DIPLOMARBEIT

Design and Implementation of a Robot Coordination Framework using Shared Memory Middleware

Ausgeführt am
Institut für Computersprachen
Abteilung für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter Anleitung von
Ao. Univ. Prof. Dipl.-Ing. Dr. eva Kühn

durch

Stephan Zavrel
Floridsdorfer Hauptstrasse 16/1/4
A-1210 Wien
Matr.Nr.: 9726049

[Wien, August 2007] ----------

Abstract

Designing control applications for autonomous mobile robots is a very challenging task and requires a very broad spectrum of knowledge since the field of robotics combines diverse disciplines like mechanics, electronics, computer science and artificial intelligence. The process becomes even more complex if the solution to a problem requires the cooperation and coordination of multiple robots. Experience has shown that in the field of Embodied AI, which focuses on the physical body and its interactions with an environment, the particular problem arises, that a lot of time needs to be invested in the assembling and preparations of robots and suitable communication mechanisms before focus can be put on the area of interest – writing artificial intelligent control applications for the robot team.

This thesis addresses the problem by introducing a software architecture that supports a developer by providing mechanisms to automatically generate a framework for the development of distributed control applications using a virtual shared memory communication layer. To achieve this goal an existing modular and extensible autonomous robot system is enhanced by a communication system based on the principles of sensor and actuator sharing and the creation of a holistic system view for every member of a robot team. Using this approach all the resources available in a scenario can be accessed from each control application in the same way. Furthermore focus is put on the usability and simplicity of the resulting system to minimize the time and effort for users to reach the point where development of control applications can start.

The work is completed by a prototypical implementation of the proposed architecture and its application in a simple proof-of-concept scenario.

Kurzfassung

Die Entwicklung von Steuerapplikationen für autonome, mobile Roboter stellt eine große Herausforderung dar und erfordert ein weites Wissensspektrum, da die Robotik so unterschiedliche Wissenschaften wie Mechanik, Elektrotechnik, Informatik und künstliche Intelligenz vereint. Problemstellungen, die eine Kooperation und Koordination von mehreren Robotern erfordern, erhöhen die Komplexität dieses Vorgangs noch beträchtlich. Die Erfahrung hat gezeigt, daß auf dem Gebiet der Embodied AI, bei der der physische Körper und dessen Interaktion mit der Umwelt im Mittelpunkt stehen, die Problematik besteht, daß der Zusammenbau und die Vorbereitung von Robotern und geeigneten Kommunikationsmechanismen sehr aufwändig sind und somit die Möglichkeit zur Konzentration auf das eigentliche Interessensgebiet – die Entwicklung von künstlich intelligenten Steuerapplikationen für ein Roboter-Team – verzögern.

Diese Diplomarbeit nimmt sich dieser Problematik an und präsentiert eine Softwarearchitektur, die Entwickler durch die automatische Erstellung eines Frameworks für die Enwicklung verteilter Steuerappliaktionen mit Hilfe von Virtual Shared Memory Middleware unterstützt. Um dieses Ziel zu erreichen wird ein bestehendes modulares und erweiterbares Robotersystem um Kommunikationsmechanismen erweitert, die die gemeinsame Nutzung von Sensoren und Aktuatoren ermöglichen und einen gesamtheitlichen Blick auf das verteilte System aus jeder Steuerapplikation heraus bieten. Durch diesen Ansatz können sämtliche verfügbaren Ressourcen in einem Forschungsszenario auf gleiche Art und Weise in jede Steuerapplikation eingebunden werden. Besonderes Augenmerk wird auch auf die Bedienbarkeit und Einfachheit des resultierenden Systems gelegt, um für Benutzer Zeit und Aufwand, die benötigt werden, um mit dem Entwicklungsprozeß beginnen zu können, gering zu halten.

Vervollständigt wird die Arbeit durch die Implementierung eines Prototypen, der auf der entworfenen Architektur basiert und abschließend in einem einfachen proof-of-concept Szenario zum Einsatz kommt.

Danksagung

Von den Personen, die direkt an der Entstehung dieser Arbeit beteiligt waren, möchte ich danken:

Frau Prof. eva Kühn für die Betreuung dieser Arbeit. Ihre Einsichten und Ideen waren ein wertvoller Beitrag zum erfolgreichen Gelingen.

Meinem Freund Jörg Irran, dessen Arbeit Inspiration und Grundlage für die vorliegende Diplomarbeit waren, für die stundenlangen Diskussionen und all die guten Ratschläge.

Richard Mordinyi vom XVSM Team, der immer sehr hilfsbereit war und zahlreiche Fragen beantwortete, die es mir ermöglichten, meine Arbeit fortzusetzen.

Am meisten von allen aber möchte ich meinen Eltern Marianne und Werner danken, die mir in meinem Leben alles was ich wollte ermöglicht haben. Sie lebten mir vor, was bedingungslose Liebe und Unterstützung bedeuten und ich bin unendlich dankbar, solche Eltern zu haben.

Meiner Schwester Monika, die immer für mich da ist und deren Zielstrebigkeit und Ausdauer stets ein Vorbild waren und die ich nie erreicht habe.

Nachdem ich diese Arbeit auch als Abschluß meiner Ausbildung betrachte, möchte ich mich noch bei Personen bedanken, die mich auf diesem Weg begleitet haben:

Meinem Klassenvorstand Dr. Michael Kellner, der ein ganz außergewöhnlicher Lehrer war und auch abseits der schulischen Belange ein Umfeld geschaffen hat, aus dem eine Klassengemeinschaft und Freundschaften bis in den heutigen Tag hervorgegangen sind.

All meinen Freunden aus der Schul- und Studienzeit – ganz besonders Peter, Arash, Martin, Michael, Flo, Christoph, Christian, Nicole, Theresia, Marlene und Andi – die dafür gesorgt haben, daß ich auf mein Studium immer als eine der schönsten Zeiten meines Lebens zurückblicken werde.

Contents

1 Introduction

The field of robotics is a highly complex area of the technology industry and research. It combines such diverse disciplines as electronics, mechanics, computer science and artificial intelligence, requiring researchers to have a very broad spectrum of knowledge across all those fields or – more likely – necessitates the cooperation of experts from each area. Robots typically consist of a set of sensors for acquiring information about the environment, some actuators to manipulate the surroundings and some kind of programmable logic to determine how those sensors and actuators work together. Depending on the field of use robots vary widely in appearance, function and complexity of the applications necessary to control them. Examples range from industrial manufacturing robots for repetitive tasks used in production lines, over mainly remote controlled models such as war mine defusing or medical surgery robots, which extend the abilities of a human operator, to mobile autonomous robots, which come in the form of automated lawnmowers up to rovers used in space missions. The group mentioned last – mobile autonomous robots – typically requires the most complex control applications implementing some kind of intelligent behavior, since they are expected to function independently from an operator once they are activated. Applying a leveled view to the sciences included one can define a low level layer containing mechanics and electronics, providing the groundwork for technologies used in sensors and actuators. On the next level mainly electronics and computer science care about the definition of transmission techniques, specification of bus logics and programming frameworks. Finally, in the topmost layer, high level programming language control applications require skills in computer science and provide a basis for research done in the field of Embodied AI [Broo91]. Due to the complexity of the matter the problem arises that for research on the higher level aspects of mobile autonomous robots a platform handling all the issues on the layers below is necessary. An artificial intelligence researcher cannot be

expected to own the skills to design and construct a robot, but restriction to existing, often inflexible models may confine a project and hinder the achievement of the ultimate goal. In his thesis Jörg Irran [Irr04] addresses this problem by presenting a modular and extensible robot system, that allows for the easy assembling of different configurations supported by a plug&play-like mechanism. In addition he introduces an abstraction layer between the low level hardware and high level control applications by requiring every module to supply a proxy class that contains the implementation details to properly address the hardware on the one side and exposes a programmable interface on the other. This way the amount of knowledge necessary for a robot construction process can be minimized for researchers mostly interested in developing control applications, while at the same time experts may work on the hardware layer independently.

Inspired by Jörg Irran's work, this thesis aims to extend the approach to scenarios with multiple robots and distributed control applications. The motivations for the use of robot teams are manifold. Some problem solving tasks may require the cooperation of different robot types, each specialized for a specific part, to be successfully completed. In other cases distributing the workload to more than one robot or computational unit can speed up a process and increase efficiency [Park95]. The issue of fault tolerance is another example where multiple robot instances can proof beneficial, because if one member of a team fails to work properly, another one can take over its duties. But the scope does not have to be limited to solely inter robot communication, the integration of external resources can be considered as well when some hardware's size, weight or energy requirements prevent it from being mobile. An example could be a lunar space mission, where a rover robot depends on some actuator that can only be hosted by the bigger landing module. Another – literally – more down to earth scenario, is providing household robots access to a home automation system in case they need light to operate autonomously.

A goal of this work is to simplify the process of control application development by applying a holistic view of the resources available to a scenario. For every host the

programmer has access to a pool of sensors and actuators that can be used to solve a task, no matter whether they are attached to the robot running the control application or on a remote host. This approach is realized using virtual shared memory middleware to create that uniform view for every team member. Implementation details of the communication necessary to enable this are hidden from the programmer and every resource is programmable in the same way.

Furthermore focus is put on the usability and user friendliness of the resulting system. The process of getting to the point where control application development for a distributed robot system can start should be as pain- and effortless as possible, hence easy to learn and pick up and is therefore addressed as well.

The following chapters provide an overview of the existing technologies this work is based on (chapter 2), cover the requirements analysis and design of the software architecture (chapter 3), followed by a prototypical implementation based on the developed architecture (chapter 4). Subsequently the developed software is put to use in a proof-of-concept scenario (chapter 5) and the design evaluated in a conclusion (chapter 6) including some ideas for future development.

2 Used technologies and systems

To gain a better understanding of the architecture and design decisions proposed in this thesis it is necessary to introduce the used technologies and systems as well as their functionality. Having a basic knowledge of the systems is essential to recognize the possibilities and problems that arise when trying to integrate those technologies into a single system. In detail, this work interconnects robots based on an existing modular robot system and sensors and actuators placed in the robot's operating environment using a virtual shared memory middleware. Therefore this chapter describes the type of robots used together with the tool set to assemble them and write control applications, followed by an introduction to the X10 home automation system which was chosen to realize a proof of concept implementation of the environmental interoperability aspects. Finally the mechanisms of the XVSM software, a representative of coordination based middleware, which was selected to realize the communication layer, are explained.

2.1 Autonomous mobile Robots

In today's world robots find many areas of application and show a great variety in capabilities and appearance, usually adapted to best suit the tasks they are intended to perform. One of the obvious and most successful usage scenarios is the deployment on industrial manufacturing lines to perform repetitive tasks requiring a specific level of precision. Such **industrial robots** are typically big in size, heavy and stationary, controlled by relatively simple programs. Another class of robots can be defined as **remote manipulators** which don't operate (completely) on their own but rather extend the abilities of a human operator. Examples for this type are mine detecting and defusing bots, repair drones for pipelines or automated mini-submarines,

generally operating in an environment hostile to humans. Oftentimes those models are mobile but they depend on some external control input. This leads to the group relevant for this thesis, **autonomous mobile robots**. As the name already states one criteria is for the robot to be mobile, meaning it possesses some kind of driving mechanism, with wheels, caterpillars or legs being popular choices. Furthermore autonomous operation is a requirement, meaning that it performs the tasks intended without requiring any further control input from an external operator once it is placed in the environment it was meant to operate in and is activated. A typical example of autonomous mobile robots garnering much public attention are the rovers used in space missions (Figure 2.1) and nowadays some models are already starting to invade private homes in the form of service bots like fully automatic vacuum cleaners (Figure 2.2).

Figure 2.2.: Electrolux trilobite 2.0 vacuum cleaner [http://trilobite.electrolux.com]

Figure 2.1.: NASA Rocky7 experimental robot [http://www-robotics.jpl.nasa.gov/systems]

The complexity of the environment is a huge factor influencing the sensors and actuators needed to solve a task as well as the demands on the control applications. Some commercial systems try to circumvent the problem by requiring additional measures for proper operation, for example automated lawn mower bots usually require the user to set up a special wire around the grass area which the robot

interprets as boundaries. In experimental setups for research fixed and predetermined fields are used to allow concentration on certain aspects of the control application. But the fact that not all possibilities of even an restricted environment can always be considered at application design time or the impracticality of setting boundaries for an environment (e. g. space missions) led to the wish of dynamically adapting agents. This idea is a driving force behind a whole sub area of artificial intelligence research, namely Embodied AI [Broo91], which puts a physical "body" – in this case the robot – and its interaction with the environment into the center of the research of learning processes and intelligent behavior.

2.2 Overview of the existing robot system

The design and implementation of robot control applications is a very complex task, even more so in the area of autonomous mobile robots and artificial intelligence research. Furthermore, depending on the intended use case, the robot platform ideally needs to provide the flexibility to easily adapt the hardware to the given needs by adding or exchanging sensors and actuators. Those changes should also reflect on the software level, providing easily accessible application programming interfaces (APIs), which assist researchers and developers in creating the control applications. As Jörg Irran points out in his master thesis "Entwurf und Konstruktion eines in Hard- und Software erweiterbaren und modularen mobilen Robotersystems"[1] [Irr04], the systems available to researchers fall short of those requirements regarding extensibility and uncomplicated adaptability. This leads either to the use of robot platforms not ideally suited to the task at hand and restrictions of the originally planned scenario to be investigated or the development of proprietary custom robots. Designing a custom robot is a very time consuming task since it involves many steps starting at hardware design and engineering continuing with the implementation of

1 Engl: "Design and Implementation of an in Hard- and Software Extensible and Modular Mobile Robot System"

bus protocols to connect the hardware with some processing unit able to run simple control loops, concluding with the development of high level programing language APIs to enable the application programming, the actual area of research. A very demanding procedure that – if designed without an open and flexible concept in mind – leads to a system well suited for the respective project, but being potentially useless for the following research, requiring the repetition of the process. As a solution to this problem Irran introduces a design following a modular concept on both the hardware and the software level. The description below concentrates on the design principles and software aspects of the system and a brief overview of the available hardware put to use throughout this thesis. For a detailed description of hardware standards and bus protocols at work in the robot platform the reader may be referred to the referenced work.

2.2.1 The modular approach

In his thesis Irran defines a module as the basic building block of a robot system. A module consists of a hardware component as well as some additional software and information required for application development. Figure 2.3 shows an overview of all the components involved which are described in more detail below.

Module

A module integrates at least one processing unit and an arbitrary number of sensors and/or actuators. Modules acting as communication interfaces between different systems may not have any kind of sensor or actuator at all. The processing unit can reach from a simple microcontroller to a processor used in embedded systems and personal digital assistants (PDAs) to a complete personal computer. It is used to implement an abstraction layer by on the one hand covering the details of fetching data from a specific sensor hardware or sending commands to a special actuator model and on the other hand providing a specified set of services through an

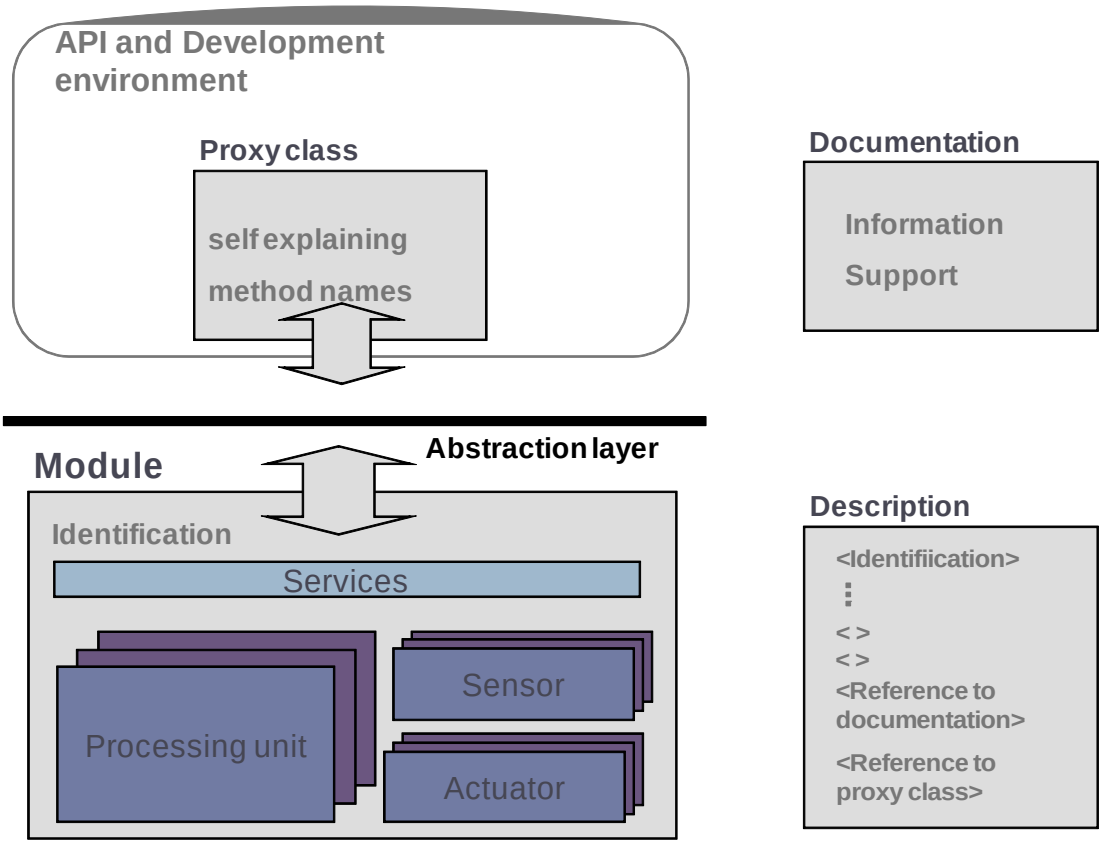

Figure 2.3: schematic view of a module and its additional components (Source:[Irr04])

external interface. The details of the implementation like internal bus logic and programming languages are irrelevant, as long as they satisfy the needs imposed through the externally accessible services. The external interface can be accessed using a data bus. The use of a bus system allows for the interconnection of several modules that together comprise the whole robot system. Each module has to implement a mandatory set of commands being:

- **identification**
  the module has to return a unique identifier, used for a plug&play mechanism during the robot building process
- **status**
  if working properly the module should return an "OK" message

- **echo**
  to test the data transmission to and from a module the message sent should be returned unaltered

Those commands are used for configuration and service tasks. Additionally a module will implement commands specific to its capabilities, for example when equipped with an infrared distance sensor commands to retrieve values indicating the distance to an object are necessary.

Proxy class

For every module exists a proxy class providing access to its services and functionality in a high level programming language. It hides the bus communication protocol and instruction set from an application developer and provides access to the special capabilities of a module by supplying an API of ideally self explaining method names for control application development.

Description

Each module contains a description with some details about the module hardware and references to its proxy class and documentation. The information therein is needed for the plug&play mechanisms and code framework generation during the construction of a robot. As a description language XML is used. Defined tags are:

`<NodeID>`	the unique identifier of the module
`<Version>`	a version number of the module
`<Name>`	a human readable name for the module
`<Details>`	details of interest about a module
`<JavaClass>`	a reference to a proxy class implemented in Java
`<DocRef>`	a reference to the module documentation

Documentation

To every module a detailed documentation should be provided containing information of the actual hardware used, operating parameters and limitations, value range and support for programming issues.

2.2.2 Important modules

Following some of the modules which are used and extended in the scope of this thesis are introduced. To get an impression of how a module looks like, a servo module with two servos connected is depicted in Figure 2.4.

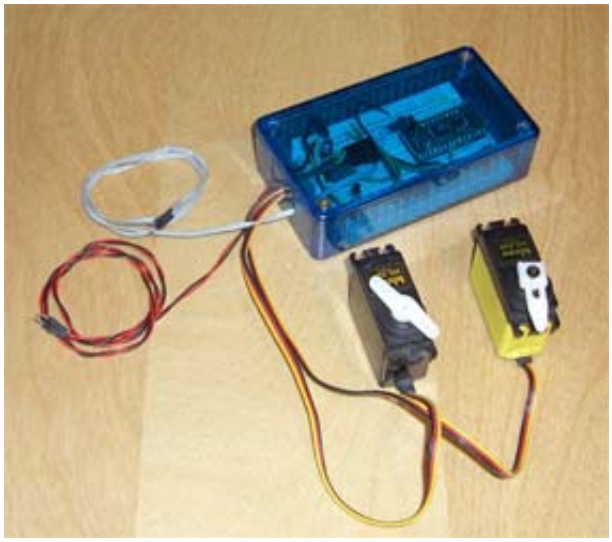

Figure 2.4.: a servo module with two connected servos

Interface converter module

Since the RS485 data bus was chosen to implement the inter module communication, but most PC systems and PDAs available today only support the RS232 bus, better known as *serial port*, the interface converter module is essential for control applications running on common computers to interact with the modules of a robot system. This module communicates according to the RS232 bus

specification on one side and translates bus messages adhering to the RS485 standard on the other side.

Bumper sensor module

Bumper sensors are used to detect collisions, i. e. whenever the robot establishes physical contact with objects in its environment. The bumper sensors are realized as a 50mm metal strip attached to a digital micro switch. Whenever some kind of force is exerted on the metal strip the micro switch is pushed down and indicates a collision. The module is equipped with eight bumper sensors allowing for placement on different positions on the robot platform. Sensors can be queried separately or simultaneously.

Infrared distance sensor module

This module combines a set of seven infrared sensors that can measure the distance to another object. Distance calculation is based on the principle of triangulation, where an infrared beam is sent out and reflected by an object. The reflected beam is registered at a sensor and depending on the angle of incidence the distance to the reflecting object can be deduced. Sensors can be queried separately or simultaneously.

Servo control module

A servo is an actuator that allows for the rotary movement of a cantilever arm to an angle between 0 and 180 degrees. Depending on the way how servos are assembled they can be put to use as parts of some kind of grabbing device like an robotic arm or crane. The servo control module enables the separate movement of up to eight servos.

Motor control module

The motor control module enables the control of direct current electronic motor controllers which can be used to determine the movement of the mobile robot or some movable parts attached to it. The module supports up to eight motor controllers which can be addressed independently.

2.2.3 Creating a Robot

The modular design described above provides the flexibility to create robots with lots of different configurations. Because every module has to provide a proxy class in a high level programming language the complicated aspects of the underlying hardware design and bus protocols are hidden from the control application developer. The presented system implements a plug&play like approach to creating a robot. A researcher simply chooses a variety of modules that can provide the functionality needed for a given scenario. All these modules are then connected using the data bus. A special interface module allows for access to the bus from a computer using the standard serial port (RS232). Furthermore a piece of software named the *RobotExplorer* that queries the data bus for attached modules via the serial port is supplied. Once a module is recognized on the bus its description is fetched to the *RobotExplorer* software, which collects all the module descriptions and assembles a complete description of the created robot. This robot description is the basis for the creation of the control application programming framework and robot documentation. Subsequently the *RobotExplorer* generates a programmatic representation of the whole robot by instantiating the proxy class of each module present in the robot description and initializing it with the respective parameters again found within the robot description. The final output is a Java application framework integrating the APIs of all modules used in the created robot, ready for the development of a probably complex control application. Additionally the references to the documentation – realized as HTML documents – are assembled to provide a

complete overview of the robots capabilities. Figure 2.5 shows a picture of a completely assembled robot consisting of a caterpillar drive controlled by a motor module, an arm realized with the help of a servo module and a laptop computer for running the control application.

Figure 2.5.: example of a completely assembled robot

To summarize, the modular robot system used in this work simplifies the process of adapting a robot platform to a specific scenario to the point of connecting some modules and running a program to be provided with a high level programming language framework ready for application development. It allows researchers to use a rapid prototyping approach even in the field of robots and Embodied AI. This thesis aims to elevate this goal to the level of multi robot and coordination scenarios.

2.3 The X10 System

To extend the idea of robot coordination beyond mobile autonomous robots, the architecture proposed in this thesis allows for interaction with sensors and actuators placed in the environment the robots operate in. As a simple solution to demonstrate how such a scenario could look like, the X10 Home Automation System was chosen

as a provider for environmental manipulators. X10 is an international industry standard for the communication between electrical devices used for home automation tasks, such as the automated and centralized switching of lights spread out throughout a house, and was originally developed by Scottish firm Pico Electronics in 1975. It is comprised of a technical description of the physical signal transmission process using existing power line wiring in private homes or short range radio signals as well as a specification of a binary communication protocol. Because no additional wiring is required and the availability of cheap devices and adapters, X10 quickly grew popular in the North American markets and became the dominating system for home automation. Due to differences in the domestic power supply systems which made production of supporting devices more expensive for the European market, the technology only caught on in Europe in recent years.

2.3.1 The physical layer

As mentioned before the physical medium for message transmission used by X10 is the existing power line infrastructure within apartments or houses. American and European households use alternate current (AC) power supply with a frequency of 60Hz (America) or 50Hz (Europe). That means the current changes following a sinus wave crossing the zero point 120 or 100 times per second respectively. X10 uses those zero crossings as points of synchronization where devices try to transmit data as close to a zero crossing point as possible, but at least within an interval of 200 µs (microseconds) around an intersection. A binary 1 is represented as a 1 ms (millisecond) burst of a 120kHz signal with a power of 300mW followed by the absence of a pulse at the next crossing whereas the absence of such a burst followed by a pulse stands for a binary 0 (Figure 2.6).

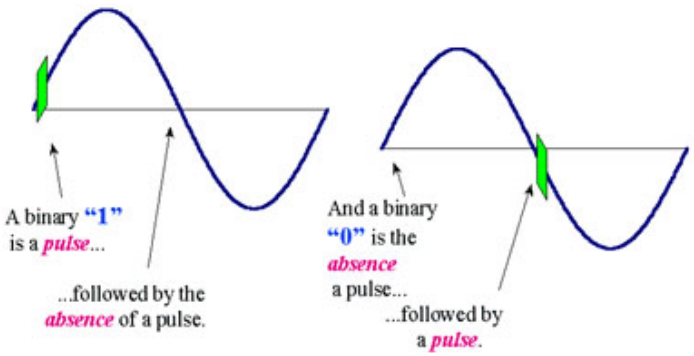

Figure 2.6.: physical representation of binary data in X10 (Source: [King99])

2.3.2 The binary protocol

The X10 protocol allows for the addressing of 256 devices, where the address space is divided into a 4 bit house code and a 4 bit device code. Each device – also called module – listens to one address, which can be set on the hardware itself. For better usability the house code is mapped to the letters A through P whereas the device code is a choice among the numbers 1 to 16. Each message sent from a sender to one or more receivers has the format depicted in Figure 2.8.

HOUSE CODES					KEY CODES					
	H1	H2	H4	H8		D1	D2	D4	D8	D16
A	0	1	1	0	1	0	1	1	0	0
B	1	1	1	0	2	1	1	1	0	0
C	0	0	1	0	3	0	0	1	0	0
D	1	0	1	0	4	1	0	1	0	0
E	0	0	0	1	5	0	0	0	1	0
F	1	0	0	1	6	1	0	0	1	0
G	0	1	0	1	7	0	1	0	1	0
H	1	1	0	1	8	1	1	0	1	0
I	0	1	1	1	9	0	1	1	1	0
J	1	1	1	1	10	1	1	1	1	0
K	0	0	1	1	11	0	0	1	1	0
L	1	0	1	1	12	1	0	1	1	0
M	0	0	0	0	13	0	0	0	0	0
N	1	0	0	0	14	1	0	0	0	0
O	0	1	0	0	15	0	1	0	0	0
P	1	1	0	0	16	1	1	0	0	0
					All Units Off	0	0	0	0	1
					All Lights On	0	0	0	1	1
					On	0	0	1	0	1
					Off	0	0	1	1	1
					Dim	0	1	0	0	1
					Bright	0	1	0	1	1
					All Lights Off	0	1	1	0	1
					Extended Code	0	1	1	1	1
					Hail Request	1	0	0	0	1①
					Hail Acknowledge	1	0	0	1	1
					Pre-Set Dim	1	0	1	X	1②
					Extended Data (analog)	1	1	0	0	1③
					Status=on	1	1	0	1	1
					Status=off	1	1	1	0	1
					Status Request	1	1	1	1	1

Figure 2.7.: complete code table of the X10 binary protocol (Source: [Smar07])

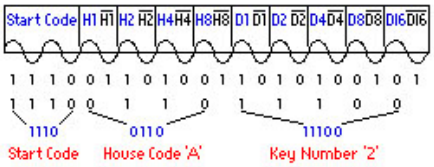

Figure 2.8.: format of a X10 message (Source: [Smar07])

A message is always started with a start code, the predetermined 4 bit sequence 1110. The subsequent 4 bits designate the house code. Then an arbitrary number of 4 bit device codes can follow, where after each one a single function code bit is sent. The function code bit tells the receivers whether to interpret the preceding 4 bits as a device number or a function. If the function code bit is a 0, a device number was sent, in case of a 1, the bits transmitted before represent the function to be executed. Due to this protocol design, a function code always ends a message and the number of clients addressed with one message is limited to 16 – all the ones that share one house code. The minimum allowed X10 data packet of one start code, on house code and one function code is called a X10 frame. For means of fault tolerance and error correction, every bit except for the start code is transmitted first in its true followed immediately by its complement form and every frame is sent twice without interruption in between frames. Two different messages have to be separated by six consecutive zeros. A full description of the protocol can be found at [X1006].

As a concluding remark the X10 system shows several weaknesses. Originally it was designed as a pure one way communication protocol, which means that the receiving devices had no possibility to send any feedback about the success or failure of the execution of a function to the controller issuing the command. Therefore it is not suitable for critical tasks where permanent and exact knowledge about the state of every component in a system is important. Newer revisions of the protocol added extensions that allowed for response messages without breaking backwards compatibility. However, special hardware is needed to use those back communication mechanisms. Furthermore, compared to todays standards, X10 is very limited in regards of bandwidth and speed. Considering all necessary idle circles and redundancies leaves an effective average bandwidth of 20 bit/sec, which is only useful for the transmission of simple commands and not for data transfers. Also due

to the use of the alternate current frequency of the power line infrastructure of 50Hz or 60Hz for synchronization the speed is locked at a very low level.

Despite those disadvantages, the simplicity of the protocol and inexpensive hardware outweighed the negatives and lead to X10 being used as a proof of concept platform for this thesis.

2.4 Virtual Shared Memory and XVSM

The basis for all communication between robots and environmental controllers presented in this thesis is the virtual shared memory middleware XVSM, an acronym for e**X**tensible **V**irtual **S**hared **M**emory. Basically the presented architecture allows for arbitrary communication mechanisms to be implemented, but the combination of virtual shared memory and robotics was one of the driving ideas behind this work and hence XVSM was chosen as a representative of a virtual shared memory system.

The term middleware designates a layer of software used in networked systems to hide the heterogeneity of the underlying platforms in a distributed system and to improve distribution transparency in applications [Tann02]. This layer is placed between the operating system and applications to provide a higher level of abstraction of communication mechanisms. An important goal of middleware is to provide as complete a set of services as possible, so that applications only rely on the interfaces provided and the direct use of operating system services is discouraged. Such services include communication facilities, naming, synchronization, persistence, replication, fault tolerance and transactional safety.

Most middleware in use today is based on some model or paradigm for describing the mechanisms of distribution. A relatively simple approach is to use only files for means of communication, hence those systems are classified as **distributed file systems**. As an example the Network File System (NFS) from Sun Microsystems can be named. Given the fact that in Unix based systems even hardware devices possess a representative entry in the file system which allows for programmatic access and

manipulation, a shared file system provides a common view of available resources and allows for the file to be the common denominator in application development. Another model is followed by **Remote Procedure Calls** (RPC) where the emphasis lies on hiding the network communication by allowing programs to call procedures where the implementation is located on a remote machine. The marshaling of passed parameters and results as well as the transmission to and from the remote host are all handled by the middleware and thus the calling program remains unaware of the network exchange. With the development away from procedural programming languages to the object oriented paradigm the need for object invocation across machine boundaries arose and led to the development of **distributed object systems** that kept the advantages like inheritance, polymorphism and encapsulation. Distributed objects are usually specified by their interface, with the object residing on one machine and the interface made available on many others. In case of a method invocation on a remote host the interface implementation connects to the object, executes the respective method and fetches the result. The model of a **Virtual Shared Memory** adds another layer of abstraction on top of distributed objects. Programs are completely unaware of where an object resides. Instead a virtual storage space for all distributed objects is created and every program only interacts with this virtual space. Objects are being made accessible for others by being written into the space. The space itself can either be realized on a server accessible for all participants or, following a peer-to-peer approach, emerge from a fusion of local memory areas of participating clients with the help of coordination mechanisms. Corso (Co-Ordinated Shared Objects), developed at the Vienna University of Technology is a representative of a Virtual Shared Memory middleware. **Coordination based systems** pick up the idea of a common space shared between members of a distributed system but put the emphasis on object use for coordination tasks rather than data storage or complex distributed data structures. Object access is restricted to a set of simple operations and a template matching system. Such a system was first proposed with the introduction of the Linda coordination language [Gele85].

JavaSpaces from Sun Microsystems and XVSM used in this thesis belong to this category and provide several extensions to the original concept.

2.4.1 XVSM – eXtensible Virtual Shared Memory

The e**X**tensible **V**irtual **S**hared **M**emory system is a coordination based middleware currently in development at the computer language department of the Vienna University of Technology which extends the Linda model by further coordination possibilities [Kühn05]. It provides a set of basic data structures, so called containers, defines the objects that can be put into those containers, namely entries (and their specialized form tuples), and specifies operations possible to perform on the shared objects. Additionally it implements a notification mechanism to raise events at participating clients in case a predefined action of interest occurs on an observed data structure. The following sections give a short overview of the mechanisms at work in XVSM and the possibilities created for distributed application development.

2.4.1.1 Entries and Tuples

In XVSM the basic entity to be placed in a shared space is called an entry, which is a single value of one of the predetermined types. Those entries are stored in so called containers (see below). Valid types are **integer**, **string**, a reference to a container (**cref**) or a **tuple**. Tuples are an arbitrary sequence of values of the types above. The length of the sequence is called the arity of the tuple. Following pointed brackets (<, >) are used to denote a tuple. An example for a tuple would be <4, 7, “test”> using the type structure <integer, integer, string>. Note that since any of the predetermined value types can be used, tuples inside tuples are also a legal construct, e. g. <integer, <string, cref>>. Tuples describe themselves by their structure, used data types and implicitly by their values. They are explained here because XVSM

allows for tuple access using pattern matching with provided template tuples, a fundamental concept of the original Linda coordination language.

2.4.1.2 Coordination Types

A major goal of XVSM is to provide a variety of coordination patterns to enable efficient communication in distributed applications. Therefore multiple coordination types are specified which can be categorized as follows.

Order

A container always has a basic coordination type that defines the way entries inside the container are accessed. Possible values for order are:

- **Random** (default):
  entries are written and read from the container in a completely random manner
- **FIFO** – **f**irst **i**n, **f**irst **o**ut:
  the Entry written to the container first will be the first returned on a read operation (Queue)
- **LIFO** – **l**ast **i**n, **f**irst **o**ut:
  the Entry written last to the container will be the first returned on a read operation (Stack)
- **LRU** - **l**east **r**ecently **u**sed:
  the entry that hasn't been in use for the longest time is accessed

View

Views can be set on a container in addition to their base coordination type. They require additional information to be supplied for operations to be successful. Available views are:

- **Key**
  provides the possibility to identify entries by a unique identifier (similar to database keys)
- **Vector**
  gives a list view of all entries in a container and allows for access using an index

Template

In XVSM it is possible to use tuples as entries in a container. The use of templates allows to filter tuples based on their content and structure. Of course such a filter may return more than one matching result. In this case the results are sorted according to the base coordination type of the housing container. When defining a template the parameters **arity**, **value types**, **values** and **count** influence the output of the matching process. The following example illustrates the functionality of template matching coordination:

A template defined as <integer, ”test”, ?> matches all templates in a container having the arity 3, an arbitrary value of type integer on the first position, an exact matching string “test” on the second and any type or value on the third. If no value for count is explicitly specified only one result is returned in case of multiple matches.

2.4.1.3 Containers

The basic data structure in XVSM is the container, a construct that can be viewed as a structured subspace offering a variety of possibilities for coordination. In XVSM the use of containers is mandatory, shared objects always have to be written into a container. Writing an entry directly into the shared space is not possible. For each container certain properties can be set:

- **Capacity** (mandatory)
  The number of entries that a container can hold at a time can be limited to a certain size. If no size is specified the container is unbounded and will grow dynamically. Size can be of importance when analyzing the blocking behavior of operations.

- **Uniqueness** (mandatory)
  Containers support two different strategies concerning the uniqueness of entries. If set to *BAG* multiple entries with the same values may exists. This is the default behavior. Alternatively the uniqueness *SET* doesn't allow for duplicates.

- **Base Coordination Order** (mandatory)
  Every container always has exactly one base coordination order of the types Random, FIFO, LIFO or LRU as described above.

Additionally XVSM provides a **naming** service for containers, which means that every container can be given a system wide unique name and a reference to it can be retrieved by performing a lookup using that identifier.

2.4.1.4 Selectors

Now that the coordination types of XVSM have been outlined it is time to take a look how they can be applied to operations on containers. To be able to specifically choose the entries on which the desired operations should execute a mechanism named selectors is provided. Since every container has a base coordination order, operations without any attached selectors choose the entries following that policy. But for the coordination types view and template additional information is needed, namely a key, a vector index or a template tuple. This is passed to the operation by attaching a selector of the appropriate type. Sometimes it might even be useful to

choose entries in an order deviating from the base order type. Hence also alternative order types can be used to select entries through selectors. To sum up, selectors of every coordination type exist to provide the greatest flexibility possible for choosing entries on which operations should be performed.

2.4.1.5 Operations

For putting entries into containers and retrieving them XVSM defines a set of operations which can be categorized into two main groups, read operations and write operations.

Read Operations:

- **read**

  Entries are read out taking any given selectors into consideration. If no additional selectors are specified return values follow the policy defined through the base coordination order.

- **take**

  This operation can be seen as a consuming read or atomic read and destroy operation. Entries accessed during this operation are removed from the container.

- **destroy**

  In this case the selected entries are removed from the container but they are not returned to the process that issued the function call.

Write Operations:

- **write**

  This operation puts entries into a container taking the base coordination order into consideration.

- **shift**
  Shifting entries into a container is similar to a write but in case capacity boundaries of any form are met an existing entry is shifted out of the container regarding base coordination order or a possibly given selector.

On an additional note it must be mentioned that some of the operations can show a **blocking** behavior. Read operations will block whenever the entries selected for reading are not yet present or not to an sufficient amount satisfying a requested count. Also the write operation may block in case a container met its boundary, the position determined by a selector is already occupied or the containers *SET* - property prevents a duplicate from being inserted. A shift operation never blocks.

2.4.1.6 Notifications

Another mechanism provided by XVSM are notifications. Clients are able to declare their interest in changes of a container by registering for notifications. When certain actions occur an event is raised at the registered participants of the distributed application, giving them the possibility to trigger the desired reaction. Notifications can be configured to fire only when a specific type of the available operations read, take, destroy, write and shift occurs or on any type of operation. Additionally the destruction of a complete container can be observed. How long the registration for a notification is valid is determined through the combination of a timeout value and the selection of a notification mode, a choice between:

- **once**
  the notification is canceled after the first time it fired
- **prolong**
  each time the notification fires it is prolonged for the full timeout period
- **restricted**
  the notification fires every time until the timeout expires

- **infinite**
  the notification stays active until it is canceled

At the time of the writing of this thesis the development of XVSM has not been finished, therefore available implementations still lack some important middleware services like replication, transactional safety or fault tolerance.

3 Description of the architecture

In this chapter an architecture that facilitates the goals pursued by this work is outlined. First the basic ideas of sensor and actuator sharing and a system component description as the central point of the system architecture are introduced. Those principles impose requirements onto the software to be designed which leads to an analysis thereof. Subsequent some definitions of importance for the understanding of the proposed system are given. Finally a description of the components and the way they work together to build the complete architecture concludes the chapter.

3.1 Basic concepts

Sensor and actuator sharing

A basic idea of the proposed design is to use sensor and actuator sharing as the means of solving tasks requiring the coordination of multiple robots, supporting distributed control applications and enabling resource monitoring. The decision for this concept is based on several considerations. First there are some more obvious advantages of sharing resources in a team of robots. One factor is cost. A particular kind of equipment needed can be rather expensive and thus it can be an advantage if only one member of a team is outfitted with it, but the others can still utilize that resource they do not possess by receiving access to the data provided by the single shared instance. Another aspect that can be covered by opening up the sensors and actuators of a robot for use by other members of a robot team is fault tolerance and hence greater reliability on a group to solve the given task. Whenever some kind of detector on one robot fails that is also present on another one, granting access to a still working one might provide the ability to cover the defect and keep the affected

team member operational and in the best case make the difference between a successful mission and the necessity to abort, as investigated in [Park05].

A holistic system view

Aside from economic and reliability reasons the idea enables new ways how to approach the development of a distributed control application. Taking a look at a single robot scenario, a developer is aware of the hardware at hand as well as its capabilities, has direct access to it using some kind of programming interface and is guided and supported by the development platform to write a control application that covers all the needs and requirements imposed by the task to be solved. By enabling resource sharing in a way that the developer is provided with a holistic view of all the sensors and actuators of all robots in a team or placed in the teams environment, more or less the same approach as in a single robot scenario can be taken to develop an application that controls the team. Virtual shared memory middleware as described in chapter 2.4 lends itself as a fitting analogy for the realization of the holistic system view as it follows a similar principle. All shared objects are written into a shared space and all clients solely communicate with the space. The space contains the objects created by all clients and hence provides a holistic view of the objects used in the distributed system. Creating object representations of each sensor and actuator in the space leads to a uniform view of all the resources available in a distributed control application from each robot's point of view, thus making application development for each robot a homogeneous process.

Taking this approach to the extreme it eliminates the need for multiple control applications even though there is more than one robot involved in a task solving process. Since every participant has access to all resources it is perfectly possible that one robot, probably with the most processing power, acts as a master running the application which puts the others to use as agents controlled through subroutines. The agents only need to run the code necessary to share their resources, which can be automatically generated and needs little alterations. Of course such a scenario puts

high demands on the bandwidth and response time of the communication layer since every command used in the agent subroutines must be transmitted using some kind of (wireless) network and still time critical issues might be involved in the solution of a task.

Another – maybe more feasible – option is to rely on specific control applications for each robot and only make occasional use of the resource sharing possibilities when needed in the collaborative parts of a mission or to solve fault tolerance issues. This way implies a completely different set of challenges. While sharing sensor data poses no real problem, the sharing of actuators can become quite tricky. What if more than one client wants to use the same shared actuator at the same time? The application designer needs to take precautions to prevent such situations and define to which *sphere of control* ([Kope97], page 98) the actuator belongs at any given time. Also the location of the resource providing data may have an influence on whether the supplied data is usable, as environment conditions can vary between different points of the area a robot team operates in. But these are problems common to distributed control applications and have to be considered during the application design. This task cannot be taken away from the programmer by the communication layer, the architecture simply aims to enable the sharing of sensors and actuators and providing a uniform way to access them.

System component description

Another important design goal is to enable distributed development of control applications for a single robot as well as multi robot scenarios. The programming of a distributed system oftentimes leads to a set of different applications each covering a certain aspect of the complete system. This is also true in the area of multi robot research projects where multiple models with different configurations may participate in the task solving process. When choosing the path of creating separate control applications for each robot outlined above, distributed development becomes a necessity since more than one application is needed. Since it is essential to provide

the ability to describe the resources available in a scenario the introduced system utilizes a description language and uses that description as a common denominator for the development of all the necessary control applications. Furthermore the information contained in the description can be used in software to prepare a programmable framework for developers for each robot or give details to a monitoring application to supervise the mission. Additionally it allows for the easy exchange of scenarios between different research teams and creates the possibility to start development for resources not yet physically available.

To summarize, the principles driving the design of the presented architecture are resource sharing and a holistic view of the resources available in a scenario realized with the help of a virtual shared memory middleware as well as the existence of a system component description that builds the central point of programming framework creation and exchange between distributed development teams.

3.2 Definitions

To better understand the explanation of the architecture it is first necessary to introduce several notions describing some part or concept used in the design and give a definition for some important terms.

- **Module**

  A module describes a programmatic accessible hardware unit able to provide data supplied by sensors connected to it or accepting commands for controlling attached actuators. More specific, a module consists of the actual hardware, a proxy class for programming and a description containing additional information as described in chapter 2.2.1. Furthermore, as an extension to the existing definition, a user interface component for a monitoring application is required.

- **Environment Module**

  A special kind of module, only for use in environments and not robots. Typically characterized by a property like energy consumption, size or weight that makes it unemployable on a mobile platform. The dependency on power line communication puts X10 modules in this category.

- **Robot**

  Describes an autonomous mobile robot assembled from interconnected modules and equipped with a processing unit to run a control application for those modules. The descriptions of the particular modules added together form a description of the complete robot.

- **Environment**

  Stands for the environment robots operate in or, more precisely, the resources available in the environment of a scenario that are not part of a mobile robot platform. Typically these are modules and/or environment modules connected to a processing unit for programmatic access.

- **RobotWorld**

  This (abstract) term is used to describe a scenario or, more specific, the resources available in it. It can include multiple robots and environments.

- **Viewpoint**

  A RobotWorld typically consists of multiple robots or environments. Even though the architecture tries to provide access to all resources available for every participant using the virtual shared memory paradigm, implementation details vary whether sensors or actuators are located locally or on a remote host. Therefore it is necessary for control application development to specify a viewpoint, which is the host that the application will run on. Clearly, valid choices for a viewpoint are robots or environments. Another – maybe not that obvious – possibility is that of an external viewpoint, controlling all resources from an external processing unit.

Figure 3.1 illustrates how the introduced terms relate to each other in an informal tree structure with an *n* above a term indicating the possibility of none, one or multiple instances. For each of the leaf nodes exists a description and for each parent node the description is an aggregation of the child node descriptions plus some additional information.

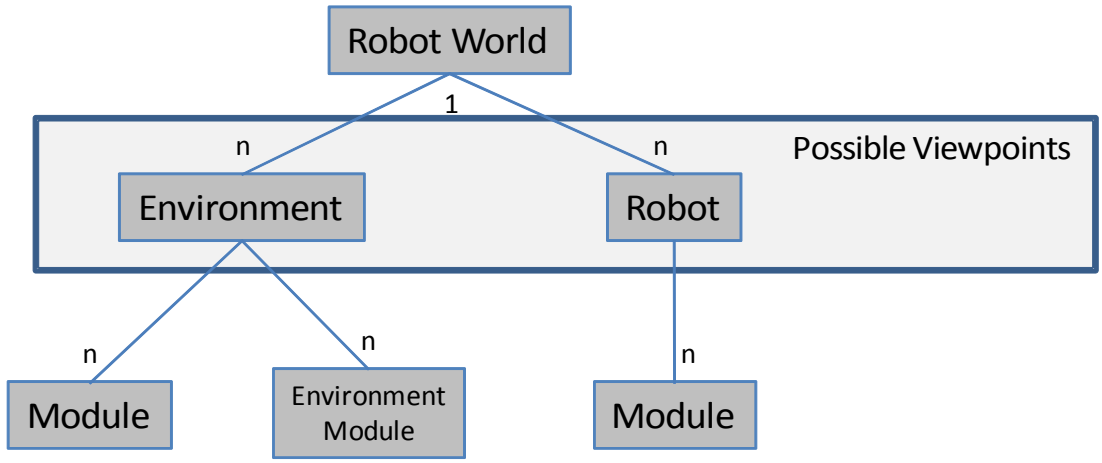

Figure 3.1: defined parts of the architecture and how they are related

3.3 Requirements

Now that the principles on which the design of the presented architecture is based are outlined, a more detailed analysis of the concrete requirements the resulting system needs to fulfill is necessary. For the sake of better understanding the description of a usage scenario helps to point out the functionality software implementing the proposed architecture should provide.

3.3.1 Usage scenario

For a research project in cooperative robotics, solutions for a specific task solving scenario have to be found and evaluated. To solve the problem a team of two robots has to be coordinated. Additionally in the environment the robots operate some

sensors and actuators are installed. Initially the two robots to be used are located at different offices of the research team. The development of the robot control applications is the essential point of the project and thus the sooner the work on that programs can start the better. At the main office, the environment for the scenario is set up and the sensors and actuators are connected to a computer. Then the team adds the environmental manipulators to a scenario description using a description language. Next a robot is configured and assembled following the modular approach described in chapter 2.2.3 and a description of the model's resources in the same description language as used for the environment is generated by software, allowing the developers to add it to the scenario description. Meanwhile, in the other office the second robot is build in the same way. Since this robot is needed for other purposes as well, the hardware can only be transferred to the main office in a few weeks time, but the project team sends the generated resource description to the main office via email. In the main office this is again added to the scenario description, completing the account of all the resources involved in the project's scenario.

The development team in the main office uses the description as an input to the software, which then creates a framework for the control application developers that provides programmatic access to all the resources using some kind of communication middleware. Furthermore the software enables the researchers to monitor the state of the whole system once a control application based on the created framework is running. For the monitoring, the values provided by the used sensors and actuators are also retrieved by accessing the data structures of the middleware layer based on the scenario description.

3.3.2 Functional requirements

Based on the usage scenario described above it is easier to deduce the exact functional requirements of the system to be contrived. It also becomes apparent, that the software is intended to support users at design time, but at the same time creates a

framework based on an architecture to be used at runtime. Hence the requirements can be analyzed taking a look at the functionality expected from each of this components.

Design time component

This part of the software needs to perform all tasks necessary to provide a starting point for control application development. When a robot assembled from several modules is connected to a computer, it should be possible to query the bus to which the parts are connected and collect information about them resulting in a description of the robots resources. This is a part already present in the existing modular robot system, albeit some modifications in the implementation will be needed. Furthermore it should be possible to load the description of a whole scenario – a RobotWorld description – which can include multiple generated robot descriptions as well as information about resources placed in the operation environment. The software should provide some kind of user interface to display an overview of the information contained in a RobotWorld description and allow for the selection of a viewpoint. Once selected a framework for control application development should be automatically generated and stored at a user specified destination. This created framework implements the design of the runtime component of the architecture.

Runtime component

The runtime component of the software has to cover the resource sharing and communication aspects of the architecture. It is realized as a framework, assembled from proxy classes of modules and exposing an easy to use API for control application development. Implementation details of the used communication layer and the locality of a resource should be hidden from the programmer. Ideally a sensor or actuator is addressed by its name – which may give an indication of the actual location – and there is no difference in programmatic access whether it is attached to the instance chosen as viewpoint or in another location.

Monitoring

To support the analysis of ongoing actions during the run of a control application the software should provide means of monitoring the state of the resources. They should be realized by interfacing with the communication layer used to implement the resource sharing at runtime. However details about the available sensors and actuators and how to establish a connection to them should be retrieved from the description of the RobotWorld and should not require manual configuration. Thus the user interface of the design time component may be extended by a possibility to load a RobotWorld description, connect to the resources defined therein and open a user interface displaying their state.

Concluding it can be seen, that the description of a RobotWorld is a central part of the architecture since the information contained in it is gathered and subsequently required in the design process as well as useful during runtime.

3.3.3 Additional requirements

Apart from the requirements on the functionality a software based on the proposed architecture should provide there are additional requirements covering properties of the resulting system or the reliance on certain technologies imposed by a specific task formulation.

Use of XML as description language

Since several mechanisms identified in the requirements analysis so far depend on the information contained in a description of the resources in a scenario, the need for an appropriate description language arises. To keep the architecture open and flexible and provide many opportunities and possible choices for its implementation, the use of XML (Extensible Markup Language) [W3C06] is a necessity. XML is the basis

for various communication standards of the Internet, the largest distributed system ever created. A plethora of accompanying technologies allows for the processing of the information described with XML. Of particular interest for this thesis are XSLT (Extensible Stylesheet Language Transformations) [W3C99] and XML Schema [W3C04]. While XSLT is ideally suited for the process of framework generation by transforming pieces of the description into initialization parameters for proxy classes, XML Schema assures that the description adheres to defined rules which is important for further processing steps. Finally there is a great variety of tools and applications available to support developers when using XML technologies and frameworks for programmatic access to XML documents exists for most programming languages of significant importance today.

Use of XVSM

Since a holistic system view and the reliance on the virtual shared memory paradigm for its realization are driving ideas of the architecture, the use of a suitable middleware for the implementation of the communication layer is necessary. XVSM, developed at the Vienna University of Technology and explained in chapter 2.4.1 is used for this purpose.

Scalability

Because the number of robots and environments and the deployed sensors and actuators in a scenario can vary greatly, it is important that the system scales well and always shows a similar behavior independent of the number of participating hosts.

Extensibility

The architecture should be designed in a way that allows to easily add new components to the system, for example new modules that are not available at the time of the implementation of the software. Also the exchange of the communication layer to other systems or newer, improved middleware versions should be possible.

3.4 Design of the Architecture

After a thorough analysis of the requirements the resulting system should fulfill, the next step is to create the actual design of the software architecture. Therefore, after an overview of the complete system, the design of the main components and the interactions between them are discussed in detail in the following chapters.

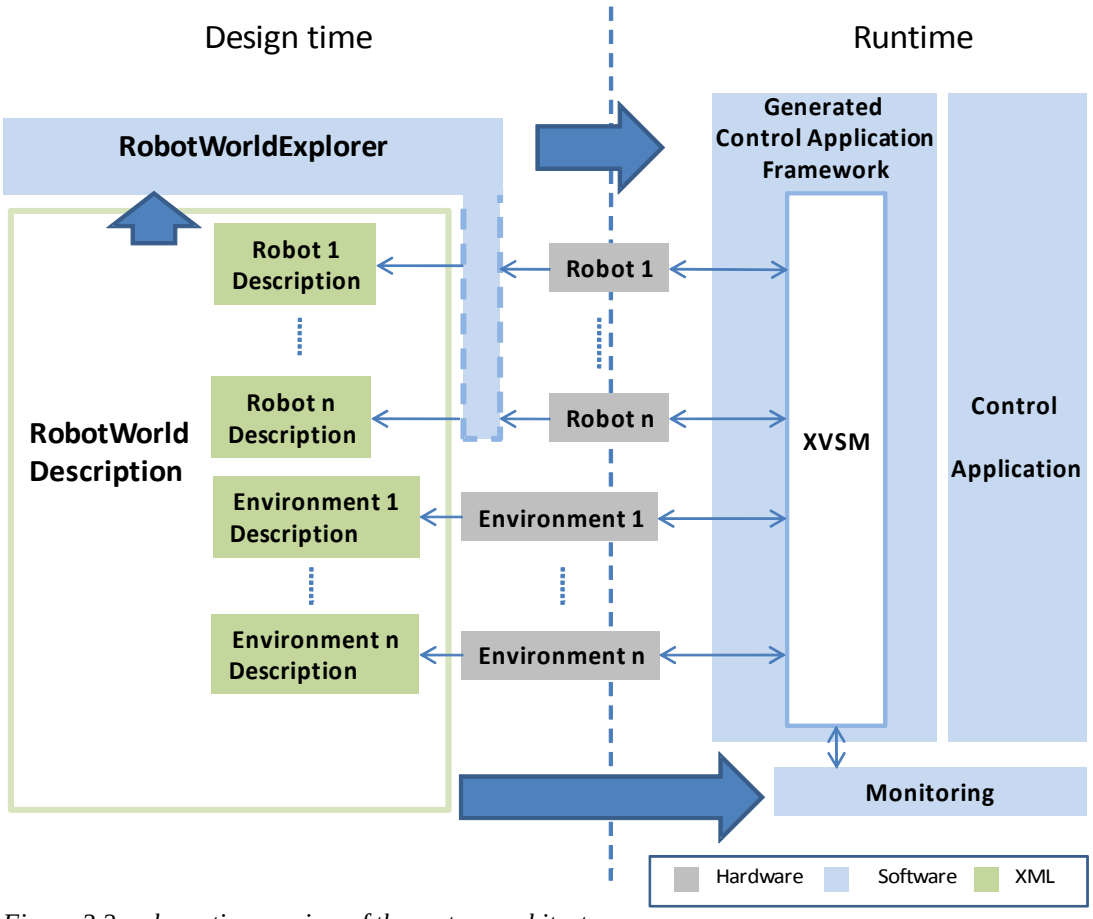

Figure 3.2: schematic overview of the system architecture

Figure 3.2 shows a schematic overview of all the components involved in the system, how they are connected and the way they depend on one another. The architecture is centered around the robots and environments or, put in another way, the hardware that should be connected through the designed system and that needs to

run at least the runtime component of the software created. For each hardware part a fitting description must exist. In case of an environment, the diversity of usable systems that allow for the placement of sensors and actuators in a scenario make the implementation of a plug&play-like mechanism for automatic creation of the description files infeasible. Therefore they have to be created manually. The descriptions for the robots however can be generated automatically through the use of the *RobotWorldExplorer*[1] software as indicated by the graphic element. The hatched area intends to outline this procedure as an optional step (manual creation is also possible) and as a part already present in the existing robot system that made its way in a slightly modified form into presented design. All those descriptions together are combined into a single RobotWorld description and used as input for the RobotWorldExplorer software that enables the automatic generation of a framework for the development of a control application for the given scenario. This framework is assembled from the proxy classes provided for every module used in the RobotWorld and represents the concluding output of the design time process. Inside the proxy classes the complete runtime architecture for the resource sharing has to be implemented. For the correct instantiation of the proxy classes information from the description is retrieved during the generation process. The control application created to actually operate the robots in a scenario relies on the architecture implemented by the framework. Finally for monitoring purposes, a software component connects to the resource representations in the XVSM space, again with the help of description input.

Since the RobotWorld description and its creation is a starting point for the development process, the next chapter will explain the details of its content and structure. Next, following the logical steps of control application development based on the proposed architecture, first the design time component and then the runtime architecture are investigated.

1The term *RobotWorldExplorer* is introduced as a name for the software implementing the architecture described in this thesis for better readability and referencing in the design description.

3.4.1 The Description

The description of the resources available in a scenario is intended to keep the development environment flexible for different configurations and simplify the startup process for a programmer by building the basis for the framework generation. However, to successfully process the information available contained in a description document, knowledge about the content and inner structure is necessary. Chapter 3.2 introduced terms and notions for all hardware components available in a scenario, which itself was named a RobotWorld. For each of this parts a description with additional details has to be provided. The goal is to finally get a complete description of the RobotWorld as it is required to trigger the framework generation process. Figure 3.1 (page 37) in chapter 3.2 depicts the relations and number of possible occurrences of each component. For the definition of the RobotWorld description it is necessary to create rules that guarantee, that for a document to be a valid and processable scenario description, its content has the structure as shown in Figure 3.1. Since the use of XML was identified as an additional requirement, the supplementing XML technology XML Schema can be used for this task. Following, the detailed XML Schema definitions for every component will be outlined, using a bottom to top approach, starting with the modules and concluding with the RobotWorld. On an additional note, the given type definitions represent the mandatory set of information necessary for the architecture to work. Depending on the concrete implementation and the hardware and middleware systems used it may be necessary to extend the given schema definitions or put some explicit restrictions onto specific element types.

Module

```
<xsd:complexType name="ModuleType">
  <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="NodeID" type="xsd:string"/>
    <xsd:element name="Class" type="xsd:string"/>
    <xsd:element name="Version" type="xsd:string"/>
    <xsd:element name="Details" type="xsd:string" minOccurs="0"/>
    <xsd:element name="DocRef" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

Listing 3.1: XML Schema definition of a module description

Listing 3.1 shows the definition of a module description. The elements can be explained as:

<Name>	the name of the module
<NodeID>	a unique identifier
<Class>	the name of the proxy class implementing the hardware access in a high level programming language.
<Version>	a version number for the module. Module capabilities may vary with different implementations.
<Details>	can contain details about some specifics of the module. (optional)
<DocRef>	a reference to the documentation of the module. (optional)

Environment Module

The XML Schema for an environment module looks exactly the same as for a module but due to the fact that there may be implementation specifics that necessitate the separate treatment of those two types, they are defined separately. Also, as can be seen in the following definitions environment modules are not allowed to be used in robots and hence need a separate type definition.

Robot

```
<xsd:complexType name="ModuleListType">
  <xsd:sequence>
    <xsd:element name="RobotName" type="xsd:string"/>
    <xsd:element name="Host" type ="xsd:string"/>
    <xsd:element name="TalkToBusClass" type="xsd:string"/>
    <xsd:element name="Module" type="ModuleType" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

Listing 3.2.: XML Schema definition of a robot description

In Listing 3.2 the definition of a robot description can be seen. Since it is basically a list of modules the specifiying type is named ModuleListType. Defined elements are:

<RobotName>	a unique name for the robot. May be necessary for the implementation of a naming service in the communication layer.
<Host>	the host computer where the control application of the robot is running. This will typically be a URL[1].
<TalkToBusClass>	the name of the class implementing the communication to the bus the modules are attached to.
<Module>	an arbitrary sequence of modules the robot is equipped with.

Environment

An environment is specified using the following elements:

<Name>	a unique name for the environment. May be necessary for the implementation of a naming service in the communication layer.
<Host>	the host computer where the control application of the robot is running. This will typically be a URL.

1 URL: Uniform Resource Locator

<COMPort>	a port where the control application can communicate with the attached modules and environment modules.
<EnvModule>	an arbitrary sequence of environment modules available
<Module>	an arbitrary sequence of modules available

```
<xsd:complexType name="EnvModuleListType">
  <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="Host" type ="xsd:string"/>
    <xsd:element name="COMPort" type="xsd:string"/>
    <xsd:element name="EnvModule" type="EnvModuleType"
                 minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="Module" type="ModuleType"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

Listing 3.3.: XML Schema definition of an environment description

RobotWorld

```
<xsd:complexType name="RobotWorldType">
  <xsd:sequence>
    <xsd:element name="WorldName" type="xsd:string" minOccurs="1"/>
    <xsd:element name="CommunicationType" type="xsd:string"
                 minOccurs="1"/>
    <xsd:element name="PackageName" type="xsd:string" minOccurs="0"/>
    <xsd:element ref="Environment" minOccurs="0"/>
    <xsd:element ref="Robot" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
```

Listing 3.4.: XML Schema definition of a RobotWorld description

Finally the description of a RobotWorld is using the elements:

<WorldName>	a name for the scenario
<CommunicationType>	allows for the specification of a communication type in case different implementations of the communication layer exist and hence the framework to be created needs to be assembled from different proxy classes.

<PackageName>	the classes of the generated framework will be grouped together into a package of the given name for better integration with development environments.
<Environment>	a sequence of environments available in the scenario.
<RobotDescription>	a sequence of robot descriptions of the available robots.

To sum up, a RobotWorld description is a XML document structured according to the type definitions above. Hence a RobotWorld can contain none, one or multiple environments and must contain at least one robot. Environments itself can be comprised of environment modules and/or modules, whereas robots are simply collections of modules. The XML Schema type definitions give at the one hand a human readable outline of the information that is expected to be provided for the framework creation process. On the other hand they are an instrument to validate existing RobotWorld descriptions for their usability with the software components of the created architecture.

The complete XML Schema for the mandatory elements of a RobotWorld can be found in Appendix B.

3.4.2 Design time component architecture

This section gives an explanation of the architecture of the software used in the design phase of a project, the component named RobotWorldExplorer in Figure 3.2. As can be seen from the schematic overview it needs to perform two tasks: first it is used to query the bus of a robot to collect information about the attached modules and create a robot description, then it has to process the information in a valid RobotWorld description and transform it into a programmable framework. Following, these two tasks are explored in more detail.

3.4.2.1 Robot description generation

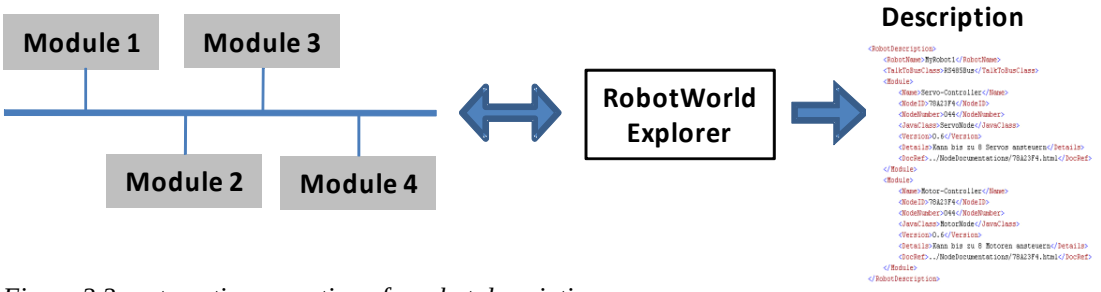

Figure 3.3: automatic generation of a robot description

Figure 3.3 depicts the process of a robot description generation. On the left side several modules attached to the robot's bus system can be seen. The bus system is connected to a PC running the RobotWorldExplorer software. To establish this communication a translator module for a widespread bus standard commonly used in PCs may be necessary, such as the interface converter module described in chapter 2.2.2. The software then queries the bus, scanning all possible bus addresses and remembering the ones where it received an answer in the form of the module's identifier. Next the module descriptions are fetched either from a storage integrated into the modules or by an ID lookup in a database and assembled into a robot description which is output into a file.

This part of the software architecture is already present in the existing modular robot system designed by Jörg Irran [Irr04] and has been identified to be a very useful mechanism to support a control application developer. It spares the manual creation of a robot description and therefore prevents mistakes of not matching descriptions and modules. Hence it was adopted for this work with some minor alterations. Changes from a design perspective are, that it is required, that the module descriptions and resulting robot descriptions conform to the XML Schema definitions introduced before, a condition not enforced in the existing solution.

3.4.2.2 Programmable framework generation

As Figure 3.2 (page 42) shows the main task of the software used at design time is to transform the information contained in a RobotWorld description into a ready-for-programming framework for a control application developer. Figure 3.4 depicts a more detailed view of the workflow the responsible part of the RobotWorldExplorer component must be able to process.

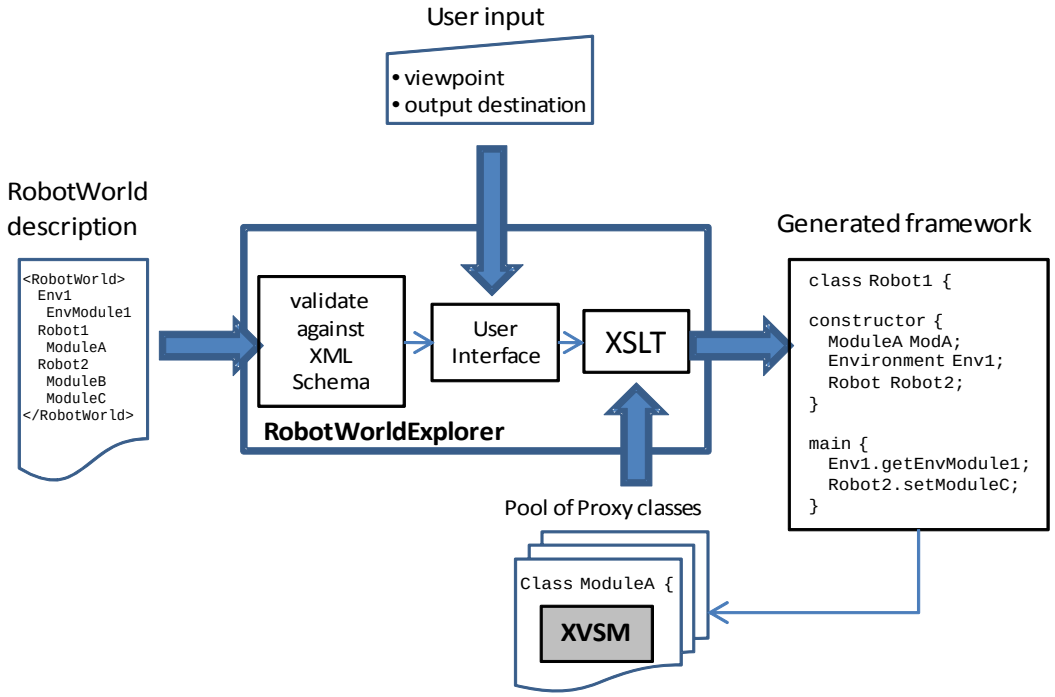

Figure 3.4: workflow of the RobotWorldExplorer software for the automated framework generation

The RobotWorldExplorer software takes a document supposedly containing a RobotWorld description. The first step is to validate the input against the XML Schema of a RobotWorld description, which is necessary to prevent easily avoidable errors in the further programmatic handling and also ensures the completeness of the information required to successfully finish the task. Once verified to be valid, the content is parsed to gain actual access to the data needed and the user has to provide additional information with the help of some kind of user interface. The user's most

important input is the choice of a viewpoint as it determines the host on which the control application is intended to run and thus influences the implementation details of the created framework. Depending on the viewpoint, proxy classes may need to be initialized with different parameters. Furthermore a name for the package or namespace under which the created classes are assembled needs to be specified if not already set in the description and a destination for the created output has to be selected.

If all necessary information is available the framework generation can be triggered. This process needs to transform the data contained in a XML document into another representation, in this case some high level programming language code. XSLT was developed just for that purpose and hence is the technology of choice for this step. For better understanding a simple example using the Java programming language as output is given to explain the procedure.

```
<RobotWorld>
  <WorldName>MyWorld</WorldName>
  <CommunicationType>none</CommunicationType>
  <packageName>mypackage</packageName>
  <RobotDescription>
    <RobotName>MyRobot</RobotName>
    <TalkToBusClass>RS485Bus</TalkToBusClass>
    <Module>
      <NodeID>15B63A4</NodeID>
      <Version>1.0</Version>
      <Name>Motor-Controller</Name>
      <Details>can control up to 8 drive controllers</Details>
      <Class>MotorNode</Class>
      <Docref>../NodeDocumentations/15B63A4.html</Docref>
    </Module>
  </RobotDescription>
</RobotWorld>
```

Listing 3.5.: example of a simple valid RobotWorld description

Listing 3.5 shows the XML of a very simple valid RobotWorld description (compare to Listing 3.4 on page 47). In this case it consists of only one robot with only one module, a motor controller, but it is sufficient to illustrate the transform procedure. As can be seen the `<Class>` tag contains the name of the corresponding proxy class for the module. Listing 3.6 below shows an XSLT document containing transformation rules to transfer the information contained in a RobotWorld

description into program code in the Java programming language. For better readability all text that is part of the Java syntax and just passed on to the output unaltered is printed in bold letters. References to elements of the XML description document are underlined. All the remaining standard text belongs to the XSL syntax.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" indent="yes"/>
<xsl:template match="RobotWorld/*">
package <xsl:value-of select="RobotWorld/packageName"/>;

<xsl:for-each select="Module">
import <xsl:value-of select="Class"/>;
</xsl:for-each>
public class <xsl:value-of select="RobotName"/> {
       private <xsl:value-of select="TalkToBusClass"/> my<xsl:value-of
                select="TalkToBusClass"/>
       <xsl:for-each select="Module">
       public <xsl:value-of select="Class"/> <xsl:value-of
              select="Class"/><xsl:number value="position()"/>;
       </xsl:for-each>
       public <xsl:value-of select="RobotName"/>() {
         try {
           my<xsl:value-of select="TalkToBusClass"/> = new <xsl:value-of
           select="TalkToBusClass"/>();
         <xsl:for-each select="Module">
          <xsl:value-of select="Class"/><xsl:number value="position()"/> =
              new <xsl:value-of select="Class"/>
              ("<xsl:value-of select="NodeID"/>",
              my<xsl:value-of select="TalkToBusClass"/>);
         </xsl:for-each>
         } catch (Exception e) {
              System.out.println("Couldn't create <xsl:value-of
                                                  select="RobotName"/>");
           System.exit(-1);
         }
       }

       public static void main(String[] args) {
              // Auto-generated method stub

              try {
                     <xsl:value-of select="RobotName"/>
                     <xsl:value-of select="RobotName"/>
                     <xsl:number value="position()"/> =
                     new <xsl:value-of select="RobotName"/>();
              } catch (Exception e) {

              }
       }
}
  </xsl:template>
</xsl:stylesheet>
```

Listing 3.6.: example for XSLT rules to transform a RobotWorld description into Java code

As can be seen, the stylesheet makes use of several XSL constructs. The `<xsl:template>` tag specifies that all the rules defined within are only applied to a RobotWorld description and its sub-elements. The `<xsl:value-of>` statements choose the literal values of the element specified by the `select` attribute in the description and are replaced by this value in the output. Furthermore the `<xsl:for-each>` constructs are very similar to loops in programming languages, iterating over all elements in the RobotWorld description that match the `select` attribute and performing the transformations enclosed between the opening and closing tag of the loop. Other than that, the function `<xsl:number value="position()"/>` is used to implement a mechanism that allows for multiple instances of the same module type in one robot or environment. In case the robot in the example would be equipped with two motor controller modules, two instances of the appropriate proxy class need to created. To differentiate the according object references the output of the mentioned function, which is the position of the element in the XML document, is used in the naming of the variables. Hence in the programming language code the two variables MotorNode1 and MotorNode2 would be used.

Now that the XSLT statements of relevance for this transformation process are explained, the merging of the Java syntax in the stylesheet with the values in the description to form a usable Java class is described in more detail.

At the top of the output file stands a package declaration where the package name is retrieved from the RobotWorld description's `<packageName>` element. Next the `import` statements for all the proxy classes of the used modules are added. Again, the name of those classes is extracted from the `<Class>` element present in each module description. This is followed by the declaration of a class named after the `<RobotName>` of the robot description, representing the entry point for the control application to be developed for the robot using this configuration. After the declaration of member variables for the bus communication implementation (`<TalkToBusClass>`) and the required proxy classes, these are instantiated in a constructor. The parameter values for the correct instantiations are retrieved from the

RobotWorld description; in case of a module the values of the `<NodeID>` and `<TalkToBusClass>` elements are needed. Finally a method stub for a `main` method creates an instance of the class just generated. Now all the modules of the robot can be accessed using this instance and the rest of the method can be filled with the actual control application logic by the developer. Listing 3.7 shows the complete output of the XSLT process.

```
package mypackage;

import RS485Bus;
import MotorNode;

public class MyRobot {

       private RS485Bus myRS485Bus;
       public MotorNode MotorNode1;

       public MyRobot() {
         try {
              myRS485Bus = new RS485Bus();
              MotorNode1 = new MotorNode("15B63A4",myRS485Bus);
         } catch (Exception e) {
           System.out.println("Couldn't create MyRobot");
           System.exit(-1);
         }
       }

       public static void main(String[] args) {
              // Auto-generated method stub
              try {
                     MyRobot MyRobot1 = new MyRobot();
              } catch (Exception e) {

              }
       }
}
```

Listing 3.7.: output of the XSLT process

In addition to the transform process the RobotWorldExplorer software needs to assemble all the proxy classes referenced by the generated class into the package and store it at the user specified destination. The result is the ready-for-programming framework for the developer who can concentrate on the programming of a control application.

Finally some of the concepts introduced for the software design that are not covered by this simple example need further explanation:

- **viewpoint:**

  For the sake of simplicity the RobotWorld in the example above consists of only one participant – a robot – equipped with one module. Since the focus of the example is on the XSLT process, the viewpoint is assumed to be this robot. In case of a scenario including multiple robots and/or environments the transformation process has to be executed for every participant and thus multiple classes are created and added to the defined package. Furthermore the transformation rules may differ whether the output is intended to be a class for the viewpoint participant or one of the others, because the viewpoint requires references to all the other's resources, the non-viewpoint participants only need to make their's accessible for the rest of the team. Also in a cooperative scenario there can be different proxy classes depending on the locality of a resources, i. e. if it is physically attached to the host running the control application or installed in a remote location. The next chapter about the runtime component architecture gives a more detailed explanation regarding these issues.

- **Communication type and resource management:**

  To keep the architecture flexible for extension and implementation changes it may be necessary to extend the XML Schema definitions, alter the XSLT stylesheets and provide alternative implementations of the proxy classes. Measures that could necessitate those changes include the change of the middleware used to realize the communication layer or the switch of the desired programming language for the output of the framework generation. Hence it would be unwise to integrate those parts as a fixed and unchangeable component into the application code. A much better solution is to keep those resources external and provide a mechanism that makes them interchangeable and accessible by the software. This is reached by the specification of a

communication type in the RobotWorld description in combination with the introduction of some naming conventions for the resource files.

As described in chapter 4.3.1 in the section explaining the XML schema definition of a RobotWorld description, it is possible to specify a communication type that indicates how the communication layer is implemented in the proxy classes. The identifier given for this type is used for a naming scheme that resources have to follow:

- XML Schema and XSLT stylesheet files for a given communication type have to be stored in a folder using this name.
- Proxy classes implementing a certain communication type have to use the identifier as a prefix in the class name. (An alternative would have been to use a folder/package/namespace named after the identifier, but this could lead to naming conflicts if different proxy class implementations exist.)

This way, adhering to the naming scheme, the RobotWorldExplorer software always accesses the files specified by the `<communicationType>` element in the RobotWorld description and produces the appropriate output. The software itself only implements a workflow it can complete as long as the description complies at least to the type definitions introduced in chapter 3.4.1 and fitting resource files are supplied.

3.4.3 Runtime component architecture

This far the design of the software involved at design time was covered. In this chapter the architecture of the runtime component implemented inside the proxy classes is explained. The goal is to provide access to all the resources in a scenario using the virtual shared memory middleware XVSM.

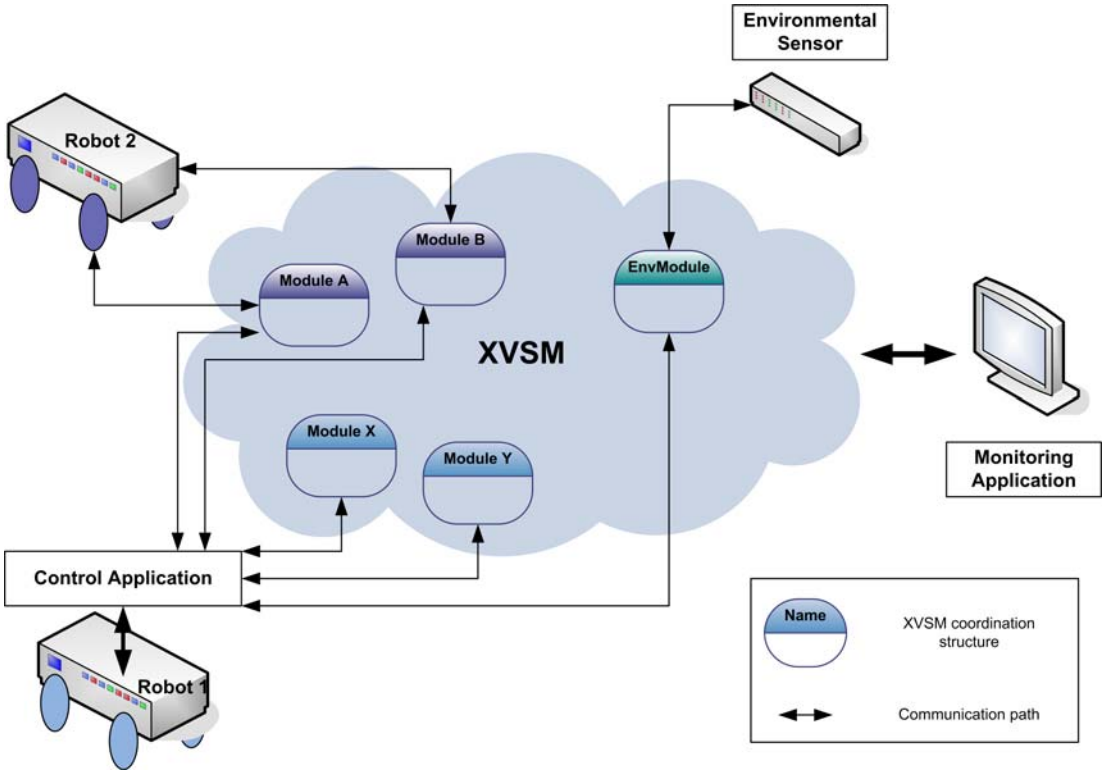

Figure 3.5.: overview of the XVSM coordination structures in a shared memory

Figure 3.5 illustrates a scenario for a control application running on Robot1. Every participant uses the XVSM coordination mechanisms to make its resources accessible to others, for example Robot2 publishes the values of the sensors attached to Module A into the space. Whenever the application running on Robot1 needs this data it can retrieve it from the space. The module hardware of Robot1 can be addressed directly, but the data is also published into the space in case another host requires it in its own control application. Furthermore a computer running a monitoring application can supervise all the coordination structures in the space. It takes the access details from the RobotWorld description of the scenario and visualizes the state of all the resources.

The next section takes a closer look at the XVSM coordination structures involved in the process.

3.4.3.1 Data structures

The architecture of the runtime component is based on a Request/Answer pattern. Whenever a control application needs the value of a sensor or wants to issue a command to an actuator it indicates this by filing a request using the corresponding data structure of the resource in the XVSM space. The host where the actual hardware is installed tries to fulfill the request, completes the desired action and reports a success value within an answer placed in a predefined structure in the space.

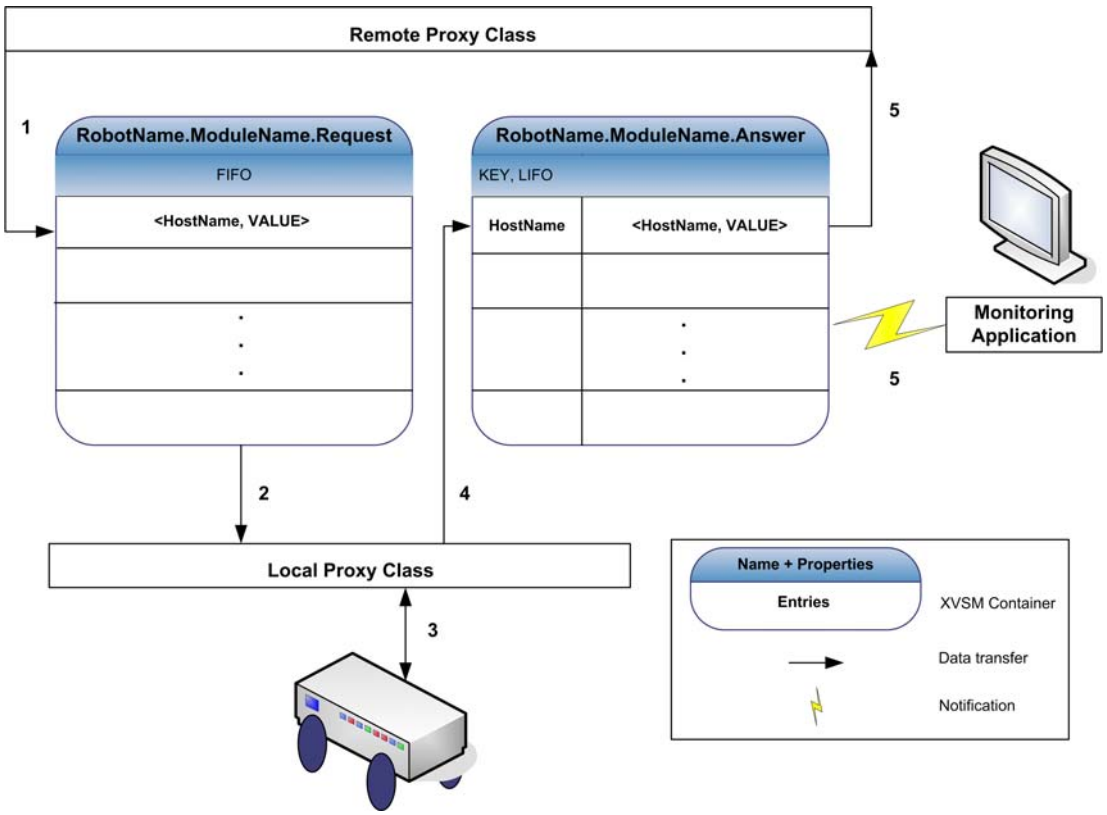

Figure 3.6.: overview of the XVSM coordination data structure – remote access

In Figure 3.6 a detailed view of the XVSM coordination data structures is depicted. It can be interpreted as a magnified view of the XVSM components in the schematic view in Figure 3.5 (page 57), which means that this structures exist once for each module. To implement the Request/Answer pattern each module has

references to two containers, one where it listens to possible requests and another one to place the respective answers. To ensure that the containers are unique within the XVSM space and can easily be referenced from other hosts they are named using the robot name or environment name specified in the RobotWorld description with the name of the module appended, followed by the suffix *REQUEST* or *ANSWER* respectively. The request container is unbounded and uses the base coordination type FIFO, hence implementing a simple queue. Whenever a remote host wants to file a request, it places a request entry into the container and the host with local access to the resource retrieves the entries in the order they have been put in the queue. The entry itself is a tuple containing the name of the requesting host and a value. Depending on whether the module manages sensors or actuators the value is interpreted in a different way. In case of sensors it may indicate the sensor number if multiple are attached to the module or it may just be ignored. If the receiving module controls an actuator the value may contain the desired action to be performed. However, the interpretation of the value and its type (integer, string, cref, tuple) are up to the implementation of the fitting proxy classes.

The answer container is also unbounded and uses LIFO as a base coordination type, but in addition the KEY property is set and entries are written using a key selector. As a key the name of the requesting host passed in every request entry is used. This way answers for a specific host are always written to the same position in the answer container. The layout for the answer structure provides the benefit that the host that initially filed the request can perform a blocking read operation on a specific entry and does not have to supervise the whole container, thereby being only activated when data is written that is meant for it. If for example the answer container would follow the same queue approach like the request container, all hosts that requested data would have to process all the entries put into the answer container, wasting computing time for filtering data not necessarily meant for them. On the other hand it allows for a notification to be placed on the whole container and thus a

monitoring application can supervise all the changes, disregarding the hosts actually involved.

Another aspect becoming obvious when analyzing the data structures depicted in Figure 3.6 is that there need to be two different implementations of a module's proxy class depending on whether the hardware resource is actually installed on the same host that runs the control application or on a remote host. In the first case the code has to cover the handling of requests, the communication to the robots module bus system and the returning of answers. When a remote host calls a method exposed by a module the proxy class implementation needs to place a request and retrieve the answer. A solutions to this problem is to describe the functions of a module in an interface and implement this interface in two different ways. This way the locality of a module makes no difference for control application programming. This is a procedure commonly used in distributed object middleware as already described in chapter 2.4.

3.4.3.2 Communication

Now that the data structures are explained, the communication steps happening when a control application accesses a remote resource are described. As a starting point for the communication sequences outlined below the following state of the system is assumed:

- On the host where the module is physically connected, a control application has instantiated the local proxy class implementation which created the two containers in the space and performs a blocking *take* operation on the request container waiting for request data.
- On the remote host that wants to access data provided by the remote module, a control application has instantiated the remote implementation of the proxy class which has retrieved references to the two containers and performs a blocking *take* operation on the entry specified by the name of the remote host used as key.

- A monitoring application is running on a third host and has retrieved a reference to the answer container and started a notification on the whole container.

Remote access

For each step the corresponding number can be found in Figure 3.6.

1. The remote implementation of the proxy class creates a request tuple containing the name of the requesting host and a value indicating the desired action. Then the tuple is put into the request container using a *write* operation.
2. On the host with physical access to the module hardware the blocking *take* operation unblocks and fetches the tuple from the request container.

The use of a *take* operation removes the first item from the queue. If more are waiting, they are fetched subsequently. Otherwise if the container is empty after the operation, the next *take* blocks until another request is put into the queue.

3. The content of the tuple is interpreted and the desired operations are performed exchanging data over the local module bus.
4. Once the operations complete the result is written to the answer container using a *shift* operation on the position chosen by the *key selector* with the host name included in the tuple.

Robot control applications often include time critical sections and are mostly sequential in nature. The use of the *shift* operation guarantees that no blocking can occur when an answer needs to be written which is important because the host can continue to process the remaining requests in the queue and does not stall the pipeline. Furthermore, if the host waiting for the answer should have been busy and did not retrieve the answer written before, the newer value will be the better choice anyway.

5. At the remote host, which initiated the communication, the *take* operation on the entry specified by the host's key unblocks. The proxy class implementation picks up the entry and interprets the content. At the same

time a notification fires at the monitoring application which in turn retrieves the entry just written by the remote host and updates the visual status representation of the corresponding module.

Local access

Figure 3.7 depicts the simplified case of accessing a module using the local implementation of its proxy class, a scenario only involving the answer container.

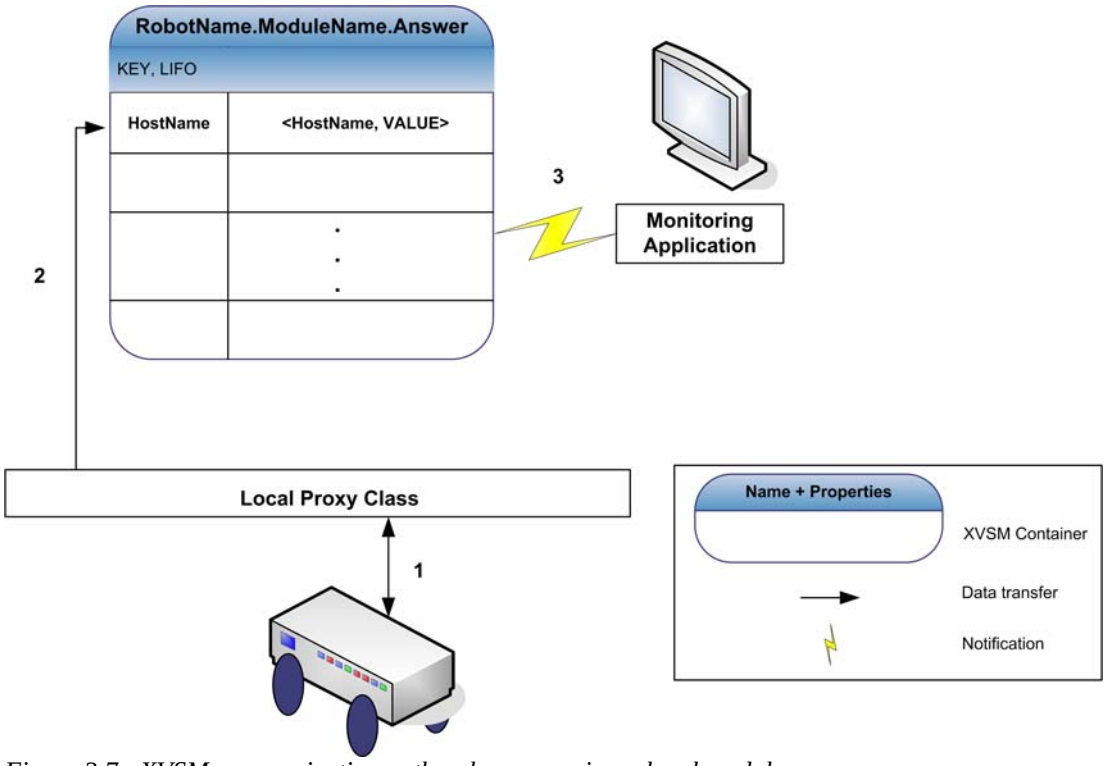

Figure 3.7.: XVSM communication paths when accessing a local module

1. A control application accesses local hardware using the module bus and retrieves a value.
2. The local proxy class implementation puts the value into the answer container using a *shift* operation with its own host name as a key for the *key selector*.

3. At the monitoring application a notification fires, the new value is retrieved using a *read* operation and the visual representation of the modules status is updated.

Depending on performance issues the local access synchronization with the XVSM space may be implemented in a way that it can optionally be deactivated, thus saving the time and processing power needed for the space communication. However the monitoring application will suffer from this and can only provide a correct status view in case a remote access occurs. In situations where monitoring is not necessary and the importance lies on optimizing the runtime behavior of a control application the trade off might be worth it.

The presented runtime architecture satisfies the requirements identified in the analysis phase. It provides access to all modules in a scenario by providing appropriate XVSM coordination structures in the virtual shared memory. The use of unbounded containers allows for an arbitrary number of clients to file requests and retrieve answers. Since a client's notification only supervises a single entry, control applications running on these clients are only alerted when a value that was explicitly requested is available. All other communication traffic of the scenario's other participants is hidden from the hosts and therefore network traffic is minimized. Still the design allows for a monitoring application to supervise all communication happening in the space without influencing the host to host performance. Furthermore the data structures also leave room for possible improvements. For example, since all requesters in a queue are interested in the same resource, an alternative implementation could take all outstanding entries from the request container simultaneously and write the value fetched from the hardware to multiple positions in the answer container, hence satisfying multiple requests with only one hardware access.

4 Implementation

After a detailed description of the software components and their architecture this chapter covers a prototypical implementation of the proposed system using the technologies outlined in chapter 2 and the Java programming language. Additionally a Java version of a subset of the X10 protocol important for this work needed to be implemented.

4.1 The development environment and used frameworks

Before the the details of the implementation are explored, the following sections give an overview of the development environment that was used to create the applications. An important tool to support programmers in the development of a complex application is the Integrated Development Environment (IDE), which provides mechanisms such as source code highlighting, code completion, refactoring and debugging features. Furthermore, as XML was chosen as an integral part of important components of the architecture, the use of existing frameworks and tools alleviate the handling of XML documents and technologies from within a programming language and can save the developer time and effort.

4.1.1 The Eclipse IDE

A popular IDE for the development of Java projects is the Eclipse IDE, an open source project hosted by the Eclipse Foundation[1]. It provides a lot of mechanisms to support programmers in the development of complex applications following the object oriented programming paradigm. Furthermore its functionality can be

1 Eclipse IDE, http://www.eclipse.org

expanded with the help of a sophisticated plug-in system for which a great variety of extensions exists supported by a very active community.

For the implementation of this prototype Eclipse 3.2.1 was used. The user interfaces for the design time component as well as the monitoring application were designed with the help of the Visual Editor 1.2.1 plug-in of the Visual Editor Project[1], which allows for drag&drop design of graphical user interfaces based on the Java technologies AWT and Swing.

To create XML documents and test and debug their processing with the associated technologies XML Schema and XSLT used in the architecture, the XML Buddy plug-in from Bocaloco Software LLC[2] was installed. It supports such helpful features as syntax highlighting, XML tag completion and the possibility to validate documents against a schema definition.

The Eclipse IDE also comes with an integrated plug-in for the support of the Apache Ant[3] build environment, outlined in more detailed in the following section.

4.1.2 Apache Ant

Apache Ant[3] is a Java based build tool for software projects. The build process is controlled by a *build* file in which so called targets are defined that specify the actions to be executed. Additionally dependencies between targets can be given, for example a target that intends to run an application usually depends on a target that compiles the sources to complete successfully. Ant provides several benefits compared to other traditional build environments like *make*[4] or the reliance on the IDE's integrated build process. Since being programmed in Java it is platform independent and its functionality can be extended by plug-ins written in Java. Furthermore it allows for the build process to work even when just the source code

1 Eclipse Visual Editor Project, http://www.eclipse.org/vep/
2 XML Buddy Eclipse plug-in, http://www.xmlbuddy.com
3 Apache Ant, http://ant.apache.org
4 GNU make, http://www.gnu.org/software/make/

files and the ant build file are available even outside of an IDE. Software projects get better interchangeable between developers and computers with different software setups. In addition to Eclipse many popular Java development IDEs like Borland JBuilder or Sun Microsystems NetBeans support Ant. Finally, instead of adhering to some proprietary format, build files are written using XML and thus easily processable using XML technologies and frameworks.

The use of Apache Ant version 1.6.9 in this thesis is twofold: first it is used as a tool configure the build process of the prototype implementation and create deployable files of the finished project. Second the framework generator implemented also creates an ant build file for the created classes so that a control application developer can easily import the sources in an IDE or can build and deploy the program.

4.1.3 XMLBeans

For the easy integration and handling of the XML description files in Java the Apache XMLBeans[1] framework is used. It automatically creates a Java type hierarchy based on supplied XML Schema type definitions and enables the programmer to access documents compliant with the schema using accessors according to the JavaBeans specification [Sun97]. Also loaded documents are validated against the schema the type structure is based on. In addition it is possible to use a cursor mechanism to select a certain position in the document an insert new elements that are not part of the original type structure.

The use of XMLBeans (version 2.2.0) takes the tasks of XML Schema validation and manual document parsing off the developer, hence saving time and programming work.

1 Apache XMLBeans, http://xmlbeans.apache.org

4.1.4 RXTX

For the connection of the X10 Home Automation system as well as the robot module bus, communication using a computers serial port (RS-232) from within a Java program needs to be enabled. Since the default Java Virtual Machine does not implement this feature, Sun Microsystems specified the JavaComm API[1] to provide the functionality. However, only implementations for Unix based operating systems are released by the company itself. The open source project RXTX[2] provides an implementation of the JavaComm API and even additional functionality and supplies binaries for Unix based operating systems, Microsoft Windows and Apple Mac OS X.

To support a great variety of platforms for running the developed prototype and to take the platform independent paradigm propagated through the use of the Java programming language into account the RXTX implementation version 2.1-7 is used.

4.1.5 Apache Tomcat

At the time of the writing of this thesis the Virtual Shared Memory middleware XVSM realizes the space as a Java Web Application[3] that needs to be hosted on a server able to act as a Web Application Container. The requirement of a dedicated server software is only specific to the early version of XVSM used in this thesis, future versions following the peer-to-peer paradigm are already in development. Apache Tomcat[4] was used by Sun Microsystems to develop the official reference implementations for Java Servlet technology [Sun03] on which a Web Application is based.

1 Sun Microsystems Java Communications, http://java.sun.com/products/javacomm/
2 RXTX, http://users.frii.com/jarvi/rxtx/
3 Sun Microsystems Java EE Web Application Technologies, http://java.sun.com/javaee/technologies/webapps/; a Web Application adheres to a set of specifications that need to be implemented by a Web Application Container
4 Apache Tomcat, http://tomcat.apache.org/

The XVSM development team chose Apache Tomcat as Web Application Container and hence version 5.5.20 is used for the implementation presented in this work.

Table 1 below summarizes the development environment used for the realization of the prototype by giving an overview of the tools and their versions.

software	**version**
Sun Java Standard Edition SDK	6.0
Eclipse IDE	3.2.1
Eclipse Visual Editor plug-in	1.2.1
Eclipse XML Buddy plug-in	2.0.72
Apache Ant	1.6.9
Apache XML Beans	2.2.0
Apache Tomcat	5.5.20
RXTX JavaComm API implementation	2.1-7
XVSM	0.8.5.1

Table 1: overview of the development environment

4.2 Description of the concrete Java implementation

The following sections give an overview of each software component of the prototypical implementation of the proposed architecture, covering the most important classes and their methods ordered by packages. First the connection to the X10 Home Automation System is outlined, an element not directly part of the architecture, but chosen to realize the environmental resources in this prototype and hence the possibility for programmatic access from within a Java application needed to be created. Thereafter the design time component, the runtime component and the monitoring are explained which are integrated into a single application, the

RobotWorldExplorer. The main user interface dialog that is presented to a user when starting the application is depicted below (Figure 4.1).

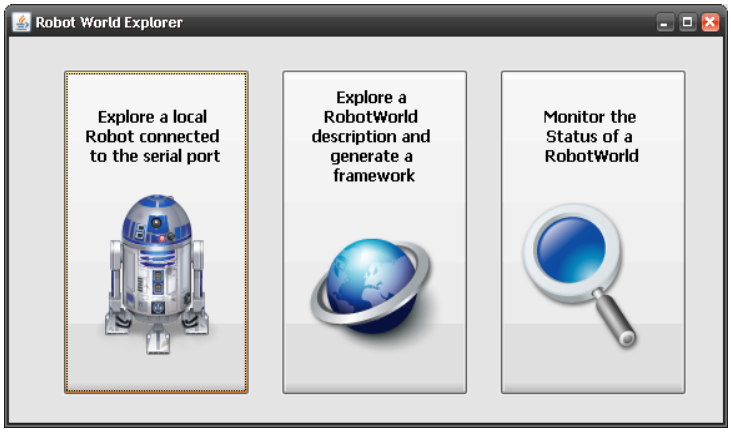

Figure 4.1.: The main menu of the RobotWorldExplorer

The left button leads to the design time component for the automated generation of a single robot description, subsequently named RobotExplorer, the middle selection starts the process of the framework generation while choosing the rightmost option allows for the monitoring of the XVSM coordination structures during the runtime of a control application.

4.2.1 X10 connection

To enable the transmission of X10 protocol messages over the domestic power line infrastructure from a computer a special interface unit is needed. For this work the Marmitek[1] CM11a controller was chosen, which can be plugged into a power outlet for power line communication on one side and connects to a PC using a serial connection (USB[2], RS-232) on the other side to receive commands. Figure 4.2 shows a picture of the device.

1 Marmitek, http://www.marmitek.com
2 USB, Universal Serial Bus

Figure 4.2: The Marmitek CM11a X10 controller

Package da9726049.x10

All classes implementing the communication to the X10 system are gathered in this package. The class *X10Constants* defines constants as byte values for all the codes of the X10 protocol, listed in Figure 2.7 (page 21), namely the house codes A through P, unit codes 1 to 16 and all the function codes. Since protocol messages are merely a concatenation of byte values this provides a convenient and better readable way to address a specific device to perform the intended function. The class *CM11aController* implements the communication to the controller Hardware using the serial port and thus requires the import of the RXTX packages to provide the JavaComm API extensions. The constructors allow for the definition of a port identifier (e.g. "COM1" in Windows environments) and transmission parameters specific to the RS-232 bus and establish a connection to the serial port endpoints using data streams. A *send()* method enables the sending of protocol messages to the controller. Furthermore predefined methods *allUnitsOff()*, *allLightsOn()* and *allLightsOff()* create and send the respective messages to trigger the according X10 functions. Since the X10 devices cannot send any feedback about the their current status, those methods might proof useful in a control application to reach a well defined starting point, for example by switching all devices off and thus have knowledge about their state. The classes *X10ApplianceModuleLocal* and *X10LampModuleLocal* can be initialized given a specific house code and unit code

and therefore each instance of on of these classes acts as a proxy object for a specific X10 device, implementing the functions it is capable off. Hence for an appliance module the methods *on()* and *off()* are available, while a lamp module supplements these with a *brighten()* and a *dim()* method. Additional device implementations can easily be added by providing an appropriate class in the package, but in the scope of this work only the two mentioned modules were available and hence realized.

4.2.2 Design time component

This section is dedicated to the implementations of the design time components of the software which are the RobotExplorer, responsible for generating a robot description of a single robot by querying the module bus using a computer's serial port, and the Framework Generator, capable of creating a ready-for-programming code framework for a distributed control application on the basis of a RobotWorld description.

4.2.2.1 RobotExplorer

This component is partly based on existing source code implemented by Jörg Irran as part of his thesis [Irr04]. However, some alterations were made to adapt the software to the requirements of this work.

Package at.oefai.diplomarbeit_e9625867.explorer

The class *RobotExplorer* in this package provides a mechanism to query the module bus of a connected robot for attached modules and assembles a robot description by fetching the module descriptions matching a module's ID from a local data storage. Again, the connection of the robot bus to a computer is realized using a serial connection. For this purpose, the existing communication layer was exchanged for the RXTX implementation of the JavaComm API extensions. Furthermore the descriptions of all the modules as well as the assembly process that creates the robot

description were reworked to comply to the XML Schema specifications introduced in chapter 3.4.1. For more details about the specifics of the bus protocol and the implementation of the querying process the reader may be referred to the work referenced above.

Package da9726049.explorer

This package and all its sub-packages implement the graphical user interface of the RobotWorldExplorer software and the processes that can be controlled with it. Of particular interest for the RobotExplorer component is the *RobotExplorerFrame* class, which allows the user to set some parameters and trigger the module bus querying process. The user interface is shown in Figure 4.3.

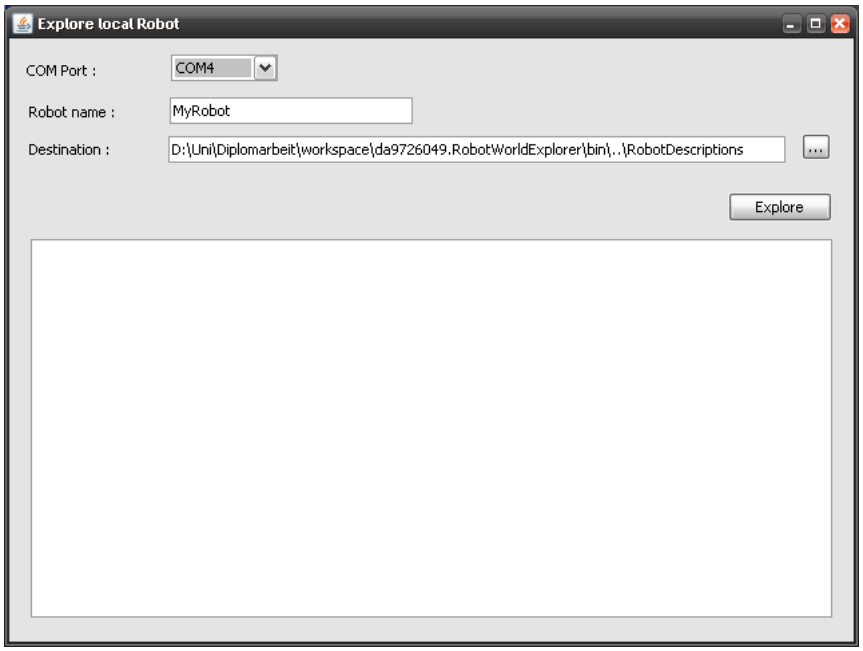

Figure 4.3.: user interface of the RobotExplorer component

On instantiation of the interface window the RXTX serial port component is initialized and searches the computer running the application for available serial port connectors, which can then be conveniently chosen from a drop down list. It also

prevents application errors from wrong user input and the querying of nonexistent ports. Furthermore a name for the connected robot can be provided that will be incorporated into the robot description and a storage location for the resulting XML document can be specified. A click on the [Explore] button will start the process and the console output with status output and progress information is redirected to the text area in the lower region. When successfully finished a valid robot description can be found in the chosen destination folder.

4.2.2.2 Framework Generator

This part of the software is responsible for the creation of a control application framework based on the information contained in a RobotWorld description. The parts a valid RobotWorld description is comprised of were outlined in chapter 3.4.1 together with mandatory XML Schema definitions for each component. However, it was also mentioned that the definitions can be extended and useful restrictions may be imposed on the possible values for certain elements should a concrete implementation of the architecture demand it. This is the case regarding the choice of the X10 Home Automation System to realize EnvironmentModules. The otherwise identical schema specification to Modules was altered to take advantage of the additional knowledge of the used system. Listing 4.1 shows the implementation-specific XML Schema definition used for an EnvironmentModule.

```
<xsd:complexType name="X10ModuleType">
  <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="HouseCode">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="[A-P] {1}"/>
        </xsd:restriction>
      </xsd:simpleType>
    <xsd:element name="UnitCode">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="ONE"/>
          ...
          <xsd:enumeration value="SIXTEEN"/>
        </xsd:restriction>
      </xsd:simpleType>
    <xsd:element name="Class" type="xsd:string"/>
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="X10LampModule"/>
          <xsd:enumeration value="X10ApplianceModule"/>
        </xsd:restriction>
      </xsd:simpleType>
    <xsd:element name="Version" type="xsd:string"/>
    <xsd:element name="Details" type="xsd:string" minOccurs="0"/>
    <xsd:element name="DocRef" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

Listing 4.1.: implementation-specific XML Schema definition of an EnvironmentModule

As can be seen, compared to the Module type definition given before, it is extended by the elements `<HouseCode>` and `<UnitCode>` since every X10 device has to be addressed using a combination of these two values. Furthermore it can be seen that restrictions are enforced on the values some elements are allowed to take. This mechanism provides the advantages that errors in the programmatic handling of the descriptions can be minimized since documents with invalid values for certain elements would already be rejected in the XML Schema verification stage. (As a side effect it can also increase the user friendliness when a developer wants to create a RobotWorld description because several XML editor programs with schema support assist the user by displaying possible choices for elements based on the according schema.) This way a house code can only be a single upper case letter in the range from A to P and unit codes are a choice among the upper case strings ONE to

SIXTEEN, thus creating a direct mapping to the constants defined in the *X10Constants* class of the package *da9726049.x10* (see chapter 4.2.1).

The use of type value restrictions is not limited to the EnvironmentModule specification, for this implementation it is applied throughout the complete XML Schema definition of a RobotWorld wherever it may proof beneficial. Such is the case for the `<Class>` element that points to a proxy class implementation of a module. Since the implementations have to adhere to a naming convention for the software to work properly, it is a good idea to ensure the correctness of the name to rule out any mistakes. The complete XML Schema type definition of a RobotWorld description for this prototype can be found in Appendix B.

Package da9726049.xvsm.xbean

The contents of this package represent the output of the Apache XMLBeans conversion tool applied to the RobotWorld description XML schema. As mentioned before, the types and elements defined in the schema are mapped to Java types and members respectively, with the latter being accessibly by *get()* and *set()* methods. This enables the easy handling of schema instances in the Java program code and spares the developer the manual parsing of the XML documents. This way the components of a RobotWorld description can be accessed using the methods *getRobotWorld()*, *getEnvironment()*, *getRobotDescription()*, *getModule()* and *getX10Module()*. Access to their elements works the same way. Should the need arise to manipulate the actual XML text content directly, on each component the *xmlText()* method can be invoked, returning the XML subtree belonging to the component as a string representation, which can then be parsed manually.

Whenever there are major changes to the schema definition, it might be necessary to rerun the conversion tool, hence rebuilding the Java type structure. However this may imply the need to change the code of the RobotWorldExplorer application. To avoid this the given implementation only makes use of XMLBeans accessors for elements included in the mandatory XML Schema definitions. Other additions and

alterations to the schema are solely handled through the XSLT stylesheet for the source code generation, thus extensibility is possible by just touching external resource files and not the existing application code.

Package da9726049.explorer

Of interest for the design time component in this package are the *GenerateFrame* and *FrameworkGenerator* classes. While the first represents the graphical user interface for the framework generation process, the latter implements all the tasks necessary to reach a usable control application development framework.

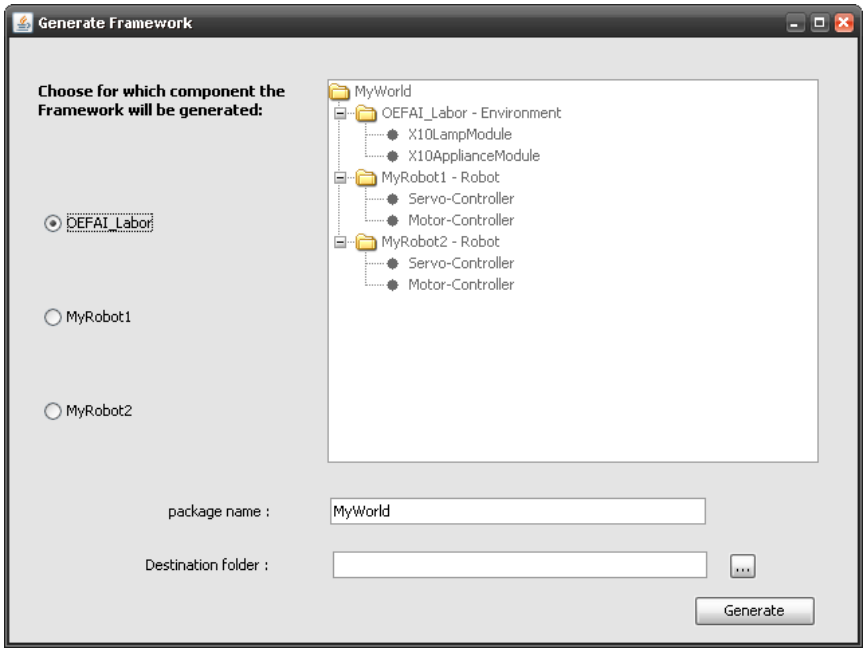

Figure 4.4.: user interface of the Framework Generator component

Figure 4.4 shows the window for configuration options of the Framework Generator displaying information about a loaded RobotWorld description. The tree view in the upper right area shows the available resources covered in the description file to the detail level of modules and environment modules. In this case the RobotWorld consists of one environment with two X10 modules and two robots with two modules

each. The radio buttons on the left hand side allow the user to choose the viewpoint, that is the participant intended to host the control application using the generated framework. Since all the generated classes are assembled into one package, the user can specify a name for this package as well as a destination folder where the resulting output will be stored. The process is then started by a click on the [Generate] button.

This is when an instance of the *FrameworkGenerator* class is created and its *generate()* method is invoked. Therein first the information about the viewpoint and the user defined package name are temporarily inserted as elements in to the RobotDescription XML using the XMLBeans cursor mechanism, since this content is needed in the XSLT transformation process. Next, depending on the communication type specified in the description the Java class loader tries to load the appropriate XSLT stylesheets, which are stored in the package *da9726049.explorer.transfer.[communication type]* adhering to the naming convention introduced in chapter 3.4.2.2. For each communication type two stylesheets need to be provided: one named *build.xslt* that contains the transformation rules that take the information from the description and produce an Apache Ant build file for the created framework source files. The other stylesheet named *source.xslt* contains two different sets of transformation rules, where the set applied depends on whether the created output source code is the proxy class for the viewpoint host or for a participant not being the viewpoint. The basic difference between the host chosen as viewpoint and the others is that in its source code the local implementations of the proxy interfaces for its modules have to be instantiated since they are physically attached to the host running the control application to be developed. The resources on the non-viewpoint participants are accessed using the remote implementations of the proxy interfaces. The output of the XSLT process with the *build.xslt* template is stored as Ant project file *build.xml* in the root folder of the user specified destination path, the classes generated using the *source.xslt* template are placed in a sub-folder named *src*. Finally all necessary libraries the generated sources depend on are copied into a *lib* sub-folder which completes the

process. The generated source files together with the implementations of the proxy interfaces in the libraries form the framework for the development of control application.

As an end result of the procedure the user is provided with complete project structure, the framework complete with a properly set up stub class for the viewpoint host and Apache Ant build environment support. Figure 4.5 shows a resulting sample project imported into the Eclipse IDE.

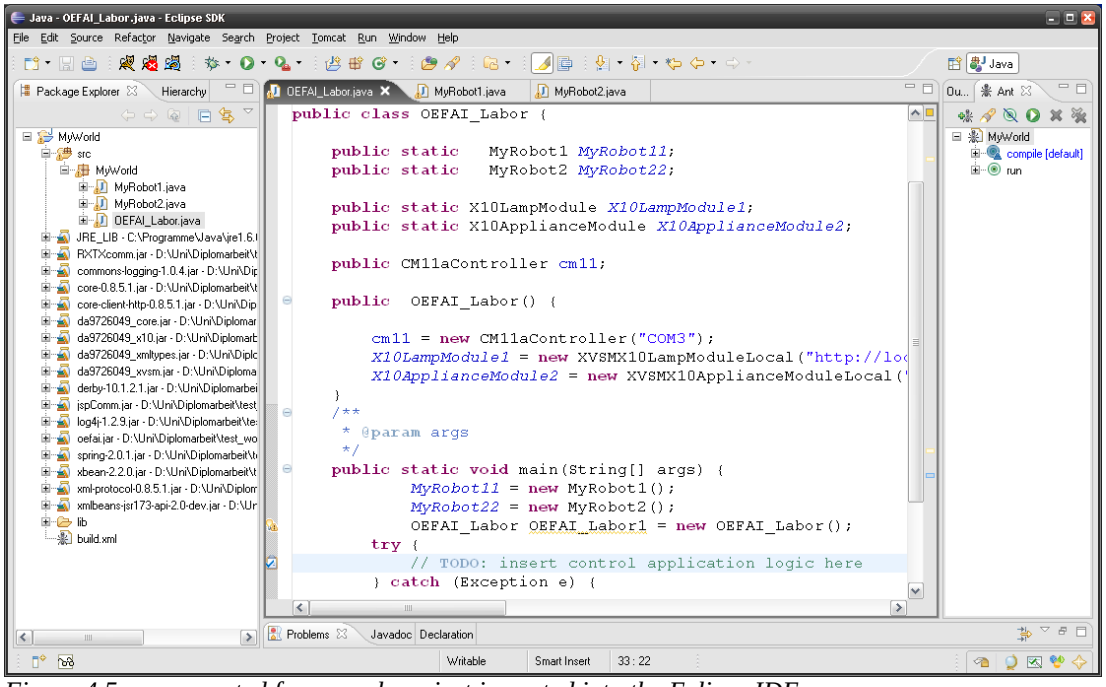

Figure 4.5.: a generated framework project imported into the Eclipse IDE

In the section on the left the complete project structure can be seen with the source files tree expanded and showing the generated classes for this project in the upper area. Underneath all the libraries forming the framework have been properly imported. The middle section shows the content of the class selected as viewpoint. All modules and references to the other participants are declared and instantiated in the constructor. The stub created for the main method is where the control

application logic has to be added. Finally the window area on the right shows the targets of the created Ant project file for compilation and running of the application.

4.2.3 Runtime component

The framework created by the Framework Generator in the previous step forms the runtime component of the architecture. It is comprised of the proxy class implementations of the interfaces specified for every module. The use of interfaces guarantees that a module always provides the functionality declared therein and thus it makes no difference from a programmers point of view what the details of the underlying implementation are. This way for a programmer, in the control application the use of a local resource is exactly the same as a method call on a remote resource – a desired behavior. It is also exploited to enable the possibility of different communication types, as for whatever system or middleware is used to realize the communication layer, as long as the classes implementing it use the specified interfaces the framework will work. All the interfaces for modules and environment modules are declared in the package *da9726049.core*. The name of those interface classes is also the name that has to be given as a reference in the module description using the `<Class>` element, the actual name of the implementing class is derived from the `<communicationType>` element which will – according to a naming convention – be added as a prefix. The proxy classes implementing those interfaces using XVSM can be found in the package *da9726049.xvsm*.

Package da9726049.xvsm

This package contains the implementation classes of the proxy interfaces declared in the package *da9726049.core* for the communication type XVSM, which means that monitoring and connections to a remote resource are realized with the help of the middleware XVSM. Adhering to the naming conventions introduced before, all class names start with the prefix XVSM. Furthermore, since the implementation differs

whether a module is physically attached to the host running the application or on a remote machine, there need to be two implementations of every module interface. The one covering local access uses the suffix *Local* in its class name, the other extends the name with the suffix *Remote*.

All of the classes in this package are derived from the class *XVSMModule* that covers the basic setup of the XVSM coordination structures required to implement the Request/Answer pattern. Furthermore to allow for an efficient parallel operation of multiple modules the class is realized as a thread, effectively handling the communication for each module in its own thread. Figure 4.6 shows an the OOD diagram illustrating the class relations and Table 2 shows the data structure of the *XVSMModule* class.

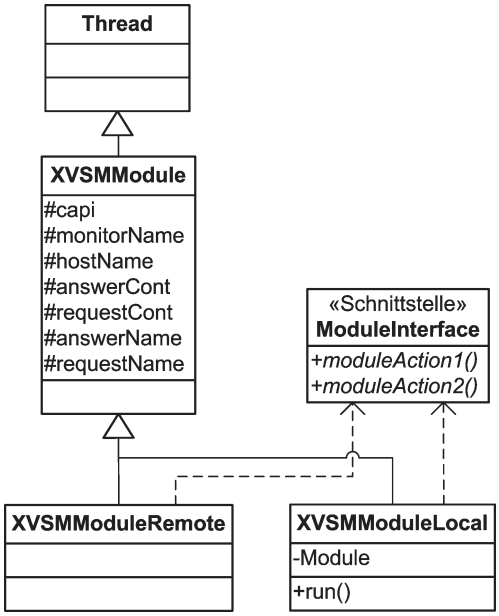

Figure 4.6.: OOD of the XVSMModule classes

member	***type***
capi	Capi
requestCont	Container
answerCont	Container
hostName	String
moduleName	String
requestName	String
answerName	String

Table 2: data structure of the XVSMModule class

The *capi* object is responsible for the connection to the XVSM space and provides methods to access the mechanisms provided by the middleware. It enables the creation of containers and notifications and calls to possible operations like read and write. For the containers belonging to a specific module to be easily identifiable and retrievable the naming service of the XVSM middleware is used. Thus containers are given a unique name that is stored in the field *moduleName*, which is comprised of the host name of the client with physical access to the hardware and the name of the module given in the RobotWorld description. This identifier is unique throughout the entire distributed system and hence identical in the local and remote implementations of the module interface. Additionally a *hostName* is stored that gives the name of the computer instantiating the class and is used as a key in requests so that the resource provider knows where to put the answers and the requester receives them. Since the architecture uses two separate containers to implement the Request/Answer pattern, the names for the each container are stored in the fields *answerName* and *requestName*, which are the *moduleName* added with the suffix *ANSWER* or *REQUEST* respectively.

The constructor of the *XVSMModule* class initializes the *capi* object, then tries to lookup the containers in the XVSM space by their respective names. In case they cannot be found, they are created. Either way the references are stored in the *requestCont* and *answerCont* fields. Finally the container properties for the answer

container are set to allow the use of a key-selector for the placement of the answer values.

Since all module implementations are derived from this class, the procedure described above is the same for all proxy classes. Implementation details for each module are covered in separate sections in chapter 4.3 below.

4.2.4 Monitoring

Another part of the runtime component is the monitoring, where the communication happening in the XVSM space and the status of the modules should be visualized in a clear way. The classes handling the monitoring task are assembled in the package *da9726049.explorer.monitors*.

Package da9726049.explorer.monitors

This package is divided into the two sub-packages *gui* and *logic* to separate the user interface from the application logic running behind it. In the package *gui* the visual representations for all modules are stored, for this implementation these are classes using Java Swing components for the graphical user interface, with each class extending the *JPanel* component. A *JPanel* is a Swing container, a component intended to help with the logical organization and grouping of user interface elements but providing no user interaction features itself. The idea behind this approach is that the RobotWorldExplorer loads a RobotWorld description and creates a monitoring window for each host declared therein. To this monitoring window for each module attached to the host a *JPanel* is added, where the user interface elements vary depending on the module type. This is realized with the help of a dynamic class loader that instantiates the appropriate class from the *logic* package. The classes in this package contain the module specific implementations for supervising the XVSM answer containers of the fitting module using a notification mechanism and create a panel from the *gui* package as a member to display the observations. Furthermore

each class implements the *Monitor* interface, which only declares the *getPanel()* method that returns the *JPanel* used to visualize the communication. Again, as a naming convention all classes in the logic package start with the communication type they implement and use the suffix *Monitor*.

For better understanding the complete process is explained step by step below:

1. The user chooses the option to monitor a RobotWorld in the RobotWorldExplorer software and subsequently selects a description document containing information about the scenario.
2. The software parses the document for all the hosts declared therein and the communication type.
3. For each host a monitoring window is created and the class loader searches for a class with the name comprised of the values of the `<communicationType>` element, the module's `<Class>` element and the suffix *Monitor* for each module.
4. If found, the class is instantiated using its first constructor, which takes the module description XML as an argument. From the description the information necessary to establish a connection to the space is retrieved.
5. The call of the *getPanel()* method of the just instantiated class returns the graphical user interface element for the module which is then added to the monitor frame of the module's host.

The final result is a window for each participant of a scenario, where each window is automatically filled with the user interface elements to display status information about the modules attached to the according participant. The use of this technique provides great flexibility for future extension of the implementation. For example if a different middleware is used, i. e. the communication type is changed, no changes to the source code of the existing RobotWorldExplorer software are necessary. Only new classes implementing the monitoring logic for the new communication layer have to be provided and are automatically loaded by the dynamic class loading mechanism.

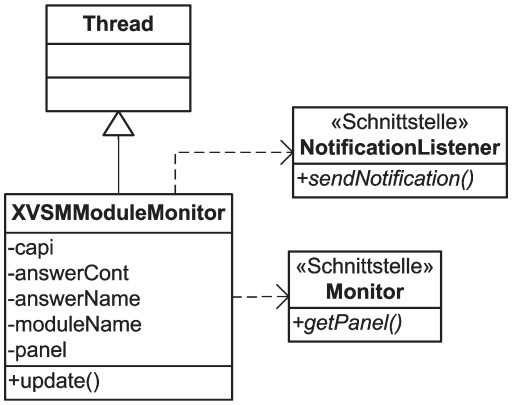

Figure 4.7.: OOD for XVSM monitors

member	**type**
capi	Capi
answerCont	Container
answerName	String
moduleName	String
panel	JPanel

Table 3: data structure for XVSM monitors

Figure 4.7 and Table 3 show the OOD diagram and data structure for XVSM monitors. All the monitor implementations have in common that they retrieve a reference to the respective module's answer container and create a notification on it. This way all changes happening to the container can be supervised. To be able to react to notifications the *NotificationListener* interface provided by the XVSM middleware framework must be implemented. As to allow the parallel monitoring of multiple containers simultaneously, the monitors are all realized as threads.

Implementation details for every module type as well as their user interface panels are explained in section 4.3 below.

4.3 Module implementations

The implementation details and data structures common to all modules have been explained it the sections above. This section is meant to give a short overview of the modules realized for this prototype, together with their interfaces, monitoring panels and the structure of the data actually transmitted through the XVSM containers. As explained in chapter 3.4.3.1 of the design description, the entries written into the containers are <host name, value> tuples, with the host name being used as a key identifying the requester and the content of the value depending on the requirements of a specific module implementation. This inner structure of the value part will also be explained in the following sections.

4.3.1 XVSMX10ApplianceModule

```
public interface X10ApplianceModule {

	public void on();
	public void off();
}
```

Listing 4.2.: interface of the X10ApplianceModule

X10ApplianceModule

field	type
host name	String
function code	int

Table 4: entry structure for an

Listing 4.2 shows the interface for the X10ApplianceModule. Its functionality is limited to the two functions *on()* and *off()* which switch a device attached to the X10 module on or off respectively. For this case the structure of the XVSM entry written to or read from the containers is a very simple tuple <host name, function code> . A simple integer value is used as a function code indicating that the X10 device should be turned off in case it is smaller than 0 and on else. The proxy class implementations and monitor classes interpret the entries received accordingly and trigger the fitting action. Figure 4.8 below shows the user interface element used for monitoring with a radio button displaying the device status.

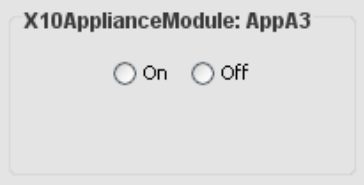

Figure 4.8.: monitoring user interface for an X10AppliancModule

4.3.2 XVSMX10LampModule

```
public interface X10LampModule {

      public void brighten(int level);
      public void dim(int level);
      public void on();
      public void off();
}
```

Listing 4.3.: interface of the X10LampModule

field	type
host name	String
level	int
function code	String

Table 5: entry structure for an X10LampModule

As can be seen in Listing 4.3 the X10LampModule provides the functions *brighten()* and *dim()* in addition to *on()* and *off()*. This environment module is intended to be used with lamps and the first two functions can be used to regulate the brightness by passing the desired brightness percentage level as a parameter. The four functions together with the parameter cannot be encoded in a single integer value, thus the tuple is extended to the form <host name, level, function code>, where the level indicates the desired brightness level and the function code contains the name of the function that should be called. The proxy class implementations and monitor classes interpret the entries received accordingly and trigger the fitting action. Figure 4.9 shows the panel used for the monitoring, with a radio button and an additional slider component to display the status.

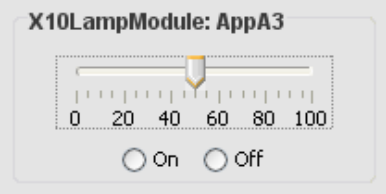

Figure 4.9.: monitoring user interface for a X10LampModule

4.3.3 XVSMServoModule and XVSMMotorModule

```
public interface ServoModule {

  public void setServo(int servoNr,
                       int position);
  public void setServoQuiet(int servoNr,
                            int position);
}
```

Listing 4.4.: interface of a ServoModule

field	type
host name	String
servo number	int
position	int
function code	String

Table 6: entry structure for a ServoModule

The ServoModule interface declares the functions *setServo()* and *setServoQuiet()*, each taking the number of the servo attached to the module and its desired new position as parameters. While the first method only completes when the addressed servo acknowledges that it has reached its final position, whereas the latter just dispatches the command to the module bus and allows the control application to continue without waiting. The functionality of a MotorModule is essentially the same, except that a first parameter indicates the motor number and the position parameter is interpreted as the speed level. But the same coordination data structure and user interface component can be used, hence the module is not specifically discussed. The tuples for XVSM coordination have the inner structure <host name, servo number, position, function code>, with the middle two used to pass the method parameters and the last one containing the name of the function to be called. The appropriate handling of the tuples passed through the containers is implemented in the proxy classes and monitors for these modules. Figure 4.10 shows the panel for the monitoring component.

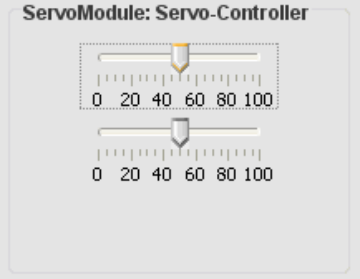

Figure 4.10.: monitoring user interface a ServoModule or MotorModule

4.3.4 XVSMBumperModule

```
public interface BumperModule {

  public int getAllBumper();
  public boolean getSingleBumper(
                          int bumperNumber);

}
```

Listing 4.5.: interface of a BumperModule

field	type
host name	String
value	int
function code	String

Table 7: entry structure for a BumperModule

The BumperModule provides functions to get the status of all the bumper sensors attached to the module at once encoded into an integer value, or to query a specific sensor returning a boolean value that is true when the sensor has contact to an object or false else. The structure for the transmitted tuples can be seen in Table 7. The function code always contains the name of the function to be called or has been called in case the tuple is written to an answer container. The value field is used to pass the bumper number when used as a request and to store the return value of the called function in case of an answer, with the boolean *false* value mapped to the integer 0. Figure 4.11 depicts the user interface panel for the BumperModule with a textfield for each bumper containing the bumper's number and the background color showing the status. A green backround indicates no contact to an object, while it turns red in case the bumper is activated.

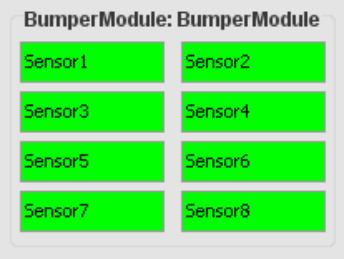

Figure 4.11.: monitoring interface for a BumperModule

4.3.5 XVSMIRDistanceModule

```
public interface IRDistanceModule {

  public String getAllSensors();
  public int[] getAllSensors_ByteRep();
  public int getSingleSensor_Cached(
                              int sensorNr);
  public int getSingleSensor(int sensorNr);

}
```

Listing 4.6.: interface of a IRDistanceModule

field	type
host name	String
value	*
function code	String

Table 8: tuple structure for an IRDistancModule

The IRDistanceModule allows for the attachment of up to seven distance sensors. The variety of the exposed methods and data types by the interface does not allow for the use of a single tuple structure for request and answer entries, therefore different answer tuples are used. The basic structure is shown in Table 8 with the field *value* taking different types according to the function called. For the request tuple it is an integer giving the number of the sensor to be queried in the *getSingleSensor*()* functions. For those two methods the tuple written to the answer container looks exactly the same with this time the integer containing the return value of the respective function. The *getAllSensors()* function requires a minor change in the way that for an answer tuple the *value* field is a String matching the return value of the method. For the *getAllSensors_ByteRep()* method the *value* field itself is a tuple of seven integer values, since the method returns an array of integers with each field storing a sensor value and the field's index matching the sensor number.

Depending on the function code the implementations of the proxy interfaces handle the different types of answer tuples accordingly. Figure 4.12 shows the graphical user interface element for visualizing the module's status. For each sensor the actual value is shown in a JTextField matching the sensor's number.

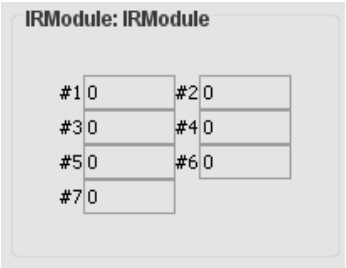

Figure 4.12.: monitoring user interface for an IRDistanceModule

4.4 Extensibility

This section is meant to give an overview in which ways the existing software can be extended, which classes and resources need to be provided and where they need to be placed to work properly. Therefore first a summary of the overall package structure and naming conventions introduced throughout the previous chapters is gathered here in one place followed by a concrete example for better understanding.

4.4.1 Naming conventions and package structure

The names of the classes created are mainly influenced by the communication type. The communication type defines the technique used to realize the information transmission between different hosts in a distributed robot system. To make use of a specific communication type in the created framework, it has to be specified in the

RobotWorld description with the `<communicationType>` element. Of course for each module a proxy class implementation using this type must be provided. For the Framework Generator to work properly, naming conventions have been introduced, so that it can find and initialize the correct classes in the resulting framework:

- **Classes:** proxy class implementations for a given communication type have to start with the name of the communication type. Should the communication type require a client/server design and thus more than one proxy class for a module, the server implementation has to end with the suffix *Local* and the client implementation with the suffix *Remote*. The class implementing the monitoring component has to end with the suffix *Monitor*.
- **XSLT:** the transformation rules for a communication type are stored in two XSLT files: *build.xslt* for the creation of an Apache Ant project file and *source.xslt* for the creation of framework source code. The files have to be placed in the folder *da9726049.explorer.transfer.[communicationType]*.

Adhering to these conventions the prototypical implementation for the communication type *XVSM* lead to the package structure below for the RobotWorldExplorer software.

Package	**content**
da9726049.core	module interfaces
da9726049.xvsm	interface implementations for XVSM
da9726049.explorer	user interfaces and logic of the RobotWorldExplorer application
da9726049.explorer.monitor.gui	graphical user interfaces for each module (extending javax.swing.JPanel)
da9726049.explorer.monitor.logic.xvsm	program logic filling the GUI elements for each module for XVSM
da9726049.explorer.transfer.xvsm	build.xslt and source.xslt for XSVM framework generation

4.4.2 Example: adding a new module for XVSM

To outline the procedure of adding a new module this example shows what a developer needs to do to make the *NewMod* module usable in control applications using XVSM:

- Add the following resources:

package	file(s)	content
da9726049.core	NewMod.java	module interface
da9726049.xvsm	XVSMNewModLocal.java XVSMNewModRemote.java	server proxy class client proxy class
da9726049.explorer.monitor.gui	NewModPanel.class	compiled GUI for monitoring (JPanel)
da9726049.explorer.logic.xvsm	XVSMNewModMonitor.class	compiled program logic behind the GUI

- Add *NewMod* to the XML Schema of a RobotWorld description as a valid value for a `<Class>` element.
- In case the module description is not stored in the hardware, it needs to be added to a folder named *ModuleDescriptions* in the same folder as the RobotWorldExplorer software.

When adhering to the naming conventions and with the help of the dynamic class loading mechanism for the monitoring components, modules can easily be added without the need to make any changes to the existing source code. As an additional note it may be mentioned, that the architecture even provides the flexibility support different middleware solutions through the specification of new communication types. Furthermore also the change of the programming language of the generated frameworks is easily doable by providing proxy class implementations in the desired language and adapting the XSLT stylesheets accordingly.

5 Putting the Framework to use

In this chapter a short proof-of-concept scenario will cover all the steps from the description of a simple scenario over assembling of the robot and creation of the framework to the implementation of control applications. The development of control applications for autonomous mobile robots is a very complex problem, even more so for multi robot scenarios. The focus of this work is to create an environment that supports the developer in the solving of this problem and gets her/him to a point where she/he can start to focus on the important task of developing the control application for a distributed robot scenario. Thus the example is intentionally kept very simple and intends to demonstrate the simplicity of the startup phase but furthermore allows for the actual development of the control application within reasonable amounts of work to show that the communication layer using XVSM works as intended.

5.1 The Scenario

A robot, equipped with a drive mechanism and bumper sensors is put into a dimly lit environment containing obstacles and the goal to navigate around obstacles. In the environment a lamp is installed which is initially turned off to save energy, but can be switch on to brighten the scenario course and allow for a better evaluation of a situation. During robot operation, the whole process shall be monitored on a computer running a monitoring application.

5.2 Hardware Requirements and Preparations

The first step is to take a look at the hardware needed to fulfill the task and the steps necessary to make it ready for development. In this case these are:

- **The robot:**

The robot needs to be assembled from the appropriate modules for the task. As a basis for the mobile robot, a caterpillar platform with a motor module is chosen. To the platform's front two bumper sensors are attached which are controlled by a bumper module. Furthermore a processing unit is necessary that can run the control application for the scenario. Since the platform is big enough, a small laptop computer is used for this task. To enable the connection of the computer to the robot's module bus an interface converter module receives messages from the computers serial port and translates them to the bus protocol.

Finally all the modules are connected together and a computer running the RobotWorldExplorer software queries the module bus and produces a valid robot description.

- **The environment:**

The lamp in the environment is realized with the help of the X10 Home automation system and an appliance module. To enable access as a shared resource a computer with a CM11a X10 controller connected to the serial port is needed to send X10 protocol commands over the domestic power line infrastructure from within the control application. Since a plug&play mechanism for module recognition for the X10 system cannot be realized, the developer has to create a valid environment description manually with an XML editor.

The two descriptions for the robot and the environment are then manually assembled into a single RobotWorld description for the monitoring and framework generation process.

```
<?xml version="1.0" encoding="UTF-8"?>
<RobotWorld xmlns="http://xbean.xvsm.da9726049">
  <WorldName>MyWorld</WorldName>
  <CommunicationType>xvsm</CommunicationType>
  <Environment>
    <EnvironmentName>OEFAI_Labor</EnvironmentName>
    <Host>http://myrobot1:8080/xvsm/xvsm</Host>
    <COMPort>COM3</COMPort>
    <X10Module>
      <Name>X10ApplianceModule</Name>
      <HouseCode>A</HouseCode>
      <UnitCode>THREE</UnitCode>
      <Class>X10ApplianceModule</Class>
      <Version>1.0</Version>
    </X10Module>
  </Environment>
  <RobotDescription>
    <RobotName>MyRobot1</RobotName>
    <TalkToBusClass>RS485Bus</TalkToBusClass>
    <Host>http://localhost:8080/xvsm/xvsm</Host>
    <Module>
      <Name>Motor-Controller</Name>
      <NodeID>15B63A4</NodeID>
      <Class>MotorModule</Class>
      <Version>1.2</Version>
      <Details>allows controlling of 2 Motors</Details>
      <DocRef>../NodeDocumentations/15B63A4.html</DocRef>
    </Module>
    <Module>
      <Name>BumperModule</Name>
      <NodeID>78A23F4</NodeID>
      <Class>BumperModule</Class>
      <Version>1.0</Version>
      <Details>receives the values of 2 bumpers</Details>
      <DocRef>../NodeDocumentations/78A23F4.html</DocRef>
    </Module>
  </RobotDescription>
</RobotWorld>
```

Listing 5.1.: the RobotWorld description for the example scenario

- **The monitoring host:**

A third host computer is required for the monitoring task. For this purpose the RobotWorldExplorer software is installed on the system. Additionally the RobotWorld description file needs to be supplied to the software to configure the monitoring.

- **Connection:**

The three host computers that build this distributed system need to be connected using a network infrastructure to enable communication between them. Since the robot should be able to move freely, a wireless LAN[1] connection is chosen for this client. The other two hosts may use either a wired or wireless connection. Furthermore a computer running the Apache Tomcat server application hosting the XVSM space is required. Because the control application of the robot is the one most likely having to deal with time critical issues, the space is hosted on its laptop computer to eliminate additional network latencies.

Now that the hardware is set up and the valid RobotWorld description has been created for the scenario, the next step is to create the frameworks for the control applications. In this specific case two control applications are needed, one for the laptop computer controlling the robot and one for the host controlling the environment resources. Hence the RobotWorld description is loaded with the RobotWorldExplorer software and two frameworks are generated, one with the robot selected as viewpoint and one with the environment being chosen. At this point all steps necessary to start with the application development are completed.

5.3 Writing the control applications

To solve the task described in the scenario two control applications are needed, one for the environment and one for the robot.

The first is very simple because the host controlling the environment modules in this case only acts as a resource provider. It does not issue any commands to remote hosts, it simply has to initialize the proxy classes for proper hardware access and listen for requests filed by the robot control application. Actually all these tasks are already performed by the framework generator, hence the created framework is

1 LAN: Local Area Network

already the complete control application. All that remains to be done is to deploy the framework project on the appropriate computer and compile and run the application.

```
while(true){
       //--- forward ---
       MotorModule1.setMotor(LEFT,0.25);
       MotorNodule1.setMotor(RIGHT,0.25);

       if (BumperModule1.getSingleBumper(6)
           || BumperModule1.getSingleBumper(4)) {
              //--- back ---
              MotorModule1.setMotor(left,-0.3);
              MotorModule1.setMotor(right,-0.3);
              Thread.sleep(500);
              //--- switch lamp on ---
              X10AppliancModule1.on();
              //--- evaluate situation (simulated) ---
              Thread.sleep(3000);
              //--- turn ---
              MotorModule1.setMotor(left,0.4);
              MotorModule1.setMotor(right,-0.4);
              Thread.sleep(1000);
              //--- switch lamp off again ---
              X10AppliancModule1.off();
       }
}
```

Listing 5.2.: a simple control application for a robot

For the robot control application a simple strategy for navigating the scenario is developed. The robot starts by simply driving straight forward. Whenever the bumper sensors indicate physical contact to an object, the robot immediately moves backward for a short distance. It then pauses, switches on the light and evaluates the situation. (Note: In this control application the evaluation phase is only simulated by a short timeout. A real life evaluation could be realized with the help of a camera module mounted on the robot platform, that can be used to detect a direction without any obstacles ahead, which is only possible in good lighting conditions. However, this would have increased the complexity of the control application considerably and the simulation is also justified by the lack of such a module implementation.) After the evaluation the robot turns in a direction with no obstacles ahead and continues its movement through the scenario. The code to implement this simple strategy is shown in Listing 5.2.

The robot starts moving by calling the *setMotor()* method of the motor module with the desired motor number (here replaced by the constants *LEFT* and *RIGHT* for better readability) and a speed value indicating the percentage of the maximum speed as parameters. If the two front bumpers detect a collision with an object the robot moves backwards. Then the remote resource, the X10 appliance module is switched on, the situation "evaluated" and the robot turned into a direction with no obstacles ahead. Before the movement is continued, the remote lamp is switched off again. All the method calls during this process cause status information to be distributed into the XVSM space and can be monitored at the third host running the monitoring application.

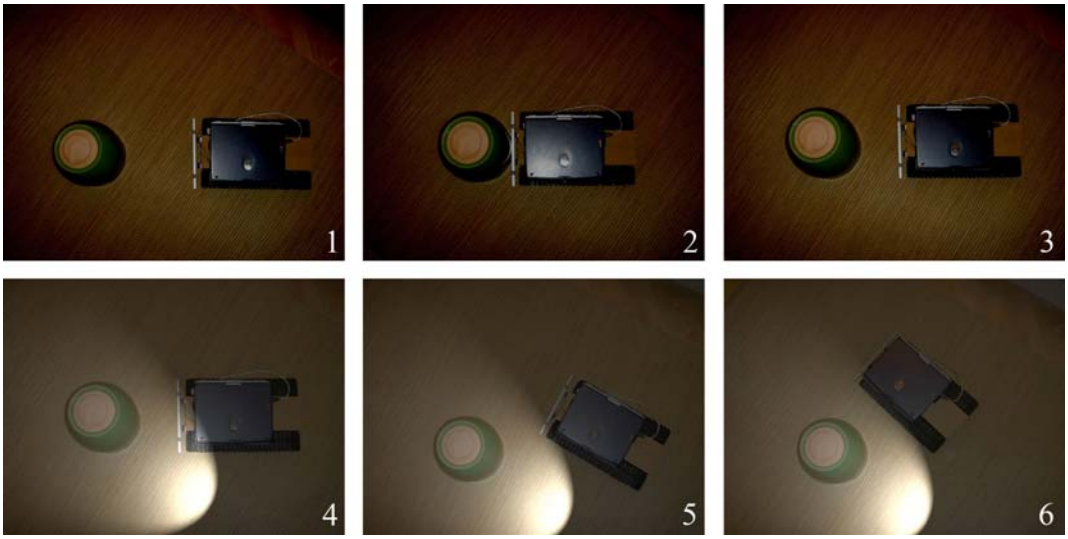

Figure 5.1.: photo series of a robot running the sample control application

Concluding, to emphasize the versatility and benefits of the holistic system view and resource sharing provided by the presented architecture, it should be highlighted, that the control application code for the robot shown above can be interchanged between the client computers in this distributed system without any changes. Put into the environment control application's main method, the robot would function the same way, albeit being remote controlled from an external host computer.

Furthermore, using the tools provided through this thesis, the realization of this scenario takes no longer than 2 to 3 hours, which again indicates the efficiency and short startup time introduced through the use of the proposed system.

6 Conclusion and future outlook

In this chapter the proposed architecture and its prototypical implementation is evaluated based on the requirements that emerged as a result from the analysis in chapter 3.3. This is done separately for the design time and the runtime components, followed by a section covering possible future extensions and alterations.

6.1 Design time component

The design goal for the design time component was to assist the developer in the start up phase of a robot coordination scenario and minimize the time required to reach the point where the most complex part of such a task, the control application development, can start. This goal is met as can be best seen in chapter 5, where the prototype software based on the proposed architecture is put to use to realize a simple example scenario. The planning and preparation process is supported and automated to a point, that a programmable framework complete with a folder structure, project file for use with IDEs and support for a monitoring application are provided to the developer. In the special case, where a participating host only acts as a resource provider, no additional programming by the developer is required. The output framework of the design time component is already the control application for the host it was created for. The only input required to reach this state is some XML content conforming to well defined rules supplied in the form of a XML Schema definition outlined in chapter 3.4.1. Through the reliance on XML technologies, developers can choose among a great variety of existing applications and tools to be supported in the creation of this content.

The architecture outlined in this work further allows to apply a layered view to the whole system depicted in Figure 6.1.

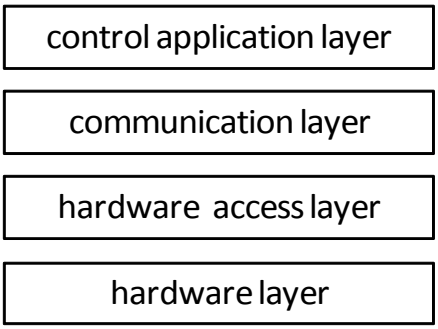

Figure 6.1: layered view of the system architecture

On the lowest level is the hardware layer, which covers all the aspects of the robot hardware including sensors and actuators, their physical implementation and connection. Stacked on top of it is a hardware access layer, which enables the access to and controlling of the hardware from within high level programing languages. Next is the communication layer, used to connect multiple clients in a robot team scenario and covered by the runtime component of this work. Finally on the topmost level the distributed control application resides, the layer implementing the logic necessary to solve a specific task and of most interest for artificial intelligence research. The benefit of this layered model is that development and research for each layer can happen simultaneously and separately from each other as long as the interfaces to the adjacent layers are kept intact. However the use of different and new technologies in one layer might necessitate at least small changes to the way two layers interact. The mechanism providing this flexibility and mask the unavoidable differences between alternative implementations is the XML description introduced in this architecture. The information contained therein is used to select the appropriate classes for the framework and to instantiate them with the correct values.

As already mentioned in the introduction to this thesis, research in the area of Embodied AI requires knowledge of many diverse areas which are reflected in the layers outlined above and allows experts from every field to concentrate on a specific layer. The output of the software's design time component covers the lower three

levels and thus allows for experiments in implementing artificial intelligent behavior, a goal this thesis set out to reach.

The requirement to enable distributed control application development is also met through the use of the XML descriptions as a basis for the framework generation process. This way the description files or parts thereof can be exchanged between different developers, which can in turn write different solutions for a task or contribute to a joint development effort. Because of the implemented communication layer even remote access to hardware resources over a network is possible.

Finally, the system realized in this thesis can be extended in several ways. For the provided prototypical implementation using XVSM new modules can be added by supplying additional source code but not requiring changes to the existing code. The same is true if instead of XVSM an alternative communication layer probably using another middleware system is used. Furthermore, even though all the software components in this work are written in Java, when providing suitable resources the framework generation process will produce output in an arbitrary programming language.

In conclusion it can be said that the design time component successfully performs the tasks as intended in the original idea. It assists a developer in the design process by hiding and automating low level aspects and thereby simplifies the demanding task of creating distributed Embodied AI applications. Its realization and modular design provides a flexibility that enables good extensibility and adaption to different platforms. This can be attributed to the use of Java and its platform independent paradigm and primarily to the use of XML technologies and their platform overlapping usability.

6.2 Runtime component

The runtime component of the architecture is the communication layer implemented in the framework generated by the design time component. It was

realized using the virtual shared memory middleware XVSM to provide a holistic system view to the programmer of a distributed control application. Using this approach, from within a control application the locality of a resource is hidden from the programmer and hence can be accessed in the same way whether it is physically connected to the host running the control application or installed in a remote location. The exemplary control application developed in chapter 5.3 demonstrates that the architecture is perfectly capable of providing this service, since the application code is interchangeable between the participating hosts and works without any changes necessary. A major benefit of the holistic system view is that – from a programming perspective – in effect the development of a distributed control application for a robot team is the same as for a single robot. Additionally it proofs beneficial for fault tolerance scenarios, since remote resources can be used in case of a hardware failure.

The proof-of-concept control application in chapter 5.3 illustrates the remote access to resources, which in this case is not time critical. For the sharing of resources in time critical situations further considerations have to be taken. Autonomous mobile robots are real time systems, where some processes are subject to hard deadlines, meaning that the meeting of certain critical time constraints is essential and missing can have fatal consequences [Kope97]. In the case of collision detection with an obstacle, if the time between detection and proper reaction is to long, the robot may continue to exert a force in the obstacle's direction instead of moving back and possibly destroy some part of its own hardware or the obstacle. Depending on the speed of the robot and factoring in the latency of the communication medium – e. g. with the technologies used in this thesis a wireless Ethernet connection having typical latencies in the area of low hundreds of milliseconds – control loops for such a task using remote resources are not carried out over not real time proof media types that cannot meet the requested time constraints. Taking this into account during control application design, such situations can be avoided. An adequate solution to this problem can be achieved by providing access to simple behavior patterns in addition to the resources for time

critical tasks that are realized using local control loops. For example a robot could expose a method that moves it forward until it collides with an obstacle and then moves back and waits for further instructions. The approach of having several control layers starting with fast and reactive control mechanisms on a lower level that can be used and influenced by commands of higher level layers can also be seen in the subsumption architecture introduced by Brooks [Broo86]. Aside from the problem mentioned above, there are many aspects in a distributed control application that are not time critical by their nature (e. g. resource access outside of critical control loops) or can be realized in a way that they do not depend on time limits (e. g. ensure that a robot is in a safe state before a task is performed).

The coordination mechanisms provided by XVSM proofed to be very valuable and versatile for implementing the communication layer, allowing for the data structures to be very simple and enabling the required holistic system view. The concept of containers accepting and delivering arbitrary entries based on predefined coordination types enables the realization of a shared queue for multiple clients just with the services offered by the middleware. Exploiting the fact that one container can house entries with different data types and inner structure, only one container instead of multiple can be used to store the data returned from all kinds of different method calls. This helps to keep the growing of the number of shared data structures on a linear level with the amount of shared resources and the same base implementation of the Request/Answer pattern can be used for each module type. Furthermore the notification mechanism turned out to be an essential service for the task of resource monitoring. The use of unbounded containers allows the system to scale to an arbitrary number of clients from an architectural point of view. At this point in time XVSM is still a work in development and little experience on the scalability of the middleware performance exists. However for future versions increased performance and reliability as well as additional services can be expected. The current development process moves away from the client server architecture to a peer-to-peer solution. This removes the need for hardware capable of running a Web

Application Container as a necessary component to realize the space and will allow every host to run a lean peer-to-peer client, which also increases reliability through the elimination of a single point of failure.

Concluding it can be said that the runtime architecture addresses all the requirements imposed on it and that XVSM proofed to be a very usable middleware to realize the task at hand. Due to the real time nature of autonomous mobile robots the distributed nature of the resources has to be taken into consideration during control application design and practicability will have to be evaluated on a case-by-case basis.

6.3 Future outlook

The following section introduces ideas for further extensions, improvements and variations of the presented work.

6.3.1 Adaptive request queue processing

Since autonomous mobile robots are real time systems, the relevance and age of retrieved sensor data plays an important role in control applications. A newer value is always more important than the preceding one, hence the older one can already be overwritten even if it has not yet been processed. In the current implementation this is already considered for example through the use of *shift* operations that never block for writing answers. However requests are placed into the queue by clients one at a time and processed by the host one at a time. This way it is possible that the number of requests in the queue increases when many clients request data from the same resource. For sensors this behavior could be optimized in a way that the host reads all the requests waiting in the queue simultaneously, since they all are interested in the same sensor value. The answers are then written to the answer container for all the requesters in the queue. This variation can optimize response time for clients waiting

at the end of the queue and also decreases the traffic on the module bus since multiple request can be satisfied with only a single sensor query.

6.3.2 Virtual resources

This extension is based on the idea that modules do not necessarily have to be some kind of sensor or actuator hardware. Basically the resource sharing process consists of a request for certain data and a response to that request. The provider of this data can as well be another piece of software or a service. For example a map service could have a map for a certain area or scenario at its disposal. By filing a request to the map service, the robot could receive information about the directions to a certain point.

6.3.3 Knowledge sharing

This is a kind of variation or extension of the virtual resources idea and of particular interest for artificial intelligence research and robot teams. A major goal of this research area is to create autonomous robots that show some kind of intelligent behavior and are capable of learning about aspects in their environment. As an example, by trying to move certain objects, a robot could acquire the knowledge that it can push and move sphere shaped object since they roll, but does not have enough power to do the same with blocks. It could store this knowledge in a knowledge base shared among a robot team and save the other team members the effort to find out on their own through trial and error. Another typical example would be a search and rescue task for a robot team, where the member finding the target can share the target location with the others so that they stop searching and move to the correct position.

6.3.4 Distributed system on a single robot

The architecture presented in this thesis allows for the sharing of hardware resources among different hosts. This design does not necessarily have to be deployed in a multi robot scenario, it may also provide benefits when applied to a single robot that uses multiple processing units instead of one. Given the discrepancy between computing power versus mobility and energy efficiency it can be a better solution to use several mobile and efficient computers to build a mobile robot instead of using a big and heavy single computer system that quickly drains the batteries. In case of humanoid robots – robots that move on two legs in an upright position mimicking human walking – the balance of the whole system is an important factor, that can be influenced by the weight distribution. Being able to use several smaller computing devices instead of having all the weight concentrated in one place can be helpful in this area as well.

As a concluding remark, the architecture proposed in this thesis and the ideas it is based on show a great potential for future developments and extensions due to its modular design and used mechanisms and might be even applicable in other ways than initially intended during its design as illustrated by the last extension idea.

7 Appendix A

7.1 Literature

[Broo86] Brooks R. A., *A Robust Layered Control System for a Mobile Robot*, IEEE Journal of Robotics and Automation, Vol. 2, No. 1, pp. 14-23,1986

[Broo91] Brooks R. A., *Intelligence Without Reason*, Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91), pp. 569-595, Morgan Kaufmann publishers, Sidney, Australia,1991

[Gele85] Gelernter D., *Generative communication in Linda*, ACM Transactions on Programming Languages and Systems (TOPLAS), pp. 80-112, ACM Press, New York,January 1985

[Irr04] Irran J., *Entwurf und Konstruktion eines in Hard- und Software erweiterbaren und modularen mobilen Robotersystems*, Master thesis executed at the Institut für Medizinische Kybernetik und Artifical Intelligence, University of Vienna ,2004

[King99] Kingery P., *Digital X-10*, http://www.hometoys.com/htinews/feb99/articles/kingery/kingery13.htm#Digital%20X-10,1999, accessed August 2007

[Kope97] Kopetz H., *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, Dordrecht, The Netherlands, Sixth Printing 2002,1997

[Kühn05] Kühn e., Riemer J., Joskowicz G., *XVSM (eXtensible Virtual Shared Memory) Architecture and Application*, Technical Report, Vienna University of Technology E185/1, SBC Group,2005

[Park05] Parker L. E., Chandra M., Tang F., *Enabling autonomous sensor sharing for tightly-coupled cooperative tasks*, Multi-Robot Systems: From Swarms to Intelligent Automata, Volume III, Springer, New York,2005

[Park95] Parker L. E., *The Effect of Action Recognition and Robot Awareness in Cooperative Robotic Teams*, Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vol. 1, pp. 212-219,August 1995

[Smar07] Smart Home Systems Inc., *How X10 Works*, http://www.smarthomeusa.com/info/x10theory/#theory,accessed August 2007

[Sun03] Coward D., Yoshida Y., *Java Servlet Specification Version 2.4*, Sun Microsystems, http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html, November 2003, accessed August 2007

[Sun97] Sun Microsystems, Hamiltion G., *Sun Microsystems JavaBeans Specification version 1.01*, http://java.sun.com/beans,August 1997, accessed August 2007

[Tann02] Tannenbaum A. S., Van Steen M., *Distributed Systems: Principles and Paradigms*, Prentice Hall Inc., New Jersey,2002

[W3C04] W3C, *XML Schema Part 0: Primer Second Edition*, http://www.w3.org/TR/xmlschema-0/,October 2004, accessed August 2007

[W3C06] W3C, *Extensible Markup Language (XML) 1.0 Fourth Edition*, http://www.w3.org/TR/2006/REC-xml-20060816/,2006, accessed August 2007

[W3C99] W3C, *XSL Transformations (XSLT) Version 1.0*, http://www.w3.org/TR/xslt,November 1999, accessed August 2007

[X1006] X10, Ltd., *Standard and Extended X10 Protocol*, ftp://ftp.x10.com/pub/manuals/xtdcode.pdf,2006, accessed August 2007

7.2 Figure index

7.3 Table index

7.4 Listing index

8 Appendix B

8.1 XML Schema definition for mandatory elements of a RobotWorld

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="RobotWorld" type="RobotWorldType"/>
    <xsd:complexType name="RobotWorldType">
     <xsd:sequence>
     <xsd:element name="WorldName" type="xsd:string" minOccurs="1"/>
     <xsd:element name="CommunicationType" type="xsd:string"
                                              minOccurs="1"/>
     <xsd:element name="PackageName" type="xsd:string" minOccurs="0"/>
     <xsd:element name="Environment" type="EnvModuleListType"
                                                   minOccurs="0"/>
     <xsd:element name="Robot" type="ModuleListType" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>
  <xsd:complexType name="EnvModuleListType">
    <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="Host" type ="xsd:string"/>
    <xsd:element name="COMPort" type="xsd:string"/>
    <xsd:element name="EnvModule" type="EnvModuleType"
                    minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="Module" type="ModuleType"
                 minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ModuleListType">
    <xsd:sequence>
    <xsd:element name="RobotName" type="xsd:string"/>
    <xsd:element name="Host" type ="xsd:string"/>
    <xsd:element name="TalkToBusClass" type="xsd:string"/>
    <xsd:element name="Module" type="ModuleType" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ModuleType">
    <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="NodeID" type="xsd:string"/>
    <xsd:element name="Class" type="xsd:string"/>
    <xsd:element name="Version" type="xsd:string"/>
    <xsd:element name="Details" type="xsd:string" minOccurs="0"/>
    <xsd:element name="DocRef" type="xsd:string" minOccurs="0"/>
   </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="X10ModuleType">
    <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="HouseCode">
    <xsd:element name="UnitCode">
    <xsd:element name="Class" type="xsd:string"/>
    <xsd:element name="Version" type="xsd:string"/>
    <xsd:element name="Details" type="xsd:string" minOccurs="0"/>
    <xsd:element name="DocRef" type="xsd:string" minOccurs="0"/>
   </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

8.2 XML Schema for a RobotWorld description with restrictions for the prototype implementation

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:sz="http://xbean.xvsm.da9726049"
targetNamespace="http://xbean.xvsm.da9726049"
elementFormDefault="qualified">
  <xsd:element name="RobotWorld" type="sz:RobotWorldType"/>
  <xsd:element name="Environment" type="sz:EnvModuleListType"/>
  <xsd:element name="RobotDescription" type="sz:ModuleListType"/>
  <xsd:complexType name="RobotWorldType">
    <xsd:sequence>
      <xsd:element name="WorldName" type="xsd:string" minOccurs="1"/>
      <xsd:element name="CommunicationType">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:enumeration value="none"/>
            <xsd:enumeration value="xvsm"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="PackageName" type="xsd:string" minOccurs="0"/>
      <xsd:element ref="sz:Environment" minOccurs="0"
       maxOccurs="unbounded"/>
      <xsd:element ref="sz:RobotDescription" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="EnvModuleListType">
     <xsd:sequence>
      <xsd:element name="EnvironmentName" type="xsd:string"/>
      <xsd:element name="Host" type ="xsd:string"/>
      <xsd:element name="COMPort" type="xsd:string"/>
      <xsd:element name="X10Module" type="sz:X10ModuleType" minOccurs="0"
       maxOccurs="unbounded"/>
      <xsd:element name="Module" type="sz:ModuleType" minOccurs="0"
       maxOccurs="unbounded"/>
     </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="ModuleListType">
     <xsd:sequence>
      <xsd:element name="RobotName" type="xsd:string"/>
      <xsd:element name="Host" type ="xsd:string"/>
      <xsd:element name="TalkToBusClass" type="xsd:string"/>
      <xsd:element name="Module" type="sz:ModuleType"
        maxOccurs="unbounded"/>
     </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="ModuleType">
     <xsd:sequence>
      <xsd:element name="Name" type="xsd:string/>
      <xsd:element name="NodeID" type="xsd:string"/>
      <xsd:element name="NodeNumber" type="xsd:string"/>
      <xsd:element name="Class" type="xsd:string"/>
       <xsd:simpleType>
        <xsd:restriction base="xsd:string">
```

```
        <xsd:enumeration value="ServoModule"/>
        <xsd:enumeration value="BumperModule"/>
        <xsd:enumeration value="IRDistanceModule"/>
        <xsd:enumeration value="MotorModule"/>
       </xsd:restriction>
      </xsd:simpleType>
     </xsd:element>
     <xsd:element name="Version" type="xsd:string"/>
     <xsd:element name="Details" type="xsd:string" minOccurs="0"/>
     <xsd:element name="DocRef" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="X10ModuleType">
    <xsd:sequence>
     <xsd:element name="Name" type="xsd:string"/>
     <xsd:element name="HouseCode">
      <xsd:simpleType>
       <xsd:restriction base="xsd:string">
        <xsd:pattern value="[A-P]{1}"/>
       </xsd:restriction>
      </xsd:simpleType>
     </xsd:element>
     <xsd:element name="UnitCode">
      <xsd:simpleType>
       <xsd:restriction base="xsd:string">
        <xsd:enumeration value="ONE"/>
        <xsd:enumeration value="TWO"/>
        <xsd:enumeration value="THREE"/>
        <xsd:enumeration value="FOUR"/>
        <xsd:enumeration value="FIVE"/>
        <xsd:enumeration value="SIX"/>
        <xsd:enumeration value="SEVEN"/>
        <xsd:enumeration value="EIGHT"/>
        <xsd:enumeration value="NINE"/>
        <xsd:enumeration value="TEN"/>
        <xsd:enumeration value="ELEVEN"/>
        <xsd:enumeration value="TWELVE"/>
        <xsd:enumeration value="THIRTEEN"/>
        <xsd:enumeration value="FOURTEEN"/>
        <xsd:enumeration value="FIFTEEN"/>
        <xsd:enumeration value="SIXTEEN"/>
       </xsd:restriction>
      </xsd:simpleType>
     </xsd:element>
     <xsd:element name="Class" type="xsd:string"/>
      <xsd:simpleType>
       <xsd:restriction base="xsd:string">
        <xsd:enumeration value="X10LampModule"/>
        <xsd:enumeration value="X10ApplianceModule"/>
       </xsd:restriction>
      </xsd:simpleType>
     </xsd:element>
     <xsd:element name="Version" type="xsd:string"/>
     <xsd:element name="Details" type="xsd:string" minOccurs="0"/>
     <xsd:element name="DocRef" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
   </xsd:complexType>
</xsd:schema>
```