. The latest revision of the web service is version 3.0, which is used in this master thesis. Although the `driverNVESet()` functions exist in both versions their parameters are quite different. After this discovery the decision was taken to not further work on the attempt to utilize the `driverNVESet()` function. The reasons for the decision were:

- The parameters of the function `driverNVESet()` in version 1.0 and version 3.0 of the i.LON 100 web service differ in structure, naming and the number of parameters. Probably it would be possible to learn about the meaning of the parameters by analyzing the version 1.1 protocol flow, but this would be a very time consuming process.
- By analyzing the version 1.0 protocol flow it can not be determined which parameters are mandatory, which are optional and which parameters depend on other parameters. This knowledge could be only established by even more time consuming trial & error procedures.
- It would be possible to switch to version 1.0 of the i.LON 100 web service. This would have the advantage to use the same version of `driverNVESet()` as the Configuration Utility. Therefore the knowledge established by analyzing the protocol could be applied easier. But a lot of software had already been written to consume version 3.0 of the web service. This software would have to be discarded in great parts if it would be decided to switch to version 1.1 of the web service.
- The last and very crucial point is that the parameters of `driverNVESet()` comprise some very low level information. For example the *Neuron ID* [Die99, p. 78] of the node, where the assigned network variable is located, is required. Furthermore some other numerical values like `nvTypeInfo`, `nvIndex` and `nvSelector` are also included in an `driverNVESet()` request message. Most probably these values transport information on the assigned network variable (this is an assumption and is not verified as `driverNVESet()` has not been used in this work). The user of `driverNVESet()` has to provide all this pretty low level information on the nodes and network variables. To get this information a tool is needed which is able to extract the information from a LonWorks network, for example LonMaker or other system integrator tools. Therefore, if a user has to use a LonWorks-enabled tool to extract this information and provide it to `driverNVESet()`, the user can just as easily use the LonWorks-enabled tool to create NVE or NVL data points. In the end this may work out to be much more comfortable for the user.

3.3.2.2 Local Data Points (NVL)

NVLs are data points whose assigned network variables are located on the i.LON 100 itself. The i.LON 100 provides per default a set of network variables to interact with the provided functionality, like alarming and scheduling. For each of these default network variables a NVL data point exists.

Most LonWorks enabled nodes provide a specific and not changeable functionality. Therefore the nodes also provide a fixed set of network variables. These network variables are usually called static network variables. However some LonWorks enabled nodes support the dynamic

creation [Ech00] of network variables, which are named *dynamic network variables*. After creating dynamic network variables they can be used just like any other network variable. To create dynamic network variables a LonWorks enabled tool – such as LonMaker – has to be used as the nodes have to be notified about the network variables the user wants to create on them.

Application scenarios for dynamic network variables are large monitoring and logging devices. With dynamic network variables these devices are able to monitor and log any number of network variables. For every network variable, which should be logged, a new dynamic network variable can be created. When the node application is notified about a new dynamic network variable, it can automatically assign its logging mechanisms to the new network variable. This allows to log every network variable on the network which is bound to the newly created dynamic network variable.

The i.LON 100 also supports the creation of dynamic network variables. For every created dynamic network variable also a NVL data point will be created. The assigned network variable of the NVL data point is the newly created dynamic network variable, as shown in Figure 3.9. The NVL data point can be used immediately after the creation of the dynamic network variable. But as long as the dynamic network variable is not bound to any other network variable the NVL data point will not provide any meaningful values on read attempts and write attempts will have no effect on other network variables. Therefore to fully integrate NVL data points into a LonWorks based system, the assigned network variables have to be bound to other network variables.

The necessity of binding network variables which are assigned to NVL data points requires more work in setting up data points than in a NVE data points only scenario. However binding eliminates the need for polling network variables as it is needed for NVE data points, because the dynamic network variable – and therefore the NVL data point – will be updated, when the bound network variable changes.

Similarly to NVE data points a *Data Point Life Time* property exists for NVL data points. The life time property of NVL data points can be changed by utilization of the i.LON 100 Configuration Utility. The default value of the property is zero. The property indicates the maximum time in seconds a data point value is valid after its last update. If this time passes the value of the data point will be refreshed by querying the assigned network variable. If the life time property is zero no such polling will happen. For NVL data points polling is not needed as the value of NVL data points is automatically updated every time the assigned dynamic network variable changes.

3.3.2.3 Constant Data Points (NVC)

NVCs are data points which present a (somewhat) constant value. NVC data points have no assigned network variable (as shown in Figure 3.9), which means NVC data points do not change if a specific network variable changes. Likewise no network variable will be changed if a NVC data point is written to. But NVC data points are not entirely constant, because a write attempt is able to change the value of a NVC data point.

Usually NVC data points are used as reference value. A user can define a temperature set point and store it in a NVC data point. The control application will compare the current temperature with the value stored in the NVC data point and can react accordingly. As the NVC data point is not really constant the reference value can be adjusted by the user.

3.3.3 Evaluation of Data Point Types

To choose a data point type for further usage in this work the types have to be compared to each other and evaluated if their functionality suffices the requirements. These requirements have to be defined before a comparison of data point types can happen.

One hard requirement is that the selected data point type allows reading and altering the values of network variables, to enable interaction with other LonWorks nodes. Therefore NVC data points are not applicable and the only remaining candidates are NVL and NVE data points.

Further requirements are related to the management of data points related. What steps has a user to do for creating a data point and how difficult are these steps? After all, people working with media façades probably have no or limited knowledge of LonWorks or other building automation systems. In this category NVE data points have a clear advantage because no binding of network variables is required. Also the steps in LonMaker which have to be taken to generate NVE data points are much easier than the steps involved in creating dynamic network variables. Also interesting from the user point of view is, if it is possible to create more than one data point in one step. This can be done for both NVE and NVL data points.

NVE data points would have a tremendous advantage if it would be possible to use the web service function `driverNVESet()`[25] to create NVE data points. Because this would allow to develop applications, which can configure the i.LON 100 and its data points dynamically according to the current usage scenario. If an application wants to switch on and off the lights in rooms it can create the needed NVE data points. If in a later version the application also wants to alter the position of sunblinds it just have to create the additionally needed data points by invoking the web service function. This would immensely reduce the burden of the user to have experience with building automation bus systems. Regrettably because of missing documentation this is not possible.

For applications using the i.LON 100 to interface with building automation it would be also interesting to retrieve the available data points from the i.LON 100 in an automated way. This is possible for both NVE and NVL data point types because data points are registered in configuration files [Ech06c, pp. 4-4–4-8] which can be retrieved from the i.LON 100 by utilizing its FTP server. Listing 3.9 shows such an entry of a configuration file. For NVE data points the path to the file is `/config/software/dataServer/dp_NVE.xml`. The name of the file for NVL data points is `dp_NVL.xml` and can be found in the same directory. Entries for NVL and NVE data points are identically structured. They contain `UCPTindex` and `UCPTpointName` tags which are used as identifiers for data points and therefore have to be unique. `UCPTlocation` is initially set to a string which represents the location of the data point on the i.LON 100. The `UCPTformatDescription` tag describes the type of the assigned network variable and `UCPTdirection` states if the assigned network variable is an input or an output. For further information on the properties of data points and the format of the data point files see [Ech06c, pp. 4-4–4-8]

However evaluating the config file for NVE data points is easier as no pre-installed NVE data points exist on the i.LON 100. Therefore the application can assume that every entry in the NVE data points file is a data point which has been created specifically for this application. The user just have to ensure that only NVE data points exist, which should be used by the

[25] See Section 3.3.2.1 why it isn't possible.

```
<DP>
  <UCPTindex>367</UCPTindex>
  <UCPTpointName>NVL_outmod2_NV008</UCPTpointName>
  <UCPTlocation>iLON100/NVL/dynamic</UCPTlocation>
  <UCPTdescription/>
  <UCPTformatDescription>SNVT_switch</UCPTformatDescription>
  <UCPTdpSize>2</UCPTdpSize>
  <UCPTbaseType>BT_STRUCT</UCPTbaseType>
  <UCPTunit/>
  <SCPTmaxSendTime>0.0</SCPTmaxSendTime>
  <SCPTminSendTime>0.0</SCPTminSendTime>
  <SCPTmaxRcvTime>0.0</SCPTmaxRcvTime>
  <UCPTsettings>0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1</UCPTsettings>
  <UCPTdirection>DIR_IN</UCPTdirection>
</DP>
```

Listing 3.9: A data point entry in an i.LON 100 configuration file is enclosed by `DP` tags. *NVE* and *NVL* data point configuration entries are identically structured. The values in the `UCPTindex` and `UCPTpointName` tags are identifiers for the data point.

application, which is relatively easy because every NVE data point is explicitly created by the user. On the other hand hundreds of pre-installed NVL data points – to interact with the functionality of the i.LON 100 – exist. Applications trying to retrieve only the data points which are useful to them, have to find a way to differentiate between pre-installed and user-defined NVL data points. This distinction can be based on the value of the `UCPTlocation` tag in Listing 3.9. For the pre-installed NVL data points the value refers to its assigned functionality, for instance an alarm flag, which can be set by the alarming functionality, the value is `iLON100/NVL/static/AG/AG00/AlarmFlag`. For every dynamically created NVL data point the `UCPTlocation` tag value equals `iLON100/NVL/dynamic`. Therefore the `UCPTlocation` tag allows to differentiate between pre-installed and dynamically created NVL data points.

Based on above requirements a comparison shows that NVE data points have advantages in several categories above NVL data points. NVE data points are easier to create and manage, are easier to retrieve from the i.LON 100 and are less invasive in the existing building automation system because no binding is needed. However this comes at the price of higher latency as NVE data points have to be polled explicitly. Therefore in the first design considerations it was decided to utilize NVE data points but to pay special attention to keeping the system compatible with other data point types.

Later this proved to be a wise decision. The used lamp actuator (see Chapter 6 nodes are not responding to updates of their network variables which are assigned to NVE data points if – and only if – the network variable is already bound. So if a network variable – for instance one to switch on/off the light – of the lamp actuator is already bound to some other network variable on a switch node[26], a NVE data point can no longer alter the value of the assigned network variable. If the assigned network variable is not bound, the NVE data point works as expected. Because of this misbehavior, the utilization of NVE data points was ruled out and it was decided to use NVL data points instead.

[26] This is a quite common scenario.

3.4 Accessing Building Automation Data

The i.LON 100 can act as gateway between the LonWorks building automation bus and other network technologies. This gateway functionality is implemented as a web service. The web service allows to interface with the applications of the i.LON 100. The applications provide the functionality described in Section 3.3.1. Amongst these applications are a data server, a alarm generator, a data logger and a task scheduler. For each of these applications the web service provides a set of functions which can be invoked by a client. The description of the web service can be found in the WSDL file located on the i.LON 100 at `/web/WSDL/V3.0/iLON100.wsdl.gz`. For more information about WSDL see Section 3.1.

3.4.1 i.LON 100 Web Service Overview

Every application of the i.LON 100 provides at least `List()`, `Get()`, `Set()` and `Delete()` web service functions. Some applications provide additional functions, like `Read()` and `Write()` functions [Ech06c, pp. 3-3–3-6]. The following paragraphs provide a small introduction into the usage and purpose of the web service functions. Some of the data server related functions are discussed in more details in the next sections.

The response of a `List` function of a specific application contains a list of all elements provided by the application. For instance invoking the function `AlarmGenerator_List()` returns a list of alarm generators which have been created on the i.LON 100. The response of the `List()` function contains information on the available elements, needed for further processing of the elements. The `Get()`, `Set()` and `Delete()` functions require as parameter a reference on which element the function should operate. Such a reference can be given in two ways. Either by providing an index, which is a unique integer value greater than zero, or by a name represented as string. The response of the `List()` function contains both index and name of each element.

A `Get()` function returns available configuration data for specific elements. The elements for which data should be retrieved have to be addressed explicitly when invoking a `Get()` function. Addressing can be done by either providing the index or the name of an element. As mentioned above the index and name of elements can be retrieved by the associated `List()` function. For example the `DataLogger_List()` function returns a list of initial information on available data loggers. With the `DataLogger_Get()` function additional information for specific data loggers can be retrieved. The `Get()` function is similar to the `List()` function in that it provides information on some elements, however the `Get()` function provide all available configuration information, while the `List()` function only provide some initial information on elements. The `DataServer_Get()` function is described in more detail in Section 3.4.3.1.

The `Set()` function of an application can be used to alter the configuration of an element or even to create elements. Creating elements is not possible with every `Set()` function, only for those whose application allow it. For instance the function `DataLogger_Set()` allows to create a new data logger element on the i.LON 100 [Ech06c, p. 5-11]. However the `Set()` function of the data server only allows to create NVC data points. It is not possible to create NVE or NVL data points with the `DataServer_Set()` function, because for creation of these data points – as described in Section 3.3.2 – information about the underlying network variables is needed. This would either require that the i.LON 100 has information of all available network

variables in the local LonWorks network or that the user of the `DataServer_Set()` function provides all needed information as parameter. Both options are not really feasible as either the i.LON 100 or the user would have to manage quite an amount of data, specifically low level data like numeric addresses of nodes and network variables. Therefore creating NVL and NVE data points is only possible with system integrator tools like LonMaker.

The `Delete()` functions allow to remove a element from a specific application. As an example the `AlarmGenerator_Delete()` requires the name or index of an alarm generator as parameter. If the alarm generator with this index respectively name exists on the i.LON 100, it will be removed. Unlike the `DataServer_Set()` function – which only allows to create NVC data points – the `DataServer_Delete()` function allows to delete any kind of data point.

The data server and the alarm notifier applications additionally provide `Read()` and `Write()` functions. These functions also require index or name of an element as parameter. The `AlarmNotifier_Read()` and `AlarmNotifier_Write()` functions allow to access the log files of alarm elements. The `DataServer_Read()` and `DataServer_Write()` functions allow to retrieve respectively alter the value of a data point. They are discussed in more detail in Section 3.4.3.2 and 3.4.3.3.

3.4.2 i.LON 100 SOAP Messages

Web services are consumed by exchanging SOAP messages. The structure of the SOAP messages is described by the WSDL file of the web service. The i.LON 100 SOAP messages are following some common design rules, which apply to every SOAP message exchanged with the i.LON 100 web service. These common design rules are discussed in this section.

Every request SOAP message sent by the client has to specify which function of the web service should be invoked. The message also has to include the necessary parameters for the invoked function. Listing 3.10 shows the basic format of a SOAP request message.

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope
      xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <FunctionName xmlns="http://wsdl.echelon.com/
          web_services_ns/ilon100/v3.0/message/">
      <Parameter1>Parameter1Value</Parameter1>
      <Parameter2>Parameter2Value</Parameter2>
       ...
    </FunctionName>
  </soap:Body>
</soap:Envelope>
```

Listing 3.10: A basic request SOAP message. The *envelope* contains only a *body*, the *header* is omitted. The body specifies with `FunctionName` which function should be invoked by the *service provider*. The parameters for the function are enclosed by the `FunctionName` tag.

The `soap:Envelope` section is used to define a XML namespaces which will be valid for the whole SOAP message. The optional SOAP Header section is omitted in the listing. The `soap:Body` section contains the information needed for the web service consumption. This includes the name of the function which should be invoked and the parameters which are required by this function. The tag `FunctionName` in Listing 3.10 has to be replaced with the

name of the function, which should be invoked. If the `DataLogger_List()` function should be invoked the string `FunctionName` has to be replaced with the string `DataLogger_List`. The tags `Parameter1` and `Parameter2` have to be renamed according to the required parameters of the invoked function. Similarly the Strings `Parameter1Value` and `Parameter2Value` have to be replaced with the actual parameter values.

A web service receiving a SOAP request message will parse the message and afterwards invoke the local function described in the SOAP message. After the local function has finished the return value of the local function will be transformed into a SOAP message and this SOAP message will be sent back to the consumer. The SOAP message which is sent to the consumer as answer to a web service consumption is called a SOAP response message. Similarly to the request messages the response messages of the i.LON 100 web service follow some common design rules. A basic example of an i.LON 100 web service response message is shown in Listing 3.11.

```
<?xml version="1.0" encoding="utf-8" ?>
<SOAP-ENV:Envelope
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/
      soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <p:messageProperties
        xmlns:p="http://wsdl.echelon.com/web_services_ns/
          ilon100/v3.0/message/">
      <p:UCPTtimeStamp>
        2007-08-20T09:33:29.150+02:00
      </p:UCPTtimeStamp>
      <p:UCPTuniqueId>03000012f6e2</p:UCPTuniqueId>
      <p:UCPTipAddress>192.168.1.222</p:UCPTipAddress>
      <p:UCPTport>80</p:UCPTport>
      <p:UCPTlastUpdate>2007-08-16T06:46:50Z</p:UCPTlastUpdate>
    </p:messageProperties>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <FunctionNameResponse
        xmlns="http://wsdl.echelon.com/web_services_ns/
          ilon100/v3.0/message/">
      <Response1>Response1Value</Response1>
      <Response2>Response2Value</Response2>
    </FunctionNameResponse
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope
```

Listing 3.11: A basic response SOAP message of the i.LON 100. The *envelope* contains a *header* and a *body* section. The header provides some optional information. The actual response data is inside the body section. The tag `FunctionNameResponse` declares the type of the response message and encloses the response values.

The response SOAP message of the i.LON 100 web service contains an envelope in which a XML namespace is defined. In the envelope reside a header and a body section. In Listing 3.11 the `SOAP-ENV:Header` section enlists a time stamp and some information about the i.LON 100, like the IP address and the port on which the web service is listening. The `SOAP-ENV:Body` section contains the actual response value of the invoked function. In a real world response SOAP message the tag `FunctionNameResponse` would be replaced with the name of the invoked function. If the `DataLogger_List()` function had been invoked, the string `FunctionNameResponse` in the response SOAP message would be replaced with the

string `DataLogger_ListResponse`. The tags `Response1` and `Response2` would have been renamed according the type of the return value of the invoked function. Similarly the strings `Response1Value` and `Response2Value` would have been replaced with the actual response values.

The software developed in this master thesis utilizes only the the data server related functions of the i.LON 100 web service, which will be discussed in the next section.

3.4.3 Web Service and Building Automation Data

The i.LON 100 can be used as a gateway to the attached building automation bus. This gateway functionality is provided by the web service of the i.LON 100. As described in the last sections, the web service of the i.LON 100 allows to interact with the applications of the i.LON 100. One of the applications is a *Data Server* which acts as abstraction layer for the attached building automation bus (see Section 3.3.1).

The parts of the web service which are used to interact with the data server of the i.LON 100 are utilized in this master thesis to implement an interface to building automation data. The functions of the web service related to the data server are:

- `DataServer_List()`
- `DataServer_Get()`
- `DataServer_Set()`
- `DataServer_Read()`
- `DataServer_Write()`
- `DataServer_ResetPriority()`
- `DataServer_Delete()`

These functions are documented in [Ech06c, pp. 4-8–4-31]. For the implementation of an interface to the building automation data only the `DataServer_Get()`, `DataServer_Read()` and `DataServer_Write()` functions were needed in this work. Therefore these three functions are described in more detail in the following sections.

3.4.3.1 DataServer_Get

The `DataServer_Get()` [Ech06c, pp .4-12–4-17] function of the i.LON 100 web service can be used to query information about existing data points. The information only contains information needed for operating on data points. There is no information available to which network variable the data point is assigned or where the assigned network variable is located.

Listing 3.12 shows a SOAP message requesting information about a data point. By adding more `<DP>...</DP>` entries to the request message, it is possible to query more than one data point at once. The data point in the listing is addressed via its name, it would also be possible to query it by its unique index by replacing the `UCPTpointName` tags with `UCPTindex`

```
<?xml version="1.0" ?>
<S:Envelope
    xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <DataServer_Get xmlns="http://wsdl.echelon.com/
        web_services_ns/ilon100/v3.0/message/">
      <iLONDataServer>
        <DPType>
          <UCPTname>NVL</UCPTname>
          <DP>
            <UCPTpointName>
              NVL_outmod2_NVI13
            </UCPTpointName>
          </DP>
        </DPType>
      </iLONDataServer>
    </DataServer_Get>
  </S:Body>
</S:Envelope>
```

Listing 3.12: A `DataServer_Get()` request SOAP message. The *envelope* omits the *header* section and contains only a *body*. In the body the `DataServer_Get` tag specifies the requested function and encloses as parameter a NVL data point identified by its name.

tags and replacing the name with an integer. For instance: `<UCPTindex>311</UCPTindex>`. Listing 3.12 also shows that the data server's abstraction technology does not hide the underlying data point types, as already described in Section 3.3.2. The request message contains the data point type which should be queried. The data point type is defined by the `UCPTname` tags in the `DPType` element. In this example the queried data point is a NVL data point type.

Listing 3.13 shows the header of a real world response SOAP message to the request message in Listing 3.12. The body section of the response message is shown in Listing 3.14. The SOAP header of the i.LON 100 web service contains a time stamp (`UCPTtimeStamp`) and a unique identification number (`UCPTuniqueId`) for the SOAP message and some additional information about the i.LON 100 web service.

The actual response is located in the `SOAP-ENV:Body` section shown in Listing 3.14. Enclosed in the body section is the `DPType` section, which provides global information on – in this case – NVL data points. For example the life time property of NVL data point types, as described in Section 3.3.2.2, is represented in the SOAP response message by the value between the `UCPTlifeTime` tags.

Enclosed in the `DP` section resides the information on the queried data point. The section contains name (`UCPTpointName`) and index (`UCPTindex`) of the data point, the type of the underlying network variable (`UCPTformatDescription`), and the direction (`UCPTdirection`) of the data point – respectively if the data point is assigned to an input or an output network variable.

Additionally presets can be defined for every data point. These presets are described in Listing 3.14 via the `ValueDef` tags. A preset is a shortcut for an explicit value of a data point. The network variable assigned to the data point in the listing is of type *SNVT_switch*. The only applicable values for such a network variable are *100.0 1* and *0.0 0* – representing the switched on respectively the switched off state. The presets can be used instead of this explicit values when altering the value of a data point. For example if an application wanted

```
<?xml version="1.0" encoding="utf-8" ?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <p:messageProperties xmlns:p="http://wsdl.echelon.com/
        web_services_ns/ilon100/v3.0/message/">
      <p:UCPTtimeStamp>
        2007-08-20T09:33:29.150+02:00
      </p:UCPTtimeStamp>
      <p:UCPTuniqueId>03000012f6e2</p:UCPTuniqueId>
      <p:UCPTipAddress>192.168.1.222</p:UCPTipAddress>
      <p:UCPTport>80</p:UCPTport>
      <p:UCPTlastUpdate>2007-08-16T06:46:50Z</p:UCPTlastUpdate>
    </p:messageProperties>
  </SOAP-ENV:Header>
```

Listing 3.13: SOAP *header* of a `DataServer_Get()` response message. The header is an optional part of a SOAP message, the i.LON 100 web service uses the header to provide additional information, such as a time stamp and configuration values.

to change the value of the data point in Listing 3.14, the application could use the string *ON* as value when writing to a data point. The data server would translate this string into *100.0 1* because this preset has been defined for this data point. For more information on writing to data points see Section 3.4.3.3.

3.4.3.2 DataServer_Read

The `DataServer_Read()` [Ech06c, pp .4-20–4-25] function of the i.LON 100 web service can be used to retrieve the current value and status information of specific data points. The data points can be addressed either by name or by index. Additionally it is possible to only query data points which have changed in a specified time frame. Listing 3.15 shows a `DataServer_Read()` request SOAP message which addresses a NVL data point via its index, because that is the common usage of this function in this work.

The `DataServer_Read()` response SOAP message to above request is shown in Listing 3.16. As in every response message the i.LON 100 web service provides a SOAP header section in its response message. The actual response data resides in the SOAP body section.

For every retrieved data point a `DP` section exist. It contains some information which is also available in the response of the `DataServer_Get()` function, for example: `UCPTpointName`, `UCPTindex` and `UCPTformatDescription`. The actual value of the data point is enclosed by the `UCPTvalue` tags. The `UCPTvalueDef` tags enclose the name of the preset assigned to the current value. The `DP` section also contains a `UCPTpointUpdateTime` tag, which contains the point in time the value of the data point was updated to the value of the assigned network variable.

Compared to the `DataServer_Get()` response the `DataServer_Read()` response message includes a `UCPTfaultCount` tag, which states the number of data points on which an error occurred while reading the values. If the fault count is zero as in this example all data point values have been retrieved successfully. If an error occurred while retrieving the current value of a data point, `UCPTfaultCount` would be greater zero and the `DP` sections representing the

```
  <SOAP-ENV:Body>
    <DataServer_GetResponse xmlns="http://wsdl.echelon.com/
        web_services_ns/ilon100/v3.0/message/">
      <iLONDataServer>
        <DPType>
          <UCPTlastUpdate>2007-08-16T06:46:49Z</UCPTlastUpdate>
          <UCPTlifeTime>0</UCPTlifeTime>
          <UCPTindex>34491</UCPTindex>
          <UCPTname>NVL</UCPTname>
          <DP>
            <UCPTindex>339</UCPTindex>
            <UCPTpointName>NVL_outmod2_NVI13</UCPTpointName>
            <UCPTlocation>iLON100/NVL/dynamic</UCPTlocation>
            <UCPTdescription></UCPTdescription>
            <UCPTformatDescription>
              SNVT_switch
            </UCPTformatDescription>
            <UCPTdpSize>2</UCPTdpSize>
            <UCPTbaseType>BT_STRUCT</UCPTbaseType>
            <UCPTunit></UCPTunit>
            <SCPTmaxSendTime>0.0</SCPTmaxSendTime>
            <SCPTminSendTime>0.0</SCPTminSendTime>
            <SCPTmaxRcvTime>0.0</SCPTmaxRcvTime>
            <UCPTsettings>
              0,0,0,0,0,0,0,0,0,0,0,0,0,1,1
            </UCPTsettings>
            <UCPTdirection>DIR_OUT</UCPTdirection>
            <ValueDef>
              <UCPTindex>0</UCPTindex>
              <UCPTname>OFF</UCPTname>
              <UCPTvalue>0.0 0</UCPTvalue>
            </ValueDef>
            <ValueDef>
              <UCPTindex>1</UCPTindex>
              <UCPTname>ON</UCPTname>
              <UCPTvalue>100.0 1</UCPTvalue>
            </ValueDef>
          </DP>
        </DPType>
      </iLONDataServer>
    </DataServer_GetResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 3.14: SOAP *body* of a `DataServer_Get()` response message. The type of the message is defined by the `DataServer_GetResponse` tag. The `DP` tag contains data on the requested data point.

```
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <DataServer_Read xmlns="http://wsdl.echelon.com/
        web_services_ns/ilon100/v3.0/message/">
      <iLONDataServer>
        <DPType>
          <UCPTname>NVL</UCPTname>
          <DP>
            <UCPTindex>311</UCPTindex>
          </DP>
        </DPType>
      </iLONDataServer>
    </DataServer_Read>
  </S:Body>
</S:Envelope>
```

Listing 3.15: A `DataServer_Read()` request SOAP message. The *envelope* omits the *header* section and contains only a *body*. In the body the `DataServer_Read` tag specifies the requested function and encloses as parameter a NVL data point identified by its index.

faulty data points would contain `faultcode` and `faultstring` elements. These two elements state the type of the error [Ech06c, pp. 2-6–2-8]. If in the case of an error a value is provided in the `Read()` response message, it is not ensured that this value is valid.

3.4.3.3 DataServer_Write

The `DataServer_Write()` [Ech06c, pp. 4-26–4.29] function of the i.LON 100 web service can be used to alter the value of data points. The data points which should be changed have to be addressed either by specifying their name or their index. Also the type of the data point has to be provided in the `DPType` section.

As the purpose of the `DataServer_Write()` function is to alter the value of a data point, the value to which the data point should be updated to, has to be provided for every data point. Listing 3.17 shows a `DataServer_Write()` request message which addresses one data point by index and wants to set the value of this data point to *100.0 1*. Instead of using the `UCPTvalue` tag and providing an explicit value also the `UCPTvalueDef` tag with a preset value could have been used.

Listing 3.18 shows the `DataServer_Write()` response SOAP message for the above request message. Similarly to the other response messages the i.LON 100 includes a SOAP header section in the `DataServer_Write()` response message. The actual response data resides in the SOAP body section. The response contains the name and number of every data point written to.

As in the `DataServer_Read()` response message a `UCPTfaultCount` is included in response message shown in Listing 3.18. The fault count value states the number of data points on which an error occured while altering its value. In the listed response message the fault count is zero, therefore all data points have been updated to the new value. If an error would have occur ed the `DP` section representing the faulty data point would contain a `faultcode` and a `faultstring` tag, which would state the type of the error [Ech06c, p. 2-6–2-8]. The value of the data point and the assigned network variable are undefined if the `DataServer_Write()` function fails.

```
<?xml version="1.0" encoding="utf-8" ?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <p:messageProperties xmlns:p="http://wsdl.echelon.com/
        web_services_ns/ilon100/v3.0/message/">
      <p:UCPTtimeStamp>
        2007-08-21T09:40:54.920+02:00
      </p:UCPTtimeStamp>
      <p:UCPTuniqueId>03000012f6e2</p:UCPTuniqueId>
      <p:UCPTipAddress>192.168.1.222</p:UCPTipAddress>
      <p:UCPTport>80</p:UCPTport>
      <p:UCPTlastUpdate>2007-08-21T07:39:00Z</p:UCPTlastUpdate>
    </p:messageProperties>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <DataServer_ReadResponse xmlns="http://wsdl.echelon.com/
        web_services_ns/ilon100/v3.0/message/">
      <iLONDataServer>
        <UCPTfaultCount>0</UCPTfaultCount>
        <DPType>
          <UCPTindex>34491</UCPTindex>
          <UCPTname>NVL</UCPTname>
          <DP>
            <UCPTindex>311</UCPTindex>
            <UCPTpointName>NVL_outmod1_NVI01</UCPTpointName>
            <UCPTpointUpdateTime>
              2007-08-21T09:38:59.260+02:00
            </UCPTpointUpdateTime>
            <UCPTformatDescription>
              SNVT_switch
            </UCPTformatDescription>
            <UCPTvalue>0.0 0</UCPTvalue>
            <UCPTvalueDef>OFF</UCPTvalueDef>
            <UCPTunit></UCPTunit>
            <UCPTpointStatus>AL_NUL</UCPTpointStatus>
            <UCPTpriority>255</UCPTpriority>
          </DP>
        </DPType>
      </iLONDataServer>
    </DataServer_ReadResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 3.16: A `DataServer_Read()` response SOAP message containing a *header* section with information on the i.LON 100 status and a *body* section, which contains the actual response data. The type of the message is defined by the `DataServer_ReadResponse` tag. The `DP` tag contains some basic information on the requested data point and the current value (`UCPTvalue`).

```
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <DataServer_Write xmlns="http://wsdl.echelon.com/
        web_services_ns/ilon100/v3.0/message/">
      <iLONDataServer>
        <DPType>
          <UCPTname>NVL</UCPTname>
          <DP>
            <UCPTindex>310</UCPTindex>
            <UCPTvalue>100.0 1</UCPTvalue>
          </DP>
        </DPType>
      </iLONDataServer>
    </DataServer_Write>
  </S:Body>
</S:Envelope>
```

Listing 3.17: A `DataServer_Write()` request SOAP message. The *envelope* omits the *header* section and contains only a *body*. In the body the `DataServer_Write` tag specifies the requested function and encloses as parameter a NVL data point identified by its index.

```
<?xml version="1.0" encoding="utf-8" ?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <p:messageProperties xmlns:p="http://wsdl.echelon.com/
        web_services_ns/ilon100/v3.0/message/">
      <p:UCPTtimeStamp>
        2007-08-20T09:33:29.490+02:00
      </p:UCPTtimeStamp>
      <p:UCPTuniqueId>03000012f6e2</p:UCPTuniqueId>
      <p:UCPTipAddress>192.168.1.222</p:UCPTipAddress>
      <p:UCPTport>80</p:UCPTport>
      <p:UCPTlastUpdate>2007-08-16T06:46:50Z</p:UCPTlastUpdate>
    </p:messageProperties>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <DataServer_WriteResponse xmlns="http://wsdl.echelon.com/
        web_services_ns/ilon100/v3.0/message/">
      <iLONDataServer>
        <UCPTfaultCount>0</UCPTfaultCount>
        <DPType>
          <UCPTindex>34491</UCPTindex>
          <UCPTname>NVL</UCPTname>
          <DP>
            <UCPTindex>310</UCPTindex>
            <UCPTpointName>NVL_outmod1_NVI00</UCPTpointName>
          </DP>
        </DPType>
      </iLONDataServer>
    </DataServer_WriteResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 3.18: A `DataServer_Write()` response SOAP message containing a *header* section with information on the i.LON 100 status and a *body* section, which contains the actual response data. The type of the message is defined by the `DataServer_WriteResponse` tag. The `DP` tag contains only basic information on the requested data point. If writing to the data point had failed, tags notifying about the error would be present.

3.5 Evaluation of Web Service Frameworks

The last section described the utilization of SOAP (Simple Object Access Protocol) and therefore XML files to communicate with the i.LON 100 web service. For every web service consumption attempt XML files have to be created and transmitted via HTTP. Then the response XML files have to be received via HTTP and parsed afterwards. All these steps have to be repeated for every web service consumption. To reduce the burden of the developers working on web services and web service clients, specialized frameworks exist.

For SOAP based web services exist a great number of frameworks[27], which are sometimes called SOAP stacks. For this master thesis some SOAP stacks have been evaluated to choose one for utilization in the building automation interfacing system.

As programming language *Java*[28] has been chosen for this work, because most of the available SOAP stacks are developed for Java and because its relative[29] platform independency ensures that the produced application will be usable on the platform the media façade producers and users are using.

The next sections will discuss some SOAP stacks available for the Java programming language. Their features, the provided tools and the ways to utilize the stacks will be described and evaluated.

3.5.1 Apache Extensible Interaction System

The *Apache eXtensible Interaction System* (Axis) is a SOAP stack developed within the Apache[30] project. It is a well-known and widely deployed web service framework because it is freely available since 2001 already. Actually there exist two totally different version of the Axis framework: Axis1[31], which was the first attempt of the Apache project to develop a web service framework and Axis2[32] which is a complete rewrite of Axis1. In this work only version 2 was tested.

The Axis2 framework provides functionality for building web service providers and for building web service clients. In the evaluation only the part relevant for clients was considered. For the implementation of clients the tools delivered with the stack are a very important part of the provided functionality, as mentioned in Section 3.1.1. Most SOAP stacks provide a tool, which is able to transfrom the description of a web service – provided in a WSDL file – to source code. The produced source code supports the developers in building a client for the web service described in the WSDL file. The relevant tool in Axis2 is called `wsdl2java`.

When `wsdl2java` is invoked it parses the provided WSDL file and essentially produces a Java class for every element included in a SOAP request message or in a SOAP response message. A `DataServer_Read` request message is shown in Listing 3.16. The SOAP body of the request message contains the elements `DataServer_Read`, `iLONDataServer`, `DPType`, `UCPTname`, `DP` and `UCPTindex`. For every of this elements a Java class is generated. The name

[27] For a not exhaustive list look at: `http://en.wikipedia.org/wiki/List_of_Web_service_Frameworks`
[28] The used version was Java Standard Edition 6, available at `http://java.sun.com/javase/6/`
[29] Java is not fully platform independent because not for every platform exists a *Java Virtual Machine*.
[30] `http://www.apache.org/`
[31] `http://ws.apache.org/axis/`
[32] `http://ws.apache.org/axis2/`

of the generated Java class is similar to the corresponding element in the SOAP message. To invoke a `DataServer_Read()` function of the web service the relevant classes have to be instantiated and nested similar to the SOAP request message. This is shown in Listing 3.19.

```
DS_Read dsread = new DS_Read();
DPType_type55 dpt = new DPType_type55();
dpt.setUCPTname("NVL");
DP_type54 dp = new DP_type54();
dp.setUCPTindex(311);
//nesting classes inside the other classes according to SOAP message
dpt.addDP(dp);
dsread.addDPType(dpt);
//invoking the web service function
DataServer_ReadResponse resp = stub.DataServer_Read(dsread);
```

Listing 3.19: Utilizing classes created by Axis2 to consume the web service function `DataServer_Read()`. The code requests the current value of a NVL data point, identified by the index *311*. The response is stored in the class `DataServer_ReadResponse`.

Code to consume the i.LON 100 web service function `DataServer_Read()` with Axis2 generated classes is shown in Listing 3.19. Execution of this code produces the request SOAP message in Listing 3.15. The class `DS_Read` corresponds to the `DataServer_Read` tag in the SOAP request message. The classes `DPType_type55` and `DP_type54` correspond to the `DPType` and `DP` tags in the request message and define the type of the data point respectively identify a specific data point. The last line in Listing 3.19 actually starts the consumption of the web service, the `DataServer_ReadResponse` tag in the response SOAP message in Listing 3.16 corresponds to the `DataServer_ReadResponse` class in Listing 3.19.

By using the classes generated by `wsdl2java` no handling of XML files is required. The postfixes `_typeXX` of `DPType` and `DP` in Listing 3.19 are added by `wsdl2java` because the `DPType` and `DP` appear more than once in the WSDL file.

3.5.2 Java API for XML – Web Services

Java API[33] *for XML – Web Services*[34] (JAX-WS) is a web service framework for implementation of web service clients. Support for developing web service providers is not included. JAX-WS has been considered for usage in this work because it is included in the Java 6 distribution and is therefore easy to retrieve and install. However JAX-WS already existed before inclusion into the standard Java distribution. The version of JAX-WS included in Java 6 is 2.0.

The tool included in JAX-WS for transforming a WSDL web service description into Java classes is called `wsimport` and is included in the Java 6 distribution. Similarly to the Axis2 tool `wsdl2java` it generates a Java class for every XML element which appears in either a SOAP request or a SOAP response message. For example every XML element in a `DataServer_Read()` request message, shown in 3.15, is transformed in a Java class. The web service function `DataServer_Read()` can be consumed by instantiating the corresponding Java classes. Listing 3.20 shows the code to consume the `DataServer_Read()` function of the i.LON 100 web service by utilizing the classes generated by `wsimport`.

[33] API: *Application Programming Interface*
[34] `https://jax-ws.dev.java.net/`

```
DSRead dsread = new DSRead();
DSRead.DPType dpt = new DSRead.DPType();
dpt.setUCPTname("NVL");
DSRead.DPType.DP dp = new DSRead.DPType.DP();
dp.setUCPTindex(311);
//nesting classes inside the other classes according to SOAP message
dpt.getDP().add(dp);
dsread.getDPType().add(dpt);
//invoking the web service function
DSReadInfo resp = stub.dataServerRead(dsread);
```

Listing 3.20: Utilizing classes created by JAX-WS to consume the web service function `DataServer_Read()`. The code requests the current value of a NVL data point, identified by the index *311*. The response is stored in the class `DataServer_ReadResponse`.

The Java classes corresponding to XML elements in SOAP messages have to be nested exactly like the XML elements are structured inside the SOAP messages. Listing 3.20 shows this nesting which is done by calling `add()` functions. The code in this listing generates a `DataServer_Read()` request messages, which queries a NVL data point with the index 311. The request message is shown in Listing 3.15. The actual consumption of the web service is initiated with the last line in Listing 3.20. The class `DSReadInfo` in the last line corresponds to the response SOAP message of `DataServer_Read()` shown in Listing 3.16.

The code for consuming the `DataServer_Read()` web service function with JAX-WS is very similar to the code required for Axis2, the biggest difference is that JAX-WS defines the classes nested according to the XML files. For example the class `DPType` resides in class `DSRead`. Because of this nesting the class names of JAX-WS generated classes do not have such postfixes as the classes generated by Axis2.

3.5.3 Results of Web Service Framework Evaluation

At first Axis (version 2) was chosen to build upon in this work, because it exists since 2001, which is a long period of time for Internet-related technologies. The Axis project is well known in the web service eco-system and is recognized as proven and stable product. However while developing the parts to consume the i.LON 100 web service a bug was discovered. The elements of the `DPType` section in the SOAP response message of the i.LON 100 `DataServer_List()` function are not arranged as described in the WSDL file of the i.LON 100 web service. According to the WSDL the `UCPTindex` and `UCPTpointName` elements should be the first elements in the `DP` section. But in the actual SOAP response message these two elements are at the end of the `DP` section[35].

The incorrect sequence of XML tags in the `DataServer_List()` response SOAP message is shown in Listing 3.21. According to the WSDL description of the i.LON 100 web service the tags `UCPTindex` and `UCPTname` at the end of the listing should be the first elements inside the `DPType` structure. Although the sequence of the elements does not comply to the WSDL description the content of the response is correct nevertheless, only the sequence of tags is incorrect. But a problem arises, because Axis2 fails when parsing this wrongly sequenced

[35] This error is documented at `http://webserv.echelon.com/default.asp?action=9&boardid=1&read=6250&fid=13`.

```
<DataServer_ListResponse
    xmlns="http://wsdl.echelon.com/web_services_ns/ilon100/v3.0/message/">
  <iLONDataServer>
    <DPType>
      <SCPTobjMajVer>3</SCPTobjMajVer>
      <SCPTobjMinVer>2</SCPTobjMinVer>
      <UCPTcurrentConfig>1.1</UCPTcurrentConfig>
      <UCPTlastUpdate>2007-09-20T07:47:41Z</UCPTlastUpdate>
      <UCPTlifeTime>0</UCPTlifeTime>
      <UCPTindex>24383</UCPTindex>
      <UCPTname>NVE</UCPTname>
      ...
```

Listing 3.21: Incorrect sequence in `DataServer_List()` response SOAP message. The `UCPTindex` and `UCPTname` elements should be the first elements inside the `DPType` structure.

response message. Therefore it was decided to abandon Axis2 and utilize another SOAP stack instead.

As Axis2 could not be used because of the incorrect sequence elements in `DataServer_List` response messages it was decided to switch[36] to another web service framework, namely JAX-WS. It was decided to use JAX-WS because it is included in the Java 6 standard distribution. This ensures availability and an easier install process for users of the developed software.

The version of JAX-WS included in the Java 6 distribution is 2.0. Regrettably this version of the tool `wsimport` produces an error when parsing the WSDL file of the i.LON 100 web service. Fortunately version 2.1 of JAX-WS was already available, which can be installed on top of the version included in Java 6 [37]. This update resolved the issues and `wsimport` was able to successfully parse the WSDL file and generate the corresponding Java classes.

[36] Ironically, the `DataServer_List` function is not used in the software at the moment, because the data points are not retrieved from the i.LON 100 but have to be pre-defined in a configuration file (see Section 6.3). However a switch back to Axis2 was not done, because if the requirement to consume `DataServer_List` arises in the future the software will be prepared for it.

[37] Instructions how to install JAX-WS 2.1 with Java 6 can be found at: `https://jax-ws.dev.java.net/guide/Using_JAX_WS_2_1_with_JavaSE6.html`

Chapter 4

Building Automation and Cognitive Science

The technology deployed in buildings has changed quite a lot in the last century. Fifty years ago most buildings comprised room lighting and, depending on climate conditions, heating. In today's modern buildings these two tasks are still among the most important, but a lot of other technologies have been added. Additionally to heating, ventilation and air conditioning have grown to be so basic functionality of a building that the term HVAC (*Heating, Ventilation, Air Conditioning*) has been coined to refer to this part of building technology. HVAC and room lighting are the origins of modern building automation. Building automation today refers to distributed and networked systems deployed in buildings and responsible for controlling environment parameters.

With the evolving computer technology and the development of field buses the possibility arose to build all the diverse systems deployed in a building on a common foundation [Rus03, p. 2]. Field buses which are specifically designed for utilization in building automation systems can act as such a foundation.

A common foundation on which systems can operate and interact also allows easier integration of new systems and functionality. Today's buildings provide much more than automated room lighting and HVAC. For example the information gathered throughout the building automation system can be used to optimize energy consumption. Access control for special sections in a building and room management are also applications which are integrated into building automation systems.

The number of applications in a building automation system will grow even faster in the future [PP05]. New functionality like automatic initiation of repairs and automatic reaction on extra-ordinary events one of the possible applications which will be seen in the near future. As the number of applications grow the number of sensors, actuators and nodes will grow rapidly too [Die00]. Management of the expected amount of sensors and actuators will become very hard or even impossible with today's approaches [PP05]. One of the goals of the *Artificial Recognition System* (ARS) is to solve this scalability problem.

ARS attempts to heave building automation onto the next evolutionary level. By hiding and abstracting the low level details of building automation and generating a more generic and overall view on buildings. Such an overall view of a building which allows to detect extra-ordinary events and situations may be of use for media façades too.

4.1 Artificial Recognition System

The ARS project aims to develop and implement a technical model of the human brain. The technical model described in [Pra06] and [Bur07] is based on models developed by scientific disciplines like neuroscience and psychoanalysis. The human brain is a system which is capable of dealing with an great number of "sensors". The human mind is able to react on input data and develop strategies by evaluating the input data. These two "features" of humans are approached in two subproject within the ARS project [DLP+06].

- Artificial Recognition System – Perception (ARS-PC) is an approach to implement the perception ability of the human brain. The chosen approach for ARS-PC is called symbolization. Symbolization refers to a "condensing" process which transforms a huge amount of input data into information of higher value. The pieces of information with higher value are called *symbols* [Rus03].
- Artificial Recognition System – Psychoanalysis (ARS-PA) is an approach to decision making. The goal is to implement an information-classifying-system based on emotions as described by psychoanalytic theories. The classified information and the emotions are utilized by an implementation of Freud's Ego-Superego-Id model to perform the actual decision making.

ARS-PC is designed to condense the data output of sensors into high level information, called symbols. All aggregated symbols represent the world as it is perceived by the system. The entirety of aggregated high level symbols is called world representation. ARS-PA operates on a world representation and makes decisions based on the state of the perceived world. Decisions lead to actions which are transformed into the real world via actuators. A more detailed description of ARS-PA can be found in Section 4.1.2. The correlation between ARS-PC, ARS-PA and the real world are shown in Figure 4.1.

4.1.1 Artificial Recognition System – Perception

As described above ARS-PC transforms sensor values into symbols, which collectively are a world representation. The transformation process from sensors to symbols is called symbolization. A symbol is defined in [Pra06, p. 31] as *collections of information, which are contained in the symbol itself and in its properties.* In ARS different types of symbols exist. The type of a symbol itself contains information. For example a symbol "chair" and a symbol "movement" contain already information on the nature of the symbol, without further investigation of the symbol properties. The properties of a symbol define further information like the position of a chair or the velocity of a movement.

In ARS a hierarchy of symbol levels is defined. The hierarchy consists of three levels: *representation symbol level*, *snapshot symbol level* and *micro symbol level* [Pra06]. Figure 4.2 shows these three levels. Symbols are, depending on their position in the hierarchy, classified as *representation symbols*, *snapshot symbols* or *micro symbols.*

- Micro symbols are located at the bottom of the hierarchical symbol model of ARS. Sensor values are represented by micro symbols. Micro symbols are generated, deleted or changed whenever sensor values change. The number of existing micro symbols is related to the number of existing sensors.

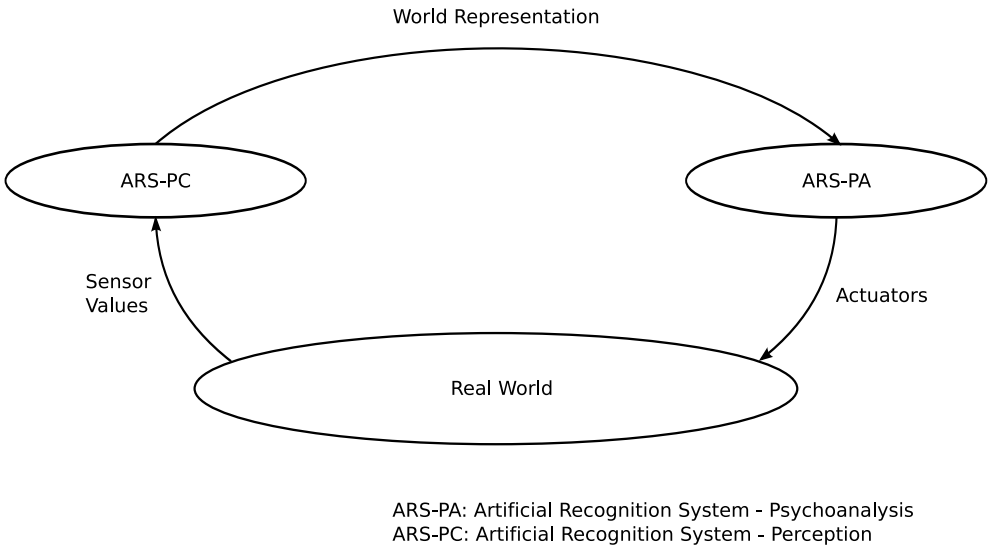

Figure 4.1: Interaction of the Artificial Recognition System with the real world [Ric07]. Sensors measure the real world and produce sensor values. ARS-PC generates a world representation, which is used by ARS-PA to make decisions. The actions derived from decisions are applied to the real world via actuators.

- Snapshot symbols are symbols which represent the perceived world in a specific moment, they represent a *snapshot* of the world. Snapshot symbols are missing continuity in time.
- Representation symbols are on the highest level of the symbol hierarchy. They collectively form the world representation. Similar to snapshot symbols representation symbols represent the perceived world, but in contrast to snapshot symbols representation symbols are not restricted to a specific moment, they also include the history of world representation.

The world representation and therefore the representation symbols are created by means of symbolization. Symbolization refers to the process of forming symbols on a specific level in the hierarchical symbol model from symbols on the level below. Micro symbols are an exception in this regard as they are not created from symbols but directly from sensor values.

Figure 4.2 shows the process of symbolization and the distribution of symbols among the three levels. Symbols are represented as cuboids of different sizes. The cuboid representations of symbols in Figure 4.2 are smallest for micro symbols because the size relates to the informational value of symbols of a specific level. The cuboid representation of representation symbols are bigger than for other symbol types because representation symbols contain the highest informational value.

The highest number of symbols exist on the micro symbol level. Snapshot symbols are created from several micro symbols, resulting in a decreased number of symbols in the snapshot symbol level than in the micro symbol level. But snapshot symbols have a higher informational value than micro symbols. As an example a person moving through a room is considered.

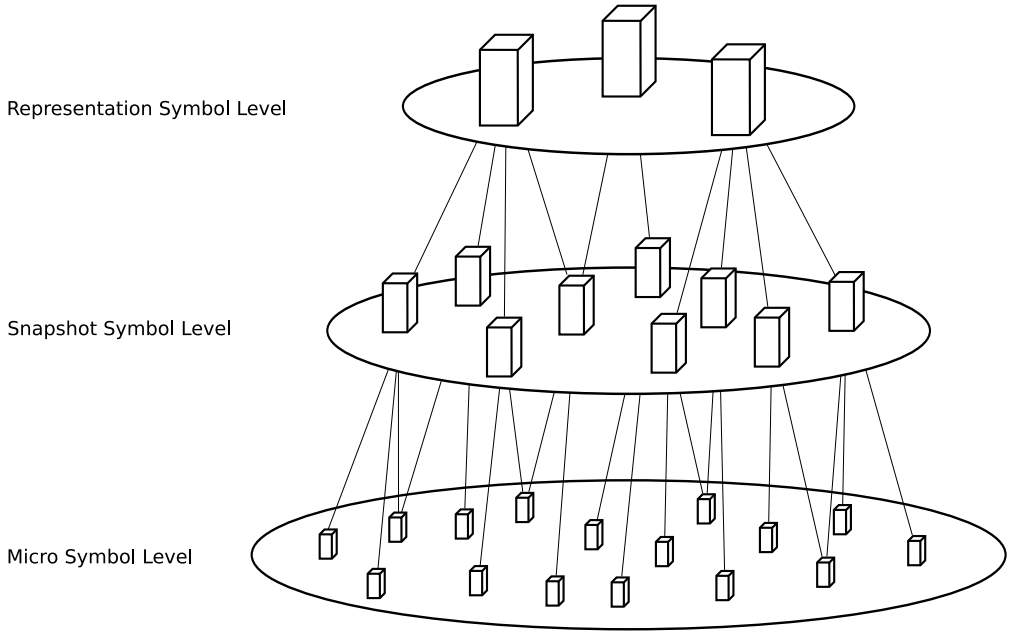

Figure 4.2: Symbol levels of ARS [Pra06]. Sensor value updates are transformed into micro symbols. Micro symbols are aggregated into snapshot levels, which only represent one point in time. Representation symbols emerge from snapshot symbols. The representation symbols form the world representation.

Assuming the room is accordingly equipped, movement sensors, light barriers and tactile sensors mounted on the floor will be triggered. The change in value of these sensors leads to creation of new micro symbols. The changes on the micro symbol level triggers creation of snapshot symbols. For example the micro symbols can be condensed to several snapshot symbols of type "person position". Each of the snapshot symbols indicate only the presence of a person at a specific position at a specific time.

On the next level, the representation symbol level, the snapshot symbols are condensed even further. The snapshot symbols, each representing the person at a specific position, can be merged into one "person" symbol. A representation symbol of type "person" is created when the person enters the room and is then updated with the new position by evaluating snapshot symbols. The person symbol can contain the previous positions of the person and several other attributes of the person. The person symbol is persistent as long as the person stays in the room, only the attributes of the symbol are updated. Updating existing symbols leads to lesser symbol creation and deletion and ensures high informational value of representation symbols.

ARS defines two more symbol types beside the already described symbol types. These symbols are named *scenario symbols* and *action symbols* and are only created by applications operating on the world representation. Scenario symbols are created by applications which observe the world representation to detect specific scenarios. When the application detects a

scenario it can create a scenario symbol add it to the world representation. Other applications can react to such scenarios and emit action symbols. An action symbol contains instructions how to react and influence the outer world.

4.1.2 Artificial Recognition System – Psychoanalysis

ARS-PA is a bionic approach to decision making. It is an attempt to implement psychoanalytical models of the human mind into a technical system. Figure 4.3 shows the basic data flow in an ARS system. It is currently worked on integration of ARS-PC as the perception module in Figure 4.3. Until then a simulator, the *Bubble Family Game* [DZL07], is used as to provide a world representation.

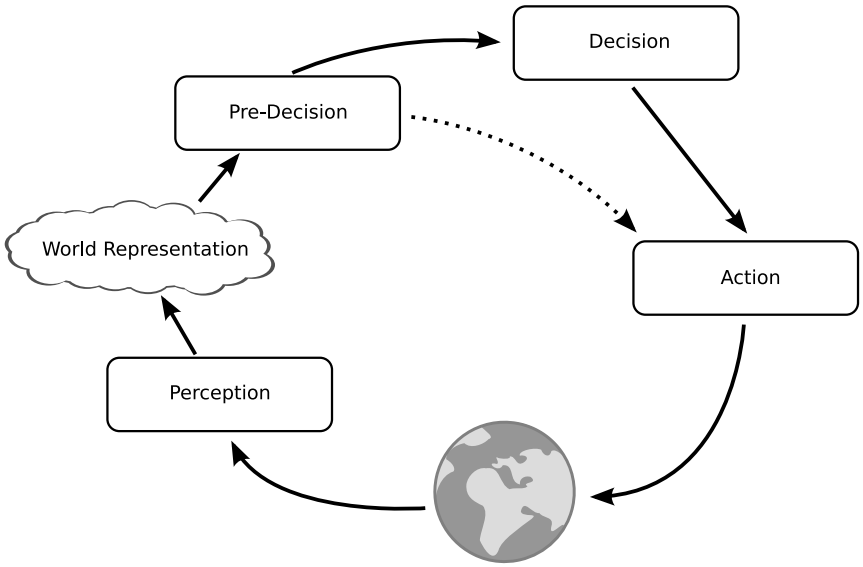

Figure 4.3: Data Flow in ARS-PA [DZL07]. The perception unit build a world representation of the inner and outer world. The pre-decision unit rates the perceived symbols based on basic emotions and drives. The decision unit contains subunits for fast reactions and strategic decisions. The action unit applies decision to the real world.

In the context of ARS-PA the world representation includes not only the perceived "outer world" but also an "inner world". The outer world is the "real world" in which the system operates. The inner world represents the internal state of the ARS-PA system. The world representation contains representation symbols for both, the outer world and the inner world. These symbols are first evaluated in the pre-decision unit. The evaluation process in the pre-decision unit is based on *basic emotions* and *drives* in analogy to the human mind. The basic emotions are rage, fear, panic and a seeking system [BLPV07]. For example if the emotion panic is very high (and the system may try to escape a dangerous situation) the perception of a barrier in its way is rated very high, while if the system were in a relaxed state a barrier would be rated much lower. The evaluation of the perception symbols therefore allows a filtering of important information.

After evaluation in the pre-decision unit the rated representation symbols are forwarded to the decision unit which consists of three sub-units:

- The *reactive decision unit* is the simplest unit and is designed for immediate reactions on dangerous situations.
- The *routine decision unit* is a more sophisticated decision unit which is responsible for execution of a sequence of actions.
- The *reflexive decision unit* is the most time-consuming decision unit. *Complex emotions* and the *ego* are responsible for decision making in the reflexive decision unit. The ego creates long-term plans and is therefore the strategic unit, the complex emotions are similar to basic emotions used as evaluation system. For example a complex emotion "appreciation" can be used to decide which decision among others should be chosen.

The decisions made by the decision units of ARS-PA are forwarded to the action unit. The action unit is responsible for conflict resolution if decisions contradict each other and is responsible for actually applying the actions in the real world.

The concepts used in ARS-PA, like emotions and drives, are not directly applicable to building automation. The concepts has to be mapped to concepts used in building automation. As example [DLP+06] proposes to interpret the drive "hunger" as ration between consumed and available energy. Therefore a high "hunger" level would result in decisions which are reducing the consumed energy.

4.1.3 Evaluation of the Artificial Recognition System

ARS was considered to act as the "translation system" described in Chapter 1. Such a translation system is required to build a bridge between the different semantics of building automation and media façades. One reason project ARS was started at the ICT is, that a traditional building automation system is not able to cope with the expected growth in the amount of data. Therefore ARS is designed to manage and cope with large amounts of data. It was decided to evaluate ARS if it could be used as the required translation system. If ARS can be used as translation system, ways have to be found how ARS can be incorporated in existing building automation and media façade systems.

Building automation operates on very low level data like temperature values of sensors and binary values of actuators. Media façades do not have any interest in such low level data. For media façades a global view of the building automation data would be interesting. Not the temperature of the room or which lights are turned on are of interest, but data on the general occupancy of the building, detection of stress scenarios or social happenings like parties.

The goal of ARS-PC could be described as abstraction of specific sensor data to generate a more global view of the environment. This is exactly what media façades could be interested in. Therefore the natural choice was to utilize ARS-PC as translation system between building automation and media façades. ARS-PA was also considered for utilization as translation system. Modifications to the ARS-PA simulation environment to enable input of building automation data was evaluated, but several problems appeared:

- First, ARS-PA was not designed to operate with low level building automation data. Maybe it would be possible to adapt ARS-PA accordingly but it is hard to estimate the complexity of the task.
- Second, its not clear what results a ARS-PA system will deliver if it is adapted to building automation data.
- Third, ARS-PC's functionality matches exactly the requirements a translation system has to fulfill.

Because of above reasons ARS-PC was chosen to be utilized as translation system. However ARS-PA may be a valuable addition once ARS-PA uses the world representation generated by ARS-PC, because the media façade could integrate decisions and the internal emotional state of ARS-PA in its appearance. Therefore a future integration of ARS-PA should be taken into account while designing the translation system.

For now it was decided to integrate ARS-PC as translation system between building automation and media façade. ARS-PC should transform the input building automation data into representation symbols. The resulting representation symbols will be forwarded to the media façade system. The implementation of ARS-PC integration and representation symbol forwarding is described in Section 6.4. The next section describes the technical design of ARS-PC.

4.2 Technical Design of the Artificial Recognition System

This section gives a brief overview of the technical design of ARS-PC. The goal of the description is to identify possible ways to integrate ARS-PC in building automation systems and how the generated world representation can be provided to media façades. Therefore description is focused on the input and output aspects of ARS-PC not on internal data processing.

4.2.1 Sources of Input Data

ARS-PC has 3 different start-up configurations. They differ in what *graphical user interface* (GUI) is started to show the generated symbols and what input data is used. ARS will be used without a GUI, while acting as translation system between building automation and media façades therefore they will be neglected in the further description of ARS-PC. The configuration of ARS-PC are:

- The *live* configuration uses sensor values from the *Smart Kitchen*, see Section 4.2.3, as input values.
- The *database* configuration retrieves sensor values from a database. The sensor values originate from the Smart Kitchen and have been recorded for testing purposes.
- The *simulation* configuration also retrieves sensor values from a database. But these sensor values originate from a simulation environment [Har07], which is able to simulate sensor values in an occupied building.

Each of the configurations implements its own start-up classes and its own *micro symbol factory* (`msf_live`, `msf_database` and `msf_simulation`). A micro symbol factory is a software component which creates micro symbols. The micro symbol factories are described in more detail in [Ric07]

As shown in Figure 4.2 micro symbols are on the lowest level of the hierarchical symbol model of ARS. Micro symbols are generated when sensor values change. Depending on the configuration the sensor values are retrieved from different sources. The micro symbols are traversed through the symbolization process and the output of symbolization is a world representation which consists of representation, scenario and action symbols. A brief overview of the structure of ARS-PC is shown in Figure 4.4.

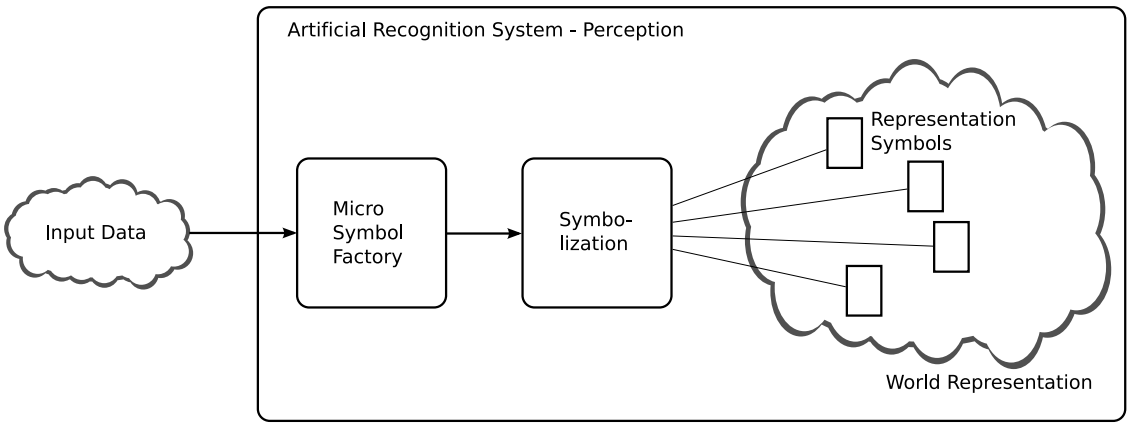

Figure 4.4: Data processing of ARS-PC. Input values are retrieved by a micro symbol factory, which generates micro symbols. The source of input values depends on the type of the micro symbol factory. The output of the symbolization process is a world representation.

The live configuration utilizes the specially equipped kitchen, named Smart Kitchen, of the ICT (Institute of Computer Technology) as data source. A more detailed description of the equipment of the smart kitchen can be found in Section 4.2.3. The sensor values which are retrieved from the Smart Kitchen are used to generate according micro symbols in the micro symbol factory `msf_live`. The generated micro symbols are distributed to other software modules via a mechanism called *SymbolNet*, which is described in Section 4.2.4.

The database configuration uses a database as source of sensor values. The sensors in the database originate from the Smart Kitchen, but they have been recorded while actions have been performed in the kitchen, which ARS is capable to detect as scenarios at the moment. The database configuration is used for test and demonstration cases. The micro symbol factory of the database configuration is called `msf_database`. It retrieves the sensor values from the database and takes care of correct ordering and timing when emitting the according micro symbols.

The simulation configuration is similar to the database configuration only the origin and nature of sensor values differ. The sensor values which are retrieved in the simulation configuration originate from a simulation environment especially designed to produce testing data

for (not only) ARS [Har07]. The micro symbol factory `msf_simulation` retrieves the sensors from the database and emits the corresponding micro symbols.

4.2.2 Symbolization and Observing the World Representation

The symbolization in ARS-PC as shown in Figure 4.4 is actually done in independent software modules, which are called symbolization modules in this work. Between these symbolization modules a tight mesh of connections exists, which is used for symbol exchange (see Section 4.2.4). Each of the internal symbolization modules of ARS-PC is connected to other modules awaiting notifications on specific symbols. If a symbolization module detects the specific pattern it is programmed for in the received symbols, it emits itself a symbol. The emitting of symbols if a specific pattern is detected is the core of the implementation of the symbolization process. The symbolization modules are only connected to symbolization modules, which are one level beneath them in the hierarchical symbol model of ARS-PC (Figure 4.2). If a symbolization module detects a specific sequence it emits a new symbol, which belongs to the higher hierarchical level. The new symbol may be received by another symbolization module which again can emit new symbols if it detects a specific pattern.

The "output" of the symbolization process of ARS-PC is a world representation consisting of symbols. These symbols can be representation symbols, scenario symbols or action symbols. An application which wants to observe the world representation generated by ARS-PC has to connect to the symbolization modules which emit symbols belonging to the world representations. For example if an application wants to be informed when a person enters a specific room, the application has to connect to the symbolization module, which generates "person" symbols.

The number of connections an application or symbolization module is allowed to have is not limited. Therefore an application can establish a great number of connections to different symbolization modules. Without such an limitation applications are able to receive a lot of symbols and therefore build an comprehensive view on the environment. For example, in Figure 4.4 an application could connect to all the representation symbols in the world representation and would be informed if updates in the world representation occur.

4.2.3 Smart Kitchen and the Artificial Recognition System

Project ARS has its origins in the *Smart Kitchen Project* described in [SRT00]. The Smart Kitchen is the kitchen of the Institute of Computer Technology (ICT), which has been adapted to act testing ground for new technologies. The goal of the initial Smart Kitchen project was to research and develop new applications for building automation.

The ARS project emerged from the Smart Kitchen project and is therefore still tightly bound to the Smart Kitchen. The Smart Kitchen has been equipped with numerous sensors, so that it can act as one of the possible data sources of the ARS-PC implementation.

The sensors in the Smart Kitchen comprise motion sensors, contact sensors at the kitchen door and the fridge door, temperature sensors at the stove, a vibration sensor on the coffee machine and tactile sensors mounted on the floor. The motion sensor detects motions in the room (without locating the position). The vibration sensor on the coffee machine is activated when the coffee machine starts grinding coffee beans and therefore provides information if

the coffee machine is in use. The tactile sensors on the floor allow detection of foots and foot steps and therefore allow to determine the location of a person. Currently it is worked on integration of a camera into the Smart Kitchen and the ARS project. The sensors and their installation is described in more detail in [Göt06].

The values of the sensors deployed in the Smart Kitchen are retrieved by a computer platform [Göt06] developed at the ICT. The sensor values are then sent to a remote PC via a Transmission Control Protocol [Pos81b] (TCP) connection. A running ARS implementation in the live configuration and therefore with a micro symbol factory `msf_live` receives sensor values. The micro symbol factory then creates micro symbols for every received sensor value and forwards micro symbols into the symbolization chain as shown in Figure 4.4.

4.2.4 Exchange of Symbols

Until now the *connection* between symbolization modules and *emitting* of symbols has not been described in more detail. The mechanism used for symbol exchange is called *SymbolNet* [Hol06]. It is a software component developed at the ICT. SymbolNet was designed to allow exchange of symbols between symbolization modules. The exchange is possible between modules running in the same process or even between applications on remote computers over TCP connections.

Before discussing the exchange of symbols via SymbolNet the implementation of symbols in ARS and SymbolNet will be described. As most parts of ARS-PC and SymbolNet are developed in the programming language Java, Symbols are implemented as Java classes. Some of the attributes a symbol contains are listed below:

- The *type* of a symbol indicates on which level in the hierarchical symbol model in Figure 4.2 belongs to.
- The *class* attribute of a symbol defines the semantic meaning of the symbol. A lot of semantic types, called symbol classes, are defined in ARS. For example symbol classes for *door open*, *gait* and *person* exist.
- The *id* is a unique number for every symbol instance and is used as identifier.
- The *timestamp* indicates the point in time when the symbol has been created.
- The *lifetime* of a symbol defines how long, starting at timestamp, a symbol is valid.
- *Properties* of a symbol are optional attributes. Properties contain additional information about the symbol. For example if the symbol is of class *person* the properties can contain the position or the identity of the person.

SymbolNet allows transmission of symbolic information either locally or to remote computers via a TCP connection. The actually transmitted content are not Java classes but *symbol messages*. The symbol messages are symbols represented in *Abstract Syntax Notation number One* [ISO02] (ASN.1). ASN.1 allows to describe data structures in a programming language independent way. For an in detail description of the used message format see [Hol06].

The interfaces of software modules which want to transmit symbol messages are defined by two *Java Interfaces* called `SymbolPipeA` and `SymbolPipeB`. A software module which is intended to emit symbol messages has to implement interface `SymbolPipeA`, a software module which has to be able to receive symbol messages has to implement interface `SymbolPipeB`. Software modules are allowed to implement both interfaces and can then act as sender and receiver of symbol messages.

SymbolNet also provides functionality to ease the implementation of receiving and transmitting symbol messages by transforming them automatically back into symbol instances. `SymbolContainer` is a class included in SymbolNet which automatically creates symbols from received messages. It implements the interfaces `SymbolPipeA` and `SymbolPipeB` and therefore can act as sender and receiver of symbol messages. A `SymbolContainer` instance creates the according symbol if a symbol message is received and puts the symbol in an internal storage. A user of SymbolNet which is not interested in the symbol messages but only in the resulting symbols can subclass `SymbolContainer` and add its own code to the available functionality. The `SymbolContainer` takes care of symbol creation while the user can operate on symbols and react on creation of new symbols without caring about symbol messages.

Chapter 5

System Design

In the previous chapters the various technologies involved in establishing a cooperation of media façades and building automation have been described. This chapter discusses the design of the system which enables the cooperation. The system consists of parts related to building automation, media façades and the *Artificial Recognition System* (ARS). For every part the chapter contains a section which describes the design of the system. The actual implementation is discussed in the next chapter.

5.1 Accessing Building Automation Data

In Chapter 3 building automation technologies have been discussed. The i.LON 100 was introduced as it allows to interact with building automation data via its web service. Before the i.LON 100 can be used for interaction purposes the attached building automation system and the i.LON 100 has to be configured accordingly. The next section describes the used configuration of the i.LON 100.

5.1.1 i.LON 100 Configuration

Section 3.4 describes ways the i.LON 100 can be utilized to access building automation data. The i.LON 100 web service provides an interface to a *data server* which allows to read and alter building automation data. The data server operates on *data points*, which represent data located in the attached building automation system. The i.LON 100 differentiates between several types of data points. It was decided to use *NVL* (local data points), because they require an established *binding* between local *network variables* and remote[1] network variables. The binding allows reading of remote data without polling and ensures that local values are propagated to remote network variables. For an evaluation of the different data point types see Section 3.3.3.

Data points have to be created on the i.LON 100 before the data server can operate on them. NVL data points can be created by creating a *dynamic network variable* on the i.LON 100. For every new dynamic network variable a NVL data point is created and assigned to the

[1] The network variables are remote from i.LON 100's point of view.

new dynamic network variable. Therefore altering the NVL data point directly affects the assigned dynamic network variable.

The nodes integrated in a *LonWorks* based building automation system comprise network variables. To actually interact with the building automation system these remote network variables have to be bound to the dynamic network variables of the i.LON 100. By doing this, changing the value of the NVL data point will change the value of the assigned dynamic network variable, which will cause an update of the bound remote network variable. Figure 5.1 shows an example configuration where remote network variables (`nvi_L1` and `nvi_L2` on the remote node) can be altered via bound dynamic network variables (`nvi_L1` and `nvi_L2` on the i.LON 100) which again can be altered via its assigned NVL data points (`NVL_nvi_L1` and `NVL_nvi_L2`).

The remote network variables may[2] also be bound to other network variables, therefore their value may be changed by other applications too. To avoid inconsistencies among the applications dealing with the same data point, most nodes provide *feedback* output network variables, which can be used to retrieve the current value of a corresponding input network variable. In Figure 5.1 every input network variable on the remote node (`nvi_L1` and `nvi_L2`) has a corresponding feedback output network variable (`nvi_L1_fb` and `nvi_L2_fb`). Another example for feedback network variables is the lamp actuator functional profile, shown in Figure 3.6.

Most applications utilizing an i.LON 100 for interacting with building automation systems may require knowledge about the current value of a network variable. Therefore in this configuration dynamic network variables are created for both, the actual remote network variable which has been chosen to be used via the i.LON 100 web service and the corresponding feedback network variable. Figure 5.1 shows some remote network variables with their corresponding feedback network variables and how they are bound to dynamic network variables on the i.LON 100. The dynamic network variables are assigned to NVL data points which can be read and altered via the i.LON 100 web service.

Therefore to allow an external application to interact with the building automation system attached to the i.LON 100, dynamic network variables have to be created for every remote network variable which should be included in the interaction and its feedback network variable. Afterwards bindings have to be established between the dynamic network variables and the remote network variables. Both of these steps have to be done by a LonWorks-enabled system integration tool. Section 6.1 describes the configuration of the i.LON 100 with the *LonMaker* system integration tool.

5.1.2 Hiding the i.LON 100 Web Service

Section 3.4 describes the web service provided by an i.LON 100 and how it can be consumed by a client. Consuming a web service is a quite big and tedious task. Web service frameworks as described in Section 3.5 provide functionality to support the developer in consuming web services. The web service framework JAX-WS (*Java API for XML – Web Services*) was

[2] Actually it is very likely, that the used remote network variables are bound to other network variables, because they are needed to provide the day-to-day building automation functionality. One should not forget that the building automation system was not implemented to be utilized by a media façade, but to provide functionality which is required and expected by everyone who spends her time in the building. Therefore the network variables of every lamp node and every sunblind node are for sure bound to a switch node.

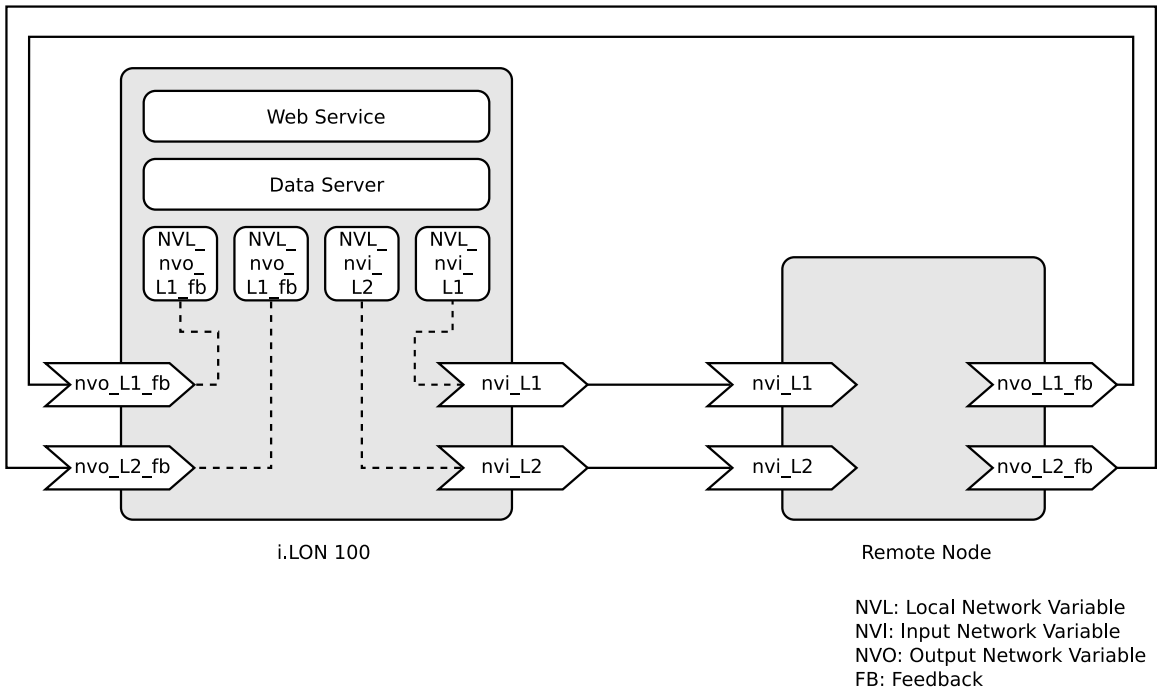

Figure 5.1: Relation of NVL data points, dynamic network variables and remote network variables. The dynamic network variables on the i.LON 100 are bound to remote network variables and therefore can be used to alter them. The current value of a remote network variable can be retrieved via its associated feedback network variable. The dynamic network variables can be read and altered via its assigned NVL data point.

chosen for further utilization, because it is already included in the Java 6 distribution and does not error-out while consuming the web service of the i.LON 100. See Section 3.5.3 for a comparison of web service frameworks and detailed reasons why JAX-WS was chosen for utilization.

The tool `wsimport`, distributed with JAX-WS, is able to transfrom the WSDL (*Web Service Description Language*) description of a web service into Java classes which can be utilized to consume the respective web service. For the i.LON 100 web service, the number of classes generated by `wsimport` is 317. The most interesting of them are request and response classes for every function provided by the web service, for example the classes `DSRead` and `DSReadInfo` for invoking the web service function `DataServer_Read()` (Section 3.4.3.2) and retrieving the response of the `DataServer_Read()` web service function. Listing 3.20 shows how these classes are utilized. Most of the other classes generated by `wsimport` are just representations of sub elements in the request and response SOAP (*Simple Object Access Protocol*) messages like the class `DPType` which is also used in Listing 3.20.

Only some of the generated classes are required for invoking a specific web service function. But even so, quite a few different classes are required to consume a web service. Additionally, the web service has to be consumed over and over again to alter and read values of assigned network variables, maybe even from different software modules.

To avoid code duplication amongst software modules, it was decided to implement an abstraction layer to hide the i.LON 100 web service consumption. An abstraction layer, which hides the specifics of the i.LON 100 web service, also allows a possible implementation of similar abstraction layers for other building automation gateways in the future. If the abstraction layers provide a similar interface the abstraction layers can be exchanged easily, enabling interaction with different building automation technologies.

The name of the Java package which contains the abstraction layer for the i.LON 100 web service is `ilonif`. The name indicates that the package is a software interface (also called API – *Application Programming Interface*) for the i.LON 100 web service. The `ilonif` abstraction layer also hides the steps of building a request SOAP message as shown in Listing 3.20 and the evaluation of the return value when consuming a web service function.

The classes which represent request SOAP messages have to be initialized with information about the data point on which the web service function should be invoked. Also the classes which represent response SOAP messages contain information about the data point on which the web service function has been invoked. The SOAP messages of the i.LON 100 web service are described in Section 3.4.

The abstraction layer `ilonif` therefore requires information on data points and returns information on data points. To ease the exchange of parameters and return values it was decided to implement a Java representation of data points as a Java class. The Java representation is called `DataPoint` and is discussed in Section 5.1.3.

When invoking a method of `ilonif` which starts a web service consumption, the method will require at least one `DataPoint` instance. The request SOAP message will be constructed by using the information contained in the `DataStorage` instance. The information in the response SOAP message will be stored in the same `DataStorage` instance. This approach ensures that information belonging to a specific data point is not spread around at different places.

The i.LON 100 web service comprise 44 web service functions [Ech06c, p. 3-3]. The *data server* related web service functions are able to read and alter data points. But only three of the data server related functions are required in this work. The required web service functions are: `DataServer_Get()`, `DataServer_Read()` and `DataServer_Write()`. The first one for retrieving all available configuration data of specific data points, the other two for reading and altering the values of data points. The three functions are discussed in Section 3.4.3.

At least for the required web service functions `DataServer_Get()`, `DataServer_Read()` and `DataServer_Write()` respective methods have to be implemented in `ilonif`. The implementation of the `ilonif` abstraction layer is described in Section 6.2.4.

5.1.3 Handling Data Points

As already mentioned in the previous section a representation of a data point as Java class is required. The class is named `DataPoint` and contains all relevant information on a data point.

Information on a data point can be retrieved from the i.LON 100 in two ways. Either by invoking a web service function which contains information on the data point in its response or by downloading the file `dp_NVL.xml`. See Listing 3.9 for an excerpt of this file.

The file `dp_NVL.xml` contains configuration values of a data point. The same information can also be retrieved by consuming the `DataServer_Get()` web service function. The relevant information which class `DataPoint` should contain is shown in Table 5.1.

Source	XML Tag Name	Description
`DataServer_Get()`	`UCPTindex`	Unique Index of the Data Point
	`UCPTpointName`	Name of the Data Point
	`UCPTdirection`	Designates Input or Output
	`UCPTformatDescription`	Designates the Data Point Type
	`UCPTunit`	Unit of the Value
`DataServer_Read()`	`UCPTvalue`	Current Value of the Data Point
	`UCPTvalueDef`	Current Value as String
	`UCPTpointUpdateTime`	Time of Value Update
`DataServer_Write()`	None	No return Value on Success
All	`UCPTfaultCode`	Fault Number
	`UCPTfaultString`	Fault Message

Table 5.1: Relevant information about data points. The source row indicates where the information can be retrieved from. XML tag names refer to the tags in the respective response SOAP messages. A small explanation of the tags is given in the description row.

The class `DataPoint` has to contain the current value of a DataPoint. The value will be updated whenever the method of the abstraction layer `ilonif` is invoked which consumes web service function `DataServer_Read()`. The method of `ilonif` will parse the response value of `DataServer_Read()` and write the value into a `DataPoint` instance.

Also the class `DataPoint` has to store the value which should be written to the data point. This value is required when consuming the `DataServer_Write()` web service function. For every web service consumption, fault messages have to be stored in `DataPoint` if an error occurs.

5.1.4 Hiding Data Points

The Java package `ilonif` provides functionality to ease the consumption of the i.LON 100 web service. However `ilonif` still uses the concept of data points, which should not be visible for users and applications usually dealing with media façades. Artists and developers who design the content of media façades typically have no (or very little) knowledge about building automation in general and LonWorks in particular. An API which is familiar for people dealing with media façades and which hides the underlying building automation technology is therefore required.

Typically media façade control systems provide a lot of possibilities to upload content. For example they may allow movies, images and/or may expose an API to define and provide content. However most of them do have on some level (not necessarily publicly exposed) a *bitmap* based interface. Bitmaps are raster graphic images which are typically represented in a rectangular grid of *pixels*.

Pixels have attributes like color and brightness. A bitmap can be shown on displays by mapping the pixels of the bitmap to the pixels of the display. A simple example of a bitmap is shown in Figure 5.2. The shown bitmap has only 16x16 pixels, but still simple images are recognizable. A bitmap based approach can also be applied to façades of buildings. The windows of the building are used as pixels, the whole façade acts as the display. To use the façade of a building as bitmap based display is not an entirely new idea. Something quite similar has been done in the project *Blinkenlights*[3], but in this work no modification of the building is required. The already deployed building automation system of the building is used. Furthermore the approach allows easy – and bitmap oriented – coordination of the content shown on the media façade and the content displayed by the windows of a building.

To provide a convenient interface to people dealing with media façades it was decided to implement a Java package which provides a pixel oriented API to its users. The name chosen for the Java package is `pixelif`.

The package `pixelif` should provide "drawing" functionality in a pixel based manner, which is familiar to users of media façades. Therefore drawing methods should require a data structure as parameter, which is similar to bitmaps. Bitmaps are usually represented as a two-dimensional array of pixels. The same approach can be used for the "draw" methods of `pixelif`. Every "draw" method of `pixelif` should require a two-dimensional array comprising the value of pixels. The position of an array element maps to a position of a window on the façade. The value of the array element is applied to the respective room lighting behind the window.

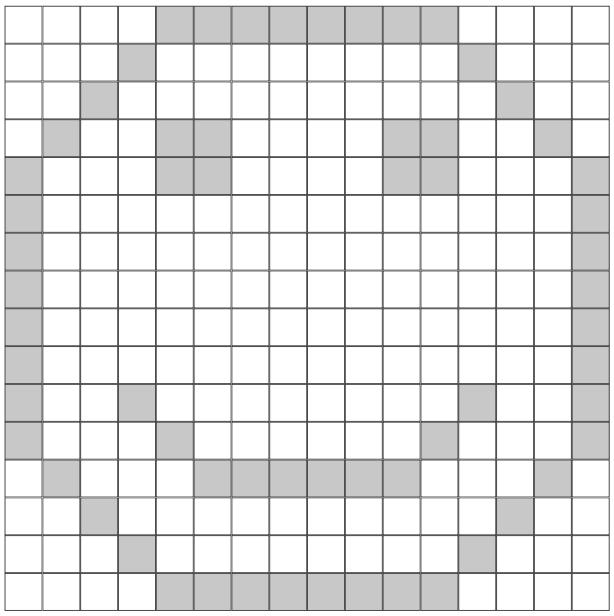

Figure 5.2: A simple bitmap graphic. With only 16x16 pixels simple images are recognizable. A bitmap based approach can be applied to façades of buildings. The windows of the building are used as pixels, the whole façade acts as the display.

[3] `http://www.blinkenlights.de/`

Beside altering actuators in the building automation system the package `pixelif` should also be able to retrieve values of specific switches. For example if a switch is designated to turn on and off the media façade, it would be nice if the media façade could retrieve the value of the switch via `pixelif`. In Java the typical approach to handle reactions on events is to implement the *Observer Pattern* [GHJV05] in the form of *event listeners*. This approach can be reused for `pixelif`. An event listener can register itself at a `pixelif` instance. Every time the observed switches change their value the registered event listener will be notified.

5.1.5 Beyond Data Points

The previous section described an approach to reuse a bitmap based API for altering data points of a building automation system. Bitmaps consist of pixels, each pixel at a specific position in the bitmap. Pixels therefore have coordinates. As the pixels of the bitmap are mapped to data points in the building automation system, the concept of coordinates has to be introduced to data points.

The class `DataPoint` acts as representation of data points. To add the concept of coordinates to `DataPoint` it is sub-classed. The name of the sub-class is `Pixel`. It comprises some information relevant for pixels, for example the coordinates. The class `Pixel` also contains information on the feedback data point as described in Section 5.1.1.

In Chapter 2 the possibilities of media façade and building automation cooperation were discussed. As a result several actuators integrated in building automation systems have been identified as possible supporters of the effects generated by the media façade. Examples are the room lighting and the sunblinds on a building. Therefore the class `Pixel` should also contain the type of the data point it represents.

With the above mentioned information (coordinates, feedback data point and the type of the pixel) a bitmap based API can be implemented in `pixelif`. The implementation of `pixelif` and `Pixel` is described in Chapter 6.

5.1.6 Storage for Data Points

Both `ilonif` and `pixelif` require access to the available data points. As all information is stored in the respective `DataPoint` objects it has to be ensured that `ilonif` and `pixelif` operate on the same `DataPoint` objects. Otherwise information could get lost or inconsistent among the various software modules.

A central place where data points are stored and where they can be retrieved from is required. The Java package which provides this functionality is called `datastorage`. It is able to store an arbitrary amount of `DataPoint` and `Pixel` objects.

The package `datastorage` also provides functionality to search and retrieve data from the storage. Parameters for which a search should be possible are the coordinates of a pixel, the name and index of data points and the type of data points (input/output) and pixels (light/sunblind).

On queries `datastorage` should only return references to `DataPoint` objects. This ensures that all users of `datastorage` operate on the same set of `DataPoint` objects and that information on datapoints is stored in only one instance of `DataPoint`.

5.1.7 Summary

In the previous sections three Java packages have been introduced: `datastorage`, `ilonif` and `pixelif`. Figure 5.3 shows schematically how they relate to each other.

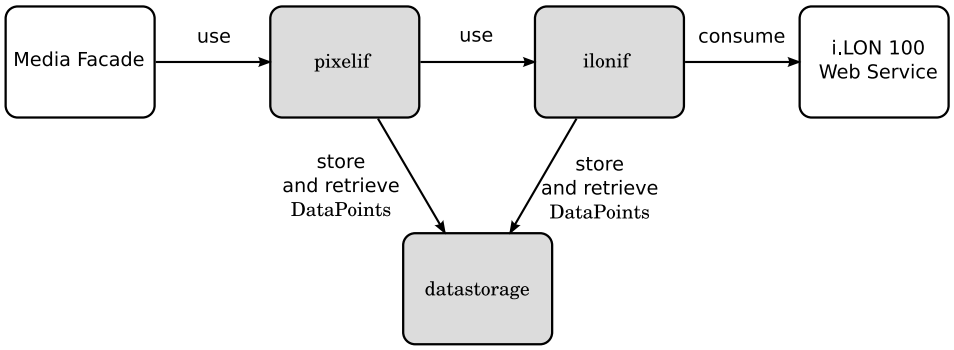

Figure 5.3: Implemented software modules and their relation. A media façade can utilize the bitmap-based API of `pixelif`. Internally the package `pixelif` uses `ilonif` as abstraction of the i.LON 100 web service and `datastorage` to store and retrieve `DataPoint` objects.

On the right side of the figure is the i.LON 100 web service located. The purpose of package `ilonif` is to deal with the consumption of the web service. The software contained in package `pixelif` uses `ilonif` and therefore does not have to care about the specifics of the i.LON 100 web service. Both, `pixelif` and `ilonif`, utilize `datastorage` to store and retrieve `DataPoint` objects. Ultimately the pixel-oriented API provided by `pixelif` is used by the media façade system as can be seen on the left side of Figure 5.3.

5.2 Utilization of the Artificial Recognition System

In Chapter 4 the *Artificial Recognition System* (ARS) has been introduced. The ARS project consists of two sub-projects: ARS-PA (*Psychoanalysis*) and ARS-PC (*Perception*). Both systems have been evaluated in Section 4.1.3 which one would be suitable to be used in this master thesis. It was decided to utilize ARS-PC. The goal of ARS-PC is to implement algorithms which allow "perception" of situations and scenarios. Perception is done by a process called *symbolization*, which refers to condensing great amounts of data with small informational value into data with higher informational value. Such an information condensing process suits exactly the requirements for a system which is able to transform building automation data into a format suitable for media façades.

5.2.1 Initial Plan

One of the goals of this master thesis was to provide pre-processed building automation data to media façade systems. For the pre-processing the ARS-PC symbolization technology is

used. The original plan was to feed data retrieved via the i.LON 100 web service into the ARS-PC symbolization process and to forward the resulting information to the media façade control system.

In theory ARS-PC would be able to accept any kind of input data and to transform it into data with higher informational value called *symbols*. However ARS-PC is still a young project and the symbolization concept in its current implementation is merely a proof that symbolization is a functional approach. Therefore it was clear that the utilization of ARS-PC in the intended way would not be straight-forward and that problems would arise.

5.2.2 Problems

One problem of the current implementation of ARS-PC is its lack of configurability. The different available input sources (*live*, *database* and *simulation* as described in Section 4.2.1) are not configured by some sort of configuration environment, but are hard-coded in their respective start-up routines. If a user wants to use the database as input source for ARS-PC the corresponding application has to be launched which sets up the internals for this use case. For each existing input source a corresponding application exists.

Additionally to the separated start-up sequences of ARS-PC, for each input source a corresponding micro symbol factory exists. The micro symbol factories retrieve values from their respective input sources, transform the values into *micro symbols* and forward them to the symbolization process as described in Section 4.2. Adding a new input source to ARS-PC therefore requires the implementation of a start-up application and a micro symbol factory.

Another problem is the current number of scenarios detected by ARS-PC. [Ric07] describes scenarios currently implemented in ARS-PC, for example “meeting” and “person makes coffee”. Most of the implemented scenarios are similar to those two, as they address situation with single persons or a small number of persons. At the moment no scenarios which detect situations with a great number of people respectively all people in a building are implemented. But exactly such scenarios would be of interest for media façade systems because they give a more abstract view on situations in the building.

The currently implemented recognition of scenarios depends on knowledge of the position of persons. This knowledge is gained via tactile sensors mounted on the floor of the *Smart Kitchen* [SRT00] at the Institute of Computer Technology (ICT). Such sensors are usually not (yet) deployed in today’s buildings. To utilize ARS-PC in today’s buildings the recognition implementation either have to be ported to other sensors or new scenarios have to be defined and implemented which are based on sensors available today.

An implementation of the original approach – retrieving sensor values via the i.LON 100 web service, feed the values into ARS-PC and to forward the resulting high level symbols to the media façade system – would require a implementation of a micro symbol factory, its corresponding start-up application and the actual symbolization to recognize scenarios adequate for the available sensors. Especially the implementation of a micro symbol factory and the recognition of additional scenarios are a huge amount of work and it was decided that such an implementation would go beyond the scope of this master thesis. However the implementation may be part of future work.

5.2.3 Solution

Because of the above stated problems it was decided to not actually use the sensor values available via the i.LON 100 web service. Instead another solution to provide information on the environment to media façade systems had to be found.

It was decided to not abandon ARS-PC as its concepts are scientifically sound and further development on ARS-PC and evolution of building automation systems may allow utilization of ARS-PC in typical buildings in the future. Therefore ARS-PC is used in this master thesis in its current form to show as a proof of concept how information retrieved from building automation data can be forwarded to and utilized by media façade systems.

As input source for ARS-PC the database configuration is used, because it ensures that data is available and symbols are generated. To enable this a launch configuration has to be implemented, which sets up ARS-PC as required. Furthermore a filter, respectively abstraction layer, is required which receives symbol messages from ARS-PC on the one side and delivers notifications to the media façade system. Figure 5.4 shows this abstraction layer named `arsif` – following the style of `pixelif` and `ilonif` – between ARS-PC and a media façade system.

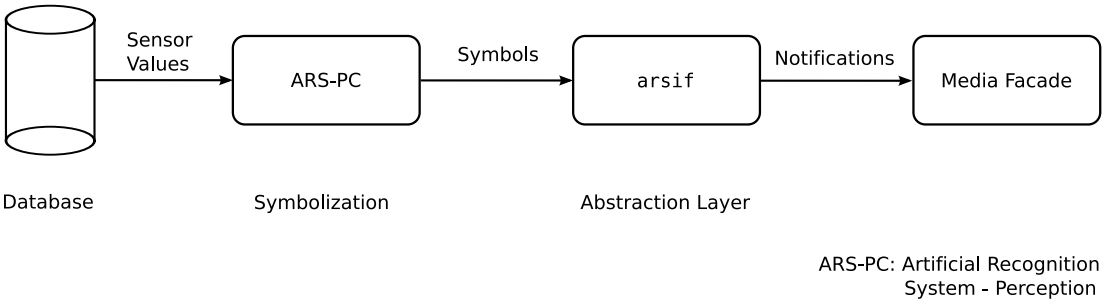

Figure 5.4: Role of the `arsif` abstraction layer. As input source of ARS-PC a database is used. ARS-PC processes the sensor values stored in the database and generates symbols. The symbols are received by `arsif`. If relevant information is found among the received symbols the media façade is notified.

Receiving symbols generated by ARS-PC can be done by utilizing *SymbolNet* (see Section 4.2.4). SymbolNet provides functionality to send, receive and manage symbols. The specifics of SymbolNet should be hidden from media façade systems. For this reason *arsif* acts as abstraction layer between ARS-PC and media façades as shown in Figure 5.4. The abstraction layer `arsif` is implemented in a Java package with the name `arsif`. It provides two interfaces, one SymbolNet-based interface to receive symbol messages and a second interface to forward information on symbol updates to the media façade.

The interface for media façade systems is similar to `SwitchListener` in `pixelif` (Section 5.1.4) based on the Observer Pattern [GHJV05]. The usual approach to the Observer Pattern in Java is to implement Listeners which will be notified when events occur. In the case of `arsif` and media façade systems it implies that the media façade system implements a listener which has to be registered in `arsif`. When `arsif` receives symbol updates via its

SymbolNet-based interface it will notify the registered listener, in this case the media façade control system.

5.3 Integrating Media Façades

The media façade is the consumer of the APIs (`pixelif` and `arsif`) introduced above. As described in Section 2.2.2 a *BLIP*[4] media façade control system consists of *Display Managers* and *Image Generators*. The Image Generator provides power to the LEDs and control color and brightness of LEDs by issuing commands over a communication bus. For big media façade installations a Display Manager is required, which is responsible for coordination of various Image Generators.

In this work no Display Manager will be used, as it is only required for really large media façade installations not for such a prototype implementation as it is the goal of this master thesis. To integrate media façade and the two above introduced software packages `pixelif` and `arsif` two options are possible.

1. The software packages `pixelif` and `arsif` are executed on a separate computer system. For communication between the Image Generator and the software packages additional software is required.

2. The software packages `pixelif` and `arsif` are executed on the Image Generator. The control software on the Image generator can consume the APIs of the software packages locally. No – or at least less – additional software is required.

The comparison of the two option shows that option number two has the advantage of lesser implementation effort. Also option two eliminates the need for an external computer system. Therefore option two is favored over option one. However if option two is feasible depends on the capabilities of the Image generator.

As the Image Generator has not been delivered by BLIP in time[5], it was not possible to implement option two. Also without an Image Generator it is not possible to actually activate the media façade LEDs, which *have* been provided by BLIP. the LEDs of the media façade.

Because of the lack of an Image Generator it is not possible to activate the LEDs of the media façade. Furthermore a consumer of the APIs of `pixelif` and `arsif` is missing. Therefore a substitute for the functionality of the Image Generator had to be found. It was decided to use a small application to act as replacement of the Image Generator as consumer of the APIs of `pixelif` and `arsif`. The functionality to control the LEDs of a media façade over a communication bus is not implemented as such an implementation would go beyond the goals of this master thesis and because the hope that an Image Generator will be delivered some time in the future has not yet been abandoned. The implementation of the application which acts as the consumer of the APIs of `pixelif` and `arsif` is discussed in Section 6.5.

[4] `http://www.blipcreative.com/`
[5] Till now, while the finishing touches are done on this master thesis, no Image Generator has arrived.

5.4 Demonstration Environment

A big part of the effort in software development is testing. Testing the software components described above is not easily arranged, because the components require a specific environment. On one hand ARS requires its sensor database as input source. On the other hand the building automation interface requires its associated hardware, which is at least an i.LON 100 (and a building automation system it is integrated to, otherwise the i.LON 100 is not of much use). Also a media façade is required to actually demonstrate the cooperation between building automation and media façade.

For the building automation interface an i.LON 100 and a building automation system is required. Buying an i.LON 100 is, despite its price[6], an easy solution to provide the first part of the testing environment. However a LonWorks based building automation system is required to integrate the i.LON 100. Without an attached building automation system the i.LON 100 is of no real use. Therefore the i.LON 100 was integrated into the available LonWorks system in the *Centre of Excellence for Fieldbus Systems*[7] at the ICT at first. But using the Centre of Excellence for Fieldbus System was only a temporary solution because it is used by other projects and in university courses and could not be allocated for the media façade project alone. Therefore another building automation system had to be found.

To find a testing environment which includes a media façade is rather difficult to manage. Media façades are still sparsely deployed and the owner of such media façades are quite reluctant to allow access to it for testing purposes.

The solution to those problems was to build a small testing environment which includes at least parts of a media façade and parts of a building automation system. Inclusion of sensors for ARS-PC was discarded because ARS-PC requires a great amount of sensors to provide its functionality. Including a great number of sensors would basically lead to a reimplementation of the Smart Kitchen (or parts of it) in our testing environment, which would contradict the idea of a *small* testing environment. This is one reason why the database configuration is used for ARS-PC. If the database configuration is used the sensor values are retrieved from a sensor database not directly from the Smart Kitchen. In the future it may be possible to locate the sensor database on an computer system in the demonstration environment, for example the Image generator.

For various reasons which are explained in the next section it was decided to integrate the testing environment into a suitcase. The components of the suitcase, the way it operates and the required software are described in the following sections.

5.4.1 Purpose and Requirements

As already outlined above the main purpose of the demonstration suitcase is to be used as testing environment for the developed software. Especially the building automation and media façade related software requires an environment to test the software. For the ARS-PC

[6] In the Echelon Product Price List, available at `http://www.ebv.com/media.php/EBV/Products/Hot%20Adverts/PDF/Lonworks%20Area/ResaleLonWorksFeb2007.pdf?dl=1` the i.LON 100 is listed with a price of 595 US$.

[7] `http://www.ict.tuwien.ac.at/komzent/`

related parts the sensor database can be used. Therefore hard requirements for the testing environment were to include (parts of) building automation and media façade technology.

The testing environment should be able to present the cooperation building automation and media façades are capable of with the newly developed system. Options like utilizing the existing building automation system at the ICT and to mount a small part of a real façade somewhere at the ICT and to use these parts as representation and testing environment were considered, but discarded as they were deemed as too expensive and would require too invasive changes in the furnishing of the ICT. Therefore it was decided to build a portable testing environment. The advantages of a portable solution include a lower price than a mounted part of a real façade, an easier handling and utilization as it has to be contained to be portable and portability itself. Portability of the testing environment allows to present the cooperation of media façades and building automation systems at shows, at conferences or to potential purchasers of media façades. It was decided to integrate the testing and presentation environment into a suitcase. A suitcase was chosen as platform because its quite common practice to present prototypes and inventions in suitcases. Figure 5.5 gives an overview of the design and the components included in the demonstration suitcase.

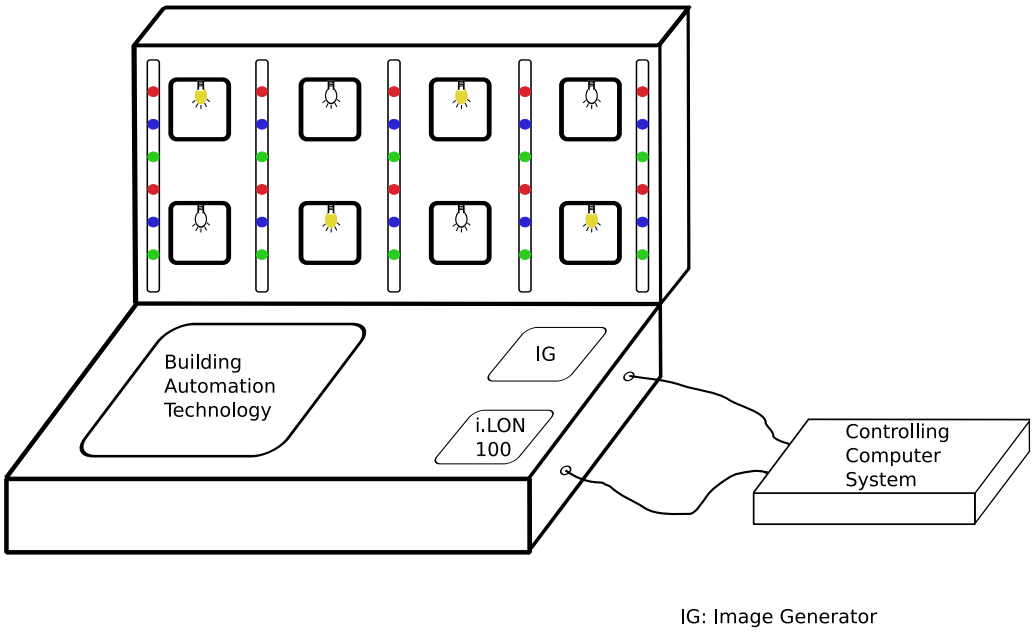

Figure 5.5: Draft of the demonstration suitcase. The top cover of the suitcase represents a façade of a building. The lights behind the windows are used to represent building automation technology. A media façade is mounted between the windows.

The top cover of the suitcase is used to represent a building façade while the base part of the suitcase contains the required technical components. The impression of a building is generated by applying representations of windows to the top cover of the suitcase and by coloring the front in a building-typical way. A requirement of the testing environment is to include a building automation system. It was decided to include a representation of room

lighting in the demonstration suitcase. The room lighting can be simulated by turning on and off lights behind the windows on the top cover of the suitcase.

As outlined in Chapter 2 not only influencing the room lighting is reasonable for media façades, but also sunblinds. Sunblinds can be used to either prevent interference between room lighting and the effects generated by the media façade or to generate effects with the sunblinds itself. Therefore it was decided to also integrate a representation of sunblinds into the demonstration suitcase. The room lighting and the sunblinds are integrated in a small LonWorks-based building automation bus, in which an i.LON 100 is also integrated.

Figure 5.5 shows that a media façade is applied to the building façade represented by the top cover of the demonstration suitcase. The media façade is aligned in "lines of light" as described in Section 1.1.3 and the lines are located between the windows. The lights of the media façade would be controlled by an *Image Generator*.

In Figure 5.5 an external "controlling computer system" is shown. As discussed in Section 5.3 if this external system is required, depends on the capabilities of the Image Generator. However as long as no Image Generator is delivered by BLIP the external system is required for sure. The demonstration application introduced as replacement for the Image Generator functionality in Section 5.3 is executed on the external computer system.

5.4.2 Components

The building automation system in the demonstration suitcase should provide means of simulating room lighting and sunblinds. Simulating room lighting can be easily done by mounting LEDs (*Light Emitting Diodes*) behind the window representations on the top cover of the suitcase.

To simulate sunblinds several approaches were evaluated. Utilizing mechanical systems as they are available for model building were abandoned because they may be easily damaged on rough transports the suitcase may have to endure. Another considered approach was to use simple, monochrome *Liquid Crystal Displays* (LCD) with backlights. The LCD solution was also abandoned because no applicable LCDs could be found and because the resulting appearance would not resemble a real façade of a building.

The solution chosen to represent sunblinds is simple but effective. In Section 2.2 it has been shown that sunblinds may assist media façades because if the sunblinds are closed no room lighting can interfere with the effects of the media façade. Therefore it was decided to use bright room lighting to represent that the sunblinds are up and to lower the brightness of the room lighting if the sunblinds are down. The reduced brightness as sunblinds simulation may not be the most intuitive solution but with some explanation the audience will understand the principle.

The simulation of sunblinds with different levels of brightness of room lighting allows to reuse the LEDs used for room lighting. The LEDs in the "rooms" just have to be supplied with different levels of electrical current for every state of a window. The four possible states of a window are shown in Table 5.2. The states of a window have only three distinguishable visual appearances, because with the chosen sunblinds representation the position of the sunblind can only be shown when the light is switched on.

	Sunblinds Up	Sunblinds Down
Lights On	bright light in window	damped light in window
Lights Off	no light in window	no light in window

Table 5.2: States of the windows in the demonstration suitcase. The lights can be dampened by lowering the sunblinds, but the position of the sunblinds is not visible if the lights are off.

Switching on and off the lights and the sunblinds should be done via the building automation system, as they represent parts of a real world building automation systems. For LonWorks the lamp actuator profile, as shown in Figure 3.6, has been designed for such purposes. A node which implements have to be found and integrated into the suitcase. The chosen nodes are described in Section 6.6.1.

The circuitry required to to show different brightness levels with a LED is shown in Figure 5.6. The switches S1 and S2 are part of a node implementing the lamp actuator profile as discussed above. Switch S1 is assigned to switching on and off the light in a room. Switch S2 is closed per default, when it opens the light of LED D is dampened, therefore S2 represents the sunblind functionality. For every window in the top cover of the suitcase two switches are required to implement the desired functionality.

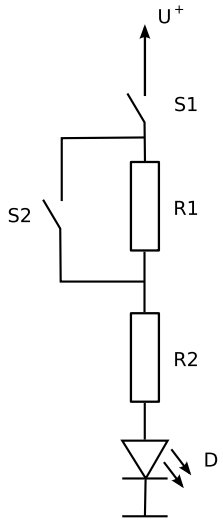

Figure 5.6: Circuit to show different brightness levels with one LED. Switch S2 is closed per default. If switch S1 is closed the LED D will produce bright light. When S2 is opened the light of LED D will be dampened.

In real world building automation systems switches are integrated to allow users to take control of some aspects of the building automation system. For turning the light on and off at least one push-button is required. For altering the position of sunblinds at least two push-buttons are required, one for moving the sunblind up and one to move the sunblind down. Three push-buttons per window, requires 48 push-buttons integrated in the demonstration suitcase, to build an building automation system which can control every window. Such an amount of push-buttons requires a lot of space which contradicts the idea of a small and

portable testing environment. Therefore it was decided to integrate only push-buttons for one window. An additional push-button is available in the demonstration suitcase, which allows to turn on and off the cooperation of media façades and building automation system. The functionality of this push-button demonstrates already the cooperation of media façade and building automation. The push-button integrated into the building automation system can influence the projected content of the media façade.

To complete the building automation related part of the demonstration suitcase an i.LON 100 and a node which implements the *LonMark* switch profile (Figure 3.6) have to be integrated into the suitcase. The push-buttons described above have to be connected to the switch profile node, which provides network variables to interact with the switches. The i.LON 100 also has to be integrated into the building automation system to allow users of its web service to interact with building automation data. The integration of both parts, switch profile node and i.LON 100, are discussed in Section 6.6.1.

5.4.3 Operation

The demonstration suitcase should be able to demonstrate the cooperation of building automation systems and media façades. Additionally it should provide a way to demonstrate some aspects of building automation. The new concept of building automation and media façades cooperation has to be introduced to architects and media designers, which have only little knowledge how building automation works. Therefore the demonstration suitcase should be able to demonstrate basic aspects of building automation technology to them.

To demonstrate building automation technology and the cooperation between building automation and media façades the demonstration suitcase is able to operate in two modes. In one mode, called *conventional operation*, the building automation system and the media façade are operating completely independent. The media façade projects its typical content without being influenced by the environment or the building automation system. One window's light and sunblind can be controlled by three push-buttons integrated into the suitcase. One push-button is for turning the light on and off, the other two are for moving up and down the sunblind. The position of the sunblind is simulated by the brightness of the light in the window as described in Section 5.4.2. A fourth push-button is available which can be used to change the operational mode of the demonstration suitcase.

If the second operational mode of the demonstration suitcase, named *effect operation* is activated, media façade and building automation system are cooperating to produce visual effects on the façade of the building. For example when the media façade produces an ascending impression by activating first the LEDs at the bottom then the LEDs above and so on, the building automation system can support such an effect by switching on the lights in the windows in a similar way.

Chapter 6

Implementation

The previous chapter described the design of the prototype system, which will be used to demonstrate the cooperation of building automation systems and media façades. The implementation of the demonstration system and its operation is discussed in this chapter.

6.1 i.LON 100 Configuration

In Section 5.1.1 dynamic *network variables* and their assigned *NVL data points* which have to be created on the i.LON 100 are discussed. For every remote network variable which should be incorporated into the media façade cooperation functionality, a dynamic network variable of same type has to be created on the i.LON 100. Between the created dynamic network variable and the remote network variable a *binding* (see Section 3.2) has to be established. Also for most of the remote network variables a designated feedback network variable exists which should also be bound to dynamic network variables on the i.LON 100. The proposed configuration is shown in Figure 5.1.

A functional block of the i.LON 100 has to be present in the *LonMaker* schematic to create dynamic network variables in it. The *virtual functional block* of the i.LON 100 proved to be the most adequate functional block of the i.LON 100 to create dynamic network variables in. Other functional blocks were also tested, but long names of network variables were truncated when created on these functional blocks. The truncation of names does not happen when the virtual functional block is used. A dynamic network variable is created in LonMaker by dragging an *Input Network Variable* or *Output Network Variable* symbol from the toolbox onto a functional block in the LonMaker schematic. The toolbox and the symbols are shown in Figure 3.5.

By creating the dynamic network variables also NVL data points are created. Their existence and configuration can be checked with the *i.LON Configuration Utility* described in Section 3.3.2.

After creating dynamic network variables, bindings to remote network variables have to be established. Establishing a binding between network variables can be done in two ways in LonMaker:

- Dragging a *Connector* symbol from the toolbox onto the LonMaker schematic and attaching the ends to the according network variables. This procedure only allows to establish a binding between two dynamic network variables at once.
- The second way is to select the *Connect...* entry in the right-click menu of a functional block. The following dialog allows to establish more than one connection at once.

After establishing bindings to all required remote network variables a user of the i.LON 100 web service is able to read and alter remote network variables. In order to avoid confusion a inconsistency in the naming of dynamic network variables is explicitly stated. The direction of the dynamic network variable and its bound remote network variable are reversed, because a binding can only be established between an output network variable and an input network variable. Therefore for every remote input network variable a dynamic output network variable (respectively output NVL) has to be created and vice versa.

The naming of the created dynamic network variable can lead to confusion because when new dynamic network variables are created LonMaker asks for an already existing network variable as template. Typically the remote network variable to which the dynamic network variable will be bound later is used as template. The template is needed to ensure that the newly created dynamic network variable is of the same type as the remote network variable. After choosing the template network variable a name similar to the name of the template network variable is suggested and this suggested name might cause confusion. It is a somewhat widespread convention to indicate the direction of a network variable in its name. Therefore a lot of network variable names contain either *nvi* or *nvo* indicating input respectively output network variables.

If the suggestion in naming the dynamic network variables is adopted the direction of the dynamic network variables conflict with its naming. On the other hand the dynamic network variable is similarly named as the bound remote network variable and that eases finding assigned pairs of network variables. Because of the eased association of dynamic network variable and remote network variable the suggested (and slightly confusing) naming of dynamic network variables was adopted. In Figure 6.1 such a configuration is shown. In the figure the output network variables on the functional block of the i.LON 100 on the left have the term nvi in their name altough they are outputs. Similar the input network variables on the i.LON 100 have the term nvo in their name. But this inconsistency allows easier mapping between the network variables on the i.LON 100 and the remote network variables.

6.2 Building Automation Interface

Even though the JAX-WS (Java API for XML – Web Services, see Section 3.5.2) web service framework is used to facilitate web service consumption, programming code to invoke a web service function is still tedious and error-prone. An abstraction layer was desperately needed to hide the actual web service invocation, the return value evaluation and error handling. Such an abstraction layer was developed as a Java package named `ilonif`. The name indicates that the package is a software interface (also called API – *Application Programming Interface*) for the i.LON 100 web service.

The package `ilonif` operates on `DataPoint` objects which are Java classes designed to represent data points of the i.LON 100. As discussed in Section 5.1.6 a central place for storing and

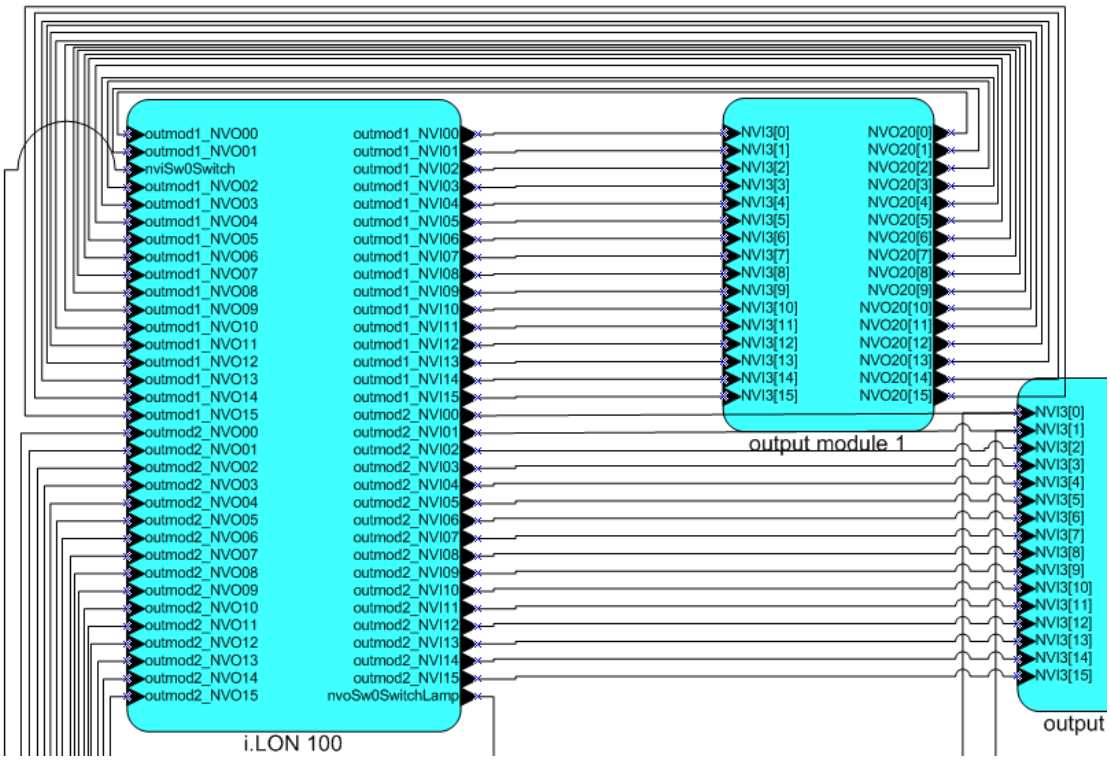

Figure 6.1: Network variable naming and binding. The functional block on the left is the *virtual functional block* of the i.LON 100. It comprises dynamic network variables which are connected to network variables on remote nodes.

retrieving information on data points is required, to avoid information to be spread to different places. The central storage functionality is provided by the Java package `datastorage`. The packages `datastorage` and `ilonif` and its associated classes are described in the next sections.

6.2.1 Java Representation of Data Points

Data points are abstracted entities which allow interaction with the underlying building automation system. NVL data points represent local dynamic network variable on the i.LON 100. By establishing a binding – with a LonWorks enabled system integrator tool – between local network variables of the i.LON 100 and remote network variables located on nodes connected to the building automation system, it is possible to read and change the value of the remote network variable via the i.LON 100 web service.

The web service of the i.LON 100 operates on data points. The consumer of the web service has to specify which operation has to be performed on a set of data points and has to provide information about the set of data points. The class `DataPoint` was developed to be used as a representation of a data point and to store relevant information about data points.

Information on NVL data points located on an i.LON 100 is provided in XML files. The information can be retrieved by either downloading the `dp_NVL.xml` file, which is described

in Section 3.3.3, from the i.LON 100 or by consuming the i.LON 100 web service function `DataServer_Get()`. The information which is relevant and should be available for `ilonif` and `pixelif` is shown in Table 5.1. The class `DataPoint` is used to store all relevant information on data points which can be retrieved from the i.LON 100. Listing 6.1 shows the member variables of class `DataPoint`.

```
/*** Basic information about the data point ***/
  protected final int index; //Index of the data point
  protected final NvType nvType; //Type of the data point
  protected String name = null; //Name of the data point

/*** Information on faults while consuming the web service ***/
  protected boolean fault; //Indicates if an error occurred on the last access attempt
  protected String faultString; //Stores value of UCPTfaultString tag

/*** Information related to DataServer_Read() and DataServer_Write functions ***/
  protected ValuePreset value = null; //Last retrieved value
  protected ValuePreset valueToWrite = null; //Value to write on next write attempt
  protected Date updateTime; //Last time value was updated
  private boolean valueChanged = false; //Indicates if value changed on last update

/*** Information retrieved via the DataServer_Get() function ***/
  protected String unit; //Unit string
  protected String location; //Stores value of UCPTlocation tag
  protected String description; //Stores value of UCPTdescription tag
  protected String formatDescription; //Stores value of UCPTformatDescription tag
  private String direction; //Stores the value of UCPTdirection tag
  private boolean isOutput; //Is set according to the direction property
  protected String invalidValue; //Stores the optional UCPTinvalidValue tag
  protected String minValue; //Stores the optional UCPTminValue tag
  protected String maxValue; //Store the optional UCPTmaxValue tag
```

Listing 6.1: Member variables of class `DataPoint`. The member variables are grouped according to their purpose.

Most of the member variables of class `DataPoint` correspond to the XML elements in the SOAP (*Simple Object Access Protocol*) messages of the i.LON 100 web service described in Section 3.4.2. As example, the variable `direction`, which indicates if the data point is an input or an output, corresponds to the tag `UCPTdirection` in a `DataServer_Get()` response SOAP message, as shown in Listing 3.14.

6.2.2 Beyond Data Points

Data points are used by the i.LON 100 to allow web service consumers to interact with the attached building automation system. The class `DataPoint`, introduced in the last section, is a representation of such a data point in Java and stores the relevant information about data points. `DataPoint` is mainly used to store all the data `ilonif` needs for consuming the i.LON 100 web service and to store the return values of web service consumption.

For media façades data points are of no (or very little) interest, because a building automation data point is alien to media façades. In Section 5.1.4 a bitmap based approach was suggested and decided to be implemented.

For the bitmap based approach the concept of data points has to be adapted to act as pixels. The class `DataPoint` has been sub-classed to introduce properties of pixels to data points. The name of the sub-class is `Pixel`.

The class `Pixel` extends `DataPoint` and adds attributes required for a "building automation based pixel". These attributes comprise coordinates of the pixel, a type and a feedback attribute. The coordinates define the position of the pixel on the façade of the building. For example if the data point represented by the pixel allows to control the lights of a room, the coordinates define the position of the window of this room on the façade of the building. If the data point controls the sunblinds of a room the coordinates define the position in a similar way.

The type of a `Pixel` defines if the pixel represents a light or a sunblind[1]. The feedback attribute of the class `Pixel` stores the unique index of the data point which is assigned to the feedback network variable of the data point represented by the pixel. Using the configuration in Figure 5.1 as example the instance of class `Pixel` representing the data point named NVL_nvi_L1 would have set its feedback attribute to the index of the data point named NVL_nvi_L1_fb. The feedback data point and therefore the feedback attribute are needed to retrieve the current value of a pixel.

6.2.3 Storage for Data Points

Both `ilonif` and `pixelif` require access to the available data points. Functionality to store and query data points is provided by the Java package `datastorage`. The class in package `datastorage`, which implements the the storage functionality is called `DataStorage`. The class `DataStorage` is the central place where `Pixel` and `DataPoint` objects can be stored and retrieved. `DataStorage` stores an arbitrary number of `DataPoint` and `Pixel` objects and provides methods to retrieve them via their unique index, name, coordinates (if the data point is a pixel) or type. A `DataStorage` instance is filled with data points in the initialization phase of `pixelif` (see Section 6.3).

The objects retrieved from a `DataStorage` are no copies but references to an existing instance. Therefore operating on such an object alters an object shared by different software modules, which is potentially hazardous when dealing with more than one thread, because the classes `DataPoint`, `Pixel` and `DataStorage` are not thread-safe. On the other hand shared objects allow easy transport of information among different methods and classes. When return values of a `DataServer_Read()` web service function (see Section 3.4.3.2) are written to a `DataPoint` object, every other class can retrieve this `DataPoint` instance from the data storage and read the updated values.

6.2.4 Hiding the i.LON 100 Web Service

In Section 5.1.2 it was decided to implement an abstraction layer to hide the specifics of the i.LON 100 web service. The name of the Java package which contains the abstraction layer is `ilonif`. Its methods solely operate on `DataPoint` instances. The request SOAP messages is constructed by evaluating data from a `DataPoint` instance. The values in the response SOAP message are written back into the same `DataPoint` instance. This way the specifics of the i.LON 100 web service are not visible outside the package `ilonif`.

The package comprise the class `ILon` and the sub-package `ilonws`. The latter contains all classes produced by `wsimport`, which is a tool distributed with JAX-WS (see Section 3.5.2)

[1] For now the types are restricted to these two options, but new types can be easily incorporated.

to parse the description of a web service contained in a WSDL (*Web Service Description Language*) file and generate Java classes. The generated Java classes support the developer in consuming the web service.

Most of the classes generated by `wsimport` represent the XML elements in SOAP messages. For an developer implementing the consumption of the i.LON 100 web service two classes are of special interest: `ILON100` and `MainSoapPort`. Class `ILON100` contains information on the location of the web service and provides a function to retrieve a `MainSoapPort` object. `MainSoapPort` is the "stub" class, as described in Section 3.1.1, which can be used to invoke the functions provided by the web service. The class `MainSoapPort` comprises functions which are named and associated to respective web service functions. By invoking such a function of `MainSoapPort` actually the associated function of the webservice is consumed and the return value of the `MainSoapPort` function is a representation of the response SOAP message sent by the web service.

The main API of `ilonif` is implemented in the class `Ilon`. For every required web service function `Ilon` has its own method which directly operates on `DataPoints`. For example the method `readDataPoint()` in class `Ilon` requires as parameter a set of data points stored in a `java.util.Collection<DataPoint>`. The information needed to invoke a web service function on a specific data point is contained in the class `DataPoint`. The method extracts the needed information, then invokes the web service function `DataServer_Read()` (see Section 3.4.3.2) and stores the results in the `DataPoint` instances again.

6.3 Building Automation Interface for Media Façades

The Java package `ilonif` hides the specifics of the i.LON 100 web service. But `ilonif` still uses the concept of data points, which is alien in the domain of media façades. Artists and developers who design the content of media façades usually have no (or very little) knowledge about building automation in general and LonWorks in particular. In Section 5.1.4 it was decided to implement an API which is familiar to people dealing with media façades and hides the underlying building automation technology. This functionality is implemented in the Java package `pixelif`.

6.3.1 Bitmaps and Building Automation

To provide a convenient interface for people dealing with media façades it was decided to implement a pixel oriented API in `pixelif`. Its main class `PixelIF` provides an interface which operates on bitmaps. For representation of the bitmaps two-dimensional arrays are used. Each entry of the array contains the value of the according pixel. For showing the bitmaps on the façade of the building `PixelIF` provides "draw" methods, which were introduced in Section 5.1.4. The draw methods require as parameter a two-dimensional array containing the values for the pixels. The methods request the respective `Pixel` instances from the `DataStorage` and invoke the write function of the class `Ilon` which is part of the Java package `ilonif`. The user of `PixelIF` does not have to care about the underlying `Pixel`/`DataPoint` functionality nor about the i.LON 100 and its web service. The user has only to provide the bitmaps representing the image he wants to be applied. For example if

the user wants to close all sunblinds on the building a two-dimensional array filled with the "sunblinds down" values is all the user has to care about[2].

In Section 5.1.4 it was decided that `pixelif` should notify the user of the API if designated switches alter their value. To enable the notification `PixelIF` has to be accordingly configured (see Section 6.3.2).

The implementation of the notification functionality follows the usual approach in Java by implementing the *Observer Pattern* [GHJV05] in the form of *listeners*. A `SwitchListener` can register itself at a `PixelIF` instance and will be notified every time the value of a switch has changed. To ensure the `SwitchListener` will be notified as soon as possible, every time before a bitmap is drawn the values of all known switches will be retrieved from the i.LON 100 and checked if the value changed. For every changed switch value the registered `SwitchListener` will be notified.

Typically a user will run `pixelif` in a separate thread to avoid timing problems and minimize the effects of other tasks which have to be performed. Threading in Java is quite easy to implement by utilizing the class `Thread`, included in the Java distribution. However a thread once started can not be stopped from the outside in a safe and reliable manner[3]. Therefore a thread has to stop itself. If the thread does not know when to stop, as it is in the case of `PixelIF` the thread has to be notified that it is supposed to stop. The registered `SwitchListener` is allowed to return a `boolean` value. When notifying the `SwitchListener` its return value is checked and if it equals true an *exception* is thrown. Processing control is transferred to whoever catches the exception. The code which catches the exception can take care of clean up and quit the thread.

If a switch listener is registered, the value of switches will be retrieved and checked every time before bitmaps are actually drawn. The retrieving of values may have substantial effect on timing of writing data points. An user of `pixelif` could choose another approach if timing is critical. The i.LON 100 web service allows multiple clients at once, therefore an user could instantiate two `PixelIF` objects each in separate threads. One instance can be used to draw bitmaps without registering a switch listener, therefore switches will not be checked and timing issues can be avoided. The other instance of `PixelIF` can be solely used to periodically check the switches (a switch listener have to be registered for that) and react accordingly.

6.3.2 Configuring Pixels

As described above `pixelif` is capable of mapping the pixels of a bitmap to the data points located on an i.LON 100. However it can not detect the position of data points automatically. In `pixelif` a configuration file is used to specify coordinates for data points.

The configuration file of `pixelif` is plain text, has a simple structure, and is human-readable. It contains a *section* for every available pixel. A section starts with the name of the section enclosed in brackets. The name of the section can be chosen freely. A pixel section contains the name of the associated data point on the i.LON 100, the name of the feedback data point,

[2] For this special case there is even a convenience method `setAllSunblinds()` in class `PixelIF`. So the user does not even have to care about an array.

[3] The class `Thread` comprise the method `stop()` but the method is deprecated and declared as *"inherently unsafe", see:* `http://java.sun.com/javase/6/docs/api/java/lang/Thread.html`.

the type of the pixel and the location in x and y coordinates. In the initialization phase of `PixelIF` the configuration file is parsed and the configuration values stored in an instance of `DataStorage`. Also it is checked if the data points defined in the configuration file exist on the i.LON 100. An excerpt of a configuration file can be found in Listing 6.2.

```
[Point1]
NameOut = NVL_outmod1_NVI00
NameFb = NVL_outmod1_NVO00
Type = Light
PosX = 2
PosY = 0

[Point2]
NameOut = NVL_outmod1_NVI01
NameFb = NVL_outmod1_NVO01
Type = Sunblind
PosX = 2
PosY = 0
```

Listing 6.2: Excerpt from the `pixelif` configuration file. Every section starts with the name of the section enclosed in brackets. A section contains the name of a data point used as pixel (`NameOut`) and its feedback data point (`NameFb`). Also the type of the pixel (`Type`) and its coordinates (`PosX` and `PoxY`) are required.

As discussed in Section 6.2.2 two types (lights and sunblinds) of pixels are implemented at the moment. The type of the pixel has to be defined in the associated section of the configuration file. The configuration file can also be used to specify switches, which should be observed by `pixelif`.

A user of `pixelif` has to create a configuration file and define a section for every data point, which should be used to draw bitmaps. Writing such a configuration file is an error-prone task because every typo leads to an invalid or incomplete configuration. To relieve the user from creating a configuration file from scratch and to reduce the probability of errors in the resulting configuration file, a small tool was developed to support the user in creating the configuration file.

The developed tool downloads the file `dp_NVL.xml`, described in Section 3.3.3, from the i.LON 100. The `dp_NVL.xml` file contains descriptions of all NVL data points, which exist on the particular i.LON 100. After downloading the tool extracts the names of data points which are assigned to dynamic network variables. The data points assigned to dynamic network variables can be differentiated from other data points because the `UCPTlocation` tag as described in Section 3.3.3 differ. Therefore the tool parses file `dp_NVL.xml` and extracts the names of data points whose `UCPTlocation` tag define them as assigned to dynamic network variables. After extracting the names of the data points a template configuration file is created. The user still has to edit the configuration file afterwards, but he has only to add the coordinates and the type (light or sunblind) to every pixel section.

6.3.3 Producing Visual Effects

The functionality included in `pixelif` allows developers to create visual effects on building façade without having to care about the underlying building automation technology. The developer just has to invoke the “draw” methods of the class `PixelIF` in a timely manner

and the bitmaps which are required parameters of the draw methods will be applied to the façade of the building.

To show and test the functionality of class `PixelIF` and to provide some initial aid for developers starting to work with `pixelif` *effect classes* were added to the package. These effect classes produce some specific effects on a `PixelIF` instance. At the moment classes to produce wave effects and to show an effect which resembles a snake creeping over the façade of the building are implemented. A sequence of pictures showing the snake effect is shown in Figure 6.2.

Figure 6.2: Demonstration suitcase showing the snake effect. The lights in the windows and the sunblinds are activated to generate the effect of snake creeping on the façade of the building.

6.4 Utilization of the Artificial Recognition System

In Chapter 4 the functionality of the *Artificial Recognition System* (ARS) and how the *symbolization* technology, implemented in a subproject called ARS-PC (*Artificial Recognition System – Perception*), can be utilized in this master thesis is discussed. Symbolization refers to a data transformation process, which condenses great amounts of input information with relative small informational value into a smaller number of information points but with higher informational value.

The concepts and data values used in building automation systems are of no or only small use for media façade systems. A more holistic view on the data available in the building automation system would be more suitable for media façades. The symbolization process can be used to transform ordinary building automation data into information suitable for media façades because it reduces the amount of data points and produces high level information.

However as discussed in Section 5.2 the current implementation of ARS-PC is not designed to operate on arbitrary building automation data as input but is specialized on the sensory data available at the *Smart Kitchen* of the *Institute of Computer Technology* (ICT). Therefore it was decided to not use building automation data retrieved via the i.LON 100 web service as input source but use one of the already implemented input sources of ARS-PC.

6.4.1 Setting up the Artificial Recognition System

As described in Section 4.2.1 the start-up process of ARS-PC is responsible for setting up the input source called *micro symbol factory*. A micro symbol factory reads sensor values from its designated source and transforms the values into *symbols*. At the moment three micro symbol factories exist: *live*, *database* and *simulation*. The difference among the micro symbol factories is their source of sensor values. The live micro symbol factory retrieves sensor values from the *Smart Kitchen*, the database micro symbol factory from a database where sensor values originating from the Smart Kitchen are stored and the simulation micro symbol factory retrieves sensor values from a database where sensor values originating from a simulation environment [Har07] are stored. In Section 5.2.3 it was decided to utilize the database micro symbol factory in this master thesis as it is more independent from the Smart Kitchen than the live micro symbol factory and nevertheless provides real world data values.

When launching ARS-PC also the connections between the *symbolization modules* have to be set up. The connections are used to exchange symbols between modules. The implementation of connections is called *SymbolNet* and is discussed in Section 4.2.4. SymbolNet allows exchange of symbol information via *symbol messages* and provides infrastructure to send, receive, store and update symbols. The symbolization process is implemented in modules which evaluate incoming symbols and generate new symbols if they detect patterns they are programmed for. Therefore establishing connections between symbolization modules is crucial and has to be done correctly.

In this master thesis the launching and setting-up of ARS-PC is done by a class called `ArsLauncher`. The class resides in a Java package named `arsif`. As shown in Figure 5.4 `arsif` acts as abstraction layer between ARS-PC and media façade systems. `ArsLauncher` can be used by media façade systems to launch and set-up ARS-PC. The configuration used in `ArsLauncher` is similar to the configuration used to set up the database micro symbol factory in the current ARS-PC implementation. The symbolization process of ARS-PC is launched in a separate thread and therefore does not block the execution of the user of `ArsLauncher`.

6.4.2 Retrieving Information from the Artificial Recognition System

The class `ArsLauncher` is used to configure and start-up the symbolization process of ARS-PC. A part of the configuration of ARS-PC are the connections established between symbolization modules. For `arsif` to be able to receive symbol messages it has to be connected to other symbolization modules. For this purpose the class `MFSymbolFilter` has been implemented. It comprises a SymbolNet interface and is therefore capable of receiving symbol messages.

In its initialization phase the class `ArsLauncher` instantiates various symbolization modules and an instance of `MFSymbolFilter`. Then it establishes a connection between some

of the symbolization modules which generate symbols on the higher levels of the symbolization hierarchy shown in Figure 4.2 and the `MFSymbolFilter` instance. This connections ensure that the `MFSymbolFilter` object will receive symbol messages when the respective scenarios are detected. Amongst others the following symbolization modules are connected to `MFSymbolFilter`:

- Person makes coffee
- Meeting
- Presence of person

After ARS-PC has been configured and started up by `ArsLauncher` a media façade should be able to retrieve the information which is generated by ARS-PC. By utilizing `arsif` software a media façade system need not to implement a SymbolNet interface but can use another approach.

The used approach in `arsif` to forward symbol information generated by ARS-PC to a media façade system follows the Observer Pattern [GHJV05]. In Java the Observer Pattern is usually implemented by listeners. A listener is a software interface which can be registered at objects, if special events occur in this object the listener will be notified.

A media façade system utilizing the class `ArsLauncher` can register a `SymbolListener`. This listener will be notified whenever the internals of `arsif` receive a new symbol message. The media façade system does not have to care about the internals of `arsif` like `MFSymbolFilter`. The only relevant class for media façade systems is `ArsLauncher`.

6.5 Integrating Media Façades

The software modules `arsif` and `pixelif` described above are abstraction layers and provide APIs to hide the underlying technology. APIs itself only provide functionality, an application which consumes the API is needed to actually show the possible cooperation of building automation and media façades. Section 5.3 described that it would be advantageous if the consuming application would be executed on the *Image Generator* (see Section 2.2.2) because it allows easier ways to communicate with the media façade system. However as the Image Generator was not delivered by *BLIP*[4] another solution had to be found. It was decided to write an demonstration application, which acts as the consumer of the APIs of `arsif` and `pixelif` and acts therefore as replacement of the Image Generator for the time being.

The demonstration suitcase described in Section 5.4 is a prototype to demonstrate the cooperation possibilities of building automation and media façades. As the demonstration suitcase is for now the only system utilizing `arsif` and `pixelif`, the demonstration software has been tailored for the use with the demonstration suitcase. Therefore it also implements the operational modes (conventional mode and effect mode) of the suitcase as described in Section 5.4.3.

The demonstration application is a simple Java program with a *Graphical User Interface* (GUI) which consumes the APIs of `pixelif` and `arsif`. Figure 6.3 shows the GUI of the

[4] `http://www.blipcreative.com/`

demonstration application. With the control elements shown in the figure the behavior of the demonstration suitcase can be altered. If the user activates one of the control elements the application utilizes the API of `pixelif` to produce the according effect with the building automation system. The text label below the control elements is used to display information retrieved via the API of `arsif`. The screenshot in Figure 6.3 shows that the last information reported by `arsif` was about a person presence.

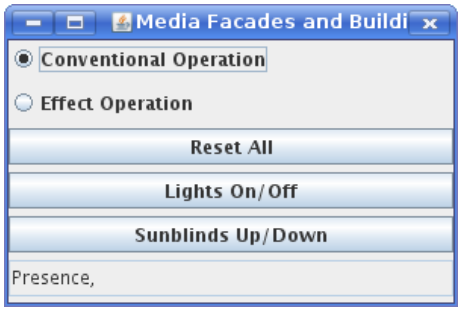

Figure 6.3: Demonstration application for the demonstration suitcase. The *checkboxes* at the top of the GUI can be used for switching between the operational modes. The *buttons* below allow to directly affect the building automation system. In the text *label* at the bottom information delivered by ARS-PC is shown.

6.6 Demonstration Suitcase

In Section 5.4 it was decided to implement a demonstration environment in a suitcase. The demonstration suitcase should be able to show the cooperation building automation and media façades are capable of, therefore a small building automation system and parts of a media façade have to be integrated into the demonstration suitcase. The integrated components are discussed in the next section.

6.6.1 Components and Assembling

In Figure 5.5 a draft of the suitcase and its main components is shown. The impression of a building façade on the top cover is mostly produced by the presence of windows. For every window the light in the "room" behind the windows should be controlled by an integrated building automation system. It was decided to produce the impression of room lighting by filling the top cover of the suitcase with polystyrene[5]. Rectangular holes in the polystyrene represent the room and in the back center of the hole a LED, which emits a "natural white" light is mounted. The polystyrene and the holes are covered with a light diffusing sheet. Figure 6.5 shows the holes in the polystyrene fitted into the top cover of the demonstration suitcase.

A plan of the technical components required to implement the functionality described in Section 5.4.1 is shown in Figure 6.4. The parts related to building automation are the push

[5]Polystyrene is better known as *Styropor* in German speaking countries and as *Styrofoam* in English speaking countries.

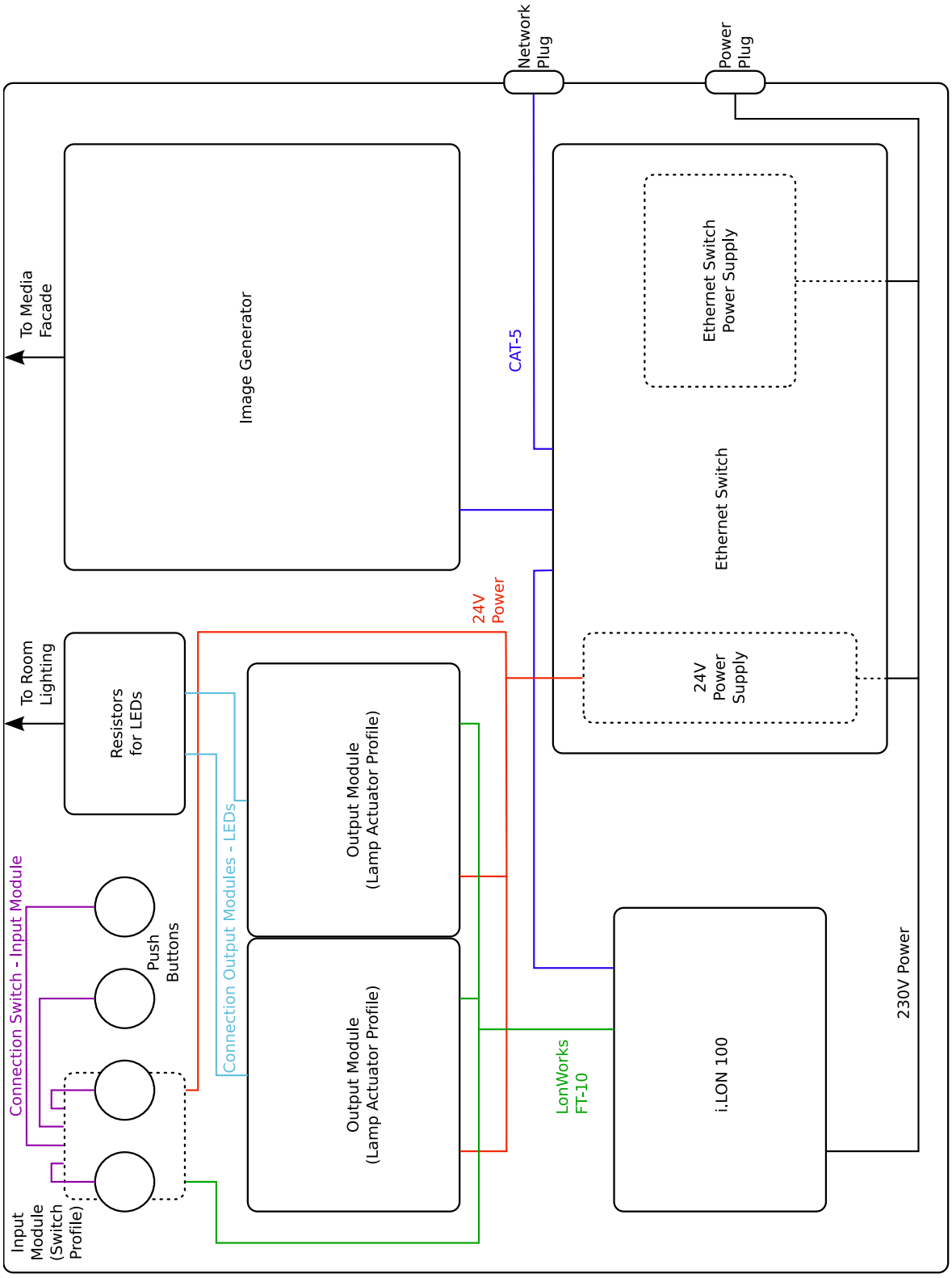

Figure 6.4: Components of the demonstration suitcase. The parts related to building automation are the input modules, the output modules and the i.LON 100. The Image Generator controlls the LEDs of the media façade. The Ethernet switch is required to enable communications between i.LON 100, Image Generator and outside world.

buttons, the input module, the output modules and the i.LON 100. The output modules are LonWorks nodes which implement the lamp actuator profile (Figure 3.6) and are responsible for switching on and off room lighting and sunblind representations. The push buttons are connected to the input module, which is a LonWorks node implementing the switch profile. The i.LON 100 allows external applications to access building automation data via its web service.

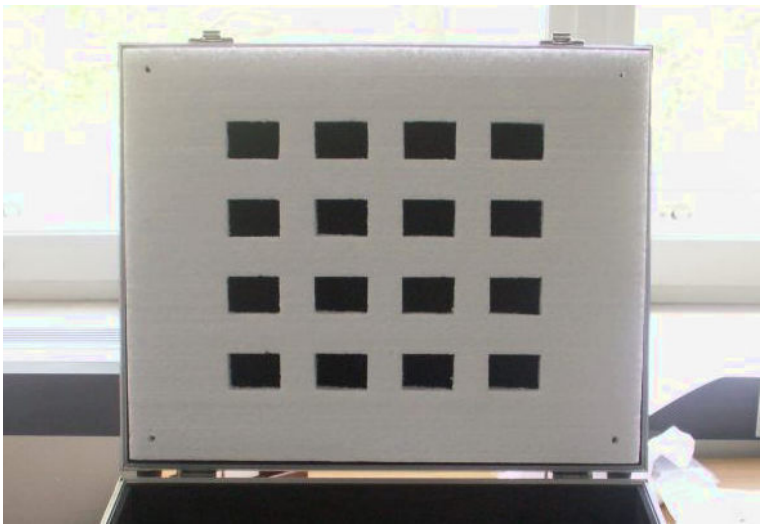

Figure 6.5: Representation of windows in the demonstration suitcase. The top cover of the suitcase is filled with polystyrene. The holes represent rooms. At the back of the holes LEDs are mounted to represent the room lighting.

On the part named "Resistors for LEDs" in Figure 6.4 the resistors for the LEDs, shown in Figure 5.6 are mounted. The LEDs in the top cover of the suitcase are connected to these resistors.

The Image Generator would be connected to the media façade in the top cover of the suitcase. The Ethernet switch in Figure 6.4 enables communication between i.LON 100, Image Generator and the outside world via.

For switching the lights and sunblinds representations on and off, digital switches are required. Switching of lights and sunblinds should be done by a building automation system as that is common practice in real buildings. Therefore a LonWorks-enabled node with a lamp actuator profile, as shown in Figure 3.6 is required. There are a lot of nodes from different vendors which implement the lamp actuator profile available. The *LM 016R* [Uni07] produced by *Unitro*[6] was chosen to be used as lamp actuator. The output module, as it is called in this work and shown in Figure 6.4, comprise 16 relais-driven digital outputs and can be mounted on a *DIN rail* (DIN: Deutsches Institut für Normung) which is also known as *top hat rail* or *mounting rail*. Both of these attributes are relevant because the demonstration suitcase comprise 16 windows. Each window requires two switches to control light and sunblind functionality, as shown in Figure 5.6. Therefore 32 digital outputs, which can be provided by two LM 016Rs, are required for the whole suitcase. Furthermore DIN rails have been used to mount the components inside the demonstration suitcase, therefore it was a hard

[6] `http://www.unitro.de/`

Figure 6.6: Picture of the components of the demonstration suitcase. At the bottom of the picture the i.LON 100 resides on the left the Ethernet switch on the right side. At the top on the left the LonWorks input and output modules and the push-buttons are visible. The Image Generator would be located at the free space in the upper right corner, but is missing in the picture.

requirement that the utilized output modules can be mounted on DIN rails. The output modules are mounted on the left side of the suitcase as shown in Figure 6.6.

Output modules alone are not enough to represent a building automation system. The lights the output modules can turn on and off should be controllable by switches. Therefore switches and a node which implements the switch functional profile, as shown in Figure 3.6, are integrated into the demonstration suitcase. The switches are small and visually appealing push-buttons. The switches are connected to a LonWorks-enabled input module. The chosen input module is the *lumina T6* [Spe06] produced by *Spega*[7]. The input module and the switches can be seen in the upper left corner in Figure 6.6. The lumina T6 was chosen because it has already been utilized at the ICT and proved to be a reliable and well-priced component.

The last component required for the building automation system in the demonstration suitcase is an i.LON 100. The input module and the output modules are connected to a LonWorks bus. The i.LON 100 is also integrated into the bus system. The i.LON 100 is configured as described in Section 6.1 and a web service consumer can therefore alter and read the values of the input and output modules. In Figure 6.6 all building automation related parts are located on the left side of the picture, the i.LON 100 is located at the bottom of the left side.

The building automation part of the demonstration suitcase is complete with the above mentioned components. The media façade related components are described below. As shown in Figure 5.5 the media façade is mounted vertically between the windows in the top cover of the suitcase. The media façade consists of a "LED string" shown in Figure 6.7. This type of LED string is used in real world media façades. The LED string has been kindly provided by BLIP. The LEDs would have been connected to the Image Generator if it would be available. However the suitcase is prepared for integration of an Image Generator. Figure 6.6 shows that free space in the upper right corner of the suitcase is available and designated for an Image Generator.

Figure 6.7: LEDs used in the demonstration suitcase to represent a media façade. The LED elements are multi-color. This type of LED string is used in real world media façades.

[7] `http://www.spega.com/`

The image generator can be supplied with power via a *Power over Ethernet* [IEE03] (PoE) connector. An Ethernet switch which provides PoE functionality was integrated into the demonstration suitcase. The Ethernet switch is shown in Figure 6.6 at the bottom of the right side of the picture.

6.6.2 Operation of the Demonstration Suitcase

The fully assembled demonstration suitcase is shown in Figure 6.8. The picture shows the room lighting turned on. The LEDs of the media façade are not activated because of the missing Image Generator. As discussed in Chapter 5 the demonstration suitcase has two operational modes. One mode for demonstrating aspects of building automation functionality (*conventional operation*) and one mode for demonstrating the cooperation between media façades and building automation (*effect operation*).

The operational mode of the demonstration suitcase can be switched by either the most-left push-button or via the demonstration application described in Section 6.5. The application is executed on a notebook, which is connected to the Ethernet switch integrated into the demonstration suitcase.

Figure 6.3 shows the GUI of the demonstration application. If the suitcase is in conventional operation mode the building automation system is not altered by the application, but the value of the push-button, which is designated to change the operational mode, is periodically read. If the push-button is pressed or a user changes the mode in the GUI of the application the effect mode is entered. In the effect mode the application would coordinate the behavior of the media façade and the building automation system to produce enhanced effects on the representation of a building façade, if an Image Generator would be available. As it is missing only room lighting is used for effects at the moment.

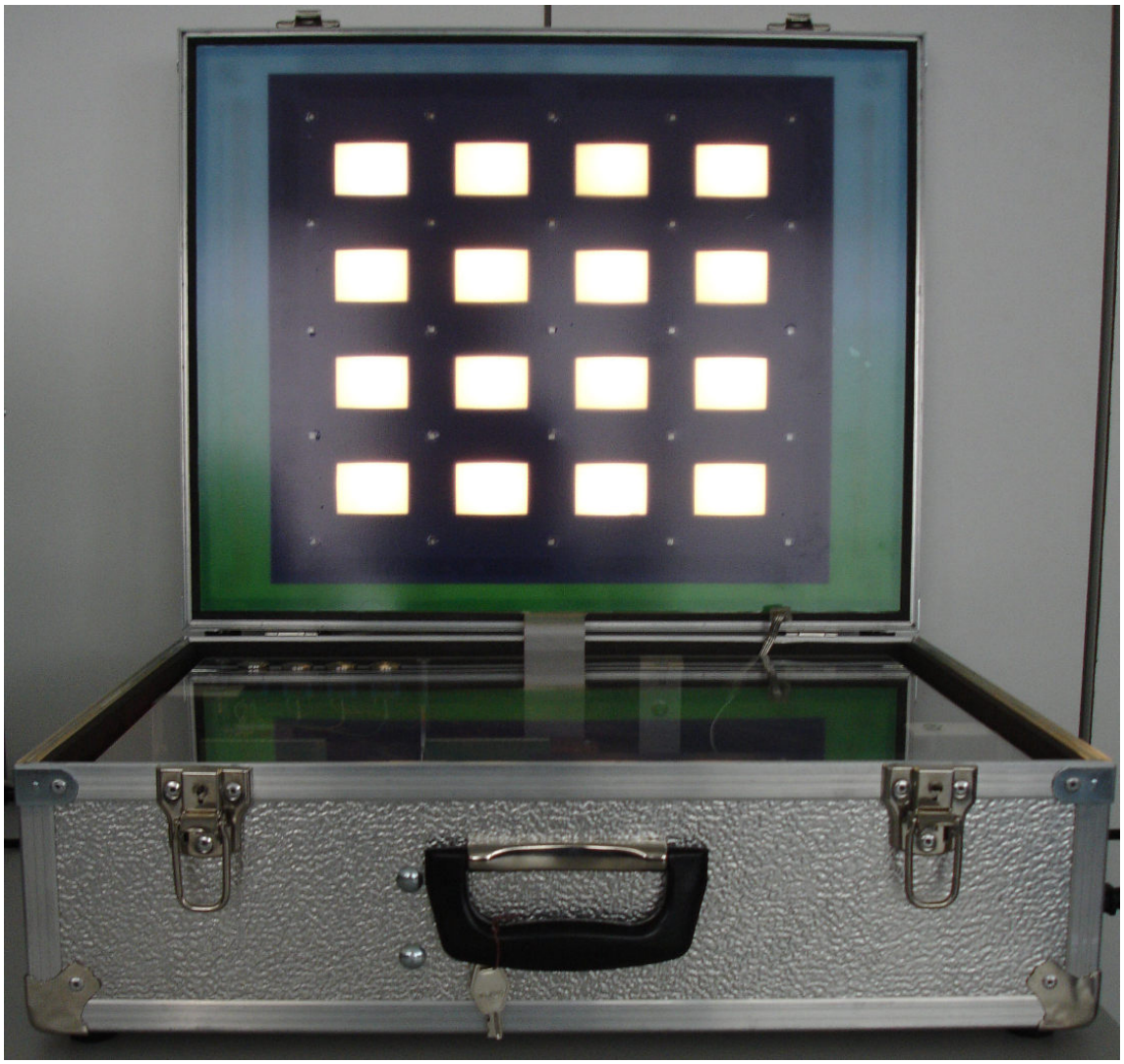

Figure 6.8: Picture of the demonstration suitcase. The lights in the windows are turned on. The LEDs of the media façade are not activated in the picture, because the Image Generator which controls the LEDs is not available. Nevertheless the placement of LEDs can be seen by the small holes between the windows.

Chapter 7

Conclusion and Further Work

In the course of this work a system has been implemented which allows media façades to cooperate and communicate with building automation systems. However some obstacles have been encountered which have to be solved in future work.

7.1 Conclusion

The system implemented in this master thesis enables media façades to communicate with building automation systems. In Chapter 2 media façade systems and how they can benefit of cooperation with building automation systems has been discussed. Cooperation is enabled by a system which acts as "translator" system as shown in Figure 1.2. Communication between building automation system and media façade is possible in two ways. One data channel originating at the building automation system and ending at the media façade system. The second data channel allows communication in the opposite direction.

For communication between building automation system and the translator system a gateway node as shown in Figure 1.2 is used. The gateway node is an i.LON 100 which provides a *web service* to allow access to data located in the building automation system. Both, the i.LON 100 and its web service are discussed in Chapter 3.

The translator system mentioned above is implemented in several software modules which are described in Chapter 5 and Chapter 6. The software module `pixelif` allows media façades to control aspects of a building automation systems. For example the media façade may be allowed to lower the sunblinds of rooms where the lights are turned on, so that room lighting is not interfering with the visual effects produced by the media façade. Furthermore the media façade may be allowed to utilize room lighting and sunblinds for visual effects.

To hide the specifics of building automation systems from media façade systems the software module `pixelif` provides a *pixel* based interface. The windows of a building façade are considered as pixels of a display and coordinates are assigned to them. The user of `pixelif` can supply bitmap images which are applied to the façade of the building by `pixelif`. For this to work, coordinates have to be assigned to actuators which control room lighting and sunblinds. Coordinates are defined in a configuration file.

The second aspect of communication between media façades and building automation deals with informing media façades about its environment. Building automation gathers a lot

of information about the state the building is in. However this information can not be directly used by media façades because building automation data and building automation data structures are alien in the media façade domain.

The information gathered by building automation typically consists of a huge number of sensor values with only small informational value. Examples for data gathered by building automation are the state of a switch or the temperature in a room. Such data does not have much value for media façade systems and therefore has to be pre-processed to be useful for media façades.

In this master thesis ARS-PC (*Artificial Recognition System – Perception*, see Chapter 4) has been used to pre-process building automation data. ARS-PC implements a functionality called *symbolization* which transforms great amounts of input data (specifically sensor values) into lesser data points with higher informational value. These data points typically describe scenarios like "person makes coffee" and are therefore more useful for media façades.

Unfortunately it was not possible to use the data retrieved via the i.LON 100 as input data for the symbolization process of ARS-PC. Instead an already existing input data source of ARS-PC, which retrieves sensor values from a database, has been used in this work. The software module which delivers information gathered by ARS-PC is called `arsif`.

To test and present the functionality of cooperation between building automation and media façades a prototype system has been built. The prototype is integrated in a suitcase as described in Chapter 5 and Chapter 6. The suitcase comprise a representation of a building façade in the top cover. The representation consists of windows which are able to show the state of room lighting and sunblinds. LEDs (*Light Emitting Diodes*) of a media façade are mounted between the windows. A picture of the demonstration suitcase is shown in Figure 6.8.

The demonstration suitcase is intended to be used as test platform of the implemented software modules and as demonstration of the cooperation between building automation and media façades at trade shows and similar events. However the suitcase has not been completed because the *Image Generator*, which actually controls the LEDs of the media façade, has not been delivered by our *Mediafacade.net*[1] project partner *BLIP*[2].

As a temporary replacement for the Image Generator a small application has been developed. At the moment the application utilizes the software modules `pixelif` and `arsif` to produce effects on the representation of a building façade on the top cover of the suitcase respectively to retrieve information from ARS-PC. The media façade LEDs can not be activated because the Image Generator is missing.

7.2 Further Work

While the basics for cooperation of media façades and building automation have been laid out in this master thesis, there is still some work left to do. One of the most pressing jobs left to be done is the integration of the Image Generator into the demonstration suitcase. The first steps in integration would be to add to the application, which is currently used as

[1] `http://www.mediafacade.net/`
[2] `http://www.blipcreative.com/`

replacement for the Image Generator, means to communicate with the Image Generator. The replacement application would have to be stripped of its GUI (*Graphical User Interface*) and act as "translator" system as shown in Figure 1.2.

The next step of integration would be to get rid of the external computer system where the replacement application is executed. Instead of the external computer system the Image Generator could be used. Moving all software modules to the Image Generator would make the demonstration suitcase much more attractive, because it would allow autonomous operation of the suitcase.

A requirement to port the software modules implemented in this master thesis to the Image Generator is the availability of a Java runtime environment for the Image Generator platform. Supposedly the Image Generator CPU has a MIPS architecture, for which only a few Java runtime environments are available. The web service framework JAX-WS (*Java API for XML – Web Services*) used in this master thesis requires at least Java Version 5. *Sun*[3] does not provide an official Java runtime environment for the MIPS architecture, but with the recent open sourcing[4] of Java and with advances made by the *GNU Classpath*[5] project there might already be a runtime environment which works with JAX-WS. Future work is required to find out which runtime environment is capable of running the software modules implemented in this master thesis and which steps are necessary to port the software modules to the Image Generator platform.

For real world applications new scenarios which can be detected by ARS-PC have to be defined and implemented. These scenarios should be detectable by evaluation of sensors which are already deployed in modern building automation systems or at least only require very simple and cheap sensors. As recent work has introduced image recognition functionality to ARS-PC another possibility is to use video cameras as input sources.

Scenarios based on the above mentioned sensors have to be defined and implemented in ARS-PC. The software module `arsif` does not have to be altered, the new scenarios are automatically forwarded to `arsif`'s user — the media façade.

[3] `http://www.sun.com/`
[4] `http://www.sun.com/software/opensource/java/`
[5] `http://www.gnu.org/software/classpath/`

List of Figures

List of Tables

Listings

Bibliography

[ASH04] ASHRAE: *BACnet – A Data Communication Protocol for Building Automation and Control Networks*. ANSI/ASHRAE Standard 135-2004, 2004 16

[ASH06] ASHRAE: *BACnet – A Data Communication Protocol for Building Automation and Control Networks*. Addendum c to ANSI/ASHRAE Standard 135-2004, 2006 22

[Atk03] Atkins, S.: *Size and cost of the problem*. IETF Meeting. `http://www3.ietf.org/proceedings/03mar/slides/asrg-1/index.html`. Version: March 2003 16

[BLFF96] Berners-Lee, T. ; Fielding, R. ; Frystyk, H.: *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945 (Informational). `http://www.ietf.org/rfc/rfc1945.txt`. Version: Mai 1996 (Request for Comments) 19

[BLFM05] Berners-Lee, T. ; Fielding, R. ; Masinter, L.: *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986 (Standard). `http://www.ietf.org/rfc/rfc3986.txt`. Version: Januar 2005 (Request for Comments) 23

[BLI07] BLIP (Hrsg.): *BLIP Display Manager*. Unit 19A, Perseverance Works, 38 Kingsland Road, UK: BLIP, 2007. `http://www.blipcreative.com/pdf/displaymanager0307.pdf` 13

[BLPV07] Burgstaller, W. ; Lang, R. ; Pörscht, P. ; Velik, R.: Technical Model for Basic and Complex Emotions. In: *5th IEEE International Conference on Industrial Informatics (INDIN)* Bd. 2, 2007, S. 1033–1038 57

[Bur07] Burgstaller, W.: *Interpretation of Situations in Buildings*. Vienna, Technical University of Vienna, Ph.D., 2007. – to be published 54

[Bus97] Bushby, Steven T.: BACnet: a standard communication infrastructure for intelligent buildings. In: *Automation in Construction 6* (1997), S. 529–540 15

[Cal03] Callaway, Edgar H.: *Wireless Sensor Networks: Architectures and Protocols*. CRC Press Inc., 2003. – ISBN 0849318238 3

[CEA02] CEA: *Control Network Protocol Specification*. ANSI/EIA/CEA Std. 709.1 Rev. B, 2002 16, 25, 29

[Die99] Dietrich, D.: *LON-Technologie : verteilte Systeme in der Anwendung*. 2. berarb. Aufl. Hthig, 1999. – ISBN 3–7785–2770–3 25, 27, 34

[Die00] Dietrich, D.: Evolution potentials for fieldbus systems. In: *Factory Communication Systems, 2000. Proceedings. 2000 IEEE International Workshop on*, 2000, S. 145–146 53

[DLP+06] Deutsch, T. ; Lang, R. ; Pratl, G. ; Brainin, E. ; Teicher, S.: Applying Psychoanalytic and Neuroscientific Models to Automation. In: *2nd IET International Conference on Intelligent Environments* Bd. 1, 2006, S. 111–118 54, 58

[DZL07] Deutsch, T. ; Zeilinger, H. ; Lang, R.: Simulation Results for the ARS-PA Model. In: *5th IEEE International Conference on Industrial Informatics (INDIN)* Bd. 2, 2007, S. 1021–1026 57

[Ech99] Echelon Corporation (Hrsg.): *Introduction to the LONWORKS Platform*. 550 Meridian Ave., San Jose, CA 95126, USA: Echelon Corporation, 1999. `http://www.echelon.com/support/documentation/manuals/general/078-0183-01A.pdf` 27

[Ech00] Echelon Corporation (Hrsg.): *Implementing Dynamic Network Variables*. 550 Meridian Ave., San Jose, CA 95126, USA: Echelon Corporation, August 2000. `http://www.echelon.com/support/documentation/docs/ImplementingDynamicNetworkVariables.pdf` 35

[Ech05] Echelon Corporation (Hrsg.): *i.LON 100 e3 Plug-in Supplement*. 550 Meridian Ave., San Jose, CA 95126, USA: Echelon Corporation, 2005. `http://www.echelon.com/support/documentation/manuals/cis/078-0315-01A.pdf` 32

[Ech06a] Echelon Corporation (Hrsg.): *i.LON 100 e3 Hardware Guide*. 550 Meridian Ave., San Jose, CA 95126, USA: Echelon Corporation, 2006. `http://www.echelon.com/support/documentation/manuals/cis/078-0311-01A.pdf` 29

[Ech06b] Echelon Corporation (Hrsg.): *i.LON 100 e3 Internet Server*. 550 Meridian Ave., San Jose, CA 95126, USA: Echelon Corporation, 2006. `http://www.echelon.com/products/cis/presentations/i.LON100e3_overview.pdf` 29

[Ech06c] Echelon Corporation (Hrsg.): *i.LON 100 e3 Programmer's Reference*. 550 Meridian Ave., San Jose, CA 95126, USA: Echelon Corporation, 2006. `http://www.echelon.com/Support/documentation/manuals/cis/078-0250-01E.pdf` 33, 36, 38, 41, 43, 45, 68

[Ech06d] Echelon Corporation (Hrsg.): *i.LON 100 e3 User's Guide*. 550 Meridian Ave., San Jose, CA 95126, USA: Echelon Corporation, 2006. `http://www.echelon.com/support/documentation/manuals/cis/078-0310-01B.pdf` 30, 33

[Ehr04] Ehrlich, P.: The Future of Facility Management. In: *HPAC (Heating, Piping, Air Conditioning) Engineering* (2004), May 16

[EN05] EN: *DIN EN 13757 Communication systems for and remote reading of meters*. Feb 2005 29

[GHJV05] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design patterns : Elements of Reusable Object-Oriented Software*. 32. print. Boston : Addison-Wesley, 2005 (Addison Wesley Professional Computing Series). – ISBN 0–201–63361–2 71, 74, 87, 91

[Göt06] Götzinger, S.: *Scenario Recognition based on a Bionic Model for Multi-Level Symbolization*, Vienna University of Technology, Faculty of Electrical Engineering and Information Technology, Master Thesis, 2006 62

[GTL94] Garas, F. K. ; T., Armer G. S. ; L., Clarke J.: *Building the Future : Innovation in Design, Materials and Construction*. London : Spon Press, 1994. – ISBN 0–419–18380–9 1

[Har07] Hareter, H.: *Worst Case Szenarien Simulator für die Gebäudeautomation*. Vienna, Technical University of Vienna, Ph.D., 2007. – to be published 59, 61, 90

[Hol06] Holleis, E.: *SymbolNet – Ein Application Framework für symbolische Kommunikation*. June 2006 62

[HSW+00] Hill, J. ; Szewczyk, R. ; Woo, A. ; Hollar, S. ; Culler, D. ; Pister, K.: System architecture directions for network sensors. In: *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000, S. 93–104 3

[IEE95] IEEE: *IEEE standards for local and metropolitan area networks : supplement to carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications : media access control (MAC) parameters, physical layer, medium attachment units, and repeater for 100 Mb/s operation, type 100BASE-T (Clauses 21-30)*. `http://ieeexplore.ieee.org/servlet/opac?punumber=9535`. Version: 1995 14, 16

[IEE03] IEEE: *Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications, Amendment: Data Terminal Equipment (DTE), Power via Media Dependent Interface (MDI)*. `http://ieeexplore.ieee.org/servlet/opac?punumber=8612`. Version: 2003 97

[ISO02] ISO: *Abstract Syntax Notation One (ASN.1): Specification of Basic Notation (ISO/IEC 8824–1:2002)*. Geneva, Switzerland, 2002 62

[KA04] Konnex-Association: *KNX Specifications, Version 1.1*. 2004 16

[Kle01] Klensin, J.: *Simple Mail Transfer Protocol*. RFC 2821 (Proposed Standard). `http://www.ietf.org/rfc/rfc2821.txt`. Version: April 2001 (Request for Comments) 30

[KNS05] Kastner, W. ; Neugschwandtner, G. ; Soucek, St.: Communication Systems for Building Automation and Control, 2005, S. 1178–1203 1

[MGH05] Mahlknecht, S. ; Glaser, J. ; Herndl, T.: PAWiS: towards a power aware system architecture for a soc/sip wireless sensor and actor node implementation. In: *Proceedings of the 6th IFAC International Conference on Fieldbus Systems and their Applications*, 2005, S. 129–134 3

[MI06] Modbus-IDA: *Modbus Application Protocol Specification v1.1b*. Dec 2006 29

[Mil92] Mills, D.: *Network Time Protocol (Version 3) Specification, Implementation and Analysis*. RFC 1305 (Draft Standard). `http://www.ietf.org/rfc/rfc1305.txt`. Version: März 1992 (Request for Comments) 30

[MSMV06] Machado, Guilherme B. ; Siqueira, Frank ; Mittmann, Robinson ; Vieira, Carlos Augusto V.: Embedded Systems Integration Using Web Services. In: *ICNICONSMCL '06: Proceedings of the International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies*, 2006, S. 18 22

[OAS04] OASIS: *UDDI Version 3.0.2*. Committee Draft, Oct 2004 18

[OAS06] OASIS: *oBIX 1.0*. Committee Specification, Dec 2006 22

[PM03] Palensky, P. ; Mahlknecht, S.: Latest Trends in Building Automation. In: *Proceedings of the IGW (Intelligente Gebäude und Wohnungen)*, 2003, S. 9–18 3

[Pos81a] Postel, J.: *Internet Protocol*. RFC 791 (Standard). `http://www.ietf.org/rfc/rfc791.txt`. Version: September 1981 (Request for Comments) 19

[Pos81b] Postel, J.: *Transmission Control Protocol*. RFC 793 (Standard). `http://www.ietf.org/rfc/rfc793.txt`. Version: September 1981 (Request for Comments) 62

[PP05] Pratl, G. ; Palensky, P.: Project ARS – The next Step towards an Intelligent Environment. In: *The IEEE International Workshop on Intelligent Environments*, 2005, S. 55–62 2, 3, 11, 53

[PR85] Postel, J. ; Reynolds, J.: *File Transfer Protocol*. RFC 959 (Standard). `http://www.ietf.org/rfc/rfc959.txt`. Version: Oktober 1985 (Request for Comments) 31

[Pra06] Pratl, G.: *Processing and Symbolization of Ambient Sensor Data*, Vienna University of Technology, Ph.D., 2006 3, 54, 56

[PSKD06] Palensky, P. ; Soucek, S. ; Klot, S. von ; Dietrich, D.: Netzwerke und Gebäude. In: *e&i (Elektrotechnik und Informationstechnik)* (2006), Nr. 6 2

[Ric07] Richtsfeld, A.: *Szenarienerkennung durch symbolische Datenverarbeitung mit Fuzzy-Logic*, Vienna University of Technology, Faculty of Electrical Engineering and Information Technology, Master Thesis, 2007 55, 60, 73

[Rus03] Russ, G.: *Situation-dependent Behavior in Building Automation*. Vienna, Technical University of Vienna, Ph.D., 2003 3, 53, 54

[RW07] Ramzan, Z. ; Wüest, C.: Phishing Attacks: Analyzing Trends in 2006. In: *CEAS (Conference on E-Mail and Anti-Spam)* (2007), August 16

[Spe06] Spega (Hrsg.): *lumina T6 Binary input*. bismarckstr.142a, 47057 Duisburg, Germany: Spega, 2006. `http://www.spega.de/downloads/datasheets/EN/211006_EN_TD.pdf` 96

[SRT00] Soucek, S. ; Russ, G. ; Tamarit, C.: The Smart Kitchen Project – An Application of Fieldbus Technology to Domotics. In: *Proceedings of the 2nd IEEE International Workshop on Networked Appliances (IWNA)*, 2000 61, 73

[TS01] Tanenbaum, Andrew S. ; Steen, Maarten V.: *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, 2001. – ISBN 0130888931 18

[Uni07] Unitron-Fleischmann (Hrsg.): *LM 0/16R digital 16 Kanal Ausgangsmodul (Relais)*. Gaildorfer Str. 15, 71522 Backnang, Germany: Unitron-Fleischmann, 2007. `http://www.unitro.de/Daten/PDF/LM_016R.pdf` 94

[W3C04] W3C: *Web Services Glossary*. `http://www.w3.org/TR/ws-gloss/`. Version: Feb 2004 17

[W3C07a] W3C: *SOAP Version 1.2 Part 1: Messaging Framework*. W3C Recommendation, Apr 2007 18

[W3C07b] W3C: *Web Services Description Language (WSDL) Version 2.0*. W3C Recommendation, Jun 2007 18

[Wei98] Weinmann, A.: *Regelungen I. Analyse und technischer Entwurf*. 3. berarb. Aufl. Springer, 1998. – ISBN 3211825568 15

[XP03] Xing, S. ; Paris, B. P.: Mapping the growth of the Internet. In: *Proceedings of the 12th ICCCN International Conference on Computer Communications and Networks*, 2003, S. 199–204 16