Abteilung Neuroinformatik
Universität Ulm
Prof. Dr. Günther Palm

Advanced Software Concepts and Technologies for Autonomous Mobile Robotics

Dissertation
zur Erlangung des Doktorgrades
Dr. rer. nat.
der Fakultät für Informatik der Universität Ulm

Hans Heinrich Utz
aus Heidelberg
2005

Dekan :	Prof. Dr. Helmuth Partsch
Gutachter :	Prof. Dr. Günther Palm
Gutachter :	Prof. Dr. Helmuth Partsch
Gutachter :	Prof. Dr. Davide Brugali
Tag der Promotion :	2. November 2005

Contents

Abstract

Research on autonomous mobile robots and robot teams has gained significant momentum over the last ten years and robotics application scenarios are becoming increasingly ambitious. But developing software for mobile robot applications is a tedious and error-prone task. Modern mobile robot systems are distributed systems and their designs exhibit large heterogeneity in terms of hardware and have to deal with soft real-time constraints as well as the tight coupling of the software solutions to the physical properties of the robot. Most of today's robot software environments provide little conceptual support to structure and encapsulate these domain characteristics and to foster the development of reusable solutions of robotics software.

This thesis introduces research into the application of advanced software methods and technologies within the field of autonomous mobile robots to make the development of mobile robot applications easier and faster and to foster reusability and scalability of robot software. The conceptual solutions identified during this research process are exemplified in the design of the *Miro* software architecture. It provides extensive support for the management of the domain-specific difficulties by addressing them in the form of a middleware-oriented architecture.

The architecture is divided into four layers that lie between the operating system and the robotics application. The device layer encapsulates the communication with the low-level controller boards. The infrastructure layer provides standard-based communication facilities and support for configuration and parameter management. On the service layer, the robots sensor and actuator devices are modeled as abstract services with network transparent, generalized interfaces. On top, the framework layer provides semi-complete solutions for often encounter robotics problems such as behavior-based reactive control or robot vision in form of scalable, user extensible frameworks.

Miro has been carefully assessed and is successfully applied on various robot platform, in different scenarios, and in multiple research projects. It is also used for robotics software development at other laboratories.

Acknowledgments

The work described in this thesis was partially funded by the German Science Foundation (DFG) under a multi-year grant for the project *Adaptivity and Learning in Teams of Cooperating Mobile Robots* within the focal program SPP-1125. The University of Ulm provided additional funding for enabling the students of the autonomous mobile robots lab to participate at national and international RoboCup tournaments, which were a valuable testbed for the work discussed in this thesis.

Among the many people that supported me through the years of work on this thesis, I want to especially thank the following people:

Prof. Dr. Günther Palm enabled my work at his department and always showed a large confidence in my research. He also managed to provide funding to satisfy the constant needs of the robots for new motors, servos or other spare parts.

Prof. Dr. Gerhard Kraetzschmar did a lot of pioneering work for the autonomous mobile robotics and especially RoboCup at the University of Ulm. Despite his very packed time table he managed to devote time for supporting me in my research with many discussions and provided valuable advice.

Prof. Dr. Helmuth Partsch I want to thank for supporting our RoboCup activities, for providing feedback on my thesis, and for serving on my thesis committee.

My former and current fellow PhD students in the robtics lab, *Stefan Sablatnög, Stefan Enderle, Gerd Mayer, Ulrich Kaufmann* and *Guillem Pages Gassull* facilitated me to work in a creative, open atmosphere and together we explored many ideas on robot software and hardware design. Especially I want to thank *Gerd Mayer* for the many fruitful discussions, which often lasted long after working hours until late at night.

The students of the The Ulm Sparrows RoboCup team always devoted lots of their spare time to the development of the robots' software and hardware. They impressed me with their team spirit during the tournaments and never stopped with challenging my beliefs on autonomous mobile robots in general and in RoboCup environments in particular.

Last, but not least I am most grateful to my parents, my family, and my girlfriend Yiman, who endured many work-through weekends before RoboCup tournaments, and provided moral support whenever it was needed.

Chapter 1

Introduction

Research on autonomous mobile robots and robot teams has gained significant momentum over the last ten years. With the necessary computational power and wireless networking technology becoming available for mobile platforms, also robotics application scenarios are becoming increasingly ambitious. Modern autonomous mobile robots are equipped with a comprehensive set of sensors and actuators, and teams of robots are expected to share their sensory observations and to cooperate in the execution of tasks. Because of this growth in complexity, the problem set addressed by autonomous mobile robot research is large and will become even more all-encompassing in the near future. Therefore, more and more disciplines of computer science such as distributed systems, computer vision, or neural information processing and learning will have to be incorporated into robotics applications in order to meet the growing challenges of modern autonomous mobile robots.

Because of its innate characteristics, the application domain of autonomous mobile robots holds huge challenges for software development. It features a heterogeneous set of hardware devices, the need for concurrent and distributed information processing, a tight coupling of algorithmic solutions to the physical properties of the robot and its environment, the stochastic properties of the physical world, as well as time and resource constraints. Moreover, all these features need to be addressed simultaneously when developing solutions in the field of autonomous mobile robots.

Most of today's software architectures for this domain provide little conceptual support for the application programmer to structure and encapsulate these domain characteristics. An additional problem is the severe lack of research in the robotics community to address the integration challenge posed by the diverse methods and technologies applied within this domain. Many of the robotics applications spend most of their time managing the domain's elementary problems instead of solving the targeted ones. This hinders the implementation of portable, reusable and interoperable generalized solutions that form building blocks for subsequent robot applications. In consequence, software development for autonomous mobile robots is tedious and error-prone, which, in turn, limits

research, the exchange of scientific results and jeopardizes commercialization.

This thesis introduces research into the application of advanced software methods and technologies within the field of autonomous mobile robots in order to address the intrinsic software development challenges of this application domain. The conceptual solutions identified during this research process are exemplified in the design of the *Miro* software architecture, the "middleware for robots" [140]. The basic idea of *Miro* is to provide extensive support for the management of the domain-specific difficulties by addressing them in the form of a middleware-oriented architecture.

1.1 Motivation

There is a growing interest within the robotics community to address the various software quality issues in autonomous mobile robot applications. The goal is to enhance the non-functional aspects of software for autonomous mobile robots such as stability, maintainability, or scalability to form building blocks of robot software functionality that can be reused by subsequent robotic applications.

Currently, hardly any sophisticated toolkits for the engineering of autonomous mobile robot applications exist. Even the direct reuse of existing solutions in different subsequent research projects is rarely seen. Difficulties often arise even within one single laboratory for various reasons. For instance, robot platforms become outdated and are often replaced between projects. But robots from different manufacturers usually come with different, incompatible software environments. Also, senior students and PhD students often leave the university after successful completion of their research. Research prototypes of robotics software especially are notoriously undocumented. But new robotic systems usually encompass and extend the functionality provided by preceeding projects. Therefore, the ability to reuse existing developments in robotics research is crucial for the success of a robotics project.

Reusability is closely related to scalability in two different dimensions: the manageable growth of project size and the manageable load related to system runtime performance. While the first depends heavily on a clear and natural structuring of the application, the second is usually dependent on the ability to identify and resolve hot-spots in the runtime environment. Both these aspects can be enhanced tremendously by the underlying software architecture.

Adaptivity is another key feature for reusable components especially with autonomous mobile robots. As the correct algorithmic solution of a robotic task is usually heavily dependent on the physical aspects of the robot and its environment, it is inevitable to adapt these parameters when applying a solution to another robot or environment. The infrastructure can support this adaptation process in two ways: firstly by making known physical properties of the robot accessible to the application, and secondly by providing tools that facilitate the application of technologies for learning and adaptation such as neural networks.

Scalability can be improved by using properly debugged and assessed high quality components. For this purpose roboticists need conceptual support in the development of robotic applications. Specialized tools that support developers in solving the specific problems of the robotics domain could therefore help to raise the software quality in this respect.

To successfully manage the growth in size and complexity of robotic applications, research into and development of autonomous mobile robots could be helped tremendously if supported by a solid software infrastructure, i.e. toolkits, frameworks and libraries that provide portable solutions usable for various robotics applications. Additionally, progress in this domain will facilitate the broader application and evaluation of proposed research solutions and will therefore help to raise the scientific standards within this particular research community.

1.2 The Problem

To foster reusability, scalability and adaptivity in robotic applications, various problems have to be addressed on different levels of an individual autonomous mobile robot application.

On the lowest level, the sensors and actuators of a robot are usually controlled by microcontroller boards that are attached to one or more PCs, which perform all higher-level processing. Due to the diversity of the available controller boards and communication links, there is little potential for reusable standardization of the communication on this level. Therefore, applications that directly interface with the robotics sensors and actuators on this level are hardly portable between robot platforms.

Another set of problems arises due to the fact that robotic applications are inherently distributed. The sensors and actuators often reside on multiple controller boards and higher-level processing of an autonomous mobile robot is frequently distributed over multiple PCs. Additionally, researchers usually supervise their robots from workstations in the laboratory. For multirobot scenarios, the scenarios themselves already imply the distributedness of the application. Such heavily distributed systems introduce their own intriguing problem set, ranging from inherent concurrency and latency within the application to the failure or temporary breakdown of communication links or entire sub-systems.

Reusability could be facilitated by defining generalizations for similar sensors and actuators of different robot platforms. But the properties of the different devices available from different manufacturers vary especially in crucial aspects such as the order and frequency in which e.g. ultra-sonic range sensors can be queried. So in practice, even similar devices from the same category often confront the application developer with widely varying interfaces.

Generalizing robotics solutions in order to make them work flexibly on different versions of sensor and actuator modalities makes it necessary to deal with the

variance in properties such as the reliability and precision of the sensor readings or the velocity and accelerations of motors. This is a major challenge, as even small differences can change the overall requirements of a subsystem significantly. Lower deceleration capabilities, for instance, require to scan a wider area for the task of obstacle avoidance. This might in turn alter the evaluation of range sensor scans or even the focus of attention control. Thus, the capabilities of every sensor and actuator are tightly related to the overall performance of the entire system and have to be individually weighted in different application scenarios.

Because robots are intended for use in the real world, they also need to be able to cope with real-time requirements at least to some extent. Hard real time, that is maximum response times of only a few μsec, is usually only required for small subsystems, that mostly reside on their own microcontroller boards. Nevertheless, timeliness and reliable response times are also an issue for the PC-based higher-level services. So the design of priorities in sensor processing and actuator control also need to be added to the problem set.

With autonomous mobile robots becoming a wide spread and mature technology, the need to systematically assess and benchmark new and existing methods is becoming mandatory. The stochastic nature of natural environments however, requires an empirical, statistical evaluation, preferably based on series of experiments in the actual application environment. Apart from the many unsolved methodological problems, there is also a lack of readily available methods and tools for acquiring the relevant data on individual robots and multirobot teams in an effective and efficient manner.

It is obvious that many of the problems sketched above cannot be solved once and for all by means of one fixed solution. Instead, many approaches need calibration and adaption of their parameters in order to cope with changes in environment or the robot itself, like lightning conditions or the dynamics of the robot when holding another object. These require the use of adaptive technology and learning algorithms. While such abilities are usually associated with higher-level services, that are beyond the scope of a generic robot architecture, they have to be considered in the design of the lower-level services. Otherwise it can become significantly more complicated to embody learning and adaptive behavior on the higher levels of the entire robot application.

1.3 Solution Approach

This thesis discusses and evaluates solutions for the problem set described above for modern autonomous mobile robot software development. The research undertaken focused on three main topics: (1) the applicability of standard distributed systems technology for robotics, (2) the design of suitable abstractions for robotic devices and (3) providing support for the non-functional aspects such as scalability and reactivity in the development of robotics applications. The proposed solutions are exemplified in the *Miro* architecture: the Middleware

for Robots [140].

The inherent distributedness of robotic applications is addressed in *Miro* by the application and customization of a CORBA-based communication infrastructure. *Miro* supports both the classical method-call-oriented client/server model as well as a message-based publisher/subscriber protocol. Special precautions are taken to address the needs of group communication in teams of robots using WLAN technology.

The different sensor and actuator devices in their various flavors are wrapped by generalized interfaces and implemented as network transparent services. Inheritance is used to make the individual features of the devices available.

Higher-level robotic functionality is supported by specialized frameworks that are used to build customizable high-level services. These have been carefully assessed for their applicability in real-time-constrained environments. By inverting the control flow of time-critical tasks like image processing or reactive control, the timeliness of information processing can be managed and controlled centrally by the middleware itself. For performance assessment the user is provided with sophisticated instruments for monitoring and logging.

1.4 Major Results

The scientific contributions of this thesis lie in the identification and application of supportive technology as well as the design and exemplification of services and frameworks for high-level robotic applications to be used in upcoming application scenarios. These are bundled in the concise design of a middleware-oriented, flexible, scalable and portable distributed software architecture for autonomous mobile robots. The discussed implementation could prove its applicability in various successful applications for different scenarios.

Communication: Within this architecture, distributed systems technology has been carefully assessed for its applicability to autonomous mobile robots, especially with respect to highly dynamic environments. Bottlenecks, e.g. in event-based team communication, were identified and successfully resolved by transparent extensions [141].

Data acquisition: The design and the performance of an efficient service for generic data acquisition in distributed multirobot scenarios, as required for various means in autonomous mobile systems such as debugging, empirical evaluation and learning was discussed [137].

Service design: A design for service-oriented abstraction of sensory and actuatory devices was proposed and successfully applied to the devices of various robot platforms [140]. The services and higher-level frameworks did independently prove their applicability.

Video image processing: The benefits of the middleware-oriented approach in addressing the intrinsic difficulties of robotics applications could be clearly demonstrated in the design and performance of a framework for video image processing in real-time constrained robotic scenarios. It allows to define priorization and synchronization of image evaluations that are both sensor triggered as well as demand driven [135]. It successfully fosters code reuse for different scenarios such as robot soccer and biomimetic learning.

Behavior-based reactive control: The proposed formally specified model for behavior-based reactive control provides a scalable way for specifying rich sets of emergent, reactive behavior in a reusable way. The integration with the middleware, especially the use of device abstractions, proliferates the development of flexible behavior implementations that are reusable on various robot platforms in different scenarios [136]. In the RoboCup scenarios it proved itself as a sustainable foundation for the development of a rich set of reactive repertoire.

Miro has been actively developed and maintained for several years and was successfully applied in such different scenarios as robot soccer [70] or biologically motivated neural learning [29]. The architecture was also ported by external research groups that use *Miro* for their robots [125], contributing additional interfaces, services and device implementations [142, 71] that adhere to the proposed design.

1.5 Overview of the Thesis

The remainder of this thesis is organized as follows. In the following chapter an overview of the challenges of robot software development as reflected in the literature is given and work related to this thesis is discussed. Chapter 3 then introduces the *Miro* architecture and its design principles, showing how it helps to overcome the problems identified earlier. The chapters following thereafter discuss the different layers of the *Miro* architecture. Chapter 4 describes the infrastructure provided for communication and configuration, Chapter 5 discusses the main features of the service layer, while Chapter 6 presents different higher-level frameworks for robotics application development. After a short chapter on the tools that were developed to support robotics software development with *Miro*, various aspects of the implementation are evaluated in Chapter 8. Thereafter, successful applications and further results are described in Chapter 9. The thesis ends with conclusions and indications for possible future work in chapter 10.

Chapter 2

Robotics Software Development

Developing software for mobile robot applications is a tedious and error-prone task. Modern mobile robots are usually composed of heterogeneous sets of hardware components which are connected using different networking technologies and communication protocols with widely varying bandwidths. A large number of different methods for processing sensor information and controlling actuators, for performing computer vision and cognitive tasks like planning and navigation, and also user interactions must be integrated into a well-engineered piece of software. All these issues contribute to the enormous complexity of the mobile robot software development task. There is growing concern in the robotics community about programming environments provided by manufacturers not keeping pace with recent developments in software technology. Also, their closed, platform-bound design does not meet the requirements of today's mobile robot application scenarios [76, 38].

In order to assess the requirements of today's robot software architectures, a more detailed view on robot software and application development needs to be taken. Therefore, an overview of current and upcoming autonomous mobile robots, autonomous mobile robot application scenarios and the challenging characteristics of autonomous mobile robotics as an application domain will be given here. This overview will be followed by a discussion of related robot software architectures and projects that address challenges of robotics software development.

2.1 Autonomous Mobile Robots

Today, mobile robots come in various different shapes and sizes that can be divided roughly into three main categories based on the computational hardware infrastructure provided:

a)  b)  c) 

Figure 2.1: Different categories of autonomous mobile robots: A Tetrixx robot, the B21 platform and the Pino

1. Miniature robots with limited computational power like the Lego Mindstorms, the Khepera robots or the Tetrixx robot kit [27]. These are predominantly used for educational purposes and usually only provide an embedded micro-controller for the control of various analog and digital I/O-ports (Figure 2.1a).

2. Mid-size PC-based platforms which nowadays are the standard robots used for research in autonomous mobile systems. There are various commercial platforms available for indoor and outdoor use, like the B21 or the Pioneer series and many custom-built robots populate the laboratories (Figure 2.1b).

3. Legged and humanoid robots equipped with specialized high-end embedded systems, like the Sony AIBO or the Curio platform. Those are not generally available and usually bring their own proprietary operating system (Figure 2.1c).

In this thesis the focus lies on the PC-based research platforms, which usually come with a wheeled locomotion system. The results presented also apply to the other two categories, but the limited computational resources on the one side and the specialized hardware and non-standard operating systems on the other make a direct transfer somewhat cumbersome.

2.2 Robotic Scenarios

In the robotics community research is usually centered around a scenario, in which results are demonstrated and evaluated. Many application scenarios are actually market-driven, but the main value of a scenario is to provide the community with a common ground on which different solutions can be more easily compared against each other.

Robotic scenarios define general system properties that often impose challenges for the robots' software architecture. These also imply requirements from non-

functional aspects of a robot's application such as reliability, fault tolerance and responsiveness. Newly upcoming scenarios usually feature a more complex set of research challenges that mark the progress in robotics research. Different robotic scenarios are also designed to stress different aspects of advanced future robot applications and robotics solution should therefore be flexible enough to be applicable to different scenarios. Thus, robot software architecture needs to support robot platforms with solutions for these different research challenges and address their different demands.

Robotic scenarios today range from artificial toy robot scenarios such as "trash can cleaning", which are used for educational purposes, to autonomous driving on highways or on unpaved terrain. The research undertaken for this thesis mainly focused on two scenarios popular with the robotics community. These two scenarios stress fairly different system properties of an autonomous mobile robot. On the one side there is the classical office scenario, and on the other the more and more popular environment of robot soccer. While the architectural issues addressed within these scenario most probably will be too complex for educational toy robot scenarios, they also cover many aspects of other upcoming robotics research such as robot companions that can also operate outdoors.

2.2.1 Office Robotics

The office scenario addresses issues such as simultaneous localization and mapping (SLAM) [24], navigation or simple interactive tasks within an office environment [69]. Figure 2.2 is a screenshot taken from a visualization tool for such a scenario. It shows a 2-D map drawn up with a laser scanner. The red rectangles and green squares visualize different granularities of a hierarchical path planner. The red line denotes the path actually taken and the red circle denotes the current position of the robot. Robots operate at moderate speed and feature higher-level cognitive capabilities such as task scheduling or deliberative planning. Human-robot interaction, web-based user interfaces [35] as well as adaptivity and learning [29] are popular research topics. Cooperation among multiple robots is also becoming of greater interest within this scenario. The multirobot SLAM, for example, was evaluated in the CentiBOTS project with a fleet of a hundred robots [63].

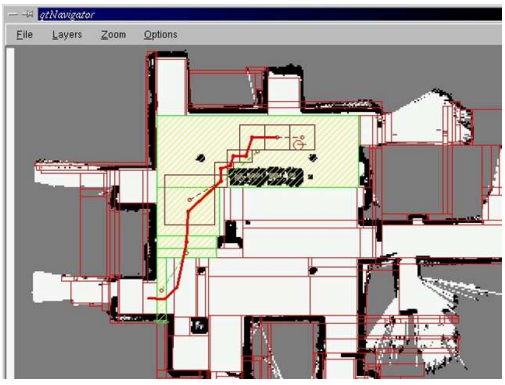

Figure 2.2: Robot navigating in an autonomously built map.

A major feature of research platforms designed for these scenarios is their rich set of sensory and actuator devices for perceiving and manipulating the natural environment. Often special purpose sensors are installed for extracting task-

relevant information such as landmarks and maps, static and moving obstacles, stairs or tables. Apart from the robot's locomotion system, actuators usually include simple grippers and multiple-degrees of freedom robot arms. The applications integrate a comprehensive set of methodologies for sensory, actuatory and cognitive purposes. The different focus of future research projects in this domain makes adaptations and modifications to previously developed solutions or even complete rearrangements of functional subsystems necessary.

The difficulties of this scenario from a robot software architecture point of view lie in the diversity of the robot's sensors and actuators as well as the number of modules and their complexity within typical applications. A robot software architecture can support application development in such a domain in the area of management and interfacing of sensory and actuatory devices, by providing domain-specific frameworks that facilitate the implementation of subsystems such as localization or reactive execution, by supporting a proper modularization and by keeping the overall application manageable.

2.2.2 Robot Soccer

Figure 2.3: Kickoff at a RoboCup game in the medium-size robot league.

Since 1997 the RoboCup scenario has been putting autonomous mobile robots in a highly dynamic environment, introducing robot teams as well as competition between robots [7, 144]. There are several leagues for robot soccer, ranging from simple 2-D simulation to humanoid robots. The team of the robotics lab at the University of Ulm, the The Ulm Sparrows, has been participating in the medium-size robot league since 1999. In this league, teams of 4 to 6 robots with a maximum size of 50 x 50 cm compete on a field of 12 x 8 m (see Figure 2.3). The most interesting aspect of the scenario is that the problem can not be 'solved' as such, but rather promotes an evolutionary process, where new solutions compete with the existing state of the art [74].

The high dynamics of the scenario impose tight timeliness constraints and promote the reactive capabilities of the system. The multirobot teams make the distributed nature of robotic applications explicit. Also, vision is the dominant sensor within this scenario, since the autonomous, robust detection and classification of various objects in real time (ball, goal, opponents, team mates) is the dominant cognitive problem [81, 82, 79]. On the other hand, the higher-level deliberative capabilities of such robots are usually rather limited.

A robot software architecture can contribute to the development of robot soccer applications by managing the difficulties of the distributed systems, the assessment of the timing characteristics of the different subsystems and by providing

scalable support for the deployment of a rich set of reactive behavior.

2.3 Challenges of the Application Domain

Apart from the different domain specific emphasis, the above described two application scenarios feature a set of difficulties that are prominent in almost every non-trivial application of autonomous mobile robots and which are all individually considered difficult. These characteristics need to be addressed in parallel to the actual robotics application. They are often underestimated and form a major obstacle for the successful integration of solved research problems into a running application.

Extreme heterogeneity of hardware components. A huge variety of sensors and actuators are available for use on autonomous mobile robots. As a consequence, almost every robot has its individual sensor suite and actuatory capabilities. It would be desirable to build logical groups of similar sensor and actuator devices that are accessible by a generalized interface. But even very similar devices offer different capabilities such as maximum sensor sampling rates, different sensing range and accuracy etc. that need to be taken into account when writing software targeted for use on more than one individual robot.

Inherent concurrency. An autonomous mobile system has to process several tasks at the same time. Various sensory and cognitive processing procedures as well as the control of multiple actuators need to take place in parallel and on different time scales. The computational power of dual-processor boards, hyper-threading technologies and the upcoming multi-core CPUs can only be fully exploited when these processes are also implemented as parallel processing threads.

Developing multi-threaded applications is considered difficult and development support for parallel processes is sparse in most programming languages and development environments. Race conditions between different processing threads are hard to achieve by classical single-step debugging and provoke non-deterministic system failure. The necessity to synchronize concurrent data access can also introduce subtle performance bottlenecks.

Inherent distribution. Multi-robot scenarios make the distributed nature of robotics applications apparent. But also a single robot often delegates computationally expensive tasks to remote workstations to overcome computational bottlenecks and receives input from web-based user interfaces. Teleoperation also requires network transparent access to the robots' interfaces. Additionally the monitoring and visualization of task execution is usually performed from a remote workstation. Some robots are also equipped with multiple PCs to provide extended computational power. Thus, the problematic characteristics of distributed systems are an in-

tegral part of the challenges associated with autonomous mobile robot applications.

Device and environment dependency. Algorithmic solutions for robotic problems are heavily dependent on the properties of the physical devices that are controlled or deliver sensory information. An algorithm for obstacle avoidance reliably working on a robot running at $1m/s$ and accelerating/decelerating with $1.5m/s^2$ is unlikely to work with $2m/s$ and $0.5m/s^2$ deceleration without major adjustments on the sensory side. This could even include adaptations of the visual attention control to recognize obstacles earlier. Algorithmic solutions therefore not only need to precisely model their physical dependencies for their flexible application. The system as a whole also needs to be flexibly adapted to the requirements of and interdependencies between its various modules.

Stochastic properties of the physical world. Robot experiments and applications in the physical world have to deal with its stochastic properties that make it impossible to guarantee a deterministic outcome. Robot sensor data is noisy and for instance lighting conditions are not controllable in an outdoor scenario. Additionally, many important parameters of the environment vary over time. This has various implications for the design, implementation and evaluation of software within this domain, as the gathering and processing of sensory information has to model its stochastic nature and the experimental evaluation needs to take the non-deterministic properties into account.

Real-time constraints. A robot acting in the physical world needs deterministic response times of various subsystems in order not to endanger others or himself. The admissible response times and jitter do not usually lie in the area of hard real-time performance. Nevertheless, tasks like obstacle avoidance require the system to satisfy at least soft real-time capabilities. In such reactive control tasks, the right answer delivered too late becomes the wrong answer, and most of these control loops run with a pace of 5Hz to 50Hz.

Resource constraints. Apart from the usual constraints on a computer's resources such as the available processing power or physical memory, a robot has additional resource limitations. Actuators can only have one physical state, so concurrent requests from multiple modules of the system need to be arbitrated. And while sensory information can easily be replicated for use by multiple sub-systems, these systems still have a maximum frequency at which new sensor readings can be sampled.

A software architecture that does not provide sufficient support for the management of the intrinsic difficulties of the robotics domain force the application programmer to cope with these problems in the limited scope of the program domain. This leads to a limited generality of the solutions, a frequent ad-hoc

reinvention of the wheel, the heavy use of heuristic solutions and proliferation of inadequate evaluation and assessment, especially of the resulting solutions.

This thesis covers research on how to address these domain characteristics on the level of a system architecture. This form of central management will make it possible to solve the general difficulties inherent in the domain from other aspects of the application.

2.4 Environments for Robot Software Development

Various architectures for programming robot applications currently exist. Most commercial robots come within their own proprietary robot and control programming environment. In the last five years various projects have been started that address the challenge of providing more sophisticated cross-platform environment in order to resolve different issues like portability and real-time control. In this section an overview of the most relevant projects in this area will be given.

2.4.1 Commercial Vendor System

A typical commercial robot software environment supports only the robots from one manufacturer. Two prominent typical representatives of such environments will be covered in this section.

Saphira

Saphira [1, 62] is the software development environment delivered with the Pioneer mobile robot family. Its core is a C library for accessing the controller board. By using the Saphira library, the programmer implicitly imports elements of a particular robot control architecture, including mechanisms like state reflection (a kind of implicit communication between client programs and server), data structures like a Local Perceptual Space (LPS) and a Global Perceptual Space (GPS), a fuzzy control-based behavior specification language, and Colbert, a language for reactive control [64]. Saphira lacks location transparency, and when integrating hardware from other manufacturers (vision systems, manipulators), the programmer must carefully synchronize the communication of client programs with add-on hardware with the basic Saphira control loop.

Mobility

Mobility is a distributed, object-oriented software development framework for the B21r/B14r family of robots by RWI [98]. It scales well with respect to user

interaction, client libraries for visualization and remote access of sensory information. Client libraries are available in C++ as well as in Java. On the other hand, Mobility lacks conceptual support for robot control. As a manufacturer-provided package, it is only available for mobile platforms provided by its manufacturer. Since not even our old B21 robot was supported by Mobility it could only be evaluated by investigating its documentation and header files.

Mobility also exposes another problem of vendor-supported robot libraries: even though RWI followed a rather liberal policy in opening up their source for scientific researchers, Mobility vanished with the buyout of RWI by Evolution Robotics in 2002.

2.4.2 Research Architectures

Due to the limitations of vendor-provided environments, some groups of researchers started to invent their own architectures. The most successful of these are actually an Open Source community project. These robot software architectures usually support robot platforms from several different manufacturers and are designed to address different aspects of the application domain's problem set.

Player/Stage

The Player/Stage project was started in 1999 and aims to provide standard APIs for autonomous mobile robots, the Player Abstract Device Interfaces (APDI) [36]. The project consists of two components. One is the Player software, which provides client/server interfaces for robot devices [143]. It supports several commercial robot platforms such as the RWI B21r robot and the Active Media Pioneer series. The other component is Second Stage, a 2D simulator able to provide virtual robots offering Player interfaces for multirobot systems research [37]. A 3D simulator named Gazebo is also under development. The Player project probably has the biggest user base of the projects discussed here.

The Player project omits OO-techniques and middleware technology for the sake of simplicity. The device abstraction model is based on the simple serial device API of read/write/ioctl [31] and does not reflect the inherent asynchronicity of message-based low-level controller communication. It does not even support event-demultiplexing based on the UNIX select() function. The servers provide TCP/IP based socket interfaces, lacking the type safety and scalability of modern middleware technology and missing all possible optimization opportunities for colocated clients and servers. They also lack support for sensor-driven event-based communication as the architecture is targeted towards robotic applications with moderate timing constraints. But this tends to introduce unnecessary and avoidable latencies to a core system design. Add-on libraries provide object-oriented wrappers for the low-level communication links on the client side.

OROCOS

The acronym OROCOS actually stands for two related projects, both for the "Open Robot Control Software" and for "Open Real-time Control Services". The OROCOS project was also started in 1999 and pursued similar goals as the Player project, but it was targeted at real-time-oriented robot control [16]. Thus, it provides a real-time kernel for low-level control for devices ranging from PID servo controllers to robot arms [17] and a software framework for robotic control software development.

The integration of a middleware-based communication infrastructure for higher-level autonomous mobile robot control is a stated project goal, but all attempts to do so led to different, non-interoperable projects, which will be discussed in the next two subsections.

SmartSOFT and OROCOS::SmartSOFT

SmartSOFT is an early attempt to introduce modern software technology such as object orientation and component-ware into robotics application development. It provided object-oriented interfaces for the sensors and actuators of the B21 robot and templates for the implementation of typical communication patterns identified in robotic applications [107, 105]. The software framework was structured after the classical 3T architecture and also provides modules for task decomposition and localization [106]. SmartSOFT was developed further in the OROCOS project and became OROCOS::SmartSOFT.

The project was based upon the original B21 control software BeeSoft and its message-based proprietary communications infrastructure TCX [30]. Later versions also use CORBA as their communications infrastructure. As most of the identified communication patterns such as one-way communication (commands in their terminology), synchronous and asynchronous queries and publisher/-subscriber protocols are not exclusive to robotic applications, they have recently become available in modern standardized communication middleware architectures. CORBA, for instance, offers specialized communication technologies with an advanced feature set for all patterns listed above: reliable one-ways [116], synchronous and asynchronous method invocation [23], and the Notification Service [41].

OROCOS@KTH and Orca

Orca, previously called OROCOS@KTH, is a component framework for autonomous mobile robots [92, 12]. Components are software entities with a sufficiently strict definition of interfaces and their semantics, which allow for their flexible use and interchangeability. A component framework provides the necessary support to deploy, configure and initialize components within a system and link their communication ports to form a modular, highly configurable system.

Orca uses an XML-based specification language to describe the configuration and initialization of such a component system.

Component approaches are a highly relevant concept for improving the modularity of robotics software. Several component models such as Enterprise Java Beans, the .NET architecture, or the CORBA Component Model (CCM) are currently available. However, there are several unsolved challenges as regards defining generalizations that could be implemented as interchangeable components. Additionally, Orca refrains from committing itself to any concrete transport protocol, thereby aiming to provide additional flexibility. This means that the communication needs to be implemented by hand instead of being generated automatically from the interface definitions.

Distributed Robotics Framework

The robotics research group at the department of electrical and computer engineering at the University of Auckland, New Zealand also conducted research in the area of robot software architectures. Their distributed robotics framework features a CORBA-based design similar to the *Miro* system. Woo et al proposed a three-tier software infrastructure based on CORBA and designed around the CORBA Trader Service [150].

Lately, these researchers have been focusing on the application of real-time features of CORBA to improve the reactivity and predictability of the response times of autonomous mobile systems. Kuo and MacDonald proposed using RT-CORBA features to ensure predictable response times for high-priority tasks within a distributed robot control architecture. In [73] they show how the proper prioritization of system tasks can ensure the end to end quality of service (QoS) requirements in a distributed robotic application.

CLARAty

The Coupled Layered Architecture for Robotic Autonomy (CLARAty) is being developed at the Jet Propulsion Laboratory (JPL) and the Carnegie Mellon University (CMU) as a common platform for controlling rover platforms as used in space missions to Mars [91]. They revert the classical 3T architecture consisting of a functional, an executive and a planner layer, in favor of a two layer architecture which divides a robot application into a functional and a decision layer. Additionally, a set of functional components for mobile robot systems are provided [146]. A major challenge addressed on the functional layer is the design of a locomotion system that generalizes the different steering capabilities of the different planetary rover platforms.

The architecture is implemented in C++ for the real-time operating system VxWorks and for Linux. It uses a client/server approach for separating the decision and the functional layer, but no higher-level standard communications infrastructure is used.

Marie

Marie is a fairly new project that addresses rapid prototyping for autonomous mobile robot applications [21]. On the system level, its developers propose the mediator pattern for integrating complete software modules. On the higher level they illustrate how graphical tools like the RobotFlow/FlowDesigner, a flow control tool originated in the DSP domain, can facilitate the programming of robot software applications.

The attempt to integrate existing implemented solutions directly instead of proposing yet another architecture for integrating components for complex autonomous mobile robot applications is very promising, but also exhibits limited applicability for various application scenarios. The communications adapter, for instance, might introduce additional latencies to the sensor processing, which severely limits the quality of available sensory information in highly dynamic environments. A centralized adapter also represents a potential bottleneck in the inherently distributed environment of robotics applications. The high-level tools discussed here primarily rely on a synchronous data processing model, driven by the pull model of querying sensor information. This concept is fairly limited when it comes to handling the complexities of the various concurrently executing sensory, cognitive and actuatory processes that run on the different modules and components of a complex autonomous mobile robot application.

LAAS robot software repository

In [3], Alami et al document the task of integrating a demonstration from scratch on a newly obtained mobile platform within 40 days. They utilize their LAAS architecture [2] for an office navigation scenario. Even though most of the tools' functionalities are located on a higher control level as currently provided within *Miro*, this work nicely documents the potential for generalization and code reuse within the mobile robotics domain. The LAAS laboratory also tries to open up their sources for reuse by the community. But due to the original closed source design of their projects, there are significant difficulties involved in the process and the different libraries and APIs currently present themselves as unrelated modules.

2.5 Assessment of Robot Software Development

Over the last two decades, significant progress has been made in the field of software engineering with respect to mastering the challenges of modern application development. Autonomous mobile robot applications impose severe difficulties which exceed those in many other application domains. Nevertheless, most available software environments for robotic application development show significant deficiencies when it comes to modern programming abstractions and programming paradigms.

	Multi-Platform	Comm.	RT	Availability	High Level Features
Saphira	-	-	-	com.[1]	control language
Mobility	-	CORBA	-	com./-[2]	-
Player	✓	proprietary	-	OS[3]	obstacle avoidance, navigation
OROCOS	✓	-	✓	OS	RT control
SmartSoft	✓	prop./CORBA	-	OS	communication patterns
Orca	✓(?)	open	-	OS	component framework
Aukland	✓/ -	CORBA	- / ✓	-	RT communication
CLARAty	✓[4]	proprietary	✓	-[5]	task control mission planning
Marie	-	open	-	OS	interoperability
LAAS	✓	-	-	OS[6]	deliberative planning

Table 2.1: Properties of commercial and research software architectures for autonomous mobile robot programming.

This has become a bottleneck in the development of autonomous mobile robot applications and hinders both the integration of further research disciplines and the use of new technologies that are needed to master the challenges of future robot application scenarios. Thus, robotic software tends to scale badly in the areas of project growth, performance, reliability and maintenance.

The growing number of projects on robot libraries and robot device servers shows the community's increasing awareness of urgent need to address deficiencies in current robot software development. As summarized in Table 2.1, the related work discussed in the last section features some common capabilities, but they diverge in the provided higher-level features. Research in this area covers very different aspects such as generic low-level control, real-time-related research, large scale system integration and higher-level control. The software architectures partly differ due to the targeted application domain, for instance RT control targeted by OROCOS vs. indoor navigation targeted by Player. Others deliberately provide different approaches for addressing similar higher-level robot applications such as the closely coupled two-tiered approach of CLARAty vs. the classical deliberative 3T architecture of LAAS.

Currently, not a single standard software architecture for autonomous mobile robots exists. There is also still no consensus on how exactly such an architecture should look. One reason for this certainly is the fact that the field of autonomous mobile robots is still evolving too rapidly to be able to define the requirements of tomorrow's robot applications precisely enough. Therefore, identifying the concepts and technologies which provide most valuable benefit for autonomous mobile robot software architectures and evaluating their applicability are still unsolved research problems.

[1] Commercial
[2] No longer available due to buyout
[3] Open Source
[4] Proprietary research platforms and planetary rovers from various labs
[5] Announced for open source release
[6] Not a complete package

Chapter 3

Miro Architecture and Design

This thesis discusses the design and application of a distributed object-oriented robot software architecture. The fundamental design goal of such an architecture is to make the development of mobile robot applications easier and faster, and to foster the modularity, portability, reusability, scalability, and maintainability of robotics applications. The core idea of the design of *Miro* is to address the challenges of autonomous mobile robotics as an application domain by means of a flexible and scalable middleware-oriented software architecture.

Therefore, the requirements for this software architecture are twofold. On the one hand, it needs to be portable, reusable, scalable, and maintainable in order to provide a solid infrastructure for software development. On the other hand, the design of the architecture needs to address the reusability and scalability challenges associated with robotics applications that use this architecture.

The research on this topic is exemplified and evaluated in the *Miro* project. *Miro* generically supports robotic applications with the peculiarities of the application domain and separates them from other research problems targeted by roboticists. Instead of providing another complete model for autonomous mobile robot control, we follow a middleware-oriented design, providing extensible abstractions for robotics software development. The architecture is therefore organized into four interdependent layers that reside between the bare operating system and the application. The thesis especially focuses on how the proposed architecture provides support in mastering the aforementioned challenges of robotics as an application domain.

In this chapter, the design of the *Miro* architecture will be introduced. First, the basic requirements as well as the design principles of the architecture will be discussed. Afterwards, the general idea of middleware technology, as well as the actual design decisions of layering will be sketched and design aspects that target specific issues of robotic software development will be highlighted.

3.1 Requirements and Design Goals

From the related work covered in Section 2.4, a number of basic requirements for a robot software architecture can be derived. They address the applicability of the architecture for different scenarios as well as its portability and reusability. The superset of requirements identified by the majority of related research projects as listed in Table 2.1 reflects the consolidating design issues of robotics software architectures, but does not address the second question of how to propagate the reusability and scalability into the actual robotics applications.

Distributed systems design. The software architecture should provide a carefully designed set of interfaces for communication between objects and for communications transparency, i.e. the programmer should not need to worry much about where objects are actually allocated and which communications protocol he needs to apply. Single robot and multi-robot applications should be treated equally.

We suggest to adopt a client/server view at least for larger modules of a robot control architecture: the modules, implemented as objects, provide services to other modules (objects) through a set of interfaces. Most modules of an application fulfill both the client and the server role, albeit for different subsystems, using the services of sensor objects as clients and providing services to actuator objects as servers.

Multi-platform support and interoperability. The applied software technologies should be available on a wide range of hardware platforms and common operating systems. The use of different programming languages for implementing different modules should be supported without requiring much extra overhead cost for integration.

Real-time orientation. Even though classical robot scenarios like indoor navigation typically have very lax timing constraints, the reactivity and deterministic response times become an important issue in highly dynamic environments and have to be solved for robotic applications that need to perform safely in natural environments.

Open development model. Considering the immense spectrum of different hardware and software components used in mobile robot systems, adopting an open architecture approach including availability of all source code seems indispensable. Only if applications programmers can access, modify and fine-tune all components of the software environment can the integration of a heterogeneous set of hardware and software components be successfully accomplished. Also, integration of new hardware developments and new computational methods developed in ongoing research usually can be achieved much more easily and quickly, if all source code is available for inspection, reuse and debugging purposes.

Modularity, reusability, stability and efficiency are major building blocks for scalable, maintainable high-quality software applications. In order to achieve

this in robotics, it is necessary to apply, extend and exploit the software technologies and concepts that target software development for large-scale applications. Such technologies are currently missing in commercially available robotic software architectures. The approach taken in this thesis is to provide generic support in the form of a robotics middleware for such non-functional aspects of robotics applications. This goal is reflected in the design principles of *Miro*.

Object-Oriented Design: The software architecture should be designed using thoroughly the object-oriented paradigm. Object-oriented concepts like information hiding, name spaces to prevent naming ambiguities when using multiple mutually independent libraries, exception handling to separate normal program control flow from error recovery, abstraction, type polymorphism, and inheritance can all significantly contribute to improve the design and implementation of mobile robot software when applied deliberately and consistently.

Suitable abstractions: A major goal for a successful design is to define suitable abstractions for the different (physical) devices of the various robots. They need to precisely model their properties while at the same time provide generalizations that allow for their abstract usage by client applications.

The object-oriented paradigm permits a clean abstraction of sensor and actuator subsystems as well as low-level system services like communications, and provides a uniform interface design. A suitable level of abstraction from low-level system details may be a key element in attaining a better understanding of how to integrate a significant number of different hardware devices and computational methods.

Employment of existing technologies: Reusability and scalability of software is often enhanced by actually reusing existing solutions. Therefore, the application of existing software technology is favored over the reinvention of functionality specialized for the target domain. Necessary customizations for the target domain actually provide valuable feedback for the development of such base technologies.

Actively targeting the integration of additional methods: Autonomous mobile robot applications require the integration of solutions from various other research disciplines such as computer vision or neuroinformatics. But the robotics domain, in turn, imposes severe additional constraints on solutions from other research disciplines. Therefore, the application of such methods requires active support from the infrastructure of the application domain in order to actually become applicable.

Generalization and scaling of existing robotic core functionality: Various solutions for many of the classical robotics problems such as reactive control or localization do actually exist. It is not the goal of the proposed design to select one apparently best suited solution for exclusive application. Instead, the goal is to provide an infrastructure

that supports a generalized, common view on the problem domain and is applicable to different solution approaches. The design on such generalizations should be focused on improving the scalability and reusability of applied solutions.

Flexible Integration of User Interfaces: A commonly underestimated problem in mobile robot research is the development and use of suitable user interfaces for various tasks. The task-oriented user interface for end users interacting with the robot requires careful design in order to avoid compromising the robot's autonomy and integrity. People operating and supervising the robots need interfaces for monitoring its operational state, while programmers may need even further internal state information for debugging purposes. These user interfaces might be activated only occasionally or under special circumstances. Nevertheless, their effect on the runtime performance should be limited, and the architecture should support the programmer in meeting this goal.

3.2 Middleware

The acronym *Miro*, which denotes the software project implementing the solutions identified and discussed in this thesis, stands for *"The Middleware for Robots"*. Instead of providing a complete model for a robot control architecture, we propose a middleware-oriented design, that provides abstract, flexible services and required functionality for the robotics domain without restricting roboticists in their choice of solutions for the various sub-domains of an application. The term middleware mostly denotes a software layer between the bare operating system and the applications that provides application-independent functionalities beyond what a usual operating system (OS) would support. Wikipedia defines middleware as follows:

> In computing, middleware consists of software agents acting as an intermediary between different application components. It is used most often to support complex, distributed applications. The software agents involved may be one or many. [148].

The term middleware is often used as a synonym for distributed systems middleware as defined by the 'Internet 2' architecture. But apart from distributed systems middleware other application domains also establish their own middleware frameworks. A central goal of middleware is to hide the intrinsic complexities of the necessary infrastructure from the applications [148]. There is also growing interest in the integration of non-functional properties such as fault tolerance, response times, security and scalability into middleware technologies. It has become evident that support for these characteristics of an application relating to "Quality of Service" (QoS) can often be efficiently managed by a middleware layer.

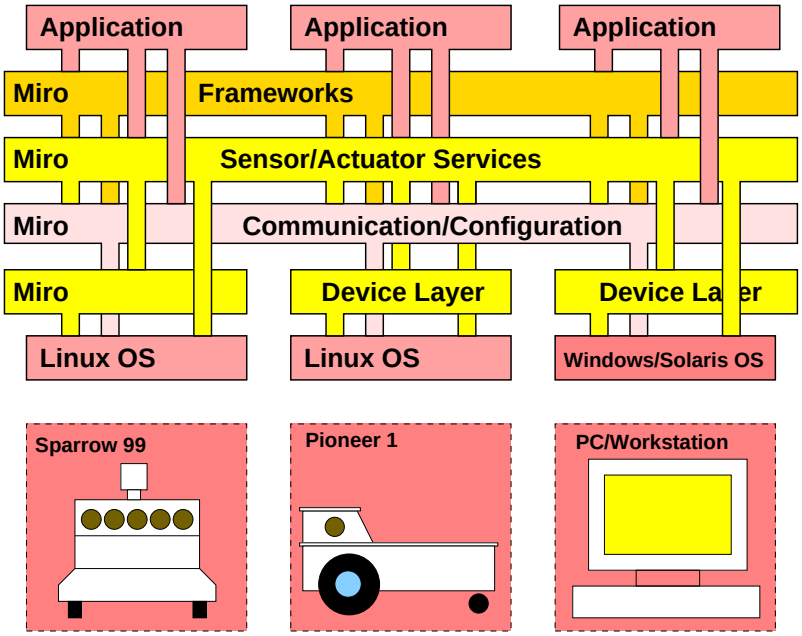

Figure 3.1: The *Miro* architecture. The *Miro* device layer provides abstract interfaces for specific platforms. The Communication and Service layers achieve platform-independence, network and location transparency. The *Miro* frameworks provides reusable designs for generic robot control functionality.

3.3 Architectural Layers

To further structure the various aspects of functionality required from a robotics middleware, *Miro* is structured into four interwoven architectural layers that reside between the bare OS and the robotics applications (see Figure 3.1). As illustrated in the figure, robotic applications only interact with the two upper layers and can also use the infrastructure provided by the communication and configuration layer, but the low-level details of the robot's sensor and actuator devices are hidden from the application programmer in the device layer. The four layers address different problems of the application domain that were identified in Section 2.3 for meeting the requirements and design goals defined at the beginning of this chapter.

3.3.1 Hardware Abstraction

The *Device Layer* addresses the problem of the extreme heterogeneity in the hardware of robotics devices by providing object-oriented interface abstractions for all sensory and actuatory facilities of a robot. This is the platform-dependent part of *Miro*. As the reuse of code is difficult in a programming environment tightly bound to the individual hardware, conceptual reuse on this layer is propagated by a pattern-oriented approach. This device layer-wraps the message or

package-based communication links to the microcontroller boards (serial link, Can bus, IEEE 1394, etc.) into classes to provide appropriate abstractions for interaction with the service layer. At this level the asynchronous nature of the low-level communication protocols is not yet hidden by the interfaces and can be exploited for higher efficiency. Also, the application of the reactor pattern circumvents multi-threaded polling of devices [108]. To the service, layer the device layer provides a synchronous view of the data flow, using the half synch half async pattern [109] for shielding most of the inherent concurrency in low-level device processing from the higher layers of the architecture. The device layer will not be discussed in more detail here. Further information can be found in [133].

3.3.2 Infrastructure

The *Communication Layer* addresses the inherent distributedness of robotics applications by offering a network transparent and programming language independent communication infrastructure, which allows strictly typed communication in a distributed environment. For this purpose a consolidated distributed computing technology, an open source implementation of CORBA, is employed [111]. CORBA offers a rich set of features that are pre-configured and customized on this layer for simplified use in the application domain. For this purpose the communications infrastructure was carefully assessed and appropriate services were selected for extended use within *Miro*. The infrastructure layer provides both the pull as well as the push model of communication. Additionally, bottlenecks in the communication, as they arise within every concrete setup, are addressed on this layer.

A second important aspect is addressed on this layer. Complex distributed applications, especially in the context of robotics, require extensive configurability and parameterization of the various modules within the system to provide the expected flexibility as reusable components. On the other hand, large parameter sets allow for another source of subtle and hard to trace errors, namely configuration errors, that can jeopardize the reliability of the resulting application. This problem set is addressed in *Miro* by a parameter description language that allows for a generic, extensible definition, use and initialization of application parameters. It is used for automatic code generation for the target programming language (C++) as well as by different GUI-based tools that support the development process. The communication and configuration infrastructure will be discussed in Section 4.

3.3.3 Services

The *Miro Service Layer* provides service abstractions for sensors and actuators to overcome the of robotics solutions' dependencies on the concrete physical devices by decoupling them through a client/server-based design. On this layer,

the sensors and actuators are therefore modeled as components with network-transparent, platform-independent interfaces. The programmer uses standard distributed systems technology to interface to any device either on the local or a remote robot.

To ensure consistency in the access of abstract robotics devices, a set of required functionality for sensors and actuators has been identified that is to be provided by the service interfaces. The abstraction from the individual properties of the physical devices is especially supported by two design concepts: metainformation provided by interfaces allows client applications to adapt to the properties of an individual device, and interface hierarchies allow to interface to robotics devices on different levels of abstraction. The various design aspects of different service types are discussed in Section 5.

3.3.4 Extensible Frameworks

The higher-level *Miro Class Frameworks* provide a number of often used functional modules for mobile robot control such as video processing, behavior generation, and self-localization. They provide generalizations of existing robotic core functionalities and can be easily extended to the needs of individual robot applications. The higher-level frameworks export their functionalities as services to other modules of an application by the use of the communication layer.

Frameworks provide extensible modules of reusable code and reusable design for the development of applications. Their integration into a middleware-oriented, multi-platform robot software architecture facilitates the development of reusable client applications. Their goal is to provide support for the generic problem set of the application domain, allowing the application programmer to focus on functionalities provided by the respective module targeted by the framework. Thus, the frameworks address the problem of how the requirements for reactivity and response times are met by managing the control flow and organizing the data flow in the sub-domain of the application. They also assist in the handling of resource constraints of robot devices by providing sensor-triggered evaluation models and arbitration methods for concurrent actuator access. The most prominent class frameworks are discussed in Section 6.

3.4 *Miro* Support for Real-time Processing

As autonomous mobile robots operate in the real world, real-time constraints are an issue in modern robot control architectures. The requirements of timing awareness vary greatly in an autonomous system. Hard real-time (i.e. guaranteed) response times in the range of a few μ-seconds are only needed on rare occasions for state of the art autonomous mobile robots. This concerns mostly actuator controllers and can usually be restricted to microcontroller boards or encapsulated in an RT-control kernel.

Motor commands need to be processed with low latencies and sensor data such as distance information has to be available with little jitter to ensure safe operation in dynamic environments. But these requirements on response times currently only lie in the range of milliseconds, since the maximum speed of autonomous mobile robots usually is only a few meters per second. For example if the reactive control loop of a robot running at 3 m/s is delayed by $10ms$ due to OS latencies, the stopping distance will be prolonged by 0.03 m. On the other hand, the controllers' setpoint can often be altered to a much lower rate, typically to between 30 ms to 100 ms. These response time requirements can be met by modern general-purpose operating systems, if properly supported by the robotics middleware [110].

Real-time constraints need to be met on all levels of an application. Therefore, several points on the different layers of *Miro* were identified, where the software architecture was able to facilitate the fulfillment of response time requirements and improve the reactivity of its applications.

- On the device layer, all incoming sensor data is time-stamped and these stamps are propagated to the sensor service implementations. This allows to properly align different sensory sources and to compensate for latencies introduced by sensor processing (e.g. image processing). The accuracy of such time-stamps is improved by carefully applying real-time scheduling and thread priorities to the low-level device handlers.

- The communication layer can improve the adherence to real-time constraints by the use of middleware technology aware of real time and quality of service (QoS) such as RT-CORBA for prioritizing the processing of service interfaces. These features of the underlying implementation of CORBA are currently not used within *Miro*. The application of this technology within robotics applications is limited by the lack of quality of service (QoS) features in current wireless network technology (WLAN) as used in teams of autonomous mobile robots. WLAN even exhibits large variations in bandwidth, which make it almost impossible to reliably stay within a low network traffic situation where acceptable network latencies could be provided. *Miro* approaches this issue in the context of team communication on the communication layer.

- Sufficiently quick processing of sensory information favors a push model of communication. Otherwise, a prompt reaction on new input data requires excessive polling or frequent context switching to the various waiting threads of control. The *Miro* architecture therefore requires for sensor services to not only support the pull model within their interfaces, but to also actively push sensory information as it becomes available.

- Higher-level frameworks are critical for ensuring the reactivity of an application and the adherence to reactivity constraints. As these are designed to be extended by the application developers' code, it is almost impossible to assess these properties a priori without severely restricting the

applicability of such frameworks. On the other hand, the very nature of class frameworks offers capabilities to facilitate the preservation of responsiveness and to allow for prioritized processing as well as for generic performance assessment of end user applications.

3.5 *Miro* Support for Development

Integrated development environments (IDE), built systems like autoconf and automake or version management systems like CVS are essential for managing the software development process of large-scale applications. Robotic applications need additional tools to speed up development and prototyping of robotic applications. It is part of the task of a robotics software architecture to offer thorough development support and to help to prevent where possible the subtle, hard to find, fatal or at least tedious errors that eat up development time and take their toll on the roboticists' nerves. In order to meet this goal, *Miro* offers a set of GUI-based end user tools, that lower the bar for beginners and speed up the development and debugging process by preventing avoidable errors and facilitating the inspection and understanding of runtime processes (also see Section 7).

3.5.1 Assessment, Adaptation and Learning

The inherent non-determinism of the environment, the robots' mobility and the physical dynamics of the environment have related implications for the development process as well as for the methodology used within the applications.

Debugging, testing and evaluation of software always is difficult. But robotic applications face additional problems in this respect due to the nature of this application domain. The environment is inherently non-deterministic, the robot moves, and the physical dynamics of the environment are also part of the problem domain. This makes it very difficult to reproduce occasional failures in the system for closer inspection or to assess the interactions of the robot with its environment in detail. This requires empirical evaluation of robotic applications based on large amounts of data retrieved in real-world experiments.

The ubiquitous presence of sensor noise and the high variability of sensory observations from the same object (e.g. in images taken from different sides) call for the application of statistical and neural methods for information processing. Also, the abilities to learn and adapt are considered to be key factors for successful robot applications, especially in natural environments. (The actual application of neural methods for robots in highly dynamic environments cannot be covered in this thesis. Interested readers might refer to [80, 60].)

A crucial feature in supporting robotic applications with these problems is the availability of technology for convenient sensor data acquisition and logging from experiments that can be used for assessment, system adaptation and the

training of learning algorithms. Therefore, *Miro* provides sophisticated logging facilities that allow to store complete traces of the robot's sensory and actuatory processes. Along with appropriate tools for replaying and visualization, they allow to carefully inspect the system's internal states during an experimental run.

3.6 Summary

In this chapter, important design issues for robotics middleware were discussed. The identified requirements ensure the software architecture's reusability with and applicability to different robot platforms and the fulfillment of the requirements imposed by various scenarios. The formulated design goals target the reusability, scalability and maintainability of modular client applications.

In order to not restrict developers in the design of an overall model of robot control, a middleware-oriented design was chosen for the software architecture. The proposed *Miro* architecture provides four layers of functionality that lie between the bare operating system and the client applications. These provide libraries for creating abstractions from the low-level interaction with microcontroller boards, a standard-based communication infrastructure, sensor and actuator abstractions with network-transparent interfaces and class frameworks for the implementation of scalable and reusable modules of typical robot functionalities.

Additional aspects identified for the applicability of the architecture are the awareness of the real-time constraints present in more advanced robotics scenarios, the availability of tools that support the development process, and capabilities for assessment, adaptivity and learning from experimental real-world data.

Chapter 4

Miro Infrastructure Layer

The communication and configuration layer of the *Miro* architecture is the lowest layer visible to the applications programmer. It is the least robotic-centered layer of *Miro*. Important infrastructure for the programming environment has been provided and the applied technology has been configured and adapted for the use in this application domain.

Communication is a central issue in a distributed environment. Sophisticated solutions for distributed computing exist, but the feature set provided by these solutions needs to be carefully assessed for its applicability within the robotics domain. The related work discussed in Section 2.4 provided multiple examples for the use of CORBA. But these projects differ in its configuration as well as in their use of the provided CORBA services. Furthermore, different philosophies about the visibility of CORBA technology to the application developer exist. The mapping of CORBA IDL to C++ does not represent the state of the art in C++ design and could be vastly improved [117]. Nevertheless, it defines a well documented API with consistent semantics. Therefore the infrastructure layer discussed here does not provide wrappers. They can shield the application developer from some of the complexity, but they can also limit him in his possibilities for using the infrastructure. Instead, we concentrated on a good default configuration and a set of helper classes that facilitate the initialization process.

Configurability is an important aspect of large-scale application design. Component frameworks are developed to manage the entire lifecycle of the different parts of an application such as the instantiation of components and the initialization of communication connections between components. However, *Miro* does not make use of a full featured component architecture. Implementations of the CORBA component model (CCM) [115] were not available when the project was started and available component architectures like Enterprise Java Beans [10] did not provide the required generality (i.e. programming language transparency). Nevertheless, *Miro* incorporates many aspects of component-based application design. While the idea of interchangeable components is appealing, it is an open question how sensor and actuator devices can be defined

abstract and yet precise enough to be interchangeably used by robotics algorithms such as obstacle avoidance or environment mapping. From a low-level perspective, parameterization is a key factor for the flexible reuse of implemented components. Therefore, the software architecture provides an infrastructure for efficient, flexible and extensible parameter management.

In this chapter the infrastructure provided by *Miro* will be introduced. The communication and configuration facilities are discussed in detail. In addition, important third-party toolkits and their provided functionalities will be briefly sketched.

4.1 Communication Infrastructure

Robotic applications are inherently distributed. While this is fairly obvious for multirobot scenarios, a single robot also often contains multiple on-board computers to increase processing power, or parts of the computing may be sourced out to workstation computers. Also, in the age of Bluetooth, WLAN, and wireless ad hoc networking, users expect to communicate with a robot through their personal digital assistant (PDA) or mobile phone. This makes the appropriation of scalable, network-transparent communication facilities a central issue. For this purpose, a type-safe high-level protocol is necessary to keep the communicated data easily accessible and prevent subtle programming errors that are hard to trace through a heterogeneous distributed environment. It also facilitates adjustments to the communicated data structures as needed during development.

Early robot control architectures introduced lightweight but proprietary and very poorly scalable messaging protocols like TCX [30], or IPC [120]. Additionally theses have failed badly in providing compile time type checking and runtime type safety. Therefore, instead of developing proprietary solutions, the design of *Miro* is based on existing distributed systems technology. Distributed middleware is an active research topic of a large scientific community and multiple powerful solutions for this domain exist. Distributedness is a central aspect of the application domain. So it is doubtful that a proprietary solution would actually allow to do without much of the structural overhead introduced by a full featured middleware in the long run. *Miro* uses distributed object computing (DOC) middleware, namely the Common Object Request Broker Architecture (CORBA) as part of a robot software architecture. Therefore, important concepts of distributed systems middleware and different available DOC technologies will be discussed in this section.

4.1.1 Distributed Systems Technology

A DOC middleware is usually divided into two parts: firstly the technical framework that provides the communications infrastructure, and secondly a set of ser-

vices that is implemented using the technical infrastructure and that supports basic generalized tasks needed for making effective use of the infrastructure.

Most state-of-the-art middleware provides an object-request-broker-based (ORB) communication model and adheres to the stub/skeleton paradigm. ORB-style middleware allows to invoke methods on an object across the network in a client/server model. For this purpose, the stub, a proxy object, resides in the client's address space. It offers the same interface as the remote server. The client invokes the operations of the stub and the stub handles the actual communication with the remote server. The skeleton handles the communication on the server side and invokes the corresponding method on the remote object. The matched pair stub/skeleton is automatically generated at compile time or at runtime by the middleware technology used [15].

Various middleware technologies currently exist. The most important ones today are probably CORBA, Sun's Java/RMI and Microsoft .NET. These middleware technologies offer a similar set of functionality but differ in the types of scenarios they are aimed at.

Java/RMI is heavily centered on the Java programming language and therefore lacks interoperability. The Microsoft .NET technology is very new. It does not yet address issues of application environments with real-time constraints. Also, the XML-based representation of communicated data produces undesirable bandwidth and runtime overhead. Therefore, CORBA was selected as the DOC technology for the *Miro* project. For an extensive review and comparison of the various distributed systems technologies and their application in robotics, please refer to [15, 72]

4.1.2 Service Interfaces

To overcome location and programming language dependencies, all of *Miro*'s interfaces for sensor and actuator services have been designed as network transparent CORBA objects, which can be accessed in any language and from any platform for which language bindings are defined and an ORB implementation is available. This enables the seamless integration of high-level robot control subsystems like Lisp-based planners or Java-based user interfaces as demonstrated in [35]. This in itself solves the basic problem of type-safe network transparency and language interoperability. But the setup and configuration of the functionalities and the services provided by the DOC middleware are crucial for ensuring scalability and maintainability. For this reason, *Miro* provides a customized model for using the communications infrastructure provided by CORBA in teams of autonomous mobile robots that need to communicate via unreliable wireless networks.

Pull Model

The communication primitives used for the interface design of services also implies an intended processing model on the client side. The pull model is the most common way of object interaction. The client side invokes a method on an object, performing an operation such as setting a value in the remote object.

Push Model

The classical pull model does not scale well anywhere in large-scale mobile robot applications. This is especially true for the processing of sensory information in a time-constrained, dynamic environment. Sensors usually produce measurements either on demand (bumper) or at some fixed or maximum rate (LRF, IR). In order to avoid missing a sensor reading, the consumer is required to either continuously poll the sensor or to wait for the next reading. When fusing information from multiple sensors, this can either lead to excessive polling or implies multiple threads and frequent context switching. Furthermore, in a well-behaved mobile robot a bump sensor should hardly ever be pressed. Nevertheless, for safety reasons no bump situation must be missed. Sometimes the interest lies not so much on the actual sensor reading, but on a particular event like the bumper being pressed. Therefore, in addition to the method-call-oriented interfaces, *Miro* also uses the push model in the form of an event-driven communication paradigm as a central communication method.

4.1.3 CORBA Communications Technology

The "Common Object Request Broker Architecture" (CORBA) is an open standard, developed and maintained by the "Object Management Group" (OMG) [46]. Several commercial and open source implementations of the standard are available [111, 14]. Additional specifications address the special needs of specific application domains such as resource-constrained embedded systems (Minimum CORBA [44]) and real-time-constrained environments (RT-CORBA [113]). CORBA was selected as the DOC middleware of *Miro* because of its very general design, the quality of the available implementations, and because of its high level of configurability for the needs of different application domains, A good technical communication framework provides network-transparent, type-safe communication within user applications, while making the distributedness as transparent as possible for the application developers. In CORBA, this is achieved through a set of different entities that will be briefly sketched in the following paragraphs. An extensive coverage of CORBA can for instance be found in [97, 50].

The Object Request Broker (ORB) defines the abstract client/server communications environment. Every application constructs a local reference (proxy object) of the ORB to enable communication. The different ORB proxy objects communicate with each other using a common Inter Orb Protocol (IOP).

The default protocol used is the TCP/IP-based Internet Inter Orb Protocol (IIOP). Other protocols can be used to exploit optimization opportunities like the colocation of two CORBA objects on the same machine. A servant is a remote object that is defined by its interface. Interfaces consist of a set of methods with arbitrarily structured parameters. Different object instances could also be represented by different servant implementations that support the same interface. A network-transparent object reference is denoted by an Interoperable Object Reference (IOR). An IOR is therefore a unique identifier that enables a client to address the remote object. User data, i.e. the parameters of the interface methods, have to be serialized when transferred to a remote server via a network and deserialized before the invocation of the servant. These tasks are called marshaling and demarshaling in CORBA terminology. The standardized representation of user data is called the Common Data Representation (CDR). The serialized data results in a CDR stream.

In order to obtain interfaces that are programming language independent, type-safe and network-transparent, the application programmer defines the interfaces in the CORBA Interface Definition Language (IDL). An IDL compiler then translates these definitions into the target programming language. Standardized mappings for various programming languages do exist, enabling e.g. Java-based clients to call Ada-based servers. On the client side, so called stub code is generated. This code implements proxy objects that expose the same interface as the remote object, but merely marshal the parameters and transfer the request through the ORB to the actual servant. Proxy objects are initialized by being provided with the IOR of the remote servant. On the server side the IDL compiler generates so-called skeletons. In object-oriented language mapping, these are virtual base classes that are invoked on behalf of a request. The ORB hands the request to the skeleton code that performs the demarshaling of the parameters and invokes the actual interface methods of the servant. A servant implementation is created through a simple derivation from the pure virtual base class generated from its IDL-based definition and it implements the defined interface methods.

4.1.4 CORBA Services

Apart from the communications infrastructure a communication middleware has to provide a set of basic services in order to allow the applications to make efficient use of the provided capabilites. The CORBA standard defines the interfaces and semantics of a broad set of services that can be provided by a CORBA implementation such as resolving objects by name, clock synchronization or services for event-based communication. The two most relevant CORBA services for this thesis are the naming service and the notification service, because they are extensively used in *Miro*.

Naming Service

The CORBA naming service [43] allows to associate an IOR with a plain text name, allowing clients to resolve remote objects by name, much like the DNS service that maps domain names (www.uni-ulm.de) to numerical IP addresses (134.60.1.2). It allows to define sub-partitioning by the use of so called naming contexts, much like subdirectories in a file system.

When a servant is created, the ORB generates a unique IOR for the object. As the IOR encodes all information necessary to access the object by a remote client, the IOR of the same interface implementation running on a different machine (with a different network address) will differ. If the server registers the IORs of objects in the naming service under a plain text name, clients can resolve the objects' IORs by the names agreed upon. Only the IOR of the naming service itself needs to be known by the client on startup. This can be held consistent over multiple incarnations of the naming service as long as it is started on the same machine.

As the use of the naming service is mostly restricted to program startup and network latencies are usually not critical at this early stage of the program's execution, *Miro* by default uses one central naming service instance for all robots of a multirobot team. Thus, client applications can conveniently resolve the IORs for all the robots' interfaces at a central service. To easily distinguish the interfaces of different robots in the naming service, each robot registers its interfaces within its own naming context. The naming context by convention denotes the name of the robot.

Notification Service

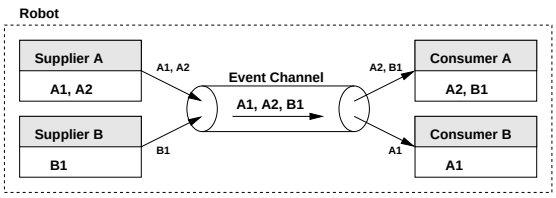

Figure 4.1: Sample sketch of the Notification Service.

The CORBA notification service [41] specifies a variant of the publisher/subscriber architecture. Publishers (suppliers in its terminology) offer events. The so-called consumers subscribe to those events. They then receive the events generated by the supplier through the event channel (see illustration in Figure 4.1). This standardized specification of a real-time event channel is part of various CORBA implementations [111, 14]. It is a rather weighty protocol, but it not only supports type checking through the use of IDL-defined data structures and the type-safety of the CORBA type code system, it also supports the specifications of QoS requirements like event timeouts or asynchronous message processing. It can also be extended to meet end-to-end quality of service requirements such as OS thread priorities to meet the requirements of the RT-CORBA specification [40].

Miro uses the notification service for providing the push model of sensor-driven processing. The general idea is that on a robot, sensor data like odometry values or distance measurements is sampled at discrete time steps. The sensor services publish these values through an event channel, allowing for sensor-driven processing in the system. Higher sensory or cognitive processes can also publish their results this way in order to distribute them to other subsystems.

4.1.5 Group Communication

Basic support for multirobot control naturally comes with a distributed robot software development environment as provided by *Miro*. Small groups of robots can address each other by exchanging the object references if their respective sensor/actuator configurations are known. Also,sharing of sensor data can be achieved through filtered event processing frameworks-based upon the notification service. However, today's wireless networks suffer from severe deficiencies like heavily varying link quality, latency and temporary breakdowns that create problems when used with CORBA, which uses a connection-oriented (usually TCP/IP based) transport layer by default, especially in conjunction with the publisher/subscriber protocol. Therefore, additional support to customize the notification service for robot group communication was designed and implemented in the context of *Miro* in order to enhance the applicability and scalability of DOC technology [141].

For this purpose, a service federation quite similar to the one described in [49] (for the proprietary "RT Event Channel" of TAO) was used. Similar to their work, an event channel instance is run locally on each robot and only the events that are subscribed to by consumers of other robots are transferred to the respective robot's local event channel instances. This message exchange is done by *Miro*'s so-called Notify-Multicast module (NMC). NMC extends the above sketched design by properties to automatically derive the minimal set of events that need to be exchanged between the robots in the group at runtime. NMC builds upon a connectionless, message-based protocol that is transferred by IP multicast. This way, network breakdowns and latencies do not block the sending robot. In consequence, events can become corrupted and get lost. But the various filter rules (such as event time-outs) applicable to the service's event channels also mandate to design the communication through an event channel in the form of unrelated, atomic messages. So as messages do not depend on each other, events from other teammates can be integrated into a robot's data processing as soon as the network becomes available again. The multicast technology used also significantly reduces the required communication bandwidth, since this way every message only needs to be broadcast once instead of being sent n times for n remote consumers.

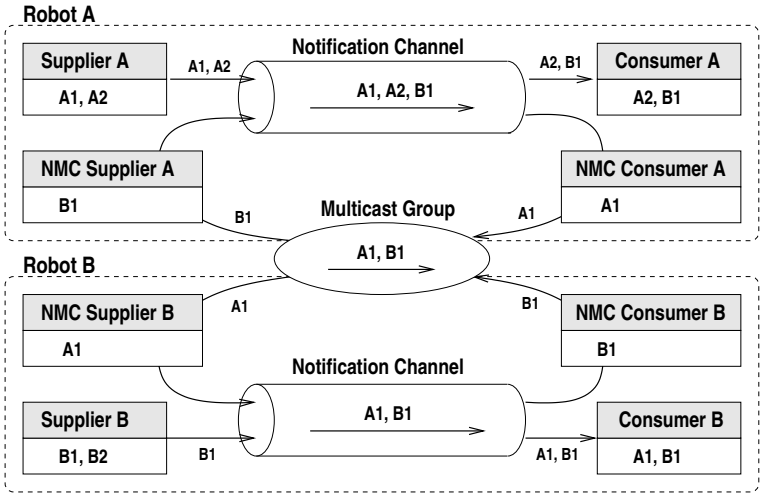

Figure 4.2: A federated notification channel setup. All events that are subscribed by remote consumers are exchanged between the event channels.

The Set of Exchanged Events

A "NMC event consumer" subscribes for all events that are offered only locally but subscribed by other team mates and sends them to the multicast group. An "NMC event supplier", in turn, listens to all events published via IP-multicast and pushes those into the local event channel that are subscribed but not offered locally. To keep track of the offered and subscribed message types and to dynamically adjust the set of events that need to be exchanged via the multicast group, NMC utilizes two fields of the standard structured event message format: the domain name and the type name. By its convention, a message's domain name contains the name of the robot producing the event. The type name describes the payload of the event. These fields are also part of the native offer/subscription management and filtering protocol of the notification service, so robots can easily determine whether events they offer are currently subscribed to in the team and skip their production entirely if there are no subscribers. To determine at runtime the minimum set of events that need to be exchanged via the multi-cast group, each NMC module periodically (by default every 5 seconds) posts the set of requested events as well as its available offers to the group. The minimum set can then easily be calculated by simple set operations.

Figure 4.2 illustrates a sample configuration of the notification channel setup. Two robots (A, B) produce two types of events $(1, 2)$, the resulting events are $\{A1, A2, B1, B2\}$. The events in the supplier and consumer boxes denote the offered and subscribed events. The events labeling the arrows denote the actual flow of events. Note that suppliers and consumers can offer or subscribe for multiple events.

4.2 Configuration and Parameter Management

Robotic applications need to be flexible, reconfigurable and adaptable to different scenarios, robot platforms etc. The reuse of modules and components in large-scale software applications usually requires the adaptation and adjustment of a large set of parameters in order to successfully adjust the configuration, the behavior, and the performance of the autonomous mobile system to the new target environment. There are several sources of configuration parameters in robot control programs and robotic algorithms

- Different robot types vary in shape and sensor configuration, but also different robots of the same type tend to vary slightly. Some robots, for instance, are equipped with further sensor or actuator devices not present in the standard configuration. Also, damage and repair of a robot's parts during its lifetime result in an individual robot which has its own unique configuration.
- Furthermore, the environment provides a complete set of parameters: lighting conditions, the shape and location of rooms and corridors, the positions of obstacles like tables or stairs. The list could be continued indefinitely. These parameters vary between different environments, but also tend to change slightly over time. Light bulbs are changed, cupboards added, tables moved etc.
- A third group of parameters is defined by the task the robot has to perform or by the scenario the robot is designed to operate in. Generalized tool boxes like planners or knowledge bases need to be populated with the relevant actions and objects, and a reactive execution engine needs to be configured with information on what actions to take in which situation.

These sets of parameters can either be configured by hand, with the help of tool-supported methods, or fully autonomously. Nevertheless, configuration remains a crucial part of a robot's setup for every scenario. Distributing these configuration parameters over a large-scale application in an unstructured manner or hard-coding them as constant values within the various modules can endanger the maintainability and adaptability of the resulting applications.

4.2.1 Related Work

Most large-scale software development environments offer facilities for configuration parameter management. But most of them rely on a simple ini-file based format that only provides simple key-value mapping and a partitioning of the configuration file into multiple sections (e.g. the KDE project, OROCOS@FAW [105]). Typically, only helper classes for parameter retrieval are provided.

The Windows registry allows to define subtrees, keys, subkeys and typed entries for storing configuration data for both the system and installed applications. On

the other hand, structured or user-defined parameter types are not supported [84].

Especially Java-based applications also use XML-based configuration syntax [85]. The ORCA project [12] also uses an XML-based description language to specify the configuration of their components. Nevertheless, the parsing of the individual parameter values at program startup is supported by helper classes, but not otherwise automated.

Generally, hardly any stringent support for the editing of parameter values by end-users exists. The `regedit` for the Windows registry, for example, cannot provide its users with the names for subkeys that are processed by the application, but currently not present in the registry. This functionality is also missing in most robotic software environments. In consequence, the manual for the Carnegie Mellon Robot Navigation Toolkit (CARMEN), for instance, explicitly states that the provided list documenting the available parameters is probably not complete [6].

Interestingly, the KDE project recently incorporated a quite similar approach to parameter management as discussed in the following section [100]. But it does not provide extensibility by user-defined types as provided by the approach discussed here.

4.2.2 Design Constraints

Providing dedicated support for configuration management not only facilitates the adaptability of the software architecture itself. A convenient, structured approach of parameter handling can also encourage developers to model the parameters of modules explicitly and that way improve the maintainability and reusability of the resulting applications.

In a time-constrained environment such as robotics, parameter management does not have to introduce significant runtime overhead compared to hard-coded implementations in order to be acceptable on all levels of an application. The parameters are not unstructured sets, but structured according to the semantics of the module's domain. For the management of the configuration options, this is valuable information and therefore needs to be representable.

Because of their nature, configuration parameters need to be altered frequently and therefore need to be easily accessible. On the down side, this can also endanger the stability of the deployed robot, because the wrong configuration can easily crash an application. Thus, ensuring the consistency and the syntactic correctness of configuration files is necessary in order not to introduce a new source of bugs which might be difficult to trace.

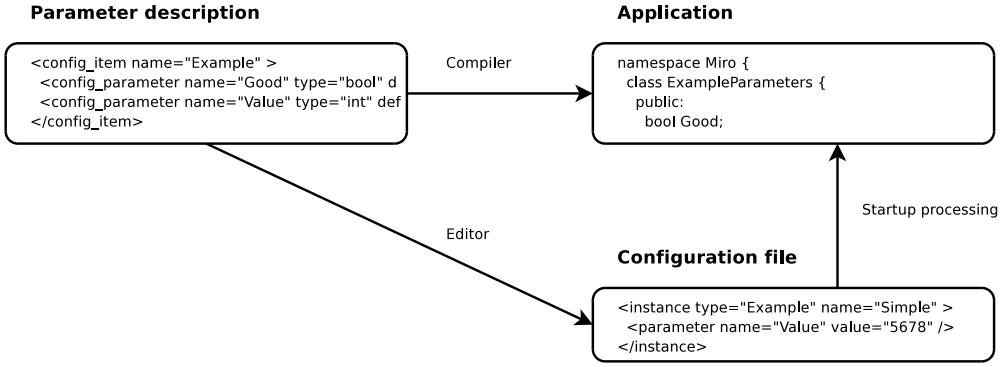

Figure 4.3: Toolkit-supported configuration and parameter management.

4.2.3 Parameter Descriptions

In order to enhance the configurability and maintainability of robotics software, a sophisticated infrastructure for parameter and configuration management was developed. It is sketched in Figure 4.3. With the *Miro* parameter toolkit, the modules' configuration parameters are separated into so-called parameter description files in an XML-based syntax. These allow describing parameter sets as arbitrarily complex structures that support aggregation, nesting, data types of variable lengths and single inheritance. A parameter description can specify properties such as its default value, the represented measure, or a verbatim description of the parameter.

Miro provides a compiler for translating these descriptions into a target programming language's data structures and code for storing and parsing this data to and from configuration files. Currently, C++ is supported. A configuration file contains various instances of parameter sets as described in the parameter description files. It is based on an XML syntax, too. The application parses those files on startup to initialize the parameter data structures of its services.

The basic syntactic correctness of configuration files can be ensured by validating it with its document type definition (DTD). However, due to the extensible nature of parameter descriptions, it cannot ensure the correctness of the mapping of a parameter set in the configuration file to its specification in the corresponding parameter description. Thus, for generic type-safe editing of configuration files, the GUI-based `ConfigEditor` has been provided. This editor will be covered in more detail in Section 7.1. Figure 4.4 shows the dialog provided by the editor for the parameter example outlined in Figure 4.3. The editor dynamically generates this dialog from the parameter description at runtime and edits the configuration file example from Figure 4.3.

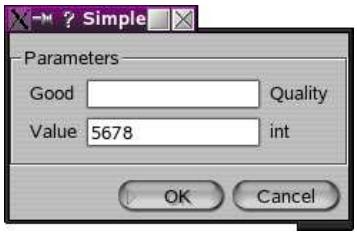

Figure 4.4: The corresponding generic dialog.

4.3 Cross Platform Toolkits

The primary implementation language of *Miro* is C++. But the power of modern programming languages is more and more defined by the available application programmer interfaces (APIs) that are provided by libraries for the language in question. Apart from the C++ standard library, various powerful libraries exist that contain APIs and frameworks for specific application domains. *Miro* makes extensive use of multi-platform libraries in order to ease portability.

4.3.1 Adaptive Communications Environment (ACE)

The Adaptive Communications Environment (ACE) is a multi-platform library available for Linux, most Unixes, Windows and some real-time operating systems. This powerful toolkit encompasses various sets of functionality targeted for network programming:

- the OS abstraction layer forms a common, C-oriented API that hides most of the inconsistencies between the different operating systems. For instance, simple methods like a query for the current system time are denoted by different names and return different data formats on the various operating systems.

- Upon this OS abstraction a set of wrapper facades is built. E.g. the socket wrapper facades form an object-oriented, strictly typed interface to the ancient C socket APIs, which usually use untyped handles for referencing the various types of connections. Another important set of wrapper facades encompass multi-threading and synchronization.

- Upon these wrappers, frameworks for scalable communications infrastructures such as the Reactor framework for event demultiplexing or the service manager framework for component-oriented service deployment are built.

While the ACE toolkit provides various APIs for basic network programming, it lacks the higher-level functionalities for type-safe, object-oriented network transparency. Additionally, as it is targeted as a multi-platform toolkit, it contains various APIs, e.g. container classes that are provided by reasonably standard-conforming C++ compilers.

Miro makes intensive use of the Reactor framework as well as the multi-threading and synchronization primitives. Also, the socket wrapper facades are used within the low-level device framework.

4.3.2 The ACE ORB (TAO)

The TAO package is an implementation of the CORBA standard based on ACE [111]. It therefore runs on almost any platform that has a reasonably complete port for ACE. This way, while delegating almost all of the platform dependency in software development to ACE, it proves the reliability and power of this toolkit. The focus of this CORBA implementation lies on scalability and applicability to real-time applications. The tradeoff between performance and scalability is deferred to the runtime configuration of various components of the architecture.

TAO comes with a rich set of implementations of CORBA services such as a naming service, various event services, a service repository, and many more. Apart from the sheer mass of APIs and services, the flexibility of configurations and options requires a fair amount of knowledge in order to provide a consistent selection of functionalities that would form the ideal solution for the problem at hand.

4.3.3 Qt

Qt is a graphical user interface application framework available for various Unix derivates including Linux as well as Windows. It is actually a commercially developed library that is also available under an open source license (GPL). Qt is used within *Miro* for the construction of GUI-based applications such as monitoring and visualization tools.

Additionally, Qt provides two standard APIs for processing XML files (Sax2 and Dom). In order to avoid further dependencies, *Miro* uses these APIs for XML processing instead of introducing another dependency on a pure XML processing library like libxml. Note that ACE also offers an XML module. But it only supports the Sax2 API which is limited to the parsing of XML files.

4.4 Summary

In this chapter, the implementation infrastructure of *Miro* has been discussed. It contributes to the state of the art in research on autonomous mobile robotics a carefully designed configuration and customization of standard DOC middleware that provides an efficient and scalable communications infrastructure for teams of autonomous mobile robots.

First, an overview of the distributed object computing middleware and its configuration and application was given. The use of CORBA technology allows to model aspects of the distributedness on a very high level of abstraction. It bases the communications infrastructure upon an open, proven and well-documented standard, which provides interoperability over various implementations and different programming languages.

Following these elaborations, the necessity to customizes its provided functionality to meet the requirements of autonomous mobile robots middleware was demonstrated by the notify multicast module for group communication. The design of the discussed NMC module successfully addresses scalability issues in this context. Providing a resource-efficient and transparent model for group communication further contributes to the solution of problems imposed by the inherent distributedness of this application domain.

Afterwards, a toolkit for managing the consistent and structured handling of configuration parameters in large-scale robotics applications was discussed. This toolkit provides a solution approach for the inevitable dependence of robotic software on external parameters without jeopardizing the reliability of the resulting application. By providing a consistent, tool-oriented model for its application, it vastly improves the maintainability of extensive configuration management necessary in this application domain.

At the end of this chapter, the open source third party libraries used within *Miro*, which significantly contribute to the infrastructure provided, were briefly introduced. The consistent use of cross-platform toolkits for the required third-party-provided infrastructure facilitates *Miro*'s portability.

Chapter 5

Miro Services

Sensors and actuators can be naturally modeled as objects, which can be controlled and queried by their respective methods. Thus, mobile robot software can be viewed as aggregations of sensory, actuatory and cognitive objects, which are able to trade information and services in an agent-like manner. The *Miro* Service Layer, provides object-oriented interfaces for all hardware components of the supported robot platforms [140].

The definition of the term "service" is rather vague. In [112], for instance, Douglas Schmidt defines a service as a "set of functionality, offered to a client by a server". This definition denotes the client/server-based design of services, which implies a modular decoupling of functionalities but specifies little more. Therefore, in this thesis the term service is used according to a stricter definition, which is actually much closer to a component-based approach. A service in the context of the *Miro* service layer implies the following:

IDL defined interfaces. The available functionality of each service is accessible by strictly typed, network-transparent, object-oriented interfaces. Compile time type safety is an important issue, because it facilitates the correct interoperability of modules within a large-scale system.

Uniform semantics. The semantics of the functionality provided by the service's interfaces need to be precisely defined. This allows to provide multiple implementations for a service with different robot platforms that expose a uniform set of functionality to client applications. For instance, a different locomotion system will require different motor speeds to produce a specific motion trajectory of the robot. This, in turn, will result in individual implementations of the service. Nevertheless, the semantics of the supported abstract interface will produce the same trajectory on different mobile platforms (within the boundaries of their maneuverability) for the same set of parameters.

Generalizations of robotics devices. The purpose of the service layer is to make devices of different physical robot platforms accessible in a uniform

manner. This requires the generalization of similar functionalities that also allows to designate their differences explicitly.

The design goal of the service layer is to provide abstractions for robotic devices such as sensors and actuators that allow roboticists to develop generalized algorithmic solutions on the client side written against abstract service specifications instead of a set of concrete physical devices.

In this chapter, *Miro*'s service layer will be covered in detail. First, a design for actuator services allowing for generalization as well as specialization of client-side access will be discussed. This design will be exemplified in a discussion of the various locomotion systems of mobile robots and their respective mapping to the `Motion` interfaces family. Second, we will take a closer look at the design of sensor services and their interfaces, in particular at the `RangeSensor` interface. It defines interface functionality common to sensors like infrareds, sonars, and laser range finders. In Section 5.4, the logging service will be introduced as an example of generic middleware services.

5.1 Actuator Services

Actuators are devices the robot can use in order to act within its environment. While a mobile robot's central actuator doubtlessly represents the mobile base itself, various other actuators are also frequently used on autonomous mobile robots. Simple grippers are often available, while more sophisticated robot arms are less common, as their control consumes more battery and computational power. Other actuators are used for active sensing tasks. Examples would be pan, tilt and pan-tilt units, which move around a directed camera or other sensors.

The purpose of actuator services is to control the actuator device and provide an interface for client applications, through which the actuator's offered functionality can be accessed. This duty especially includes the interaction with the device layer, which takes care of the communication with the low-level devices and microcontrollers that control the actuator. This way, the individual difficulties and low-level details are shielded from the application programmer, who merely uses the services' interfaces to access the actuator's functionality. In the discussion of actuator services we will concentrate on the interfaces publicly exposed to their clients.

5.1.1 Design Principles

Robot actuators, even actuators belonging to a simple category such as the robot's drive, expose an extreme heterogeneity of physical properties and designs. This results in different possible forces and velocities, differences in maneuverability and in the available control modes such as setting the target velocity versus setting the target heading of a pan unit. The challenge in

the design of actuator services is therefore to support and encourage reuse of client applications by providing uniform actuator interfaces, while at the same time offering full access to the unique feature set of the individual device. Furthermore, the bounds and limitations of individual robot devices need to be accessible in order to allow for an adaptation of client applications to properties such as the maximum permissible speed of a motor. Two design principles have been identified to address these diverging goals of the interfaces defined in the *Miro* service layer: class hierarchies and metainformation.

Class Hierarchies

Class hierarchies are used to specify generalized interfaces in the parent classes, while offering access to the advanced features of the individual actuators in the methods of the specialized, derived children. This way the generalized interface allows to access similar devices without changing the client implementation. Parallel to this, specially adopted algorithms can take advantage of the actuator's additional functionality and features.

Abstractions often make it necessary to idealize the physical properties of the devices, as the generalization does not allow to model its characteristics precisely enough. On the other hand, this simplified view of the device typically facilitates its use by the application programmer. Thus, interface hierarchies not only allow to trade generalized applicability of client applications for a potentially more powerful specialized solution, they also allow to trade a precise model for simplicity.

Metainformation

While generalizations of the service layer offer a unified interface to access different instances of an actuator family, they also need to provide information on the individual properties of the interfaced device on the corresponding abstraction level. Metainterfaces provide information about the varying properties of an instance of the actuator class such as acceleration values and maximum velocities. These can be used on the client side to adapt the parameterization of algorithms such as for instance the safety distance of a collision avoidance module to the actual physical robot.

This information also defines the permissible parameter ranges for the provided interface methods of an actuator service. Client requests (e.g. for motor speeds) that lie outside the specified ranges result in an exception to be thrown by the service.

5.1.2 Requested Functionality

In order to promote a uniform view of actuator services and their offered functionality, three types of methods are requested from each service interface.

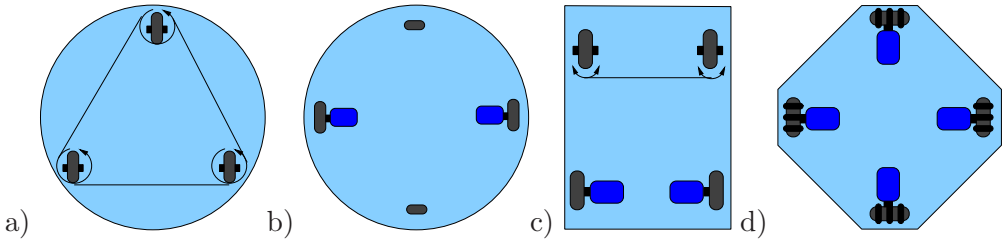

Figure 5.1: Different locomotion systems of autonomous mobile robots: a) Synchro Drive, b) Differential Drive, c) Ackerman Steering, d) Omni Drive

Set actuator. As physical devices cannot change their state instantaneously, a 'set' method actually denotes a new setpoint for the controller in charge of the actuator.

Query service state. Actuators have a physical state, such as the velocity the robot is intended to move at. This has to be reflected by designing an actuator service as a stateful service. Note that, as the control of an actuator is usually delegated to some low-level controller board or even to a part of the actuator device (e.g. a servo), the service's state is not the physical state of the actuator but actually represents the current actuator setpoints. Information reflecting the physical state of an actuator such as a robot's odometry has therefore been modeled as a sensor service.

Query service metainformation. Metainformation such as the permissible actuator values needs to be accessible to the client in order to correctly adapt itself to the devices properties.

5.1.3 Actuator Service Example: Motion

The design principles defined in the last section help to obtain a consistent set of functionality provided by actuator services. To illustrate the different design aspects, a concrete actuator service will now be discussed in detail. A motion service of *Miro* provides access to the central actuator of a mobile robot, the locomotion system.

Several different physical drives are available for the various mobile platforms. The synchro drive consists of three or more wheels, which are uniformly controlled in their speed and can be rotated synchronously around their own axis (Figure 5.1a). The locomotion properties of differential drive robots are similar, although their design is different. A differential drive robot has two fixed, individually controllable wheels as well as one or more freely moving caster wheels. Like a tracked vehicle, a differential drive robot drives forward and backward and can turn by allocating different speeds to each of the two wheels (see Figure 5.1b). Car-like "Ackerman" steering as illustrated in Figure 5.1c is not very popular for indoor robots, as it does not allow the robot to turn on the spot. Omnidirectional motion is frequently used by RoboCup midsize-league

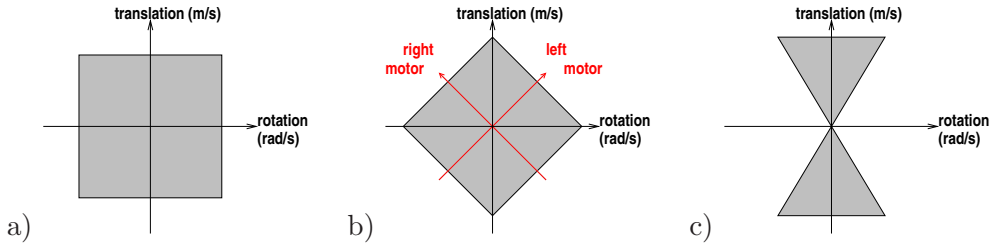

Figure 5.2: Permissible velocities for different locomotion systems: a) Synchro Drive, b) Differential Drive, c) Ackerman Steering

soccer robots. These robots can move in any direction and turn, giving the full 3 degrees of freedom (see Figure 5.1d). The special wheels have hardly any sideward traction. Therefore, driving upward with the left and right motors and to the right with the upper and lower ones, for instance, results in a diagonal movement of the robot. The capabilities of the different drives result in different applicable velocities. Figure 5.2 illustrates the varying coverage of the velocity space for the first three locomotion types. The x-axis denotes the translational velocity in forward direction. The y-axis denotes the rotational velocity. The red lines in Figure 5.2b denote the speeds of the different motors.

Interface Definition

The different locomotion systems described above need to be accessible through an appropriate set of interfaces. The resulting class hierarchy of *Miro*'s different locomotion interfaces is illustrated in Figure 5.3. The `Motion` interface forms the base class of the hierarchy. It provides methods to set the translational and rotational velocities (`setVelocity`), query the current desired translational and rotational velocities (`getVelocity`) and query the minimum and maximum translational and rotational velocities of the drive (`getMinMaxVelocity`). This interface represents a compromise. On the one hand side, it deprives the omni-drive-based platform of its most interesting feature, i.e. the ability to move sideways. On the other hand, it does not allow to model the fact that the steering-wheels-based drive cannot turn in place. Therefore, the service definition also documents how the implementations for different locomotion systems have to react to velocity commands that are permissible for the bounds reported by the metainformation query, but are nevertheless unachievable for the actual locomotion system.

The derived classes `SynchroMotion`, `DifferentialMotion`, `AckermanMotion` (which was contributed by Daniel Krüger [71]) and `OmniMotion` define additional methods for their respective kind of drive. Instances of such methods are setting the velocities for the individual wheels for a differential drive robot or setting the velocities for all axes of an omni-directional drive. Further entries, such as corresponding query methods for these methods have been omitted in the UML diagram for reasons of brevity. The specialization `OmniMotion` has

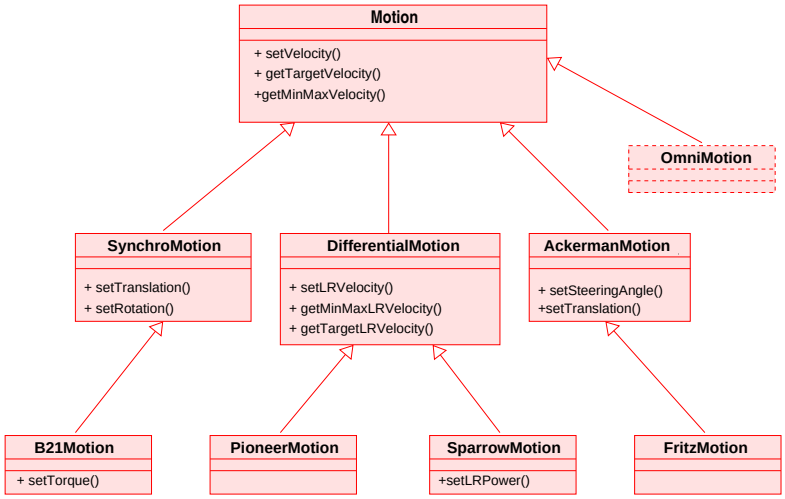

Figure 5.3: Motion interface inheritance for different mobile bases.

not been implemented yet, as at the time of writing, no robot platform with such locomotion systems is supported by the publicly available *Miro* sources (even though a port for University of Graz's omni-drive soccer robots does exist [126]).

The individual robot platforms usually provide further specializations of these interfaces in order to provide access to special features of their drives to their clients. Thus, the B21 robot provides the `B21Motion`, the Pioneer and PeopleBot platforms provide the `PioneerMotion` interface, and robots of the Sparrows family provide the `SparrowMotion`. `FritzMotion` is a specialization for a robot custom-built at the Technical University of Chemnitz. The `Sparrow99`, for instance, allows to run the robot in a wall-following mode based on infrared sensors all embedded on its low-level controller board. The application would then only set the motor power value to select the speed and choose between forward and backward translation. While this admittedly does represent a very non-portable functionality, it was extensively used to implement a very reliable goalkeeper for the The Ulm Sparrows RoboCup team.

5.1.4 Service Implementation

As mentioned above, the service interfaces are defined in CORBA IDL. From this definition, the IDL compiler generates stub and skeleton code. To ease the service implementation initially, the IDL-compiler can also generate boiler-plate files that contain empty servant implementations. These have to be populated by the service programmer.

Interface Semantics

Procedural programming languages make it difficult to ensure that the implementation of an interface definition adheres to the intended semantics. For actuator services, support for the developer of a service in this respect is provided by mix-in classes. These provide partial implementations of the service interfaces. Partial actuator service implementations provide member-variable layouts and locking primitives that state an intended design for the complete service implementation. Additionally, they provide helper methods for bounds checking of client-provided parameters' values and usually implement the query as well as the metainterface methods. The `MotionImpl` class, for example, provides an implementation structure for derived motion interfaces. `DifferentialMotionImpl` is also derived from this partial implementation and provides similar functionality for the additional methods of the `DifferentialMotion` interface and helper methods for converting the different velocity specifications translation/rotation vs. left wheel/right wheel. This way, these helper classes make it easier to perpetuate uniform semantics of the motion service interfaces across the implementations for different platforms. Note that use of these default implementations is not mandatory, if the peculiarities of a concrete device violate the assumptions taken by the intended design.

Service Configuration

Service implementations need to be parameterized to reflect the heterogeneity of robot platforms. The robots of the pioneer series, for example, all feature a differential drive and are also controllable by the same low-level controller protocols (PSOS and P2OS). On the other hand, they reach very different maximum velocities and accelerations. Therefore, the *Miro* services exploit the parameter handling infrastructure described in section 4.2. Each service interface definition is accompanied by a corresponding description of its parameter set. The parameter description of the `Motion` interface, for instance, holds the maximum translational and rotational velocities. This way, the permissible velocities can easily be defined for different robot types or adjusted within a configuration file if, for instance, additional payload on the robot such as a laser scanner reduces its permissible maximum velocity.

5.1.5 Other Services, Interfaces and Possible Extensions

Other actuator services for *Miro*'s common autonomous mobile robot actuators adhere to the design principles and requested service functionalities discussed above. The `Gripper` interface has been provided for simple robot grippers. The `Pan` and `Tilt` interfaces form the two base classes for the `PanTilt` interface used by the individual robot platforms pan-tilt services to access the different camera pointing devices. Mobile platforms designed as robot companions usually come

with capabilities for speech synthesis. A rather specialized actuator service provided by *Miro* is the support for kicking devices used by the *Sparrow* soccer robots.

An obvious omission in the above list are multiple degrees of freedom robot arms. This omission is due to the fact that none of the robots available in the robotics lab of Ulm had been equipped with a robot arm yet. But other laboratories using *Miro* already have begun working on robot arm services and appropriate metainterfaces for describing a robot arm's properties.

5.2 Sensor Services

Sensors such as cameras, microphones, or bumpers allow a robot to collect data on its environment. They can also be used to collect data on the robot itself, such as dead reckoning sensors (also known as odometry). Sensors typically used in robotics are range sensors like laser range finders (LRF), bumpers, odometry and cameras for visual sensing. An autonomous mobile robot is normally equipped with various different kinds of sensors, which make up its sensor suite. Most of the processing power is typically used to interpret and fuse sensor data and derive relevant information for the task at hand, such as recognizing obstacles for safe navigation, land marks for localization and objects (a ball and a free area in the opponent's goal for instance), for interaction (kicking the ball into the goal). The purpose of sensor services is to control the sensory devices and to provide the sensor data to their clients for further processing.

5.2.1 Design Principles

Apart from the challenges identified for the design of actuator interfaces, sensor services need to address additional difficulties. The communication with the controller boards generally is message-oriented and therefore asynchronous. Device communication therefore implements the half synch/half async pattern [109], with the servant implementation representing the synchronous part and the device layer handling the asynchronous part. The upcalling servant thread is therefore usually blocked until the data becomes available. For sensor services this is often inadequate. It results in a large number of servant threads bound by waiting on upcoming data for a commonly used sensor. On the client side, this can also result in unacceptable latencies if data from multiple sensors needs to be acquired.

An alternative implementation possibility is the asynchronous message handling (AMH) servant implementation as offered by the TAO package [23]. This model has been especially designed for middle-tier servers that hand the incoming requests (the sensor query in this case) down to the actual processing unit (the microcontroller board) for asynchronous processing, collects the replies from the processing unit, and hands them back to the requesting clients. This design

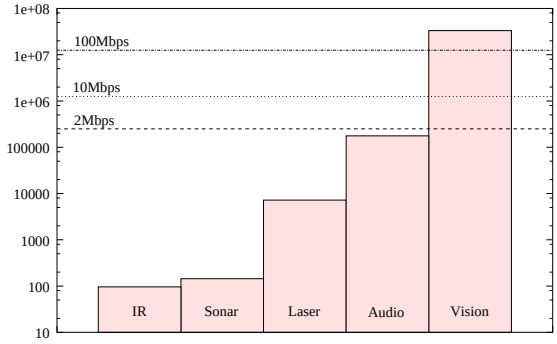

Figure 5.4: The bandwidth requirements of different sensor devices (on a logarithmic scale).

can unite the servant and the device layer processing in one event loop, as it circumvents the need for blocking calls and therefore avoids the need for multiple threads that implies latencies and jitter due to context switches. But the implementation of AMH is still in an experimental stage and does not provide colocation support.

Sensor services are instead implemented as proactive services gathering sensor data on their own initiative. The sensor data is collected proactively the device layer. This layer is usually equipped with its own thread of control to ensure quick processing of the messages exchanged with the different low-level controller boards. This is also necessary for obtaining accurate time stamps for the sensor data. The device layer pushes the data into the sensor services by a local interface method provided by the service implementation. The sensor data is then communicated to the clients by the service implementation.

Usually, sensor devices can be used to sample data at a fixed maximum rate, but the different sensory devices supply very different amounts of sensor data at different sampling rates. This results in very heterogeneous maximum bandwidth needs (see Figure 5.4), which partly lie above the bandwidths typically available. Nevertheless, choosing a uniform communication model is preferable in order to achieve consistent design of sensor services. *Miro* provides such a model with the exception of the service for video image processing. As the high bandwidth requirements of this sensor impose severe additional challenges on sensory data processing, the `Video` service actually hosts a powerful framework for real-time constrained video image processing, which is covered in Section 6.1.

Interface hierarchies are less frequent for sensor services, as the variance in the different devices less affects the interfacing to the device, but the qualitative aspects associated with the sensor reading such as its accuracy or its reliability.

5.2.2 Requested Functionality

The sensor services' different requirements result in an extended set of requested functionality in comparison to the one for actuator service interfaces. Most sensor services are stateless. If, however, internal state such as different operating modes of a sensor (e.g. normal versus differential GPS) does exist, the same requirements as for actuator services apply. A sensor service's metainformation describes the sensor configuration on the robot in order to enable flexible integration of the data.

The main purpose of a sensor service obviously is to allow access to the sensory information itself. A sensor service is requested to support three different kinds of communication patterns in order to provide clients with sensor data.

The `getSensor` method will immediately provide the last sampled sensor value and return. The `getWaitSensor` pattern actually waits on the server side until the next sensor sample becomes available. Instead of active waiting for new sensory information, the sensor-driven communications model is more adequate for sensor processing in many cases. Fusing data from multiple sensors could either require to wait in the different servants with multiple threads or result in frequent polling. Therefore, a *Miro* sensor service needs to also provide the push model for communicating sensor samples and send them through the notification service's event channel. This facilitates a clean application of the sensor-driven processing model as required frequently in time-constrained robotic applications. Client applications simply subscribe for one or more sensor streams at the event channel and then are automatically called whenever new sensor samples become available.

Sensor interfaces also provide the time stamps with the sensor data's sampling time as reported by the device layer. This is mandatory in order to allow for a precise correlation of sensor data samples (e.g. from different sensors) as needed in highly dynamic environments and for temporal integration.

5.2.3 Sensor Interface Example: RangeSensor

A good example of the additional challenges to be considered in the design of sensor services is the access to range sensor data. Such distance measurement sensors are very frequent and come in a great variety. They are normally mounted in groups in order to provide egocentric 2D-range scans. They encompass unreliable sonar, close-range infrared sensors and high-end laser range finders. But bumpers can also be regarded as supplying very primitive, binary range information. And camera images are also used to extract range sensor information, thereby providing a virtual range sensor.

All range sensor devices therefore report a set of distance readings. The `RangeSensor` service therefore organizes range sensors in scan groups, of which several are allowed on one service interface. The main differences between the devices from a performance point of view are covered in the range sensor de-

scription provided as sensor metainformation by the service interface. This description covers aspects such as minimum and maximum possible measuring range, apex angle, the mounting point on the robot and its measuring direction.

The main distinguishing points from the client interface point of view are: the number of scan points, the grouping and the frequency as well as the model of sensor data sampling. Active sensors, like the sonar sensors especially, can interfere heavily if physically neighboring sensors are fired simultaneously. Thus, the sampling sequence of these sensors is often not linear. While this information is also provided as part of the description, it has additional consequences for the interface design.

The design of a uniform service interface for range sensor devices provides an additional challenge because of a limitation of the middleware technology employed. The early versions of the CORBA specification do not allow to mix the definitions of data structures and interfaces. Inheritance especially is only supported for interface definitions. Data type definitions in IDL are therefore very similar in expressiveness as classical structs in the programming language C. This hinders the use of class hierarchies and specialization in the design of the data types that are shipped as payload of the events generated by a sensor service. On the other hand, it is desirable to only actually ship the data requested by the client, as additional data fields would just waste bandwidth. Therefore, specialized services need to offer additional event types that provide the offered information.

Interface Definition

The main design goal of the range sensor service is to provide an interface, that offers a uniform way to query for all kind of sensor scans. Like all of *Miro*'s sensor services, it is to support querying the current sensor value, a blocking query of the next sensor value and the push model, publishing range sensor data on the event channel for the subscribed consumers. In correspondence with the actuator services, the metainterface methods of the `RangeSensor` interface provide information about the sensor layout and its properties such as the minimum and maximum range measurable etc.

Different Types of Range Events In order to achieve the best possible response times, the push model has to reflect the models of data sampling of the different range sensor devices. So the range sensor service supports three different kind of range sensor events that reflect the different types of sensor data acquisition. But each service instance only produces one kind of event.

Full scan at once: The third kind of range sensor event is for reporting measurements of all scan groups at once. This consists of a vector of scan groups, each holding a vector of the successive sensor values. In contrast to a scan matrix, each scan group can differ from the others in size.

Scanning in groups: Most sensors report measurements in groups, since one microcontroller usually controls one single scan group, especially if the groups are not closely related physically. Therefore, scan group events are supported that consist of the group index and the vector of the successive sensor values.

Bunch wise scanning: Sonar sensors collect sensor data in interleaving sets to minimize cross talk. If the service waited until a full scan was completed before pushing the data to the event channel, parts of the scan would already have aged and would be less valuable. Therefore, the service can push sensor scans in bunches in order to minimize latencies in sensor processing. This event consists of a vector of sensor values, each accompanied by its scan group index and index within the group.

5.2.4 Service Implementation

Unlike with actuator services, *Miro* offers complete default implementations for sensor services such as the range sensor service.

Sensor Data from the Device Layer

Service implementations do not only offer an interface to remote clients, but also offer local methods (not defined in IDL), for easy integration of new sensory data from the device layer. The `RangeSensor` service for instance supports the different scan modes discussed above by actually offering the device layer three different methods for integrating new sensor data. The service implementations take care of race conditions between servant threads and asynchronous low level device processing by means of locking. They also dispatch the new sensory data to the event channel. This complete default implementation greatly simplifies ensuring a common semantics for services of sensor devices of the same category.

Service Configuration

The configuration parameters offered for a range sensor service instance consist of the sensor layout and sensor capabilities as queryable by the service's metainformation interface. This way, the default implementation can be configured to support any kind of range sensor by just providing the appropriate configuration data.

Derived Interfaces

The base interfaces for sensory services only export query functionality as their service interface. The service configuration is considered to be static and is initialized from the parameter and configuration management facilities at startup.

But many range sensors have additional features, that might want to be manipulated at runtime. Laser range finders often support long-range and short-range scan modes that can be switched at runtime. Sonar sensors tick during operation, which can become quite annoying. Therefore it is often desirable to disable them temporarily while they are not needed, without having to stop the entire service. Support for those additional functionalities can be given by specialized derived interfaces for individual range sensor services.

5.2.5 Other Interfaces and Possible Extensions

The design of sensory services discussed above is also reflected in *Miro*'s `Odometry` and the `Stall` sensor services. What is currently missing in the provided metainformation of sensor services is information on the sensors' error model, such as information on accuracy and reliability. While this would surely be a valuable contribution to the reusability and genericity of sensory information processing, this topic could not be fit into the scope of this thesis.

5.3 Related Work on Sensor-Actuator Interfaces

A basic requirement on robot core libraries is to provide access to the sensors and actuators of the autonomous mobile robots. Therefore, all software architectures provide interfaces to access and control these devices. Nevertheless, most existing architectures fail to provide reusable interface abstractions or provide generalizations that do not allow to precisely model the properties of specific devices for specialization. Furthermore, the notion of metainformation is mostly absent from such architectures.

Most manufacturers of commercial robot platforms provide simple libraries that provide methods for the control of the robot's basic sensors and actuators such as the locomotion system or sonar sensors [1]. Modern robot software environments as covered in Section 2.4.2 target multiple robot platforms and therefore provide more sophisticated interfaces.

TeamBots is a Java-based software environment, which is especially targeted for use in education [8]. It models each individual robot as a single object that inherits all kinds of interfaces based upon the robot's capabilities. The different interfaces do cover multiple sensor or actuator devices at the same time and also abstract sensors such as obstacle-detectors. This design often provokes naming conflicts for methods from different inherited interfaces. It also encounters difficulties in modeling a multi-PC robot design such as the B21. Distributing a single object over the network is a task not handled by most DOC models. However, approaches to integrate such a fine-grained distribution model into standard ORB architectures do exist[99].

Player/Stage also provides a C++-based client library that provides object-oriented interfaces [143]. It provides hand-coded local proxies to the various

player devices. Actuators such as the locomotion system have been designed as generalizations that offer a superset of the available functionality of the different supported devices. Clients cannot determine beforehand whether a robot actually supports the provided interface functionality (such as sideward motion). Little generalizations are provided for sensor devices. SmartSoft-/OROCOS@FAW takes a similar approach for a smaller set of supported robot platforms [105].

5.4 Logging Service

Apart from the sensor and actuator services, the *Miro* service layer includes functionality for the support of development and experimental evaluation. The generic logging facility for experimental data acquisition plays a central role for this purpose [137].

Autonomous mobile robots require advanced capabilities for data logging and processing logged data for several reasons. Learning, debugging, performance evaluation, and tuning are typical tasks that cannot be performed well without the availability and analysis of real-world data obtained from the robot's various different software subsystems like its sensor suite or its behavior modules. The domain characteristics such as the inherent parallelism or distributedness limit the applicability of classical single-step mode debugging. As these systems operate in a physical environment, executing a task on a slower time scale or with interruptions for closer inspection by the operators would also change the dynamics of the system. Simulation might help to some degree, but apart from the significant effort required to build a reasonable simulation model, most simulators fail to model the unexpected failures that occur during a real-world operation of the system. Both performance tuning and experimental evaluation of subsystems or the complete architecture also require reliable data about their performance in the target environment. Without such data, assessing the strengths and weaknesses of the subsystem becomes very difficult. Systematic acquisition of real-world data can also be essential for many system development tasks. Especially if partial system functionality is to be learned, most learning methods either require the availability of sufficiently large sets of training data, or the achievable learning performance can be enhanced if more data is available.

Data logging in complex, distributed, real-time environments is challenging. First of all, the sheer mass of sensor data in a non-trivial autonomous mobile robot can exceed the system's bandwidth constraints if the logging facility is not carefully designed. Secondly, when dealing with large amounts of data, structured data types and strict type safety become necessary requirements in order to manage the stored data effectively. Furthermore, distributed systems introduced by multirobot scenarios like RoboCup introduce further challenges to the task of building a useful logging facility such as distributed logging, synchronization, and synchronized replay.

In this section, we will introduce *Miro*'s event logging capabilities. The general

idea is the following: on a robot, sensor data like odometry values or distance measurements is sampled at discrete time steps. The sensor services publish these values through the event channel, allowing for sensor-driven processing in the system. Higher sensory or cognitive processes can also publish their results this way in order to distribute them to other subsystems. If these events are logged to persistent storage, a comparatively complete and fairly exact trace of the robot in action can be acquired.

5.4.1 Data Acquisition in Distributed Mobile Systems

As demonstrated in the paragraph above, data acquisition in distributed real-time environments like mobile robots and multirobot teams is a non-trivial problem. In the following paragraphs, required and desirable features for a logging facility for such environments will be outlined.

Logging of Structured Data: Large collections of data usually need to be structured in order to be maintainable by a program. Many software packages support printf-style logging. But the data acquired during a robot's experimental runs in a natural environment is quite different from the data which is normally written to system log files. Much of the data consists of numbers that represent time stamps, distances, 2D or 3D coordinates, classification results or even images that can not easily be interpreted in a textual representation. So instead of text-based logging facilities, capabilities are needed to store structured, strictly-typed data collections that can be interpreted by specialized visualization or other post-processing tools.

Distributed Logging: Data logging in distributed systems is especially challenging. Generating a central log is either very difficult or impossible due to network bandwidth constraints, especially if WLAN is to be used. A particularly challenging problem is logging data of several robots in a robot soccer team during an official game at RoboCup, where the available bandwidth is usually far too small to allow for logging significant amounts of data. Therefore, a distributed logging service is needed. But to locally log data on each robot imposes the problem of having to synchronize the log files afterwards.

Maintenance of Logged Data: What is stored in the files from 2001? How can we separate the irrelevant data from the interesting data (cutting logs)? As collections of logged data can become quite large, tools for the management of logged data are needed. Log files often contain lots of irrelevant data because logging is not always deactivated between experiments. It is not unusual to have logs with 10 minutes of the robot in action but also 15 minutes of inaction, where sensor data was acquired while the robot was facing a wall and was waiting for its next mission.

Data structures tend to change over time. The data in the log files from two years ago may not be readable anymore by the latest revision of the program. While this can be regarded as a question of ensuring backward compatibility on a software engineering level, help from the logging system in case of failure might still be appreciated. Therefore, metadata on the type of stored data becomes an interesting feature from a long-term perspective.

Replaying Data: It would be a pity if the acquired data were only usable for statistical analysis. Acquiring data to debug, analyze and evaluate the performance of an autonomous mobile system is essential. If we need to store such data anyway, why not think of more advanced applications for those large data collections? Having logged raw sensor data from an experiment enables a roboticist to re-run the experiment with a later version of the software. This facilitates system performance tuning and is very helpful for learning applications.

Performance: Large amounts of data (e.g. raw sensor data) need to be recorded, but the available computational power is usually already used up by the robotics application. Computational power on a robot is usually a scarce resource. Image processing, planning and all the other subsystems of a robot usually already use up most of what is available. Therefore, data acquisition needs to come with little computational overhead. Additionally, there should be almost no overhead if the logging facility is not in use.

Ease of Use: Roboticists want to concentrate on solving robotics problems, not data acquisition problems. Although data logging may play a central role especially in scientific applications of autonomous mobile robots, there is usually not enough time that can be devoted to developing systematic solutions to such peripheral problems. A successful data logging facility for a robotics system must be transparent and easy to use in order to be valuable in everyday work.

5.4.2 Technical Solution

Miro's logging facility is implemented as a simple consumer that subscribes for events on the event channel. Along with a time stamp, each event is written to persistent storage. In the current implementation, this is a memory-mapped file. Log writing is vastly simplified by using the CORBA-provided infrastructure.

CORBA comes with a complete specification for object serialization, the common data representation (CDR), as needed for communication in a distributed system [46]. Object serialization is called marshaling in CORBA technology. Every data type that can be communicated via the Notification Service can be serialized using its marshaling methods, that is every data type defined in IDL. Note that arbitrarily complex structured, nested and variable sized data types

can be defined in IDL. These are then mapped to the data types of a programming language by the IDL compiler, which additionally provides corresponding marshaling and demarshaling methods. The payload of an event in the event channel is represented in a `CORBA::Any` data type. This is a strictly-typed void pointer, which can hold any kind of data, but can only be dereferenced by casting it to the correct type. Thus, a log file is technically a CDR stream, which is written to disk instead of a network connection.

Configuring which events are to be logged, can be easily managed by the event channel's subscription/offer management protocol. Additionally, the Notification Service defines powerful filter mechanisms that can be applied to limit or preselect the persistently stored data. This is especially useful, since the data on an event channel is usually not published for logging or debugging purposes only, but also for further processing elsewhere within the system. The logging facility is just another consumer among potentially many others.

5.4.3 Logging Configurations

The conceptual approach of this logging facility is very powerful and flexible. It allows for the efficient and optimized local and remote logging of events from an event channel. Therefore, the same underlying mechanisms can be used for quite different logging configurations.

Remote Logging of Single Event Channels: The simplest configuration of the logging facility is depicted in Figure 5.5: a standalone logging program is started on a remote computer, links to the robot's event channel, and writes the events to which it is subscribed into a log file on the remote computer. This setup is very convenient for debugging purposes, where the remote computer is usually the workstation used by the programmer.

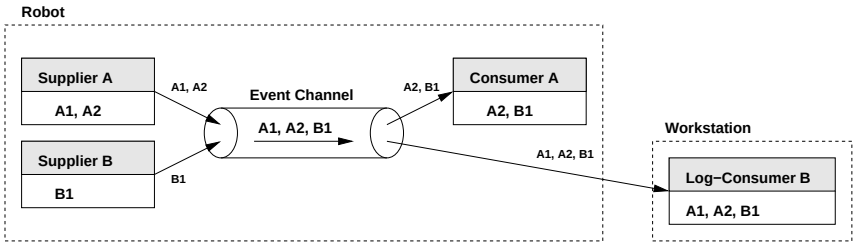

Figure 5.5: Ad-hoc logging in the lab: directly connecting to the robot's event channel from a remote machine.

Remote Logging of Multiple Event Channels: In multirobot applications such as RoboCup, it is often important to simultaneously log and analyze data from multiple robots. In this scenario, group communication is handled by a federated notification service as discussed in section 4.1.5. Thus in this configuration, a central logging client which features its own local event channel links

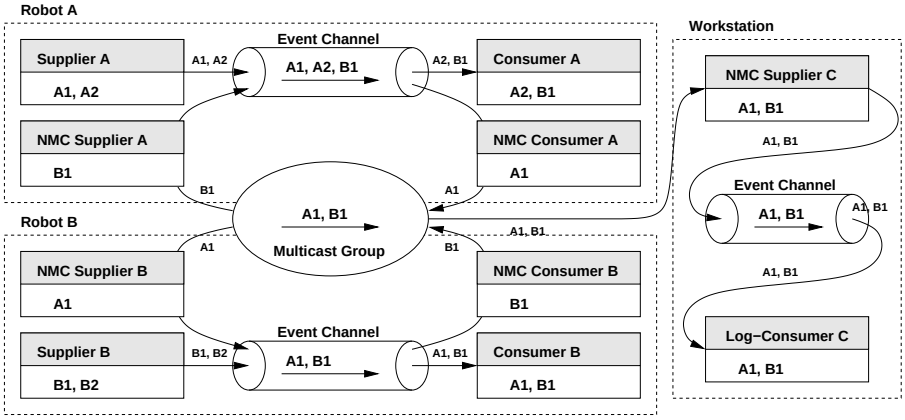

Figure 5.6: Centralized logging: using the notify-multicast-based event channel deliberation of the robots in a team to log data from multiple robots.

to all robots via NMC and writes a single log file of the delivered events (Figure 5.6). This configuration is especially useful if the data of interest has already been exchanged between the robots as part of a team coordination mechanism. In this case, logging this data requires no additional network bandwidth. NMC also helps to circumvent the problem of clock synchronization between the robots, although fairly sophisticated solutions are available for this purpose (e.g. NTP[86]).

Distributed Local Logging with Synchronized Replay: The configuration with the best performance in terms of data throughput is without doubt achieved by colocating the logging client in the same address space as the event channel on the robot itself (Figure 5.7). This configuration is used by our RoboCup team, the The Ulm Sparrows, to log vast amounts of data during a tournament game.

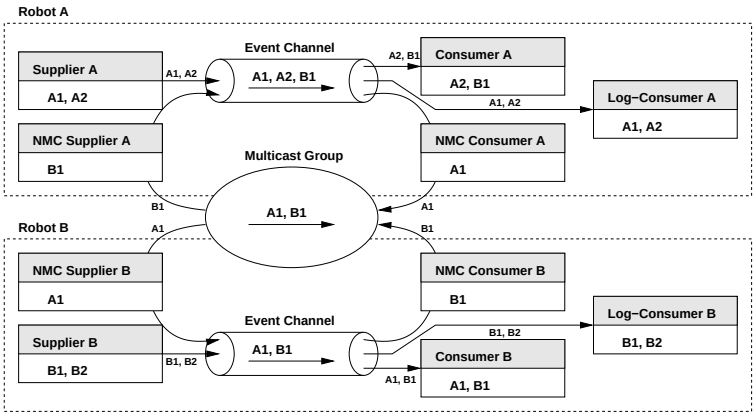

Figure 5.7: Tournament mode: decentralized logging locally on each robot.

5.4.4 Related Work On Data Acquisition

Storing program traces for debugging and system analysis has a long tradition in computer sciences. Therefore, many software development environments come with a more or less sophisticated set of logging facilities. In this section, four logging facilities related to *Miro* will be discussed.

The Adaptive Communications Environment (ACE) features a general-purpose C++-based logging facility. It is designed as a classical system log. It allows printf-like debug messages with variably-sized arguments to be written to various destinations (standard error, system log, a file) or to be sent to a remote logging server. Verbosity of logging can be configured at runtime. Additionally, various macros customize the functionality and also allow to disable logging at compile time. The log messages are plain text messages and no support is provided for parsing and post-processing of acquired data. *Miro* uses the ACE logging framework for classical debug messages and system errors.

RLog is a C-based logging library that was developed by Kortenkamp et al [66] for use in robotics. It defines data formats for storing built-in C data types like `int` and `double` and provides routines for log file parsing and post-processing. Another nice feature is the ability to register variables at the logging library for timer-based probing. Additionally, a query language allows to select sets of events from a log file, and it also contains basic generic visualization tools. The main limitation of RLog is its lack of support for structured and/or variable-length data types. Also, it lacks compile time type checking and strict runtime type safety as provided by the IDL compiler and the `CORBA::Any` data type. In [65], performance for logging to a local file is reported with 53 msec for 1100 logging calls on a PII running at 800 MHz. The logging payload contained only one single instance of a basic data types like int, float or character string. Note also that in difference to the *Miro* logging facilities, only dedicated messages are logged by the library, instead of logging general purpose system events.

The Carnegie Mellon Robot Navigation Toolkit (CARMEN) provides a typical example of a handcrafted distributed systems logging facility. CARMEN uses the message passing library IPC [120] for publisher/subscriber-based communication. The logging facility can register for a fixed set of predefined message types at the central IPC server and writes them to a file in a textual representation. Every new message type or altered message structure has to be manually added to the logging as well as the parsing facilities. This makes logging scenario-specific output from higher processing levels with this facility impractical. IPC supports structured and variable length data types, but lacks compile time type checking and needs to rely on interpretative marshaling. Performance numbers for the CARMEN logging facility are not reported.

CORBA itself also provides a specification for a logging service based on its Notification Service, the Telecom Log Service [45]. As in our approach, it writes events received from the event channel to persistent storage. The constrained language defined for event filtering by event consumers can also be used for querying the data base for sets of logged events. The specification differs

from the approach taken by *Miro* insofar as it does not define a logging client, but defines special logging-enabled channels which log the data in addition to distributing them to consumers. Also, the Telecom Log Service specification focuses on log creation and logging management but does not provide any support for log file management or for the timely replay of (possibly multiple) logs.

5.5 Summary

In this chapter the service layer of the *Miro* architecture has been discussed. First a generic design for actuator services was proposed and illustrated by the design of the interface hierarchy for the control of different locomotion devices for mobile robots. Second, an analogous solution for sensor services taking into account the requirements for flexible communication of sensor data to client applications was outlined. Afterwards, the requirements and difficulties of data acquisition in distributed environments such as robot teams were identified and a scalable service for generic logging of data in multirobot experiments was presented along with related work.

The proposed object-oriented design of interfaces to sensors and actuators, along with the service orientation addresses several of the identified problems hindering the portability and reusability of robotics software. The extreme heterogeneity of robotics devices is modeled by interface hierarchies that allow accessing a concrete sensor or actuator on different abstraction levels. The service-based decoupling of clients and servers takes the distributedness of robotic applications into account and the introduction of metainformation queries as a required feature of sensor and actuator services helps client applications to further overcome dependencies on the devices or the targeted environment. Additionally, the time-stamping and the publisher/subscriber-based distribution of sensory information fulfill important requirements for sensor processing in time-constrained, highly-dynamic environments. The distributed logging service, along with its replay capabilities plays a central role in addressing the inherent problems of applications that need to cope with the stochastic properties of the physical world.

The chapter contains two contributions to the state of the art in research on autonomous mobile robotics. First, a new, detailed design has been proposed for modeling sensor and actuator devices as network transparent, pro-active services with type-safe, object-oriented interfaces. Second, a solution for generic, high-performance data acquisition in distributed environments was proposed.

Chapter 6

Extensible Frameworks

An essential step towards easier and faster development of mobile robot software is the reuse of code and design for regularly occurring tasks. Since mobile robots research has seen a constant flow of new methods and research results in the past, this seemed to make little sense so far. However, a number of recently developed methods and techniques seem to emerge as de-facto standard solutions, e.g. grid-based probabilistic egocentric and allocentric mapping [131, 24], self-localization based on segment matching [47] or particle filters (a.k.a. Monte Carlo localization [33, 25]), behavior-based robot control [13, 104], or various path planning methods. The *Miro* Framework Layer provides extensible frameworks that implement commonly used techniques for mobile robot control in a way that enables them to be applied uniformly on the different platforms supported. Available functionality includes a framework for video image processing in time-constrained robotics applications [135], a behavior framework for structured reactive behavior-based control and hierarchical modularization [136], as well as a sample-based pose estimation based on particle filters [26, 139]. Further valuable extensions would be the integration of generic mapping such as [24, 102, 69] and path planning functionality such as [134] or the integration of frameworks for higher-level symbolic planning.

Frameworks are second-generation programming abstractions. While libraries group related algorithms into pre-built packages, frameworks do not only foster the reuse of code, but also support the adaptation of a design [58]. They often represent semi-complete applications for a specific target domain. By providing abstractions for the control flow of the programming logic, the application programmer merely needs to implement the individual functionality of the target application. The framework then executes the implemented code according to the execution logic [112]. Frameworks often organize the data flow, too. Graphical user interface programming environments such as Qt (Section 4.3.3) are usually organized that way. The framework handles the input from mouse and keyboard as well as the triggering of redraws for window elements and provides standard window elements such as buttons or entry fields that can be readily used or extended in their functionality by the programmer.

Frameworks are also a key software concept for supporting developers with the challenges of the robotics domain, since their design also allows to address non-functional aspects of an application on an abstract, generic level.

Inversion of control flow. This common feature of a software framework can be exploited in the framework design, to address issues of scalability and reactivity, by providing different models of parallel or interleaved execution. A consequently optimized control flow management can also enhance the efficiency of data processing in the target applications. Additionally, this feature allows for a natural application of the sensor-driven processing model as often required in time-constrained environments.

Management of data flow. Data flow management can shield an application programmer from the various subtle problems of locking in multi-threaded applications. Also, a poorly conceived organization of data flow can easily lead to memory leaks, which represent a subtle and hard to trace category of programming errors that compromise the overall stability of an application. Additionally, distributing large amounts of data to the various modules of a robotics application stresses the available memory bandwidth, which can jeopardize cache utilization. Consequently, a careful organization of data flow can reduce memory copying and thus improve the overall system performance.

Implemented standard functionality. Providing building blocks of standard functionality for a target application domain is very easy and efficient within the bounds of a framework. The reuse of the intended design by the application programmer ensures the matching of the interfaces and helps to eliminate the need for glue code that moderates between the pre-built components and their new application.

In this chapter two frameworks of the *Miro* project will be covered in detail. The video image processing (VIP) framework provides a software architecture for the real-time-oriented processing of video image streams for autonomous mobile robots [135]. Its goal is to enable and facilitate the use of computer vision methodologies within the heavily time-constrained environment of an autonomous mobile robot in a highly dynamic environment, like the RoboCup mid-size league. The second framework is designed for structured, reactive behavior-based control (BAP, an acronym derived from its key components: behaviors, action patterns and policies). In this framework, concepts are applied to overcome the intrinsic scalability issues of behavior-based robot control architectures. The key issue addressed by the design of BAP is how behaviors can be organized in a hierarchical way. This allows for complexity reduction and for the reuse of action patterns or complex sequences of actions either within a single control system or for different scenarios [136]. Apart from the detailed discussion of the domain-specific issues and requirements that need to be addressed by the design of the framework, an important aspect of the discussion will be how the framework-based design can provide support for meeting the challenges of software development for autonomous mobile robots

6.1 Video Filter Framework

Vision is one of the most valuable sensors for autonomous mobile robots. Cameras are relatively low cost and offer a huge and diverse set of information that can be used for very different sensing tasks. Unfortunately, there is a severe lack of advanced vision processing methodologies applied in todays robotic applications. Especially in highly dynamic environments and predominantly reactive scenarios, research is still focused on model-based color blob detection [42, 48]. In consequence, vision processing in robotics lacks flexibility and scalability, which makes it impossible to use such a vision system for different tasks and multiple scenarios. This hinders advances in the scientific view on the problem domain.

Applying advanced vision processing methodologies to autonomous mobile robotics is difficult, as the requirements of this application domain add a whole set of additional complexity to the original task of image understanding. For instance, image processing binds a lot of computational resources and most higher-level image processing operations are difficult to apply within the timeliness constraints of a real-time reactive autonomous system. Addressing such issues for a robotics vision system requires extensive architectural support, which is not available in currently available image processing systems.

6.1.1 Image Processing on Autonomous Mobile Robots

Vision systems for mobile robots bring together the two very challenging problem domains of image processing and autonomous mobile systems. E.g. most of the state of the art computer vision algorithms are computationally rather expensive, even when efficiently implemented. So a very careful assessment of their individual applicability is necessary. This on the other hand often discourages experts in computer vision to work on robot vision, as most of the advanced algorithms seem to be ruled out per se by timing constraints. In consequence solutions in robot vision are often: (1) hard coded quick hacks that try to enable micro optimizations by doing multiple operations at once, (2) heavily model-based or heuristic, exploiting special circumstances with little validity despite the one scenario they are targeted for, (3) in consequence hardly maintainable and little flexible.

So to mediate between the partially contradictory requirements of advanced vision processing in a real-time constraint environment, proper conceptual support from the vision processing architecture is necessary, to encapsulates the vision application within this application domain. In order to better understand the different requirements that need to be supported, we first take a brief look at the two problem domains: computer vision and robot vision.

Computer Vision and Image Understanding

The basic concept of computer vision is the application of operators to image data such as the conversion of a color image into gray-scale, or filtering the image for edges. Often operations transform more then one input image into a new output image as e.g. a Canny edge detector [19] usually needs two images, which are convolved using a horizontal respectively a vertical Sobel operator. Other operators may use the same image result from different time stamps as for example a operator using two timely consecutive images to detect the optical flow [52].

More sophisticated operations do not only cover filter-like processing steps, but all possible input-output mappings in general. So the result of a computer vision operation doesn't have to be again an image but can be every possible data as e.g. a color histogram, a similarity value between two images or any other image statistic measure.

Sequences of such image operators reveal features within the image that can be used to identify regions of interest (ROIs). So subsequent image operations don't need to be applied on the whole image but can be restricted only to relevant subwindows. This is done either to speed up the processing loop or to be sure not to tamper the result with unwanted image structures from outside the region. Further operators derive image features from these ROIs that enable a reliable object recognition. Various feedback loops such as integration over time [59] can speed up processing and improve classification results [80].

Robot Vision

Performing the above sketched operations on an autonomous mobile robot on the video image stream of the robots camera(s) within a medium sized robotics application adds a whole bunch of additional challenges to the problem set.

Efficient organization of control and data flow. Video image processing on a mobile robot is usually sensor triggered and is started as soon as a new image is available to the robot as an image taken one second before does not necessarily resemble anymore the actual situation in a dynamic environment. At the same time, the performed processing needs to be demand driven, to not misspend the available computational resources.

Parallel and asynchronous evaluation. More and more robots are equipped with multiple cameras for stereo vision, or to extend their field of view. Multiple image sources, but also dual CPU boards as well as the upcoming hyper-threading and multi-core processor technologies call for asynchronous, parallel processing capabilities. Multiple image sources allow for interleaving processing, and the true parallelism of the advanced hardware features stay unused by single-threaded applications. The actual challenge however, lies in the

proper synchronization between different image processing tasks for the fusion of their results.

Timeliness and resource management. Due to the computational cost of most image operations, and the fact that the CPU is also used by other concurrent tasks of the system, the available processing power will usually not be enough, to perform all possible evaluations on every single image. In order to still meet the timeliness constraints of the reactive systems, different perceptual tasks (e.g. obstacle avoidance and face recognition) need to be properly prioritized. E.g. the data for obstacle avoidance needs to be evaluated as often as possible, while the face recognition for greeting known pedestrians can be evaluated whenever some CPU cycles are left. Additionally, not all image processing tasks have to be performed over the whole time. The robots' situatedness enforces the use of special vision routines for different purposes.

Communication of results. Last but not least, images as well as extracted symbolic information of objects need to be accessible to the other modules of the robot software. Interfacing is an issue in the context of image processing on autonomous robots, as the information requested by client modules usually determines which information needs to be extracted from the image in a given situation. Also, the communication of whole images to client applications consumes large amounts of communication bandwidth and requires therefore a careful design.

Related Work

Common vision related architectures and publications can be roughly divided into three types: subroutine libraries, command languages and visual programming languages.

Subroutine libraries are the most commonly used ones. They mostly concentrate on the efficient implementation of image operators. Therefore they consist of normal functions, each responsible for a different image processing operation. Classical examples are e.g. the well known SPIDER system [129] or NAG's IPAL package [20] written in C or Fortran. More recent approaches are e.g. LTI-Lib [122] or VXL [123], which both are open-source, written in C++ and consist of a wide range of operations, ranging from image processing methods, visualization tools and I/O functions. The commercial Intel Performance Primitives (IPP) [56] are an example for highly (MMX and SSE) optimized processing routines with a normal C-API. What they all have in common is their lack of support for some kind of flow control support. Yet another collection of mutex or semaphore helper classes and some kind of thread abstraction is the maximum of assistance in this respect.

More advanced command languages for image processing are mostly implemented as scriptable command line tools that a developer can use to direct

the vision package. In case of the imlib3d package [121], the image processing operators can be called from the Unix command line, the CVIPtools [132] are delivered with an extended tcl command language. So both packages have the ability to include conditional and looping facilities. But again the programmer not only has a flexible way of complete control over the system, but also the full liability over the processing cycle. Additionally the scripting approach makes it hard to meet the required performance constraints of this application domain.

The most sophisticated solutions are the visual programming languages. They allow the user to connect a flow-chart of the intended processing pipeline using the mouse. They combine the expressiveness and the flexibility of both above groups. Often they contain not only a real mass of image processing functions and statistical tools, but also a complete integrated development environment. Most of these systems are commercial products. One of the most advanced one is VisiQuest (formerly known as Khoros/Cantata). According to their web site, it supports distributed computing capabilities for deploying applications across a heterogeneous network, data transport abstractions (file, mmap, stream, shared memory) for efficient data movement and some basic utilities for memory allocation and data structure I/O.

But as of today, there is no concise design for image processing available that combines all of our above described features like parallel and on demand processing of parts of the filter tree in a flexible yet powerful way, making the system suitable for a wider range of image processing tasks, like active vision problems on autonomous mobile robots.

6.1.2 Solution Approach

The principal idea of the VIP framework is to manage the control flow and organize the data flow of the vision application, for a clean separation of the two problem domains. That is, the vision application programmer only needs to implement the individual image operations (if not already available in form of a library) and direct the data flow for the target application. The VIP framework then executes the implemented code according to the needs of the client modules, ensuring the correct evaluation order of the various image operators.

The basic processing unit is called a filter. This denotes not only a (non-)linear image transformation function like a Sobel operator, but every input-output mapping such as a neural classificator on image features. While the control flow is evaluated in a tree in depth first order (subsequently referenced as the processing tree), the data flow is much more flexibly organized as a directed acyclic filter graph (DAG). The framework ensures the correct evaluation order. Freely definable so-called metainformation such as a list of regions of interest, histogram values etc. can also be passed through the DAG to successor filters. This actually extends each filter instance to a general image processing node.

Robotics Support

Support for the intrinsic problems of robotic vision is supplied on the basis of the configurability and adaptivity of the framework and its execution logic, by special purpose filters and also by additional development tools. As part of the *Miro* project, VIP is currently implemented as a C++ white box framework.

To prevent excessive polling or context switching between waiting threads the framework performs sensor triggered evaluation of filters. In order to maximize performance in this highly time constraint environment, VIP keeps track of which filters are actually queried by client modules. Based on this connection management, a dynamic graph pruning is performed to process only the minimal required filter tree for each image. If a client module connects to a new filter, the filter is guaranteed to be part of the processing tree, as soon as the next image becomes available. The service-based design of the VIP framework within the *Miro* software architecture provides support for network transparent as well as co-location optimized access to images or higher-level sensory results from the filter DAG to client applications. For colocated image queries a shared memory-based approach is used with zero-copying.

Source Nodes of a Filter DAG

Video devices are also modeled as filters within the framework and form the root node of a processing tree that is, source nodes in the data flow graph. The framework supports various camera connections such as analog frame grabbers (e.g. BTTV), IEEE 1394 and USB-cameras and also multiple cameras in parallel. Each processing tree its executed within its own thread and is processed in parallel with other source nodes, while the data flow can stay connected. The framework then ensures appropriate synchronization between the image streams. Note that, as the framework takes care of synchronization, developers do not need to worry about locking issues and the right usage of synchronization primitives. Also, the data flow needs not necessarily stay synchronized with the control flow, that is, a filter can hold a reference to an input image over multiple invocations to perform operations on consecutively taken images, such as optical flow.

Additional processing trees can be added to decouple time-consuming image operations (a slow path) that can not be performed on each image of the input source, from fast image evaluations, needed at full frame rate for reactive tasks in the robotics application.

Real-time Constraint Image Processing

As one of the dominant features of robot vision is the timeliness constraint, VIP integrates multiple concepts for real-time processing. Each processing tree can be executed with its own thread priority and scheduler choice, which is directly

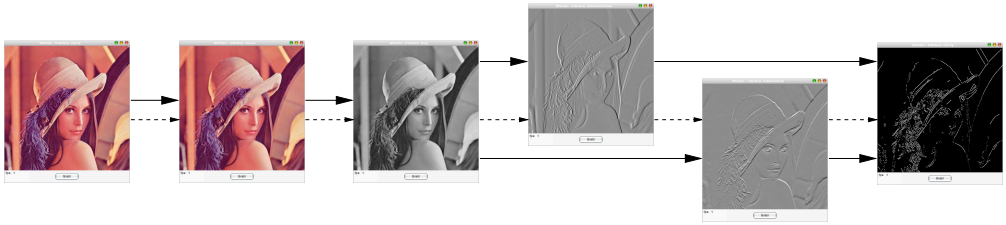

Figure 6.1: Original image, intermediate processing steps (blurred, grayed and convolved images) and resulting edge detection. The thick, solid lines denote the data flow, the thinner, dashed lines the control flow.

mapped on the OS-native process scheduler by the framework. This is necessary to minimize jitter and ensure correct priorization, especially under high load situations. Additionally, detailed timing statistics are provided for each filter. Different models for synchronization of filters between different processing trees can be used to either optimize synchronization of image sources (stereo vision) or minimize locking overhead and context switching between threads (slow/fast path processing).

Development Support

Applications in robot vision require extensive testing and tuning of filter configurations. Therefore VIP provides various concepts to ease the development process. The extensive use of the configuration management support provided by the infrastructure layer described in Section 4.2 allows to specify metainformation about newly developed filters for various means. Instead of simply defining the parameters of a service implementation, the VIP framework uses the infrastructure, to provide complete specifications of filter graph configurations by the help of the parameter description language, allowing to adapt processing parameters, or to provide an entirely different image processing DAG without recompiling a single filter. Such configurations can be built conveniently under a graphical user interface, as illustrated in Section 7.1. Also, every filter, and therefor every intermediate result, can be queried (e.g. for visualization) by simple assigning it a name for the according interface. The middleware integration also enables to change filter parameters on the fly from client applications in reaction to changes in the environment. By exchanging the physical video device with an image file set based virtual device that replays a stored image stream, the whole processing tree can equally used on- and offline.

6.1.3 Example Configuration

The above described feature set of the VIP framework is best understood by a small illustrative example. Figure 6.1 illustrates the derivation of an edge image from the classical test image of computer vision. The original image is Gaussian blurred and transformed into a grey image. Then a horizontal and vertical Sobel

operator is applied and in the last step the Canny operator is applied. The screenshots are taken from the generic inspection tool. Metainformation is not provided by these simple filters. The data flow and control flow is illustrated in Figure 6.1. The thick, solid lines denote the data, while the dashed lines illustrate the control flow.

6.1.4 Possible Extensions

Future work will be directed in two different directions. The first is to assess carefully the optimization potential for used system resources, especially memory consumption. Improving cache hit rates for instance can tremendously increase the performance of image algorithms and control flow and memory management have a significant impact on it. One possibility is to switch to inplace processing of filters, if the filter and the filter graph configuration allow it. The other direction is to connect the priorization of the image processing tasks with the real-time capabilities of the underlying distributed systems middleware (RT-CORBA), to ensure end to end quality of service between sensory and actuatory processes, especially in combination with a real-time enabled notification service configuration as discussed in [40].

6.2 BAP Framework

Mobile robots acting in dynamic environments populated by humans and other robots must be able to react quickly to unexpected situations. Behavior-based approaches have been suggested and successfully used to implement reactive robot behavior for almost 20 years now [13, 5, 78]. Nevertheless, implementing a broad set of different behavioral skills and coordinating them to achieve coherent complex behavior is still an error-prone and very tedious task. To more than a few, behavior programming therefore seems to be more of an art than a science.

Most robot programming environments provide little conceptual and tool support for behavior engineering issues such as modularization, configuration, reusability and maintainability. Furthermore, although some behavior-based approaches provide concepts for organizing reactive behavior in a hierarchical manner, there is no widely accepted methodology for creating such hierarchies in a systematic way.

In this section the *behavior, action pattern, policy* (BAP) framework for specifying hierarchical, behavior-based control systems is discussed [136]. The framework allows to adopt many well-known behavior-based approaches, such as those based on subsumption [13], fuzzy control [103, 104], or potential fields [5], but was designed to incorporate concepts of modern software technology, like modularization and reuse. The BAP framework was designed to ease the integration with planning-based methods and to foster the use of learning algorithms. It also aims to overcome some of the scalability problems associated with behavior-based approaches. The integration of the BAP framework into

the *Miro* project facilitates reuse and generalization of behavior libraries in and for different scenarios and robot platforms.

6.2.1 Desired Functionality

In practice, the design of a behavior-based reactive control system is very tedious and time-consuming for any non-trivial task. Due to the emergent nature of the overall system behavior, designing a set of behaviors producing a particular desired system behavior often proves to be very tricky. In addition, the behavior engineer must consider numerous issues related to system integration and interaction with the other system components. As a result, the design and implementation of behavior-based systems often appears to have little structure and seems to lack methodology, at least to outside observers. Programming complex behavior-based robot applications could be significantly improved by developing methods that allow for more systematic development, support prevalent software engineering desirabilities like modularity, functional abstraction, reuse, and reduce development time and effort by providing libraries, and appropriate tools for design, implementation, and evaluation of partial or complete behavior-based systems. A few particular issues are addressed in the next few paragraphs.

Reactivity by supporting concurrent behavior execution. A basic notion of all original behavior-based approaches is their intrinsic concurrency [13, 5]. This allows the programmer to factor out detection and handling of potential failure situations in separate behaviors and to retain a concise formulation of the actual task. The concurrent execution of all behaviors ensures automatic surveillance of potential failure situations. However, some more recent behavior-based approaches have traded concurrent behavior execution in favor of easier behavior sequencing. This puts the burden to ensure timely evaluation of failure conditions back on the programmer. Therefore, we consider concurrent behavior execution for a generalized reactive control framework as indispensable.

Taskability by supporting behavior sequencing. A common weakness of the early behavior-based approaches is the difficulty to implement a wide set of different tasks. Murphy uses the notion of taskability for describing how easy it is to switch between different tasks [87]. Indeed, a wide spectrum of different tasks requires a large number of behaviors not all of which are relevant in all situations. If all behaviors are concurrently executed all the time, some behaviors may get in your way in undesirable situations. A much more concise approach is to only execute the minimal set of behaviors necessary to produce the desired system functionality. Upon arisal of certain situations, the system switches to another behavior set. Thus, temporal sequencing of behavior sets is a highly desirable mechanism to provide easy taskability of the robot. Fewer

concurrent behaviors also make it simpler for the programmer to understand the emergent properties of the active behavior set.

Modularity by hierarchical policy specification. Simple temporal sequencing of behavior sets is often insufficient, if the application domain and the task set gets more complex. For example, in robot soccer (and many similar scenarios) there are numerous situations where the robot is supposed to select and execute a contingent plan out of several options, e.g. when executing one out of maybe 5 or 10 different special plays after a foul call. Each play – a contingent plan – can usually be represented by sequencing behavior sets (using e.g. a finite state machine). However, representing all such plays in a single large finite state machine of behavior sets gets soon very unwieldy and degrades maintainability. Therefore, concepts for hierarchically building policies, which implement such contingent plans and have proper entry and exit semantics, would greatly simplify the overall structure of the behavior system and increase modularity.

Functional abstraction by providing behavior parameterization. In practice, there are many situations where behaviors in different behavior sets are quite similar and differ only in a few parameter settings, like settings for maximum translational and rotational velocities and accelerations, or safety distances to be observed. This holds even for behaviors on different mobile platforms, which may have different low-level sensing and actuating capabilities and different sets of parameters like maximum speed, accuracy of odometry, and so on. Thus, there is considerable potential for code reuse, if the behavior architecture supports functional abstraction and concepts for parameterization.

As an example, consider a generic *obstacle-avoidance* behavior. The safety distance upon which it needs to react and influence the robot's movements depends both on the scenario and inherent capabilities of the robot platform. A tour guide robot should avoid crashing into exhibits and visitors at all cost, while a rescue robot may have to account only for its own stability when pushing its way through obstacles in the disaster area on its way to a victim. In robot soccer, tackling (physical contact) is okay for a soccer playing robot as long as the referee does not object, but such a behavior would need dynamic adaptation after the robot has been shown a yellow card. So the parameterization of behaviors can not solely be derived from the service-provided metainformation for sensors and actuators.

Flexibility by allowing different arbitration mechanisms If more than one behavior is active and producing inputs for motor control, then the behavior outputs are potentially in conflict and must be arbitrated. The various different behavior-based approaches, like subsumption, potential field methods, and fuzzy behaviors, mainly differ in what type of output the behaviors produce and what methods are used to arbitrate these outputs [93]. As the different

approaches often have complementary benefits and pitfalls, it is desirable to use the most adequate approach in any particular situation. A general-purpose behavior architecture should therefore support the use of multiple arbitration mechanisms.

In order to professionalize software development for autonomous mobile robots, a state-of-the-art system development environment should provide programming concepts and tool support for implementing behavior-based reactive control systems, and its design should observe the above design criteria. The next section presents the formal organization of behavior with the BAP framework, which was developed specifically with these design criteria in mind.

6.2.2 A Formal View on Modeling Reactive Control

The next two paragraphs introduce the main concepts informally, before we provide more formal definitions further on.

We assume a robot with multiple sensors, which are controlled by sensory processes that read out or interface to the sensors and deliver their data into an observation space $\mathcal{O}$. Interpretation, fusion, and integration of sensor data is performed by perceptual processes, which represent their results in a space of state variables $\mathcal{S}$. The robot's effectors are controlled via low-level motory processes, the inputs of which make up motor space $\mathcal{C}$. Observation space and state space, together constitute data space $\mathcal{D}$, i.e. the set of variables constituting the information available to the robot at any time instant and the basis for any decision making. A special type of perceptual processes are guards, which observe particular conditions on data space, e.g. that an object is visible, or detect state changes, e.g. that an previously visible object has been lost or that a goal state has been reached. Guards signal events, which can be viewed as logical predicates satisfied in a specific situation represented in data space.

Behaviors are mappings from data space to motor space. If several behaviors run concurrently, an arbiter deals with their potentially conflicting outputs. A set of concurrently running behaviors together with a set of guards and an arbiter make up an action pattern, which can be viewed as a primitive, coherent action with safeguards. By combining a set of action patterns together with an event-based transition relation we get a policy, which defines temporally coherent sequences of action patterns. The transition relation allows for the specification of quite complex control structures, including sequences, conditionals, and loops. Policies can take the place of action patterns in policy definitions, thereby permitting hierarchical specifications of policies. The temporal extent of action coherence increases, the higher we move up in policy hierarchy.

6.2.3 Behaviors

A single *behavior* b defines a mapping from data space $\mathcal{D}$, which represent observations of the environment, to motor space $\mathcal{C}$ (that is, actuator commands), which change the state of the environment:

$$b : \mathcal{D} \rightarrow \mathcal{C}$$

Usually, behaviors are parameterized mappings, which we refer to as *behavior schemata*. A set of n available behavior schemata in an actual implementation is denoted by $\mathcal{BS} = \{BS_1, \ldots, BS_n\}$. Behavior schemata must be instantiated to produce actual behaviors that can be used in action patterns. Instantiation requires providing parameter settings. The behaviors present within an implementation form the set of available behaviors:

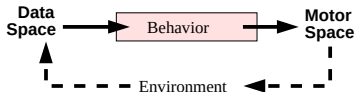

Figure 6.2: Feedback loop of a behavior.

$$\mathcal{B} = \{b_1, \ldots, b_n\}$$

The formalism does not make any assumption about the kind of mapping a behavior implements nor about how a behavior is actually implemented. This weakens the assumptions possible on a behavior, but allows to capture most of the known behavior implementation methods, including augmented finite state automata [13], potential field methods [5], and fuzzy behaviors [103, 104].

6.2.4 Arbiters

Concurrent execution of behaviors was identified as one of the design requirements. Multiple behaviors that concurrently map into motor space provoke quite undefined behavior in practice. To resolve the issue, each behavior b_i maps into a separate motorspace $\mathcal{C}_i$. The process of combining the actuator commands of different concurrently running behaviors is called arbitration. It can be regarded as mapping function f that fuses several suggested actuator commands into one:

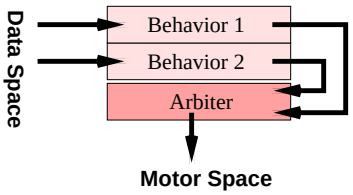

Figure 6.3: Arbitration of behaviors.

$$f : \mathcal{C}^n \rightarrow \mathcal{C}$$

Similar to behaviors above, arbiters may be parameterized, resulting in arbiter schemas analog to behavior schemas described above. The arbiters available within an implementation form the set of available arbiters:

$$\mathcal{F} = \{f_1, \ldots, f_m\}$$

As for behaviors, our definition of arbiters does not constrain the behavior designer to a particular arbitration method. Some well-known arbitration methods are static or dynamic priorities, subsumption, superposition of potential fields [5], or fuzzy inference and defuzzification [103]. However, dependencies exist between the used arbitration scheme and the types of behaviors usable with it; e.g. when using fuzzy arbitration the behavior outputs are usually required to be fuzzy variables.

6.2.5 Guards

While the control loop of a behavior forms a continuous sensor/actor mapping, there may also be discrete conditions in the data space that a reactive system has to take into account: the distance to the target falls below a certain threshold, a timeout for achieving a certain goal elapses, the object to track vanished etc. Thus conditions are recognized by so called guards g that signal the event of detecting such a condition, by emitting a guard event. A guard event e, therefore indicates a discrete event within the system. As an exception for notational convenience, e_0 denotes the empty event, that will be ignored. The set of available guard events within the system is denoted by:

$$\mathcal{E} = \{e_0, \ldots, e_l\}$$

Naturally, g does not need to be surjective. Therefor the subset of events actually generated by g is denoted $\mathcal{E}_g$, with $\mathcal{E}_g \subseteq \mathcal{E}$. A guard can be therefore described as a mapping:

$$g : \mathcal{D} \rightarrow \mathcal{E}_g, \text{ with } e_0 \in \mathcal{E}_g$$

Events from Behaviors

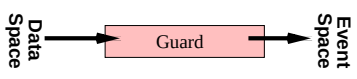

Figure 6.4: Guards signal discrete events.

Many conditions become recognized during the evaluation of the control loop of a behavior. These conditions normally indicate that a behavior has reached its goal, or that the behavior is unlikely to reach its goal anymore, as a necessary precondition no longer holds. I.e. a wall following behavior will loose its applicability, if no wall is detectable by its used range sensor(s). Therefore it will be useful in implementation practice, to allow for behaviors to supply guard events. However, for notation simplicity, we will treat such an implementation as two distinct objects in the formal framework.

Just as behaviors and arbiters, guards can also be instantiated from parameterized guard schemas. The guards present within an implementation form the set of available guards:

$$\mathcal{G} = \{g_1, \ldots, g_k\}$$

External Events

Events may also be generated by processes outside of the behavior engine, e.g. by sensor reading processes, motor control processes, or cognitive processes. This is useful for interfacing the behavior architecture with other system components and for overall system integration. Formally, we denote these events by a set $\mathcal{E}_{\triangleright} \subseteq \mathcal{E}$. These events are treated just like events generated inside the behavior architecture.

6.2.6 Action Patterns

An *action pattern* consists of a set of behaviors, a set of guards, and an arbiter, all of which are concurrently executed. An action pattern is formally denoted by a triple:

$$a = (B, G, f), \text{ with } B = \{b_1, \ldots, b_m\} \subseteq \mathcal{B}, G = \{g_1, \ldots, g_n\} \subseteq \mathcal{G}, f \in \mathcal{F}$$

When necessary, we distinguish action patterns and their components by indexing them appropriately, as e.g. in a_1 and B_a. An action pattern represents a controller for the autonomous mobile system (see Figure 6.5). Its behaviors, guards, and the arbiter form the emergent control loop. Action patterns allow for monitoring and control of many different sensory and effectory capabilities in a structured and modular way and ensure timely responses (reactivity) via concurrent execution of its constituents.

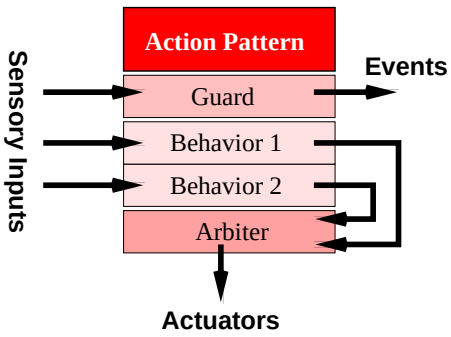

Figure 6.5: Structure of an action pattern.

For the execution of different tasks the capability of executing different action patterns is necessary. The set of available action patterns is denoted by

$$\mathcal{A} = \{a_1, \ldots, a_k\}$$

At any time, only a single action pattern is active. Behaviors, guards, and arbiters are called active, if they belong to the active action pattern. Thus, the action pattern is the central concept for capturing concurrent execution of behavior-producing functionality (and thereby taking care of the reactivity requirement), while task sequencing will require to switch between action patterns.

Of further importance is the set of events potentially signaled by the behaviors and guards of an action pattern, also called internal events, which we denote by

$$\mathcal{E}_a^{\triangleleft} = \bigcup_{g \in G_a} \mathcal{E}_g$$

The set of events possibly signaled while an action pattern is active consists of the events signaled either internally or externally

$$\mathcal{E}_a = \mathcal{E}_a^{\triangleleft} \cup \mathcal{E}^{\triangleright}$$

Whenever an (internal or external) event is signaled, the execution of an action pattern is terminated.

6.2.7 Transitions

After defining the basic concepts for producing behaviors, arbitrating conflicting behavior output, generating events for termination conditions, and the basic activity unit for concurrent execution, we can now tackle task sequencing. Assume we have some set $\mathcal{A}$ of available action patterns, for each of which we know the set $\mathcal{E}_a$ of possibly signaled events.

A *transition* specifies the successive action pattern, if an active action pattern is terminated by a specific event e, and is formally denoted by a triple

$$t = (a, e, a'), \text{ with } a, a' \in A, e \in \mathcal{E}_a$$

a is called the source, a' the target of a transition. The event e is either produced internally by a guard of the action pattern a or an external event. So, while the guard event terminates an action pattern, a transition defines a successor pattern in such an event. Transitions are atomic, that is they represent synchronization points within the concurrent execution of behaviors, arbiters and guards. Loops may be defined, i.e. transitions with $a = a'$. This can be used to explicitly ignore the request of a guard for the termination of an action pattern. For further reference, we define for a given set A of action patterns and a given set $\mathcal{E}^{\triangleright}$ of external events the set of possible transitions over A and $\mathcal{E}^{\triangleright}$ as follows:

$$\mathcal{T}(A, \mathcal{E}^{\triangleright}) = \{(a, e, a') \mid a, a' \in A \wedge e \in \mathcal{E}_a\}$$

Additionally, the successor pattern defined by a transition is required to be uniquely defined, to ensure deterministic results. That is,

$$\forall (a_0, e, a), (\widehat{a_0}, \widehat{e}, \widehat{a}) \in T : [(a_0 = \widehat{a_0} \wedge e = \widehat{e}) \rightarrow a = \widehat{a}]$$

Transition Patterns

There often do exist situations within scenarios that require a uniform reaction of the robot. Referee calls in RoboCup like the game start would qualify for such a situation. For notational convenience, but also for the explicit notion of universal validity, these events can be handled by a so called transition pattern $t^* = (*, e, a')$ that denotes a set of transitions as follows:

$$t^* = \{(a, e, a') : \forall a \in \mathcal{A}\}.$$

Note that t^* is defined for all $a \in \mathcal{A}$ even if $e \notin \mathcal{E}_a$. Analogously to normal transitions, the set of possible transition patterns $\mathcal{T}^*(A, \mathcal{E}^{\triangleright})$ and the uniqueness of successor action patterns for t^* are defined as follows:

$$\mathcal{T}^*(A, \mathcal{E}^{\triangleright}) = \{(*, e, a) \mid a \in A \wedge e \in \mathcal{E}_A\} \text{ with } \mathcal{E}_A = \bigcup_{a \in A} \mathcal{E}_a; \text{ and}$$
$$\forall (*, e, a), (*, \widehat{e}, \widehat{a}) \in T^* : [(e = \widehat{e}) \rightarrow a = \widehat{a}].$$

6.2.8 Flat Policies

The combination of action patterns and transitions allows us to specify a higher-level robot controller, which does not only handle a particular task like an action pattern, but complex task sequences. Such a higher-level robot controller is called a *flat policy* and formally specified by a quadruple

$$p = (A, T, T^*, a_0)$$

where $A \subseteq \mathcal{A}$ is a set of action patterns, $a_0 \in A$ is a uniquely determined start pattern, $T \subseteq \mathcal{T}(A, \mathcal{E}_{\triangleright})$ is a consistent set of transitions over A, and $T^* \subseteq \mathcal{T}^*(A, \mathcal{E}_{\triangleright})$ is a consistent set of transition patterns over A. Informally speaking, a flat policy consists of a set of action patterns, a set of transitions associated with the individual source patterns, a set of transition patterns associated with the policy, and a dedicated start pattern, which is executed upon activation of the policy. The targets of both transitions and transition patterns must be elements of the flat policy's set of action patterns.

Although we require both the set of transitions and the set of transition patterns to be consistent, this does not necessarily hold for their union. That is a

transition and a transition pattern could specify different successor patterns for an event e:

$$t = (a, e, a') \in T \text{ and } t^* = (*, e, a'') \in T^*, \text{ with } a' \neq a''$$

This problem is solved by the following precedence rule: *Transitions take precedence over transition patterns.* That is, if e is signaled while a is active, the successor action pattern is defined to be a'.

The enclosing (flat) policy defines the appropriate context for transition patterns as already referred to previously. In fact, within a flat policy p, each transition pattern $t^* = (*, e, a) \in T^*$ is equivalent to a *set* of transitions:

$$t^* = \{(a_i, e, a) \mid a_i \in A \wedge \forall a' = a : (a_i, e, a) \notin T\},$$

i.e. we could replace a transition pattern by a set of transitions, one from each of the policy's action patterns as source and all of which sharing the event and the target, unless there is already a different transition specified for the source and the event.

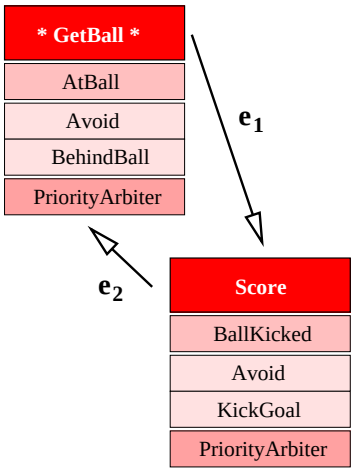

Figure 6.6: Example of a simple policy with sequencingand iteration of tasks.

Flat policies are terminated, if an event occurs that is not covered either by a transition with the currently active action pattern as source or by a transition pattern. Formally, the set of events terminating a policy, also called *unhandled events of a policy*, is determined by

$$\mathcal{E}_p = \bigcup_{i=1}^{n} \left(\mathcal{E}_{a_i} \setminus \{e \mid \exists a' : (a_i, e, a) \in (T \cup T^*)\}\right)$$

If for any event possibly occurring during execution of a policy p there is always an appropriate transition specified, i.e. if $\mathcal{E}_p = \varnothing$, the policy will never be terminated. In this case, we say that the policy is *closed*. In the less constrained case, where the policy handles at least all locally generated events, i.e. $\mathcal{E}_p \subseteq \mathcal{E}^{\triangleright}$, the policy is said to be *locally closed*.

For further reference, the set of available flat policies over $\mathcal{A}$ is denoted by

$$\begin{aligned} \mathcal{P}_0 = \{p \quad = \quad & (A, T, T^*, a_0) \mid A \subseteq \mathcal{A} \wedge \\ & T \subseteq \mathcal{T}(A, \mathcal{E}^{\triangleright}) \wedge T^* \subseteq \mathcal{T}^*(A, \mathcal{E}^{\triangleright}) \wedge a_0 \in A\} \end{aligned}$$

Figure 6.6 shows a simple example policy from the RoboCup domain. The policy consists of two action pattern, `GetBall` and `Score`. The start pattern is marked by asterisks around the pattern name. In the first action pattern, two behaviors are active, which are arbitrated by a subsumption style

`PriorityArbiter`. The `BehindBall` behavior maneuvers the robot behind the ball in direction of the opponents goal and `Avoid` ensures a collision free motion through the obstacles (opponent robots). The guard `AtBall` signals the event e_1 as soon as the robot reached its destination. The second action pattern consists of a behavior `KickGoal` that kicks the ball into the goal, again an `Avoid` behavior ensures collision avoidance and the second action pattern is terminated by the event e_2 as soon as the guard signals, that the ball was kicked. The two transitions model a simple interleaving of the two action patterns.

6.2.9 Hierarchical Policies

The concepts introduced so far cover all design requirements except for modularity by hierarchical policy specification, which we will now introduce. The basic idea is to also allow (sub)policies in the place of action patterns in the definition of policies. In order to do that, a hierarchy of *activity units* $\mathcal{U}_i$ is specified. Level 0 activity units are just the available action patterns: $\mathcal{U}_0 := \mathcal{A}$. Level 1 activity units extend $\mathcal{U}_0$ by the available flat policies defined above: $\mathcal{U}_1 := \mathcal{U}_0 \cup \mathcal{P}_0$. Level n activity units are recursively constructed by adding level $n-1$ policies (defined below) to the level $n-1$ activity units: $\mathcal{U}_n := \mathcal{U}_{n-1} \cup \mathcal{P}_{n-1}$. Sets of possible transitions and transition patterns are now defined equivalently over sets of activity units:

$$\mathcal{T}_i(U_i, \mathcal{E}^{\triangleright}) = \{(u, e, u') \mid u, u' \in U, e \in \mathcal{E}u\}$$
$$\mathcal{T}_i^*(U_i, \mathcal{E}^{\triangleright}) = \{(*, e, u') \mid u' \in U, e \in \mathcal{E}_{U_i}\}.$$

With these definitions, it is now straightforward to extend the definition of flat policies towards hierarchical policies:

$$\mathcal{P}_n := \{(U, T, T^*, u_0) \mid U \subseteq \mathcal{U}_n \wedge T \subseteq \mathcal{T}_n(U, \mathcal{E}^{\triangleright}) \wedge T^* \subseteq \mathcal{T}_n^*(U, \mathcal{E}^{\triangleright}) \wedge u_0 \in U\}$$

Hierarchical policies are illustrated in Figure 6.7. Here the action patterns of Figure 6.6 are enriched with a guard that checks, whether the ball is still seen. If not, the event e_3 is signaled and the `SearchBall` subpolicy is executed. There the robot turns in place and looks for the ball. `BallFound` will signal the event e_5 as soon as the ball is found, terminating the policy. If this takes too long, the event e_4 is signaled and the `GoHome`, subpolicy will drive the robot back to its home position. Note, that policies can be embedded multiple times in different higher-level policies, thereby facilitating code reuse. However, both direct and indirect loops and recursion are excluded, i.e. the hierarchy of policies is indeed well-defined. Note also, that in practice, events can be given arbitrary, more meaningful names.

6.2.10 Implementation

The framework-based implementation of the BAP concept of structured, behavior-based control offers various possibilities of supporting the application developer

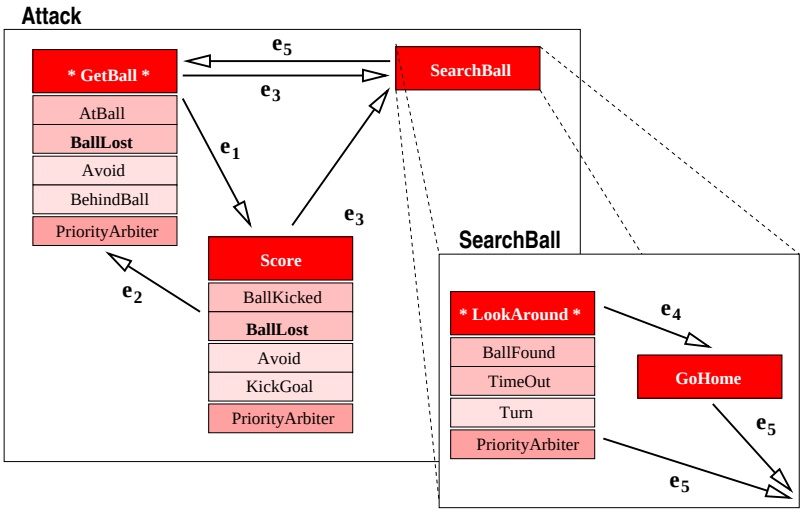

Figure 6.7: Example of a hierarchical policy. Searching for the ball is encapsulated in a subpolicy.

with scalability and reactivity issues of the application domain.

Control Flow Management

A central concept of the framework is that behaviors and guards run asynchronously in parallel. The BAP framework inverts the control flow by defining a behavior base class with virtual methods (OO callback hooks), for starting, stopping and reinitialization, as well as for calculating the output for a single iteration of the control loop of a behavior. The behavior engine runs its own control loop(s) and calls the different behavior and guard instances as specified in the configuration of the action pattern. Different concurrency models exist that can be used simultaneously by different behaviors. They can either be preemptively multi-tasked or triggered by timers. In the later case, each behavior/guard can define its own pace, at which the evaluation of the control loop is triggered. Additionally, they can also be modeled as consumers of an event channel (Section 4.1.4). This allows reactive control loops that are driven by the occurrence of sensory events, like the delivery of an infrared distance sensor measurement, or the activation of a bumper. As a consequence, the different behaviors/guards do not have to run altogether at the same pace, but can choose their own, adequate evaluation rhythm.

Data Flow Organization

The formal specification of BAP also allows to organize much of the required data flow for behavior instances such as configuration parameters and sensory events. This eliminates in many cases the necessity for application programmers

to handle locking issues for concurrently executed behaviors.

Predefined Components

The integration of BAP with the *Miro* architecture allows to implement framework components independently of the robot platform, as the service layer takes care of abstracting from the low-level details of a robots sensors and actuators and provides metainterfaces for the adaptation of client applications. Therefore the BAP framework also provides components ready for deployment into target applications.

BAP provides two different arbitration schemes. The first is a priority-based arbitration, which implements the subsumption architecture approach of Brooks [13]. The second is an experimental arbitration scheme, which favors a multi-valued approach. Its unique feature is to allow behaviors to prohibit values in the motory space. Experiments for a potential-field-based arbitration approach were also conducted.

Simple behaviors such as 'move to position' or a primitive 'wall following' are also part of the framework implementation. The discussed work provides also the foundations for the design of generic high-level behaviors such as generic obstacle avoidance for highly dynamic environments. Nevertheless, this involves the solution of severe additional challenges and therefore lies beyond the scope of this thesis.

6.2.11 Configuration Management

The configuration of behaviors, arbiters, action patterns, transitions and sub-policies into a policy is a substantial engineering effort in itself. The implemented framework does not force such configurations to be hard wired within the source code, but enables configurations (in particular parameterizations) to be separated out into a configuration file using an XML-based grammar. This allows for fast and easy reconfiguration of a policy without recompilation, which is especially useful during development.

For this purpose, the BAP framework, too, makes heavy use of the configuration management infrastructure. While the relevant properties of behaviors, arbiters and guards such as their parameters and events can be specified by the parameter descriptions, the hierarchic policy graph does not match very well the expressiveness of the configuration language. Therefore the policy configuration provides its own, extended XML-based grammar. It therefore also offers its own editor for the graphical programming of hierarchical policy configurations, which is discussed in Section 7.2.

Dynamic Reconfiguration

Static configurations of a robot's actuatory capabilities are sufficient, as long as all of the robots actions can be defined a priori. But skill learning or the interaction with navigation modules such as described in [134] or with symbolic AI planning make it desirable to overcome the limitations of static configurations.

In BAP, there are two levels of dynamic reconfiguration of a policy. The first level is the reconfiguration of the parameter sets of guard, behavior and arbiter instances. The second is addition and deletion of behaviors, action patterns, subpolicies, transitions and transition patterns. The first level of reconfiguration is fairly straightforward from the implementations perspective. Behaviors and arbiters are already defined to work on different parameter sets within different action patterns. Therefore, the change of a parameter set in a currently active action pattern is equivalent to a transition to the same action pattern with the new parameter set. The second level of reconfiguration is conceptionally well-defined, but includes subtle locking issues to prevent actions from e.g. deleting a currently executed action pattern and is therefore not implemented yet.

6.2.12 Possible Extensions

The proposed model of structured reactive control allows to describe complex behavior of autonomous mobile robots in a modular way that fosters reuse in this highly target application specific domain of robot programming. There are various extensions possible to further enhance the expressive power of the formalism.

Allowing guards to be instantiated on the policy level would allow to define additional termination conditions for sets of previously defined action patterns and subpolicies and would therefore enhance their reusability. Nevertheless, the formalism needs to ensure the deterministic nature of the transitions if the policy level guard events are also handled by transitions within the subpolicies. The simplest solution would be to require the set of policy level guard events and the set of events handled within the policy to be disjunct.

6.2.13 Related Work

The work discussed in this paper combines concepts from the theory of hybrid automata [51] with the behavior-based decomposition of control loops, which is then extended by the introduction of recursive structuring. The structuring of reactive behavior is predominantly addressed on the second level of a classical three tier architecture, that is, in the mapping of abstract, partially ordered plans onto the reactive control regime of an autonomous mobile platform. This resulted in very little support for structuring on the behavior layers itself. On the other hand the robot control languages that mediate between the deliberative and the reactive layer usually take a top down perspective and therefore are very loosely coupled to behavior concepts like arbitration.

Reactive Control Systems

The reactive layer of a robot control system is most often based upon a behavior-oriented control regime. However, structuring reactive control is usually delegated to the next higher level, so that few conceptual solutions exist on this layer. Consequently the systems that support structuring are typically not designed for a 3T architecture in the first place.

Subsumption architecture: The classical subsumption architecture [13] organizes reactive behavior in levels of competence. It uses a priority-based arbitration scheme, as higher levels of competence can override the output of lower levels. Aside of that there is little structural support, especially for behavior sequencing. On the contrary, the basic idea is that lower levels stays active all the time.

Saphira Fuzzy Controller: A formally more sophisticated arbitration scheme was introduced by [103]. It was first used by Flakey and became then part of the Saphira architecture [62]. It uses fuzzy control rules to combine outputs of different behaviors. At runtime, behaviors calculate fuzzy variables as well as their own activation value. The arbiter/defuzzyfier combines the values, weighted by the activation of the behaviors. Behaviors can be excluded from execution, but the control regime itself does not model this.

Dual Dynamics: A very interesting approach in organizing behavior sets into different modes of control is taken by the Dual Dynamics approach [57]. Behavior sequencing modeled as a continuous system based on bifurcation points in activation dynamics. Behaviors are organized by layering. Elementary behaviors calculate target dynamics (i.e. motor outputs). Higher level behaviors (level 0, 1, ...) calculate activation dynamics (modes). The activation of a behavior is calculated from the activation dynamics of behaviors of the next higher level. The system is accompanied by flow-based graphical design tools and a simulator [11]. The problem with reuse of behavior sets designed in DD is, that still all behaviors have to know all activation values of the next higher level, in order to respond correctly in all modes.

XABSL: The Extensible Agent Behavior Specification Language XABSL [75] models finite state automatons (FSAs) but uses decision trees for hierarchical structuring. The overall behavior is specified in a directed acyclic graph, with actually forms the decision tree. In contrast to BAP every leaf node consists of only one behavior. Each inner node ('option' in XABSL terminology) defines a FSA that augments the decision taking by an internal state. The decision tree is evaluated in every iteration. So options do not terminate themselves, but get disabled by taking another branch somewhere higher in the decision hierarchy. It does not support parameterization.

The specification is interpreted at runtime, so interaction with other subsystems like perception is done by registering their output variables at the XABSL runtime engine on program startup. - This introduces the possibility of runtime errors (i.e. unregistered variable) that actually should be detectable at compile time. Also, this concept interacts badly with concurrency in the form of preemptive multi-tasking.

Robot Control Languages

Robot control languages are designed to bridge the gap between the deliberative layer and the reactive execution engine. They handle sequencing and parallel execution at the task-level. But they are usually designed from a top down perspective and have little conceptual connection to the behavior level.

Reactive Action Packages: An early robot control language is the "Reactive Action Packages (RAP)"[32]. A RAP represents an action that can be arbitrarily complex and structured. RAPs fits conceptually quite good to the hierarchical planning approach, in that RAPs are executed by stepwise refinement until they represent an executable atomic action (skills in their terminology that are actually continuous processes). This includes sequencing and parallel execution. There can be multiple refinements for a RAP and preconditions can be checked for the applicability of a refinement. Failure and completion of actions is acknowledged by signals that can be linked to successor RAPs. Signals that are not linked to successor states are lost, which makes it hard to intercept a RAP from a higher-level perspective.

As most robot control languages, the coupling to the reactive layer is rather loose. The problem of arbitration between reactive processes is circumvented in that incompabilities of skills can be specified to prevent them from being executed in parallel.

Saphira Tasklevel Controller: A similar design is provided by the Saphira architecture. The two robot control languages PRS-lite [89] and Colbert [64] that are available for the Saphira architecture [62] have no apparent link to the behavior framework [103] of the system (which is based on fuzzy control theory discussed above), even though they usually are needed to activate and deactivate its behaviors for different tasks.

Task Definition Language: The Task Definition Language (TDL) [119], was designed for space missions of NASA. It is similar to RAP, as it works on a task tree that is expanded at runtime. It is implemented as an extension to C++ that is compiled down to plain C++ and supported by a runtime library. Its main constructs are a set of task identifiers (Goal, Command, Monitor...) that are prefixes for a C++ global function, the spawn keyword for starting parallel subtasks and a set of constraints for synchronization of

spawned subtasks. Exception management is used for expressing failure, using similar semantics as the event and binding semantics in BAP.

Structured Behavior-Based Control

The architecture most similar to *Miro* is probably MissionLab [4], an end-user-oriented robot task-level control software, which is designed for mission specifications in the military domain. It allows the composition of higher-level agents by combining a set of (atomic) agents with a coordination operator. The reactive behavior of robots can be specified by the use of a configuration description language (CDL), for which a graphical frontend exists [77].

Coordination operators can select and fuse outputs from the agents and allow for arbitration as well as for temporal sequencing by implementing a FSA. Events are only handled directly by a coordination operator. Termination conditions like successful completion or failure of a subtask need therefore be checked on the next higher level and are not part of the agent itself. Also the approach for multi-platform support differs significantly from the approach discussed in this thesis: MissionLab uses different compiler backends to translate the specified control program for different robots.

6.3 Summary

In this chapter, the design rationale of the framework layer of the *Miro* software architecture has been discussed. Frameworks are not only a powerful concept with which to foster the reuse of code and design for concisely defined application domains. The properties of a framework-based design also make it particularly suitable for addressing non-functional aspects as need to be considered for meeting the challenges of large-scale applications. These features were highlighted in the discussion of two of *Miro*'s frameworks.

The VIP framework addresses the difficulties of meeting the requirements of the application domain when applying advanced computer vision to autonomous mobile robots in dynamic environments. It is designed to facilitate the application of computer vision in robotics by dealing with the additional challenges of robot vision in this domain. The middleware-based framework approach especially makes it possible to support roboticists with the non-functional aspects like configuration and performance assessment. The inversion of control flow and the management of data flow provided by the frameworks also allows to effectively shield the developer from the various locking issues associated with parallel data access and synchronization, and to provide a generic model for task prioritization. Extensive development support is provided in the form of parameter management, GUI-based configuration and generic inspection of images and metainformation.

The BAP framework evaluates methods to overcome intrinsic scalability issues of behavior-based robot control architectures. The key question dealt with in

this section is how behaviors can be organized in a hierarchical way. This allows for complexity reduction and for the reuse of action patterns or complex sequences of actions either within a single control system or for different scenarios. The framework-based design allows for a flexible mixture of control loop evaluation models. The middleware-provided configuration management enables graphical modeling of action patterns and policies as well as the flexible parameterization of their components and extensive development support.

This chapter contains two contributions to the state of the art in research on autonomous mobile robotics. First, a new design for video image processing in time-constrained dynamic environments was discussed. It features a control-flow-oriented design, incorporating parallel execution and proper prioritization, along with a stringent connection management, which guarantees a minimized processing tree. Second, a formal method for the design of reusable, modular patterns of robot behavior was defined, which encompasses most of the prominent behavior-based approaches. The distinct separation of parallel execution, sequencing, and modularization allows to derive features such as sets of unhandled events of sub-policies, which are used for consistency checking by the provided development tools.

Chapter 7

Tool Support

Tool support is an essential element of modern software development processes. Modern text editors provide features that reach far beyond the capabilities of classical syntax highlighting. Debuggers allow single-stepping through running code and profiling tools help to identify hot spots in program execution and can even report on cache utilization and processor pipeline stalls. In order to support this process, additional tools for developing, debugging and evaluating robotic applications are needed.

In the previous chapters, various solutions for addressing the challenges of the robotics domain were proposed. These included specialized as well as customized technologies such as extensive configurability support, logging or behavior engineering. These technologies need to be supported further to make them easier to use and to speed up the development process.

In this chapter some of *Miro*'s high-level GUI-based tools will be briefly introduced. They illustrate how robotic applications can be supported by generic tools, a situation in which the identified solutions can be more easily propagated to the developer level. All *Miro* tools have some basic design principles in common:

Visualization: Because humans are very good at visually understanding structures and relations, the tools produce an editable visualization of the problem domain. This is done by the use of standard GUI elements as well as by producing customized visual representations.

Input Checking: *Miro* makes great efforts to prevent subtle and hard to debug programming errors. This is why the tools have to provide the best support for properties like type safety and correctness of editing functionalities.

Generic Extensibility: Robotics middleware provides functionality that is to be used and extended by the roboticists. Therefore, the tools provided by the middleware layer have to be capable of extending their functionality in order to be able to cope with the applications' growing capabilities.

This can be achieved for example by genericity within the tool itself or by the use of a plug-in architecture.

Runtime Inspection: The most difficult problem in autonomous mobile robot applications is the inspection of the processes at runtime. In order to address this problem, the tools especially support the inspection of the system during operation.

7.1 Configuration Management

Figure 7.1: ConfigEditor — Type-safe editing of structured and nested configuration parameter sets.

The `ConfigEditor` supports the application of a toolkit from the *Miro* infrastructure layer, namely the configuration and parameter toolkit as described in section 4.2. Because it allows for a powerful and flexible parameterization of services and frameworks, it is used by many of *Miro*'s components and is also frequently used in *Miro*-based robotics solutions. In consequence, the XML-based configuration files can become quite large. Even though XML is a plain text format, editing big XML files without further support from the text editor is very tedious and error-prone. A number of XML editors that facilitate the editing of files in compliance with a document type description (DTD) or a Schema do exist. But the user-defined parameter descriptions are not covered by the DTD and also contain features like inline documentation, which are hardly representable within a DTD or a Schema.

The editor provides a tree view of the configuration document, its sections and parameter instances. This allows for fully type-safe editing of parameter entries and supporting all types supported by the parameter framework. This includes nested structured types, vectors and sets. The editor obtains the definitions of parameter sets by parsing the user-defined parameter description files and thus extends its editing capabilities generically to also include parameters defined in user applications that build upon *Miro*. Also, changes in the parameter descriptions such as additional parameter fields are reflected automatically within the editor. Run-time inspection and configuration of parameter configurations is a rather new feature and is therefore only supported by only a few interfaces.

The `ConfigEditor` is designed to provide a maximum of error safety and editing comfort by making full use of the information provided by parameter description files. It displays the expected type, the physical measure if specified in the parameter description, and also the provided default values as a tooltip. Figure 7.1 shows the VIP framework sample configuration from Section 6.1.3. It performs the processing steps for simple edge detection (upper window). The lower dialog shows the parameter configuration window for the Canny filter.

7.2 Graphical Policy Programming

The design and implementation of policies for the BAP framework is much more related to a programming task then to a mere configuration effort. It consists not only of parameterization of behaviors, guards and arbiters, but also includes behavior grouping, sub-partitioning of the problem domain and conditional control flow directing by transition management in an iterative edit, test, and debug cycle.

The policy editor visualizes the policy as the graph of the hybrid automaton it describes. The functionality of the `ConfigEditor` is reused for parameter handling and additional constraint checking is available for transition management. The parameter description framework is also reused for the generic description of policies and their parameters. This enables the editor to incorporate user-written behaviors. For runtime support, policies can be sent to a running robot. This allows to replace a robot's policy without having to restart its control program. This is especially helpful during debugging and for parameter adaptation.

This tool allows for graphical programming of action patterns and policies. The available behaviors, arbiters and guards can be grouped into action patterns and their parameters can be adjusted to the peculiarities of the task. Transitions can be added for sequencing action patterns in a policy. The graphical programming environment checks whether all locally produced message are bound by a transition. Modularization by means of subpolicies is also supported.

Figure 7.2 shows a small example of graphical policy programming. The window on the left shows the graph structure of the policy document, while on the right side a simple policy is visualized. The tree view on the left lists the external transitions 'Wait' and 'Play', used for stopping/starting the robot's

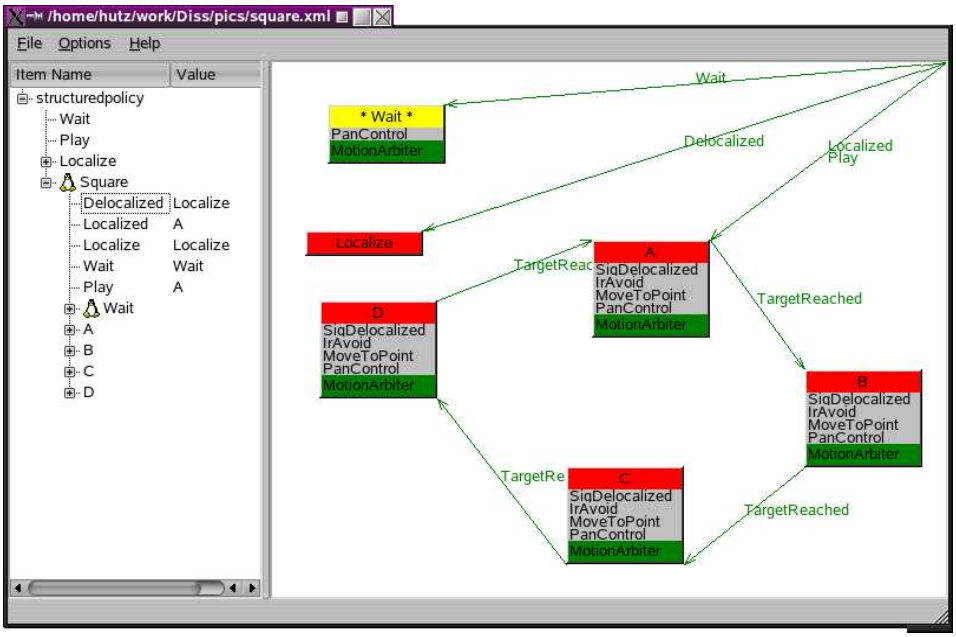

Figure 7.2: Graphical behavior programming. The tree view on the left side represents the structure of the policy document, while on the right side a policy graph is visualized. The nodes represent action patterns ('Wait', 'A', 'B', 'C', 'D') or policy instances ('Localize') and the arrows denote transition. The arrows from the upper right corner are transition patterns. The policy can be edited in both windows.

activity from an external control panel, a subpolicy 'Localize' and the main policy 'Square'. It consists of transition patterns for the events 'Wait', 'Play', 'Localized' and 'Delocalized', the action patterns 'Wait', 'A', 'B', 'C' and 'D' and a subpolicy instance of 'Localize'. The types of the entries are also displayed in the list view, but the column is scrolled out in this screenshot. The right side displays a policy graph, in this case the 'Square' policy. Transition patterns originate in the right upper corner. The start pattern is denoted by enclosing the name with two asterisks on either side. The 'MoveToPosition' behavior is parameterized with different target coordinates in each action pattern. Each time a goal is reached, a 'TargetReached' event is generated and the policy switches to the next action pattern. The guard 'SigDelocalized' keeps track of the successful localization of the robot and signals if it is delocalized by sending the 'Delocalized' event. So if the localization fails, the 'Localize' subpolicy for active relocalization is executed. Afterwards, the robot restarts the square movement at pattern A. The 'IrAvoid' behavior ensures that the robot does stay away from obstacles identified by its infrared sensors. The outputs from the different behaviors are fused by the 'MotionArbiter', a simple priority based arbiter implementing the subsumption scheme of action selection.

7.3 Log file Management and Replay

The sheer mass of log data generated by the event logging service described in Section 5.4 requires tool support for post processing, especially since the performance-optimized binary file format renders standard text processing and filtering tools such as regular expression processors useless. But on the positive side, this binary data format does not only support efficient logging of large amounts of data. One of it's primary design goals was to make it possible to replay these data logs as the original events through an event channel.

The appearance of the `LogPlayer` tool mimics a standard media player with a control panel which supports play, pause, stop, slow-motion, single-stepping and quick access to random parts of the data stream through a slider bar. An event view window allows the inspection of individual events. Generic inspection of the event payload would be possible due to the middleware-based design of the logging facility. CORBA offers interpretative demarshaling of structured data through the use of the DynAny specification. This feature, however, has not been implemented yet.

As the essential processing tool for the acquired log files, the GUI-based `LogPlayer` front end (see Figure 7.3) allows to access all the information stored within a log file. It can replay log files in a timely manner, pushing all logged events into an event channel. It can also replay multiple log files while synchronizing their time stamps as required for debugging and for the assessment of data from multirobot experiments. For log file management, functionality for extracting sequences out of log files and filtering logged events based on the domain and event name is available.

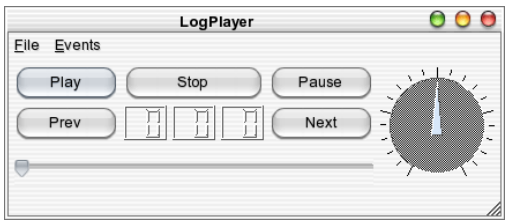

Figure 7.3: Tool to replay logged files.

7.4 Data Visualization

The data structures as communicated through the event channel, especially user application defined data types, are mostly specialized for the robot's application domain. This is why hardly any general-purpose visualization and data analysis tools exist so far. But the value of the insight provided by the event streams of structured sensor and higher-level service data nevertheless calls for an easily extensible framework for their visualization.

A powerful visualization tool, the so called Vizard was developed especially for the RoboCup domain. It offers an OpenGL-based 2D visualization of the application domain, allowing to display additional data in various dialog boxes on demand. Further features for visualization can be integrated through its visualization framework, so it would also be easily applicable to a SLAM or

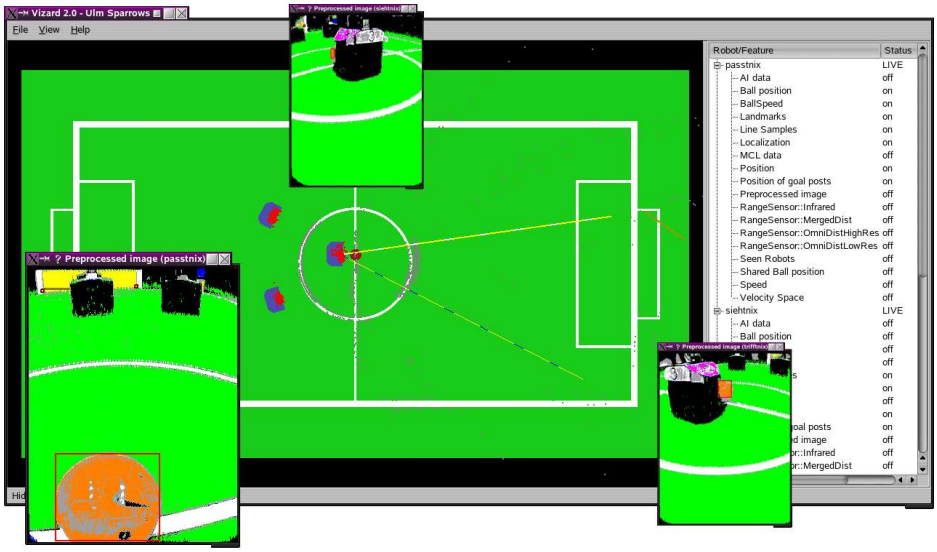

Figure 7.4: Visualization of the robot soccer team drawn up from the robots' data. The position of the ball, the free area in the opponents' goal and the visible localization landmarks are displayed as observed by the robot in the center circle.

navigation scenario, for example. Because of its strictly event-based design, it can be used for online inspection of an active robot soccer team as well as for offline visualization of multiple data streams as replayed by the `LogPlayer` introduced in the previous section.

The Vizard tool displays the sensory event streams such as the robots positions, their velocities and infrared range sensor scans as well as the results of higher-level object recognition such as ball and goal positions. Additional information like preprocessed images or the current state of the BAP engine can be displayed in additional windows. For a more detailed perspective, the display of each feature (and robot) can be turned on and off independently. Figure 7.4 shows a pre-kickoff situation at RoboCup. The main window on the left displays the football field and the visualized data. The tree view on the right allows to select the various features for display. The three overlaid windows show the preprocessed images of the three robots. The big window at the bottom left belongs to the field player in possession of the ball, all observed features in the image are marked by little boxes and are also displayed in the main window: The ball is directly in front of the robot. The directions of the left goal post and the right corner flag (which represent landmarks for localization) are displayed by the yellow and blue and by the yellow and white lines respectively. The yellow bar in the opponents' goal visible in the image represents the biggest free area observed in the goal. The field lines observed by the three robots (which are also used as localization features) are displayed as gray dots in the main window. The red dots represent position hypotheses created with the applied sample-based Markov localization method.

7.5 Summary

In this chapter a selection of GUI-based tools has been presented. They support the application of provided functionalities from the software architecture's different layers.

The `ConfigEditor` enables type-safe, generic editing of configurations, thus preventing subtle errors like misspelling of parameter names. It also immediately propagates any change in the configurability of components to the user level. The `PolicyEditor` provides graphical modeling capabilities for action patterns, transitions and policies, and supports a rapid develop-test-cycle, by utilizing the online configurability of the BAP implementation of robots in operation. The `LogPlayer` provides capabilities for the maintenance and replay of event stream acquired during robot experiments for visualization, evaluation and learning purposes. The `Vizard` application provides a framework for the visualization of event streams for online and offline inspection of multirobot experiments such as RoboCup tournaments.

Such tools are necessary stepping stones for easing the applicability of the advanced features of a robotics software architecture in end-user applications. They are therefore an integral part of the design on which *Miro* is based.

Chapter 8

Experimental Evaluation

An extensive evaluation of the functionality implemented in the *Miro* architecture is essential in order to assess the actual benefits provided by the solution approach. However, it is difficult to experimentally verify design goals such as reusability and scalability of the system as a whole. Additionally, qualitative measures like reusability are difficult to verify quantitatively with justifiable effort.

The evaluation of the discussed architecture has therefore been split up into two chapters. This chapter focuses on an empirical evaluation with respect to runtime properties achieved by the design and deliberately omits issues that are difficult to quantify within the scope of this thesis. The next chapter will present some of the proposed architecture's qualitative results as derived from various projects and applications based upon the *Miro* architecture.

In this chapter, assessments of selected functionality from the different layers will be presented. On the infrastructure layer, the runtime performance of the communications infrastructure as well as its scalability is of primary interest. On the service layer, the efficiency of the logging service with respect to runtime and space will be evaluated. On the framework layer, the support of the different application sub-domains in their non-functional requirements is of concern. Therefore, the VIP framework's ability to adhere to QoS requirements such as prioritization will be assessed. As the device layer is not part of the scope of this thesis, it has been deliberately omitted from empirical evaluation as well.

8.1 Communication Overhead

A question often raised within the robotics domain is the overhead introduced by large scale distributed systems technology such as CORBA. We evaluate these questions with two experiments. First, the impact of inter-process communication on the latencies introduced by the physical devices of a robot. Second, the general performance of the employed CORBA middleware is discussed.

8.1.1 Device Latencies and Inter-Process Communication

In a first experiment we tried to quantify the impact of the communication infrastructure on sensor/actor-based feedback loops, typical for behavior-based reactive control.

High performance and real-time conformance are still not really an issue for many commercially available mobile robot platforms, due to the limited capabilities of their hardware. The Pioneer controller board, as well as the B21 (pre-rFlex) motor controller are both attached to a PC via a slow serial link. Furthermore, they do report their status to the PC only in 100 msec intervals. The odometry resolution is in each case about 1cm. Therefore, a basic response test, that is the time between issuing a motor command and the reflection of the robots movement within an odometry reading is heavily dominated by the latencies of the low level controller communication and can hardly reflect the performance of the actual software architecture. Of the robots in our lab, the Sparrow-99 controller board seemed most suited for performance measurements. It is attached to the PC via a 1MHz CAN-Bus and, with currently used firmware, capable of reporting odometry updates at 100Hz.

The basic response test was designed as follows. A motion command was issued on a still standing robot and the time was measured till the first odometry reading was received, which indicated that the robot had moved. Afterwards the robot was halted randomly between one and two second, before the next iteration of the test. We used three implementations of a basic response test. The first was calling into the device layer directly. The second was invoking the methods via the CORBA method interface, actively waiting for the next odometry measurement. The third one was setting the velocity via a CORBA method call and evaluating the odometry messages pushed to it by means of the event channel. The CORBA-based tests were run on the same machine as the sensor/actor services, but as separate tasks. Table 8.1 shows two runs of each test with 100 iterations. The runs of the different implementations were interleaved to compensate for the decreasing battery voltage.

response time in μsec	min	mean	max	var
raw Response	4789	12519	20722	1878
	14613	20807	27164	4543
poll Response	5459	13476	21257	6208
	10176	15660	22634	5321
notify Response	8372	14115	21661	5060
	8511	15091	21917	6350

Table 8.1: Results of basic response test.

Looking at the performance stats of the CORBA implementation used within *Miro* [94] one has to expect that even on this platform the basic response time is still heavily dominated by the 100Hz update cycle of the odometry reports. By doing a random wait before issuing the motor command, this latency is

on average half an update cycle and should therefore contribute 5 msec to the averaged latency. Nevertheless the remaining jitter, is much too high, to make the existing latency of the CORBA overhead measurable. Only an increase of the variance could be measured. But note that the variance itself is also heavily biased by the 10 msec cycle of the odometry events.

8.1.2 CORBA Performance Considerations

It lies beyond the scope of this thesis to provide a detailed performance analysis of the employed CORBA implementation. But as the performance of this communication infrastructure is often provided as the standard argument against its application in this domain, we will shortly discuss some of the relevant aspects.

Generally speaking, quantifying the overhead introduced by CORBA is a very difficult task. The first counter question would be: Quantifying the overhead compared to what? Raw TCP/IP surely can provide a better bandwidth utilization and therefore higher throughput. On the other hand, DOC technology provides many features such as marshaling of structured data, interoperability, hosting of multiple objects in one remote server and colocation optimizations that are not available for most ad hoc solutions. Additionally, in many application, throughput is less important than scalability or the adherence to QoS requirements etc.

In [39], Gokhale and Schmidt compare the throughput of the standard CORBA transport protocol of the TAO CORBA implementation for various data types against a raw TCP baseline. The results vary between almost as good as TCP and about 30% of baseline TCP throughput. The great variance in the results basically confirms both. The general applicability of CORBA, especially in high-performance environments, as well as the occasional need for specialized optimizations, to meet the requirements of some application domains.

An important aspect that is often totally missing in such a discussion is the potential for generic optimizations on the transport layer. E.g. if the client and the server reside on the same machine, or within the same address space, high performance ORB architectures allow to (automatically) switch to other, optimized transport protocols, without the need for any changes on the client side interfaces. Such functionality is hardly provided by hand crafted or small scale solutions such as IPC. The effect of this feature is illustrated in Figure 8.1 for the TAO ORB deployed by the *Miro* project. It shows the throughput numbers of the standard 'Cubit' performance test provided by TAO. This test is for instance used in the performance evaluation in [114]. The test calls a CORBA object with different parameter types, resembling a representative cross section of typical CORBA parameter data such as `void`, `sequence` and `struc` types. The `struct` contains an `octed`, a `long` and a `short`. The short sequences contain 4 bytes, while the long sequences contain 4 KB of data. The tests are all run with clients and servers colocated on the same 1GHz AMD K7 machine with 512MB RAM.

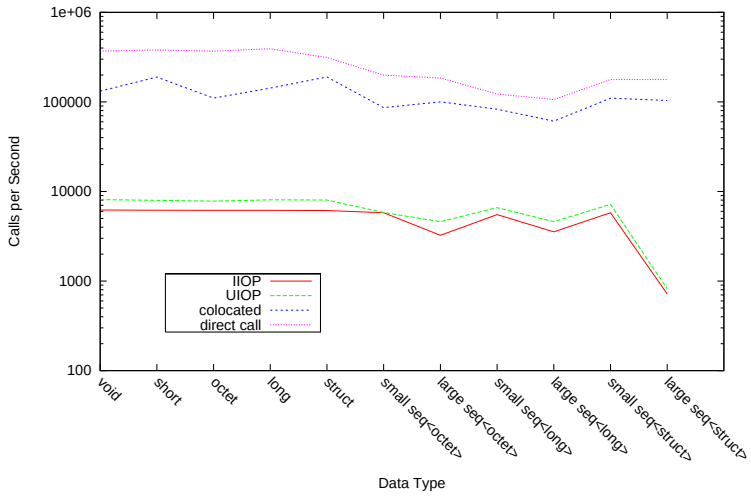

Figure 8.1: Throughput with different colocation optimizations.

We compare the standard IIOP implementation against TAO-specific protocols, optimized for colocated clients and servers on the same machine and the colocation optimizations applicable, for clients and servers that run in the same address space. The different transport protocols are: 'IIOP', this is the standard transport protocol using TCP. 'UIOP', which is a specialized protocol provided by TAO, using named pipes. Note, that this protocol is based upon a standard extension hook of the CORBA architecture. So other ORB implementations can provide similar or additional optimized transport protocols. 'Colocated' denotes the test results for clients and servers located within the same address space. Direct is an optimization that further omits some ORB-features and is therefore not applicable to all CORBA scenarios. Note, that the throughput numbers are plotted on a logarithmic scale.

The UIOP througput is on average 25% higher than the IIOP baseline. The large sequence of structs shows the impact of marshaling complex data types on the overall performance of IIOP and UIOP. The colocated configurations show a tremendously (10 to 100 fold) better performance. As the parameters need not be marshaled/demarshaled in this configuration, the performance degradation for the large sequence of structs is not observable. These experiments illustrate, that modern object-oriented distributed systems techonology allows to provide generic network transparency without geopardizing performance for none-distributed use cases.

8.2 Group Communication in Robot Teams

In Section 4.1.5 the design of a group communication facility in robot teams was discussed. It features a federation of the standard CORBA notification

service event channels by the use of IP-Multicast that automatically exchanges events of types requested by remote consumers within the federation. In this section we verify the design decisions by an empirical evaluation.

To assess the performance of the IP-Multicast-based federation we compare it against two standard configurations of the notification service. For this purpose, a fixed payload is sent through the event channel to a number of consumers and the throughput is measured for different configurations of the experiment. In each experimental run, a number of 1.000 events is send through the event channel and the time for sending each event is measured. In the different runs of each of the configurations, the number of clients is increased. The three configurations of the experiment use different locations for the consumers.

Local consumers: In the first configuration the consumers are colocated with the supplierers and the notification service on the same machine, and use the standard IIOP protocol for communication. This is by far not the most performant configuration of the notification service, but it provides a setup with very deterministic timing characteristics as a base line. A detailed analysis of event channel througput with supplieres and consumers colocated in the same address space with the notification service is implicitly given in the discussion of the performance of the logging service in Section 8.3.

Remote consumers: The second configuration is similar to the first configuration, but the consumers reside on remote machines in a WLAN. This introduces a great source of noise in the performance measurements of the experiment, but resembles the actual target environment. Also, the absolut throughput numbers are not of primary interest in this experiment. The focus lies on the relative performance numbers with different numbers of consumers.

NMC: The third configuration resembles the NMC-based federated event channel configuration in the same WLAN. The consumers reside on remote machines, but are connected to their own local event channel. The channels exchange the events through the multicast group.

The experiments were conducted on the robots of the The Ulm Sparrows RoboCup team. The four robots are connected through IEEE 802.11b standard WLAN with 11 MBit/s. The three field players have 512 MB RAM and Pentium Centrino 1.4 GHz processors. The goalie and the spare notebook are equipped with Pentium III 1.2 GHz processors. The payload of the events is a raw byte array of 1K. To minimize the variance in available network bandwidth, the five laptops were the only machines on the WLAN. The tests were run using the round-robin real-time scheduler to minimize jitter.

The results are illustrated in Figure 8.2. They show an average throughput for the local consumers configuration of about 4,162.75 $events/s$ with one consumer, corresponding to 867.90 $events/s$ when this consumer resides on a re-

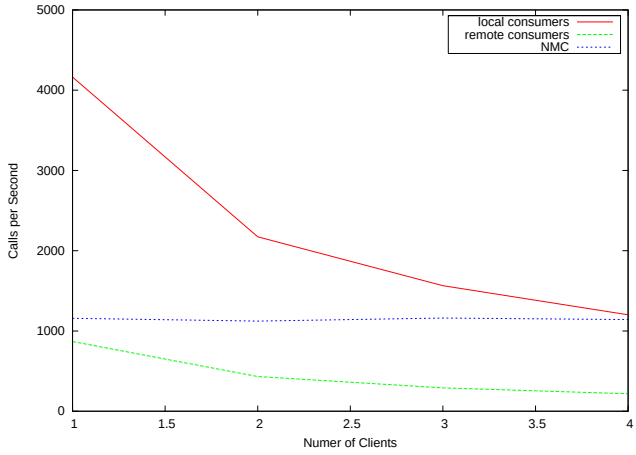

Figure 8.2: Event channel throughput in different configurations.

mote machine in the wireless LAN. The throughput of the event channel decreases as expected with a growing number of consumers. The NMC-based federated event channel setup shows an almost constant throughput of about 1,150 $events/s$, with an increasing number of consumers. In these last series of experiments a package loss of about $5 - 10\%$ was measured. The NMC-based setup performes even better than the second configuration with just one client. This is most probably due to the fact, that IIOP by default requires a response from the server to acknowledge the successful processing on the server side and retransmissions of data due to packet loss. The NMC-based configuration almost draws level with the performance of the 'local consumers' setup with 4 consumers.

8.3 Efficient Data Acquisition

Logging should cause as little computational overhead as possible in order to avoid jeopardizing the performance of the overall system. This section gives an overview on the performance of the *Miro* event logging facilities. The time as well as the space efficiency of the logging service are of concern, due to their impact on the system performance when acquiring data from robot experiments. Therefore, the performance is tested under three different aspects: *i)* time needed to log an event by the logging client, *ii)* time needed to send an event across an event channel and log it, and *iii)* size needed to store an event in a log file. In the experiments, events with different payload sizes were used, resulting also in different marshaling complexity, and each run logged 10,000 events. The different payloads were:

None An event with no payload. This gives an indication about the basic overhead for logging an event.

Octet1K Octet sequences are unstructured byte sequences in CORBA. They imply only minimal marshaling overhead. The payload size is 1KB.

Octet100K Same as before, but with 100KB payload. This comparatively large payload size tests also how the performance of the notification service scales with event size. Because storing 10,000 events of 100 KB each would result in a log file of $>$ 1GB, only 1,000 events were logged in this test.

Odometry A small event with fixed size (44 bytes), but containing a nested data structure. This event is actually generated by an odometry service of *Miro*.

RangeSensor An event which mainly consists of a variable-size array of integer values, as for example sent by a range sensor like a laser scanner. For this test, the structure is populated with 361 distance measurements, resulting in a payload of about 1460 bytes.

SharedBelief The `SharedBeliefState01` is a fairly complex structured data type consisting mostly of a variable-size array of variable-sized arrays. It is used to exchange beliefs states about the environment in the RoboCup scenario between robots [141]. In our case, the data structure was populated with 17 objects on the field, resulting in a data size of about 1260 bytes. This test addresses especially the marshaling overhead of complex data types.

The test was run on a computer with an AMD K7 1GHz CPU, with 512MB RAM, and running under Linux 2.4. The CORBA implementation used was TAO version 1.4. To minimize the influence of scheduling latencies on the timing results, the tests were run with the round robin real-time process scheduler. Note that the timing results have to be interpreted with care. As the timings are in the μsec range, cache hit rates and related factors potentially have a significant influence on the test results.

8.3.1 Runtime Performance

Table 8.2 shows the timing results for writing an event to a memory-mapped log file. The average time for writing an event to a log file ranges from 4 μsec to 595 μsec depending on the size of the events. Note that the median is always smaller than the average value. The large maximum values are probably caused by file buffer flushing operations. Access to hard disks is generally a problem in a real-time setting. Using a memory-mapped file has advantages over a plain memory buffer, as the operating system saves the logged data even in case of a program crash. However, limitations imposed by the operating system have to be kept in mind when operating on really big chunks of data. The min values actually represent the raw marshaling overhead for data serialization.

Marshaling of the SharedBelief data structure requires 40 μsec on average. This shows the penalty of serializing complex structured data types in contrast to flat

performance in μsec	min	mean	max	std.dev.	median
None	2	3	319	30	3
Octet1K	3	8	351	439	4
Octet100K	505	595	979	2621	580
Odometry	3	6	398	223	4
RangeSensor	4	12	357	646	6
SharedBelief	28	40	349	872	32

Table 8.2: Performance of the logging client.

performance in μsec	min	mean	max	std.dev.	median	delta
None	30	33	525	67	32	30
Octet1K	32	39	565	497	34	31
Octet100K	578	681	1292	3230	678	86
Odometry	32	37	680	284	34	31
RangeSensor	34	44	576	699	38	34
SharedBelief	59	73	583	935	64	33

Table 8.3: Performance including event channel overhead.

arrays (8 μsec per 1K). As an array of variable-sized arrays is implemented as an array of pointers to arrays, this results in copying about 20 memory regions independently allocated on the heap. On the other hand, higher-level cognition data structures like the SharedBelief are likely to occur less frequently than low-level data events.

Table 8.3 shows the timing results for distributing an event via the event channel and processing it by the logging consumer. It therefore encompasses the timings presented in Table 8.2. The last column, delta, denotes the difference of the mean values of tables 8.2 and 8.3. The event channel overhead seems to be almost constant between 30 and 35 μsec, with the exception of Octet100K, where the extra overhead is likely to be attributable to parameter copying within the CORBA event channel implementation. This is definitely not negligible but tolerable, given the rich feature set of this service. Also, as the logging service normally is not the only consumer of an event in the application, the event channel processing for logging events will not be the full 40 μsec either.

8.3.2 Memory Footprint

The memory overhead of the log records is listed in Table 8.4, columns three and four. As the numbers show, it can be significant. The empty payload test yields an overhead of about 48 Byte. 8 are used by the time stamp, another 9 for the domain name (robot name) and type name fields, about 20 for the fixed size part of the event data structure and the rest is used up by data alignment.

[1] On 1,000 events only.

		without Type Code Rep.		*with Type Code Rep.*	
storage footprint	payload (bytes/event)	file size (bytes)	overhead (bytes/event)	file size (bytes)	overhead (bytes/event)
None	0	480.000	48	520.012	52
Octet1K	1024	11.320.000	108	10.840.064	60
Octet100K[1]	102400	102.508.000	108	102.460.064	60
Odometry	44	5.840.000	540	1.000.508	56
RangeSensor	1460	18.520.000	392	15.160.352	56
SharedBelief	1268	28.799.996	1612	13.241.576	56

Table 8.4: Logging storage footprint for 10,000 entries.

The 60 additional bytes of overhead for the Octet1K payload are caused by encoding the data type. For every event, the complete type description of the payload is stored. This is unfortunate, as usually a small set of event types are sent hundreds or thousands of times via the event channel. The type code can easily exceed the size of the actual payload for small payloads like odometry events. Also the type description accumulates to more than 1 KB for the `SharedBeliefState01` data structure. Even though this does not seem to cause a significant performance overhead, the size overhead is still significant. Unfortunately, the use of complete type codes is hard to avoid in a generic logging facility based on the `CORBA::Any` data type. But, there is no need to store type codes repeatedly within the log file. So rows five and six show the file size and overhead achieved by using a type code repository that is prepended to the event stream and contains the needed type codes. The type code in the log record is replaced by its id in the repository. This way the size overhead can be kept constant and within an admissible range. The use of the type code repository yields the same runtime performance within the precision of measurements.

8.4 Prioritized Video Image Processing

A critical part of robot vision is the timely processing of image data. The VIP framework does not try to provide faster implementations for standard image operations, as sufficient libraries for this purpose exist. These can be easily utilized for the use by VIP, as done for IPP in our applications. Instead this framework concentrates on improving the responsiveness of a vision application, by allowing for proper priorization and synchronization of image processing tasks with parallel and asynchronous control flow.

A typical use case for the processing of multiple filter trees, is the combination of a fast path with an asynchronous slow path of vision processing, which then needs correct priorization. We therefore assess in this section the capabilities of the framework to correctly preserve processing priorities under high-load situations.

Processing tree	Prioritized		Unprioritized	
	Mean	Std. Dev.	Mean	Std. Dev.
Fast path only				
fast path	7,17	0,035	7,18	0,052
Medium load				
fast path	7,22	0,017	7,26	0,479
slow path 1	30,46	25,197	8,32	10,000
High load				
fast path	7,22	0,025	8,55	3,192
slow path 1	53,31	69,921	60,77	94,601
slow path 2	57,66	5,065	56,84	5,240

Table 8.5: Different timing statistics for the individual processing trees in both, the prioritized and unprioritized case. The values are stated in μsec.

The typical scenario would be one camera-synchronous processing tree that runs at full frame rate and extracts sensory information for the reactive control module and one or more asynchronous processing trees, that are connected to the data flow of the first tree and perform time-consuming computations not possible at full frame rate, extracting information for higher-level cognitive processes with relaxed timing constraints.

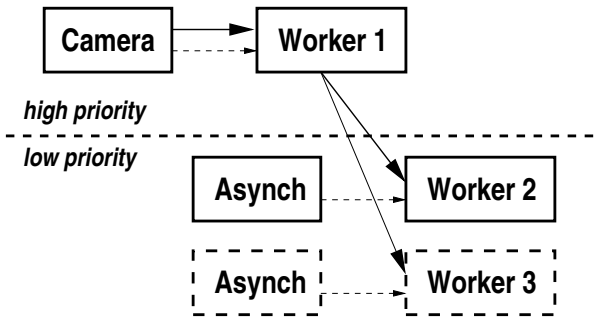

Figure 8.3: The filter configuration of the experiment.

The configuration of the VIP module for this experiment consist therefore of one high priority tree with the camera as source node, running with a round robin real-time scheduler (the fast path) and one, resp. two low priority asynchronous processing trees that are connected to the camera tree, running with default priority (the slow paths). The configuration is illustrated in Figure 8.3. The low priority load is increased incrementally in the experiment. In the first run, the synchronous processing tree is run alone. In the second run one low priority processing tree is added to the configuration, but still all processing threads can be completed at frame rate (30Hz). In run three a second low priority tree is added and the system load reaches saturation. The results are compared against the equivalent setup without priorization.

Table 8.5 shows statistics on the overall time, the different processing trees need for completion. In the unprioritized configuration, the completion time of the camera-synchronous tree drops significantly in the third configuration, as the thread is preempted before completion to perform work on the other

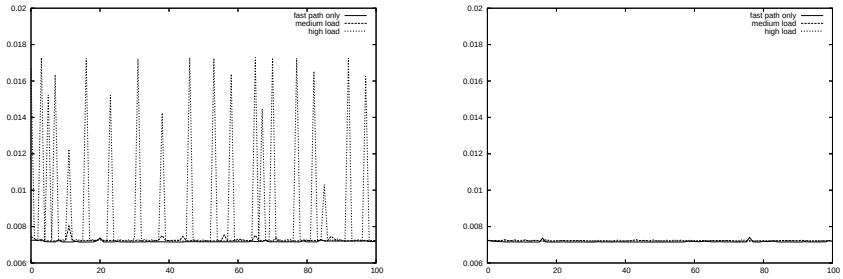

Figure 8.4: Timings of the fast path with none, one and two slow paths running concurrently. The left plot shows the unprioritized case, whereas the right plot shows the fast path running with enabled real-time scheduling.

processing trees. This would cause significant delay for the consumers of this sensor information (e.g. the reactive control unit).

In Figure 8.4 this effect is illustrates by plotting the individual timings for 100 runs of the fast path. It can be clearly observed that while the prioritized processing three still runs with predictable completion time, the timings of the unprioritized configuration worsen significantly under high load.

Another visualization of these preemptions is shown in Figure 8.5. From the third setup a small section of the interleaving processing of the three processing trees is plotted. Each tree is assigned a different color. Yellow was chosen for the fast path, the slow paths are colored red and blue. To fit into the column, a new line is added each time the processing of both slow paths is finished. The completion of a processing tree is marked with a black box at the end of the colored bar. While the real-time scheduled fast path always runs to completion before its processing stops, it is occasionally interrupted without priorization. Additional load on the system will worsen this effect. A medium complex robotics application performs many other tasks in parallel to image processing, which will contribute to the latencies in high load situations.

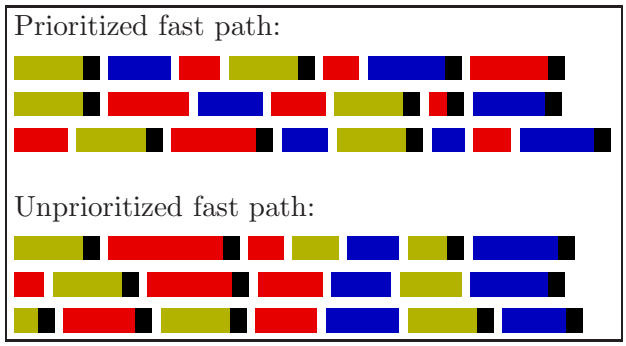

Figure 8.5: Illustration of alternation between the different running processes with a prioritized resp. unprioritized fast path.

8.5 Conclusions from Experimentation

This chapter has provided an assessment of various runtime properties of the discussed robotics software architecture. A set of experiments that evaluate the situatedness of the resulting implementation in terms of efficiency, scalability and the adherence to QoS requirements was presented.

In a first experiment, the performance-overhead of the CORBA ORB was compared with the latencies of the robot's physical devices. It was clearly shown that in this respect the abstraction penalty of the communications layer can be considered negligible.

Afterwards, a short discussion of the performance of CORBA as such was provided and its generic optimization capabilities were experimentally verified using a standard test tool provided by the ORB implementation. These generic optimizations allow to take advantage of the configuration of the distributed system without the need for any code changes. Such features are mostly missing from small-scale communication packages or hand-crafted ad-hoc solutions.

The scalability of the suggested design of group communication facilities in robot teams was verified in the third experiment. The evaluation of the IP-multicast-based event channel federation could clearly show the improvement in scalability over a standard "off the shelf" configuration of the notification service. The multicast model provides a uniform performance unrelated to the number of remote consumers.

The efficiency of the logging service is of importance, since the acquisition of data from the robot under real-world experiments must have very little overhead. Otherwise the overall behavior of the system under observation would be altered. The runtime overhead and its space efficiency could be shown to be low enough to obtain valid traces of real-world data from logged experiments.

In the last experiment, the support of QoS issues provided by the architecture for end user applications based upon the framework layer was verified. The prioritization of the VIP framework's different processing trees allows for a clean separation of evaluation tasks on image streams that perform on different time scales and with varying response time requirements.

The discussed set of experiments was able to confirm the design decisions taken to ensure efficiency, scalability and predictability in the implementation of the various sub-systems of the proposed robotics middleware architecture.

Chapter 9

Results and Applications

Miro is being successfully used in several projects striving for autonomous mobile robot control. In the different scenarios, the robots are required to perform very diverse tasks. In the office scenario, the software architecture's services were used for research on neurosymbolic mapping of indoor environments [24], hybrid multi-representation world modeling [68, 69], autonomous self-localization based on the Monte Carlo Localization method [25] and hierarchical path planning, reactive execution and human-robot interaction through web-based user interfaces [35]. The MirrorBot project applied the architecture in research on biomimetic learning processes such as the sophisticated neural processing of sensor data [88, 29] or visually guided docking [147]. In the RoboCup scenario, it was applied for research on localization based on sporadic features [139] and robust, real-time constrained image processing [81, 79, 80].

Meanwhile *Miro* is also used by other laboratories for the development of robotics applications. The RoboCup teams of the TU Munich and TU Graz use *Miro* for the control of their soccer robots [34, 127] as well as for team communication. At the TU Chemnitz it is adapted for usage on a commercial as well as a custom built autonomous outdoor robot platform [71, 142].

In this chapter, we will take a look at the results achieved within the architecture itself and at applications of *Miro* and how they benefit from the software concepts and technologies applied to the discussed architecture. We will summarize our experiences and discuss how well (or not-so-well) *Miro* does achieve the design goals.

9.1 Ensuring Software Quality

In order to ensure the longterm quality of a large scale project such as *Miro*, additional precautions have to be taken. This is especially true, as many of the well known methods for ensuring implementation quality (such as automated regression tests) are difficult to apply in this application domain.

Testing is the central method for ensuring software quality issues for the *Miro* project. This is due to the experimental nature of the approach that makes it difficult to model software properties formally in a top down manner. Also the challenging properties of the application domain such as the tight coupling of the lowest layer of *Miro* to the peculiarities of its controller boards, which often lack a detailed specification, favor this approach.

For each robot platform and service abstraction a set of related tests exists within the *Miro* sources. The test programs for robot platforms offer access to the provided functionality entirely by the use of the device layer. Test programs for service abstractions allow to test each method of the interfaces and also provide test clients for the events supplied by sensor services. Unfortunately this can not be generally modeled in the form of automated (regression) tests. As the robot operates in the real world, sensor readings are dependent on the environment the robot actually is positioned in. Actuator movements may not even be possible, without endangering the robot or its environment. Simulation is of little help for testing the device layer, as a detailed model of the low-level device characteristics, like non-deterministic failure conditions is at least as difficult to verify as the correctness of the device-layer implementation. The service layer implementations can be easily verified by comparing the output of the device-layer tests, against the results from the service abstractions tests.

For functionality such as configuration management or BAP policy parsing, sets of test files together with the expected output are provided. For higher-level frameworks test programs do exist too, to ensure the correctness of the provided infrastructure. Additionally, the logging capabilities introduced in Section 5.4 provide powerful capabilities for the testing and verification of target applications as covered in Section 9.5.

As *Miro* is meanwhile developing into a community project, patch reviewing for external contributors ensures the quality and consistency of the implementation with respect to design principles and coding standards.

9.2 Available Platforms

Presently, *Miro* supports all mobile robot platforms at the robotics lab of the University of Ulm. These are equipped with different sensors, actuators, and computational equipment. They are used in different scenarios, ranging from an office delivery to highly dynamic soccer games [28]. These platforms are:

- A B21 robot (see Figure 9.1, back row, right one), which is equipped with bumpers, IRs, sonars, a laser range finder, and a vision system. It features a synchro drive mobile base, and speech synthesis and is controlled by two on-board PCs.
- Pioneer robots from ActiveMedia. These are a family of platforms that are all based on the same low-level controller protocol. The lab owns

Figure 9.1: The robot platforms of the Ulm robotics lab.

the Pioneer-1 (Figure 9.1, front row, left robot) and the PeopleBot robot (back row, left one). But other robots such as the Pioneer-2 and the Pioneer 2 AT for outdoor scenarios are also controlled by the use of *Miro* at other labs. The base version is differential drive platform equipped with sonar sensors only, which is controlled either via a laptop mounted on top of a robot or by a host PC via a serial radio link. The various configurations options include a gripper, cameras with pan-tilt units or laser range finders.

- Sparrow-99 robots [101, 138], which are custom-built robots developed in our lab at the University of Ulm (Figure 9.1, front row, middle one). Sensors include sonars, IRs and a camera. The mobile base is a differential drive system. The robot also has a pan unit and — for its special purpose — a kicker. It is controlled via an on-board embedded PC.

- The Sparrow-2001 robots (Figure 9.1, front row, right) represent the second generation of custom-built soccer robots [67]. They are not only faster and equipped with bigger PCs. They also provide a different sensor suite. The sonar sensors where dropped and an omni-directional camera was added, the directed analog camera was exchanged by a digital camera, with wider viewing angle and higher resolution. Also the pan-unit was exchanged for a faster, more precise successor. The low-level controller boards and the chassis were completely exchanged.

9.3 Behavior Engineering

The BAP framework is used successfully in a robot indoor navigation office scenario as well as in the highly reactive robot soccer scenario of the middle-size league of RoboCup. It is the most high-level application framework of *Miro* and therefore gives a good impression on the fulfillment of the design goals such as reusability and scalability for robotics applications.

9.3.1 Reusable Components

Miro facilitates reuse by its interface abstractions for sensor and actuator devices. This allows to write generic behaviors, guards and arbiters for different robot platforms. On a second level the reuse of action-patterns and policies is enabled by the modularization enabled in their specification.

This allows for highly interoperable applications, as demonstrated in the RoboCup scenario by soccer robots of the The Ulm Sparrows team. During the transition from the Sparrow-99 platform to the new Sparrow-2001 robots the The Ulm Sparrows played with a mixed team of both robot platforms. Those used exactly the same BAP setup, with a configuration that only differed slightly in some parameters. Note that the behaviors weren't written initially with portability or reusability in mind. The interface abstractions resulted in a natural genericity of behavior implementations and the metainformation provided by interfaces could be exploited to adapt the behaviors and guards to most of the peculiarities of the different platforms. So only little parameters, such as maximum velocities for specialized maneuvers needed to be adapted in the policy configuration.

9.3.2 Specifying Complex Tasks with BAP

Scalability is also addressed on the implementation, as well as the configuration level. Behaviors and guards can be based on different processing models, timer-based, event triggered or concurrent in their own thread of control. On the configuration level the provided tool support ensures the maintainability of complex policies and the hierarchical decomposition allows for a task-oriented partitioning of the behavior.

The power, expressiveness and simplicity of the BAP method for specifying complex task for autonomous mobile robots is best illustrated by a real world example. The most sophisticated use case for the BAP framework so far is its application in the RoboCup environments. In the 2003 version of our soccer code, a flat policy describing the complete behavior of a field player consisted of 13 behaviors which were used by 20 action patterns. An action pattern is typically composed of 4 to 5 behaviors. The policy graph was connected by 70 transitions.

As the interaction of the robots with the referee is very limited at the moment, and the strongly reactive nature of the scenario imposes little need for other external event sources like higher-level reasoning systems, the number of external transition messages is limited to 4 within this policy: Kickoff, Opponent Kickoff, Stop and Formation. These are handled by transition patterns.

Although a graph of 20 nodes and 70 edges seems not to be very impressive, it is large enough so that it becomes very difficult to fully grasp its behavioral logic and to maintain and extend it. The flat policy can be naturally structured into 7 different subpolicies, that represent the different tasks of the flat policy. An 8th policy is actually a second instance of one of the 7 policies reached from a different context. The hierarchical grouping also allows to define a previous transition message as a transition pattern, as it always has a uniform successor pattern in the different task contexts. As the hierarchically structured policy has the same semantics as the flat one, the number of concrete action patterns and transitions is actually the same. Nevertheless the savings in the specification are obvious. Three action patterns could be saved by the reuse of one single subpolicy. And various transitions were either replaced by transition patterns or could be better structured by grouping them within subpolicies.

9.4 Advanced Image Processing

The VIP framework implements a very precise model for organizing robot vision applications. This does not in any way limit its applicability, though. To the contrary, it provides a flexible, scalable framework for the implementation of reusable image processing components.

It is successfully in use in different robotic scenarios such as biologically motivated neural learning and object classification [29] and reliable high speed image processing in the RoboCup mid-size league [61, 79, 80]. It also provides the basis of a large, reusable filter library shared between the different scenarios. This `libVideoFilters` currently contains about 100 filters, which range from general purpose image transformations such as scaling or edge detection, over ROI selection and ROI tracking up to neural object classifiers. To ensure efficient implementations of standard image operations, the IPP library [56] is used. It's application in RoboCup consists of a dual camera setup, combining a directed camera for object classification with an omni-directional camera for obstacle avoidance and near range ball tracking. The application combines 66 filters with 108 connections. One of the fastest path, a simple color-based football goal detection takes around 4 msec to complete, while one of the slowest paths (a complete neural robot classification) needs around 20 msec on average when seeing one robot per image (measured on a 1.4 GHz Pentium M processor).

Currently, priorization of and synchronization between processing trees is not yet used by the RoboCup application. The use of an omni-directional camera as well as the real-time features of the framework were both added fairly recently. But the promising results of Section 8.4 will definitely encourage their

prompt application.

9.5 Logging Applications

The power and flexibility of the presented logging service is best described by presenting different scenarios which exploit recorded robot data.

9.5.1 Learning-Data Acquisition

In the first example, the logging facility was used to collect data for learning a dribbling behavior for our soccer robots. The robots can "dribble" the ball by pushing it with their body towards a given target point on the field. The dribbling task is quite complex: we know neither the robot's nor the ball's precise physical properties and constraints. Therefore, it is very difficult to model the dribbling behavior by hand. But if suitable training data is available, an artificial neural network can be trained instead to learn the required input-/output mapping function. However, for supervised learning methods, large sets of high-quality training data are necessary. Sometimes such training data can be generated using a simulator, but in our case a simulator modeling the fine-grained physical dynamics and interactions between the robot and the ball was neither available nor easy to build.

The *Miro* logging facilities were used to collect the required data sets for neural network training directly on the robot. The robot was dribbling the ball while being moved around either using a joystick or by pushing it by hand. During robot motion, the relevant sensory and motion data were logged. It turned out that the dribbling task is very difficult to achieve even for a human pushing or joysticking the robot, and the logged data contained long sequences of data useless for learning purposes, because the robot had lost the ball and needed to catch it again before performing another dribbling sequence. The log files were therefore inspected in slow-motion using the `LogPlayer` in order to identify and extract subsequences usable for learning.

9.5.2 Multirobot Debugging

Debugging robot programs by just observing their behavior is extremely difficult, and sometimes impossible. Correlating a particular robot behavior to the program parts causing this behavior without knowing which program parts are actually executed and what the values of certain variables are is very hard, and often guessing. Finding the reason for a program failure, however, is almost impossible, especially in a distributed setting involving multiple robots. In order to track down the causes of misbehaviors and identify potential failures, the programmer needs to track the robot's observations, its global position data, internal states and even information communicated from other robots. The *Miro* logging facilities allow to generate and record such information either by

Figure 9.2: Old and new localization parameterization in direct comparison.

logging the required data separately on each robot or by using a centralized remote logging sink.

We used these capabilities to debug the "shared belief" functionality on our soccer robots. Each robot collects the observations (together with their uncertainty values) from each of its teammates and tries to infer a coherent world state from these informations. The amount of data shared between robots is carefully tuned in order to not overstress the wireless network connection, especially in tournament situations. Both logging configurations (remote logging of multiple robots and distributed local logging) are possible and have been used, but for practical reasons we use distributed local logging most of the time.

9.5.3 Performance Comparison

During program development in a research setting, it is often necessary to compare different algorithms, or different parameterizations of the same algorithm, in order to assess the progress being made. Logging facilities are an indispensable utility for this task.

In the example in Figure 9.2, we used logged data and the supporting tools to assess the improvements achieved by using some new features in our particle filter-based self-localization framework [139]. We first collected a couple of log files using our original, older version of self-localization. The log files included odometry values, observed objects and the localization data itself. By replicat-

ing the logged sensor information under another robots name for the improved localization, we could visualize the old, logged localization data simultaneously with those generated online by the new approach. Thereby we were able to compare how the new method behaves in comparison to the old one. By this direct comparison, it was for example easy to see how fast the position estimates converge in each approach and to determine how many particles are needed in the new method in order to achieve an equivalent performance level as the old method. In Figure 9.2 the robot drawn further below resembles the logged data, displaying the visible field lines and the direction of one goal post. The robot drawn above visualizes the result of the optimized localization run. The red dots represent localization hypothesis. The localization features match mutch better for the upper robot. – Note, that in this case a fairly extreme example was taken just for demonstration purposes. The preprocessed image from the robots camera helps to verify the localization result for the developer.

9.6 Multirobot Teams Spanning Multiple Labs

A topic not yet mentioned in this thesis is the interoperability of robots from different projects that try to achieve similar goals. This research area lies beyond the scope of this thesis. Nevertheless, *Miro* was already applied for conducting early experiments in this new and upcoming scientific topic as first discussed in [141].

Due to scientific as well as pragmatic reasons, there is a growing interest in the robotics field to join the efforts of different labs to form mixed teams of autonomous mobile robots. In RoboCup, the pragmatic reasons are compelling. The recent rule change in the mid-size league allows for more robots per team, and in the RoboCup Rescue league a group of heterogeneous robots with diverse capabilities is likely to perform better than one system that tries to encapsulate them all. However, the limited financial resources and the additional maintenance effort for further robots exceeds the capabilities of many research labs. Also, the threshold for new research groups to participate in long term scenarios such as RoboCup is lowered if they only need to contribute one or two robots to a mixed team, instead of having to build an entire team. Mixed teams are also motivated from a scientific perspective. They introduce the research challenge of cooperation within teams of extremely heterogeneous autonomous mobile systems.

As most robots in the middle-side league are custom built, or at least customized commercial research platforms with unique configurations of actuator and sensor configurations, mixed teams from different laboratories are extremely heterogeneous. There are few commonly used high-level libraries for sensor data processing and reactive actuator design in the community. Furthermore there is a multitude of methods and schools, each deliberately designing the control architecture of their robots fundamentally different to their competitors. This makes the unification of the software of the different robots of a potential

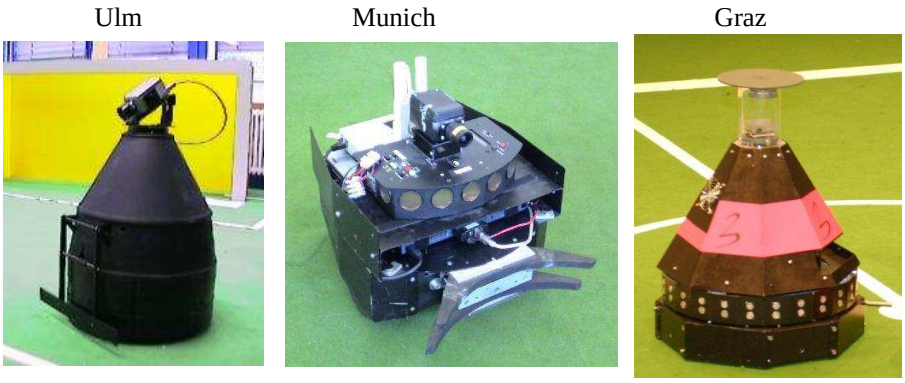

Figure 9.3: Individual robot platforms from the different labs.

mixed team almost impossible without substantial rewriting of at least one of the team's software. In our opinion it would be also undesirable. Why should an autonomous mobile robot have to commit to any kind of sensor processing or control paradigm to be able to cooperate with another team mates, if both are programmed to interact in the same problem domain?

The basic idea of the conducted experiments in mixed robot teams was, that for cooperation between robots, the sharing of information about the environment is initially sufficient for successful cooperation. If all robots share both the same belief about their environment, as well as the same set of goals, similar conclusions should be drawn. This is known as the Intentional Stance [22], and has proven to be a successful way of coordinating behavior in RoboCup scenarios [130, 18]. A central prerequisite for successful team cooperation was therefore the unification of the beliefs about the world of the different agents. The limitations of the individual sensors usually provide each robot with quite limited information about the state of its environment. So it is unlikely that the beliefs derived solely from the robots' own sensors are automatically sufficiently similar to coordinate behavior in a shared environment. Sharing of information was therefore considered essential for solving this problem. As different autonomous mobile platforms robots are equipped with different sensor suites that each provide their own unique perception of the environment, this improves the quality of the information available to the robots, even if they observe exactly the same scene from similar positions. Compare for instance laser range finders, which provide precise depth information, with color cameras, which provide more certainty about object identity.

At the RoboCup WorldCup 2004 in Lisbon a heterogeneous mixed-team experiment was conducted. For this purpose, one robot of the The Ulm Sparrows joined the teams of the middle-size league teams of the TU Munich, the "Agilo Robocuppers" [118] and the TU Graz team, "Mostly Harmless" (see Figure 9.3). For Agilo, Ulm provided the goal-keeper, in the Graz team, a striker was added. All teams used the group communication facility provided by *Miro* to exchange the beliefs of each robot with the other team mates. For this purpose,

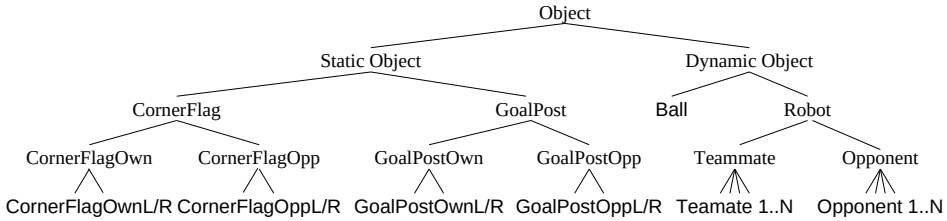

Figure 9.4: Tree of object classes for robot soccer.

a common representation for the robots observations was designed (the so-called belief state). It defines a classification hierarchy on the objects present in the robot soccer world (Figure 9.4), and models uncertainty as well as precision of the observed objects. In a followup to this experiments, a diploma thesis at the TU Munich currently evaluates methods of learning team-coordination based solely on a fused belief state and knowledge about the individual properties of the other robots in the team.

9.7 Achievement of Requirements and Design Goals

In this chapter, qualitative results achieved by the application of the proposed robot software architecture have been discussed and real-world examples of the application of *Miro* in various research projects were presented. The applications based upon the framework layer clearly reveal the benefits provided by the interaction of generic infrastructure, abstract robotics services, and application frameworks within a consistently designed robotics software architecture.

The requirements formulated in Section 3.1 were clearly met. *Miro* is highly portable. It supports multiple commercial platforms and was ported to various custom-built autonomous mobile robots from the Ulm Robotics Lab as well as from other research groups. It provides a very performant, flexible and scalable communication infrastructure based upon the CORBA standard offering compile-time type-safe interfaces to its various sensor and actuator services. Although it does not use the real-time features of CORBA, it addresses reactivity and timing constraints on the different layers of the architecture, ranging from accurate time-stamping of sensor data to prioritized asynchronous processing of image streams. It follows an open development model and is available as an open source software [1]. The stated design goals are consistently addressed throughout the entire software architecture. In consequence, *Miro* is not only reusable and scalable for various application scenarios in the robotics domain, it also fosters reusability, scalability and maintainability in applications based upon the architecture, especially through its higher-level frameworks. The experiments conducted on cooperation within teams of extremely heterogeneous robots illustrate a compelling long-range perspective for interoperability in the

[1] http://smart.informatik.uni-ulm.de/Miro

autonomous mobile robotics domain.

For the service layer, a detailed model for the design of sensor and actuator abstractions was proposed. The consistent and generalized service interfaces allow for multirobot platform development as demonstrated with the BAP framework in RoboCup. Existing software technologies such as CORBA or XML were consistently applied, especially on the infrastructure layer. This provides interoperability and allows for cooperation between very heterogeneous robots as demonstrated in the experiment on mixed robot teams spanning several labs. The architecture actively targets the integration of additional methods for use in the robotics domain, e.g. computer vision in the VIP framework or neural learning through the provided logging facilities.

Chapter 10

Conclusions

In this chapter, we will summarize the scientific contributions of this dissertation and suggest promising directions for future work.

The growing complexities of robot software development are not sufficiently addressed by existing robot software development architectures. They especially fail to assist in ensuring the reusability, scalability and maintainability of robotic software applications over time. There is a rising awareness in the robotics community, that the growing complexity of robotics applications requires a solid software infrastructure.

10.1 Contributions

This thesis contributes to this ongoing research by proposing the design for an object-oriented distributed robot software architecture aimed at providing improved reusability and scalability for robotic applications. The requirements for a robot software architecture identified during the research presented here are twofold: first, the architecture itself needs to be reusable, scalable and maintainable. Furthermore, it has to actively facilitate the development of reusable, scalable and maintainable client applications.

The proposed middleware-oriented architecture *Miro* addresses these requirements by identifying the inherent challenges of the robotics application domain and separating them into the four different layers of the software architecture (Section 3.3). The application developer only interacts with the top three layers, avoiding any dependency on the physical robot platform represented by the lowest layer.

The first layer was not discussed in this thesis. It provides abstractions for the low-level communication with robot devices. Its software pattern based design and the consequent usage of an OS abstraction toolkit (ACE) ensures its high portability.

The second layer provides an infrastructure of communication and configuration

facilities for the architecture's other layers as well as for end user applications. For this purpose, it employs proven software technology such as CORBA and XML, providing a detailed model of their configuration and application for usage within the robotics domain. The scientific contribution lies in a detailed assessment of the applicability of CORBA to robotics, and in a transparent extension to overcome a bottleneck as identified for group communication in robot teams on highly unreliable wireless networks (Section 4.1.5).

The proposed abstraction of sensor and actuator devices is addressed on the third layer by providing a definition of robotics services. The client/server-based design provides a natural decoupling of client applications from a robot's physical devices. Additionally, this layer provides generic support for high-performance data acquisition as needed for debugging, assessment and adaptivity in this application domain. The scientific contribution of this layer lies in the identification and detailed specification of functional requirements for the design of sensor and actuator abstractions, which facilitate the development of flexible and reusable client applications (Section 5.1f). Additionally, it contributes supportive technology required at various stages of the development cycle of robotic applications in the form of a logging service (Section 5.4).

The design goal of fostering the reusability of robotics applications with the help of the proposed robotics software architecture is addressed on the fourth layer of the architecture in the form of class frameworks for higher-level robotic subsystems. The thesis discusses two frameworks of *Miro*, namely the VIP framework for time-constrained video image processing (Section 6.1) and the BAP framework for hierarchical reactive behavior-based control (Section 6.2). These exemplify how the combination of communication and configuration infrastructure, along with service-based abstractions for physical robot devices can be utilized by frameworks of reusable design and reusable code that facilitate the development of reusable components for individual sub-domains of robotic applications. The scientific contribution of the VIP framework lies in providing a flexible model for control flow and data flow for prioritized, parallel image processing and a sensor-triggered but demand-driven push model of information processing. The BAP framework contributes a generalization for the modeling of reactive control that is applicable to different schools of behavior-based robotics. It extends them by introducing generic sequencing capabilities and a method for hierarchical modularization, as well as by introducing reusability on the reactive control and sequencing layer.

10.2 Open Problems

As autonomous mobile platforms evolve, research on robotics will meet new challenges and needs to fulfill additional requirements. This results in new and upcoming research problems that will need to be addressed. Some of the currently unsolved problems in research on robotics software architectures are the following.

10.2.1 Extended Service Meta-Information

The method for the modeling of sensor and actuator devices as abstract services discussed here proposes a definition of interface functionality that provides metainformation about the properties of the device. Such self-describing components enable the autonomous adaptation of client applications to individual devices. While *Miro* models basic properties such as the velocity bounds of a locomotion system etc., the question of how to express detailed sensor and actuator models in a generic extensible way in order to enable more sophisticated applications such as generic, precise sensor fusion in highly dynamic environments remains an unsolved research problem [83].

Sensor error models, for instance, can be rather complex. Laser range finders see through glass doors, sonars have difficulties with perceiving flat walls at obtuse angles and so on. Additionally, actuators also interact with sensor precision, especially the self-movement of the mobile base (motion blur), which also provides additional error sources for the integration of sensor values over time (i.e. precision of dead reckoning sensors).

This requires the specification of extensible sensor metainformation on a more general and flexible level. Additionally, since such information is partly dependent on its environment and might change over time, complex sensor-actuator systems such as mobile robots or sensor networks are required to obtain this information in an autonomous manner in order to keep the system maintainable.

10.2.2 Specification of Team Behavior

The BAP framework discussed in this thesis proposes a solution for the scalable, modular specification of reactive behavior for a single robot. Only very few solutions supporting the specification of team behavior exist so far. The MissionLab system, for example, allows for the specification of multirobot missions, but assumes a fixed assignment of roles [77]. Advanced multirobot scenarios such as RoboCup or sophisticated human-robot interaction requires the flexible assignment of roles in cooperative and competitive reactive action sequences [128, 124]. Advanced cooperation also requires specifying the options for the different individuals, as well as synchronization points of their respective actions.

The double pass, for instance, is a reactive action sequence for two cooperative players. The two passes between them form the synchronization points, where both have to perform in a matching pair pattern. The movement of the players between the passes is part of the sequence, but allows for much more individual timing and is actually heavily constrained by the opposing player(s) and their movements. This would require the extension of the BAP formalism by notions of multirobot parallelism, synchronization and sequencing, as outlined in more detail in the research proposal of the ongoing DFG SPP-1125 project at Ulm [95].

10.3 Future Work

Miro has gained some momentum as a community project and is now rapidly evolving towards new application scenarios and challenges such as outdoor robotics or large-scale robot fleets. To keep up with the growing number of requirements, future work on this project will address the issues listed below.

Dynamic service configuration. The heavy growth in supported platforms and robotics devices requires a more flexible and scalable design for meta-servers [112] used for robotics services. While at present each binary holds a preselected set of services, a flexible generic meta-server would allow to define the components hosted by the server entirely based upon its configuration requirements on startup (or even at runtime). According to the metainformation design, the server's configuration would then be required to be inspectable at runtime. These efforts will allow for easier use of *Miro* services in new robot platforms or in varying sensor/actuator configurations.

Additional high-level frameworks. The set of application frameworks provided by *Miro* needs to grow further and reflect the process of consolidation in some of the areas of robotics research. While some potential frameworks are actually already in existence at the Ulm Robotics Lab and essentially need a proper assessment of their genericity and flexibility (SLAM, navigation), other candidates would actually require an extended cooperation with other research groups (for example in planning). This would further enhance the software quality of applications based upon the *Miro* architecture.

End to end quality of service. *Miro* currently does not exploit the features of the DOC technologies employed for real-time processing on the communications layer [113]. The increase of requirements for predictability in highly dynamic environments necessitates their employment along with a stringent model for priorities in a highly reactive autonomous mobile system. This will be a major step towards enhancing the predictability of performance in autonomous mobile systems and towards ensuring that QoS requirements in the client/server model are met end to end.

Standardization. Apart from the various extensions that ensure the scalability and maintainability of *Miro*, another pressing issue is the extension of cooperation between the various other research projects on robotics software architectures. Herman Bruyninckx suggested the standardization of measures and their representations for the various projects. *Miro* already provides basic support for Player/Stage and thus provides an interesting basis for enhanced interoperability. Incorporating results from the research on RT-CORBA for robotics carried out by MacDonald et al. [73] would extend the applicability of the *Miro* robotics middleware to include further application scenarios.

10.4 Outlook

A common notion of robotics middleware will be a key factor for the successful deployment of robotic applications. It would allow to decouple robotics hardware platforms from their provided software environment and thus enable faster, more flexible software development independent of the success of an individual robot platform and visa versa.

The solutions for addressing the inherent challenges of the robotics software domain proposed in this thesis actually address a wider range of possible applications. They also naturally apply to the domain of sensor networks, i.e. large sets of sensors (and actuators) statically distributed over a specified area, where intensive research is being carried out at present. Indoor application scenarios encompass instrumented buildings, usable for surveillance, assisted living (care for the elderly) or household automatization. Sensor networks in outdoor scenarios are considered to be used for coordinating rescue forces in disaster areas, for environmental monitoring or in the military domain. These scenarios exhibit very similar inherent domain problems such as distributedness, parallelism, sensor/actuator variety etc. which are addressed by the research discussed in this thesis. The large amount of sensors additionally necessitates services for requirements-based flexible service discovery and for an extended form of metainformation for the efficient interaction of reusable client applications.

Zusammenfassung

Die Forschung im Bereich autonomer mobiler Roboter arbeitete in den letzten Jahren an zunehmend komplexeren Problemstellungen, Anwendungsszenarien und Roboterplattformen. Die Kernbereiche der anwendungsorientierten Informatik haben auf ähnliche Komplexitätssteigerungen in ihren Anwendungsdomänen mit der Entwicklung von flexiblen, skalierbaren Softwarearchitekturen und einem begleitenden Methodenarsenal reagiert. Durch die verbreitete Fokussierung der Robotikforschung auf Problemstellungen im perzeptorischen (Sensorinterpretation und -fusion, Bild- und Sprachverarbeitung) und/oder aktuatorischen (Regelung, Kollisionsvermeidung, Pfadplanung) Bereich ist eine entsprechende Entwicklung der Software-Methodik bislang nur in Ansätzen erfolgt, wie im Kapitel 2.4 diskutiert. Die dadurch noch weit verbreitet anzutreffenden konzeptuellen Defizite der Softwarearchitekturen aktueller Demonstratorplattformen erhöhen den Aufwand für die Entwicklung und die Fehleranfälligkeit der Lösungen, schränken die Austauschbarkeit und Validierbarkeit wissenschaftlicher Ergebnisse ein und stehen einer Kommerzialisierung erarbeiteter Lösungen im Wege.

Die Probleme der Software-Entwicklung in der Robotik erklären sich aus der inhärenten Komplexität der Anwendungsdomäne. So müssen extrem heterogene Hardware-Komponenten, die Notwendigkeit einer nebenläufigen und verteilten Informationsverarbeitung, eine enge Verknüpfung von algorithmischen Lösungen mit physikalischen Eigenschaften des Roboters und seiner Umgebung, die stochastischen Eigenschaften der physikalischen Welt, die Echtzeit- oder zumindest Realzeit-Anforderungen sowie Resourcenbeschränkungen bei der Entwicklung von Applikationen für autonome mobile Systeme gleichermassen berücksichtigt werden. Die strukturellen Probleme der Software führen zu einer eingeschränkten Generalisierbarkeit von Forschungsergebnissen, der häufigen Reimplementierung eigentlich gelöster Probleme, dem starken Einsatz heuristischer Lösungen und einer ungenügenden Auswertung und Analyse der Applikationen. Dadurch bleiben viele Forschungsergebnisse der Robotik auf einen reinen Demonstrationscharakter beschränkt.

In der Dissertation wurden daher verschiedene Konzepte zur Lösung der strukturellen Probleme aktueller Software für autonome mobile Roboter erarbeitet (siehe auch Kapitel 3): Der Einsatz moderner Softwaretechnologien wie objektorientierte Programmiersprachen, Middleware und Auszeichnungssprachen, die Einführung Service-orientierter Abstraktionen von Sensorik und Ak-

tuatorik zur Förderung von Portabilität und Wiederverwendbarkeit sowie die Framework-basierte Generalisierung von robotikspezifischen Aufgaben. Diese Konzepte wurden am Beispiel des Entwurfs der Roboter Softwarearchitektur *Miro* diskutiert, exemplarisch umgesetzt und in diesem Rahmen experimentell evaluiert. *Miro*, die „Middleware for Robots“ ist eine objektorientierte, Middleware-orientierte Softwarearchitektur für die Entwicklung portierbarer, wiederverwendbarer und skalierbarer Applikationen für autonome mobile Roboter, im Kontext nebenläufiger und verteilter Informationsverarbeitungsprozesse unter den Realzeit-Bedingungen natürlicher Umgebungen.

Die Architektur ist in vier funktionale Ebenen unterteilt, die zwischen dem Betriebssystem und dem Anwendungsprogramm liegen (Kapitel 3.3). Die unterste Schicht stellt die Hardware-Abstraktionsebene dar, welche die Kommunikation mit den heterogenen Bus-Systemen und Mikrocontroller-Boards kapselt und als Bibliotheken zur Verfügung stellt. Diese Ebene wird von den höheren Ebenen der Architektur weiter gekapselt, um Applikationen eine weitestgehende Unabhängigkeit von der genutzten physikalischen autonomen mobilen Plattform zu ermöglichen. Auf der zweiten Ebene wird Infrastruktur zur Kommunikation und Konfiguration auf der Basis von CORBA und XML modelliert (Kapitel 4). Die auf dieser Ebene zur Verfügung gestellte Funktionalität wird extensiv von den höheren Ebenen der Architektur aber auch von den bereits auf *Miro* basierenden Anwendungsprogrammen genutzt. Auf der dritten Ebene werden die Sensoren und Aktuatoren des autonomen mobilen Systems als netzwerktransparente Dienste mit streng getypten Schnittstellen modelliert (Kapitel 5). Die verschiedenen Sensor- und Aktuator-Typen werden dabei in ihren verschiedenen Ausprägungen in generalisierte Schnittstellen gefasst. Die individuellen Merkmale der Sensoren und Aktuatoren unterschiedlicher Hersteller werden durch abgeleitete, spezialisierte Schnittstellen zur Verfügung gestellt. Robotiktypische Aufgaben wie Video-Bildverarbeitung oder die reaktive Verhaltensmodellierung werden auf der obersten Ebene von Software-Frameworks unterstützt (Kapitel 6). Diese Frameworks sind unter besonderer Berücksichtigung der Skalierbarkeit, sowie der domänenspezifischen Echtzeitaspekte konzipiert. So ermöglicht z.B. die Framework-basierte Steuerung des Kontrollflusses in zeitkritischen Aufgaben die Priorisierung der Informationsverarbeitung auf der abstrakten Ebene des Frameworks generisch zu kontrollieren und zu konfigurieren.

Die zentrale Leistung dieser Dissertation stellt die Identifikation und Umsetzung von Konzepten der aktuellen Software-Methodik zur Bewältigung der inhärenten Komplexität der Anwendungsdomäne autonome mobile Roboter dar: die konsequente Objektorientierung, der Einsatz und die Adaption der Middleware-Technologie CORBA, die durchgängige Unterstützung nebenläufiger Informationsverarbeitung sowie die Einführung von Modellierungswerkzeugen für die Anwendungsentwicklung.

Die in dieser Arbeit konzipierten Dienste und Frameworks wurden erfolgreich in so unterschiedlichen Szenarien wie „Simultanious Localization and Mapping“ (SLAM) für Service-Roboter, Roboterfußball oder biologisch motiviertes neuronales Lernen eingesetzt. Die Frameworks konnten auch jeweils unabhängig ihre

Leistungsfähigkeit unter Beweis stellen. Das „Video Image Processing“ Framework (Kapitel 6.1) zur Bildverarbeitung unter Echtzeitbedingungen dient als Basis für eine Szenario-unabhängige Bibliothek für fortgeschrittene Methoden zur visuellen Objektklassifikation. Das „Behavior Actionpattern Policy“ Framework (Kapitel 6.2) wird zur Modellierung des umfangreichen Verhaltensrepertoires von Fußballrobotern im RoboCup eingesetzt. *Miro* wird mittlerweile auch von anderen Forschungseinrichtungen zur Entwicklung von Anwendungen in der Robotik genutzt. Die RoboCup Teams der TU München und Graz verwenden *Miro* für die Steuerung und Gruppenkommunikation ihrer Fußballroboter (siehe auch Kapitel 9.6). An der TU Chemnitz wird *Miro* für den Einsatz auf einer kommerziellen sowie einer eigenentwickelten Outdoor-Roboterplatform portiert.

Abbreviations

ACE *Adaptive Communication Environment.* A multi-platform, object-oriented network programming toolkit in C++ (Section 4.3.1).

API *Application Programmers Interface.* A set of definitions of the ways one piece of computer software communicates with another [149].

BAP *Behavior, Action pattern, Policy.* The application framework for structured behavior based control of autonomous mobile robots discussed in this thesis (Section 6.2).

CDR *Common Data Representation.* A binary format for the serialization of data structures, as defined by CORBA.

CORBA *Common Object Request Broker Architecture.* An open standard for object oriented distributed systems communication (Section 4.1.3).

DAG *Directed Acyclic Graph.*

DOC *Distributed Object Computing.* An object oriented programming model for distributed systems.

FSA *Finite State Automaton.*

IDL *Interface Definition Language.* A computer language for describing the interface of a software component [149]. CORBA also defines an IDL.

IIOP *Internet IOP.* The TCP/IP based version of IOP.

IOP *Inter ORB Protocol.* A meta-protocol that defines the low-level communication between ORBs.

IOR *Interoperable Object Reference.* A reference to a remote object, as defined by CORBA.

IPP *Intel Performance Primitives.* A commercial library with highly optimized implementations of standard image operations.

IR *Infrared sensor.* A low-cost near range distance sensor.

LRF *Laser Range Finder.* A very accurate distance sensor.

NMC *Notify Multi-Cast.* A module of the *Miro* architecture, that is used to built a federation of event channels of the standard CORBA notification service (Section 4.1.5).

OMG *Object Management Group.* The standardization consortium, which among others, maintains the CORBA standard.

ORB *Object Request Broker.* The mediator for communication between distributed objects.

OS *Operating System.*

QoS *Quality of Service.* In general it refers to the ability of a server to meet the requirements of a given service contract with a client.

ROI *Region Of Interest.* A sub-window of an image, containing potentially important information for the task at hand.

RT *Real-Time.* An operation within a larger dynamic system is called a real-time operation if the combined reaction- and operation-time of a task is shorter than the maximum delay that is allowed, in view of circumstances outside the operation [149].

RT-CORBA *Real-Time CORBA.* An extension of the CORBA standard, defining interfaces and functionality of real-time enabled ORB implementations.

SLAM *Simultaneous Localization And Mapping.* The problem of build up a map within an unknown environment while at the same time keeping track of its current position [149].

TAO *The ACE ORB.* A high-performance open-source CORBA implementation (Section 4.3.2).

UIOP *Unix IOP.* A non-standard version of IOP available with TAO, optimized for interprocess communication on a single Unix machine (Section 8.1.2).

VIP *Video Image Processing.* The application framework for computer vision on autonomous mobile robots discussed in this thesis (Section 6.1).

WLAN *Wireless Local Area Network.* A wireless networking technology.

Bibliography

[1] ActivMedia. *Pioneer 1 Software Manual*. RWI, Jaffrey, NH, 1996.

[2] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, Special Issue on "Integrated Architectures for Robot Control and Programming", 1997.

[3] R. Alami, R. Chatila, S. Fleury, M. Herrb, F. Ingrand, M. Khatib, B. Morisset, P. Moutarlier, and T. Simeon. Around the lab in 40 days. In *Proceedings of the 2000 International Conference on Robotics and Automation (ICRA 2000)*, pages 88–94, San Francisco, April 2000. IEEE.

[4] R. Arkin, T. Collins, and Y. Endo. Tactical mobile robot mission specification and execution, 1999.

[5] Ronald C. Arkin. Towards the unification of navigational planning and reactive control. In *Working Notes of the AAAI Spring Symposium on Robot Navigation*, March 1989.

[6] Greg Armstrong. Configuring carmen parameters. Available via `http://www-2.cs.cmu.edu/~carmen/config_param.html`.

[7] Minoru Asada and Hiroaki Kitano. The RoboCup challange. *Robotics and Autonomous Systems*, 29:3–12, 1999.

[8] Tucker Balch. *Behavioral Diversity in Learning Robot Teams*. PhD thesis, College of Computing, Georgia Institute of Technology, 1998.

[9] Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors. *RoboCup 2001: Robot Soccer World Cup V*, volume 2377 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, Germany, 2002.

[10] D. Blevins. Overview of the enterprise JavaBeans component model. In *Component-based Software Engineering: Putting the Pieces Together*, chapter 33. Addison-Wesley, 2001.

[11] Ansgar Bredenfeld, Thomas Christaller, Wolf Göhring, Horst Günther, Herbert Jaeger, Hans-Ulrrich Kobialka, Paul-Gerhard Plöger, Peter Schöll, Andrea Siegberg, Arend Streit, Christian Verbeek, and Jörg

Wilberg. Behavior engineering with "dual dynamics" models and design tools. In Manuela Veloso, editor, *Sixteenth International Joint Conference on Artificial Intelligence IJCAI-99*, pages 57–62, 1999.

[12] Alex Brooks, Tobias Kaupp, Alexei Makarenko, Stefan Williams, and Anders Orebäck. Towards component-based robotics. In *Proceedings of the 2005 International Conference on Intelligent Robots and Systems (IROS 2005)* [55]. – Accepted.

[13] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1), March 1986.

[14] Gerald Brose. JacORB: Implementation and design of a Java ORB. In *Proceedings of DAIS '97*, Cottbus, Germany, October 1997. Chapman & Hall.

[15] Davide Brugali and Mohamed E. Fayad. Distributed computing in robotics and automation. *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures*, 18(4):409–420, August 2002.

[16] Herman Bruyninckx. Open robot control software: The OROCOS project. In *Proceedings of the 2001 International Conference on Robotics and Automation (ICRA 2001)* [53], pages 2523–2528.

[17] Herman Bruyninckx, Peter Soetens, and Bob Koninckx. The real-time motion control core of the OROCOS project. In *Proceedings of the 2003 International Conference on Robotics and Automation (ICRA 2003)*, pages 2766–2771, Taipei, Taiwan, September 2003. IEEE.

[18] Sebastian Buck, Michael Beetz, and Thorsten Schmitt. Reliable multi robot coordination using minimal communication and neural prediction. In M. Beetz, J. Hertzberg, M. Ghallab, , and M. Pollack, editors, *Advances in Plan-based Control of Autonomous Robots*, Lecture Notes in Artificial Intelligence. Springer, 2002.

[19] John F. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 679–698, 1986.

[20] M. K. Carter, K. M. Crennell, E. Golton, R. Maybury, A. Bartlett, S. Hammarling, and R. Oldfield. The design and implementation of a portable image processing algorithms library in fortran and c. In *Proceedings of the 3rd IEE International Conference on Image Processing and its Applications*, pages 516–520, 1989.

[21] C. Côté, D. Létourneau, F. Michaud, J.-M. Valin, Y. Brosseau, C. Raievsky, M. Lemay, and V. Tran. Code reusability tools for programming mobile robots. In *Proceedings of the 2004 International Conference on Intelligent Robots and Systems (IROS 2004)*, Sendai, Japan, September-October 2004. IEEE/RSJ.

[22] Daniel C. Dennett. *Brainstorms*. Harvester Press, 1981.

[23] Mayur Deshpande, Douglas C. Schmidt, Carlos O'Ryan, and Darrell Brunsch. The design and performance of asynchronous method handling for CORBA. In *Proceedings of the Distributed Objects and Applications (DOA) Conference*, Irvine, Canada, October/November 2002.

[24] Stefan Enderle. *Probabilistic Spatial Representations for Mapping and Self-Localization in Autonomous Mobile Robots*. Dissertation, University of Ulm, Neuroinformatics, Ulm, Germany, June 2001.

[25] Stefan Enderle, Heiko Folkerts, Marcus Ritter, Stefan Sablatnög, Gerhard K. Kraetzschmar, and Günther Palm. Vision-based robot localization using sporadic features. In *Workshop Robot Vision 2001*, Auckland, New Zealand, February 2001.

[26] Stefan Enderle, Marcus Ritter, Dieter Fox, Stefan Sablatnög, Gerhard K. Kraetzschmar, and Günther Palm. Vision-Based Localization in Robocup Environments. In Peter Stone, Tucker Balch, and Gerhard K. Kraetzschmar, editors, *RoboCup-2000: Robot Soccer World Cup IV*, volume 2019 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, Berlin, Heidelberg, Geramany, 2001.

[27] Stefan Enderle, Stefan Sablatnög, Steffen Simon, and Gerhard K. Kraetzschmar. Tetrixx – A Robot Development Kit. In Thomas Christaller, Giovanni Indiveri, and Axel Poigné, editors, *First International Workshop on Edutainment Robots*, 2000.

[28] Stefan Enderle, Hans Utz, Stefan Sablatnög, Steffen Simon, Gerhard K. Kraetzschmar, and Günther Palm. Miró: Middleware for autonomous mobile robots. In *IFAC Conference on Telematics Applications in Automation and Robotics*, pages 297–303, July 2001.

[29] Rebecca Fay, Ulrich Kaufmann, Friedhelm Schwenker, and Günther Palm. Learning Object Recognition in a NeuroBotic System. In Horst-Michael Groß, Klaus Debes, and Hans-Joachim Böhme, editors, *3rd Workshop on SelfOrganization of AdaptiVE Behavior SOAVE 2004*, pages 198–209. VDI Verlag, Düsseldorf, 2004.

[30] Chris Fedor. *TCX - An Interprocess Communication System for Building Robotic Architectures: Programmer's Guide to Version 10.xx*. CMU, Pittsburgh, Pennsylvania, 1993.

[31] R. J. Feiertag and E. I. Organick. The Multics input/output systems. In *Proceedings of the Symposium on Operating Systems Principles*, pages 35–41, New York, October 1971.

[32] R. James Firby. Task networks for controlling continuous processes. In *Second International Conrerence on AI Planning Systems*, pages 49–54, Chicago, IL, June 1994.

[33] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte Carlo localization: Efficient position estimation for mobile robots. In *Proceedings of the National Conference on Artificial Intelligence*. AAAI, 1999.

[34] Gordon Fraser, Gerald Steinbauer, Arndt Mühlenfeld, and Franz Wotawa. A modular architecture for a multi-purpose mobile robot. In *In Proc. of the 17th Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, volume 3029, 2004.

[35] Guillem Pages Gassull. Communication services and user interfaces for tele-operating mobile robots via the internet. Master's thesis, University of Barcelona and University of Ulm, Neuroinformatics, Barcelona, Spain and Ulm, Germany, July 2001. (in German).

[36] B. Gerkey, R. Vaughan, K. Sty, A. Howard, G. Sukhatme, and M. Mataric. Most valuable player: A robot device server for distributed control. In *Proceedings of the 2001 International Conference on Robotics and Automation (ICRA 2001)* [53].

[37] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, pages 317–323, Coimbra, Portugal, June 2003.

[38] Christopher D. Gill and William D. Smart. Middleware for robots? In *Proceedings of the AAAI Spring Symposium on Intelligent Distributed and Embedded Systems*, 2002.

[39] Andy Gokhale and Douglas C. Schmidt. Optimizing a CORBA IIOP protocol engine for minimal footprint multimedia systems. *IEEE Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, 17(9), September 1999.

[40] P. Gore, I. Pyarali, C. Gill, and D. Schmidt. The design and performance of a real-time notification service. In *Proceedings of the 10th IEEE Real-time Technology and Application Symposium*, Toronto (Canada), May 2004.

[41] Pradeep Gore, Ron Cytron, Douglas C. Schmidt, and Carlos O'Ryan. Designing and optimizing a scalable CORBA notification service. In *LCTES/OM*, pages 196–204, 2001.

[42] Erio Grillo, Matteo Matteucci, and Domenico G. Sorrenti. Getting the most from your color camera in a color-coded world. In Nardi et al. [90], pages 221–235.

[43] Object Management Group. CORBA services: Common objet services sepcification. Technical report, OMG, Framingham, MA, 1997.

[44] Object Management Group. Minimum CORBA specification v1.0. Technical report, OMG, Framingham, MA, August 2002.

[45] Object Management Group. Telecom log service specification. Technical report, OMG, Framingham, MA, July 2003.

[46] Object Management Group. CORBA/IIOP specification. Technical report, OMG, Framingham, MA, April 2004.

[47] Jens-Steffen Gutmann and Christian Schlegel. AMOS: Comparison of scan matching approaches for self-localization in indoor environments. In *Proceedings of the 1st Euromicro Workshop on Advanced Mobile Robots*, 1996.

[48] Claudia Gönner, Martin Rous, and Karl-Friedrich Kraiss. Real-time adaptive colour segmentation for the robocup middle size league. In Nardi et al. [90].

[49] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The design and performance of a real-time CORBA event service. In *Proc. of OOPSLA '97*, Atlanta, October 1997. ACM.

[50] Michi Henning and Steve Venoski. *Advanced CORBA Programming in C++*. Addison-Wesley professional computin series. Addison Wesley, Reading, MA, 1999.

[51] T.A. Henzinger. The theory of hybrid automata. In M.K. Inan and R.P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, NATO ASI Series F: Computer and Systems Sciences 170, pages 265–292. Springer-Verlag, 2000.

[52] B.K.P. Horn and B.G. Schunck. Determining optical flow. Technical report, Massachusetts Institue of Technology, 1980.

[53] IEEE. *Proceedings of the 2001 International Conference on Robotics and Automation (ICRA 2001)*, Wailea, Hawaii, October 2001.

[54] IEEE/RSJ. *Proceedings of the 2003 International Conference on Intelligent Robots and Systems (IROS 2003)*, Las Vegas, Nevada, October 2003.

[55] IEEE/RSJ. *Proceedings of the 2005 International Conference on Intelligent Robots and Systems (IROS 2005)*, Edmonton, Canada, August 2005.

[56] Intel performance primitives (IPP). More information on `http://www.intel.com/software/products/perflib/`.

[57] Herbert Jaeger and Thomas Christaller. Dual dynamics: designing behavior systems for autonomous robots. In *The Sixth International Symposium on Artificial Life and Robotics (AROB 6th '01)*, 1997.

[58] Ralph E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42, 1997.

[59] Rudolph E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[60] Ulrich Kaufmann, Rebecca Fay, and Günther Palm. Neural networks for visual object recognition based on selective attention. In *1st International SenseMaker Workshop on Life-Like Perception Systems 2005*, 2005.

[61] Ulrich Kaufmann, Gerd Mayer, Gerhard Kraetzschmar, and Günther Palm. Visual robot detection in RoboCup using neural networks. In Polani et al. [96], pages 262–273.

[62] Kurt Knolige and Karen Myers. The Saphira architecture for autonomous mobile robots. In *Artificial Intelligent and Mobile Robots*, chapter 9, pages 211–242. AAAI Press, 1998.

[63] K. Konolige, C. Ortiz, R. Vincent, A. Agno, M. Eriksen, B. Limketkai, M. Lewis, L. Briesemeister, E. Ruspini, D. Fox, J. Ko, B. Stewart, and L. Guibas. CentiBOTS: Large-scale robot teams. In *Multi-Robot Systems: From Swarms to Intelligent Automata*, volume II. Kluwer, 2003.

[64] Kurt Konolige. Colbert: A language for reactive control in Saphira. In G. Brewka, C. Habel, and B. Nebel, editors, *Advances in Artificial Intelligence*, volume 1303 of *Lecture Notes in Computer Science*. Springer, 1997.

[65] David Kortenkamp, Tod Milam, Reid Simmons, and Joaqin Lopez Frenandez. Collecting and analyzing data from distributed control programs. *Electronic Notes in Theoretical Computer Science*, 55(2):19, 2001.

[66] David Kortenkamp, Tod Milam, Reid Simmons, and Joaqin Lopez Frenandez. Collecting and analyzing data from distributed control programs. *Formal Methods and Systems Design Journal*, (24):157–188, 2004.

[67] Gerhard K. Kraetzschmar, Gerd Mayer, Hans Utz, Philipp Baer, Martin Claus, Ulrich Kaufmann, Markus Lauer, Simon Natterer, Sebastian Przewoznik, Roland Reichle, Alexander Reisser, Axel Roth, Miriam Schmidt, Christoph Sitter, Florian Sterk, and Günther Palm. The Ulm Sparrows 2003. In Polani et al. [96], pages 238–249.

[68] Gerhard K. Kraetzschmar, Stefan Sablatnög, Stefan Enderle, and Günther Palm. Application of neurosymbolic integration for environment modelling in mobile robots. In Stefan Wermter and Ron Sun, editors, *Hybrid Neural Systems*, number 1778 in Lecture Notes in Computer Science, Berlin, Germany, March 2000. Springer. ISBN 3-540-67305-9.

[69] Gerhard K. Kraetzschmar, Stefan Sablatnög, Stefan Enderle, Hans Utz, Steffen Simon, and Günther Palm. Integration of Multiple Representations and Navigation Concepts on Autonomous Mobile Robots. In Horst-Michael Groß, Klaus Debes, and Hans-Joachim Böhme, editors, *Workshop*

on SelfOrganization of Adaptive Behavior SOAVE 2000, volume 10/643 of *Fortschrittsberichte VDI: Informatik/Kommunikationstechnik*, Düsseldorf, October 2000. VDI Verlag.

[70] Gerhard K. Kraetzschmar, Hans Utz, Stefan Sablatnög, Stefan Enderle, and Günther Palm. Miro – Middleware for Cooperative Robotics. In Birk et al. [9], pages 411–416.

[71] Daniel Krüger. Porting the MIRO middleware to a mobile platform. Master's thesis, Technical University of Chemnitz, Chemnitz, Germany, 2005.

[72] Yuan-hsin Kuo and Bruce A. MacDonald. Designing a distributed real-time software framework for robotics. In *Proc. Australasian Conf. on Robotics and Automation*, Canberra, Australia, December 2004.

[73] Yuan-hsin Kuo and Bruce A. MacDonald. A distributed real-time software framework for robotic applications. In *Proceedings of the 2005 International Conference on Robotics and Automoation (ICRA 2005)*, Barcelona, Spain, April 2005. IEEE.

[74] Pedro Lima, Tucker Balch, Masahiro Fujita, Raul Rojas, Manuela Veloso, and Holly Yanco. RoboCup 2001: A report on research issues that surfaced during the competitions and conference. *IEEE Robotics and Automation Magazine*, pages 20–30, June 2002.

[75] Martin Lötzsch, Joscha Bach, Hans-Dieter Burkhard, and Matthias Jüngel. Designing agent behavior with the extensible agent behavior specification language XABSL. In Nardi et al. [90].

[76] Bruce MacDonald, David Yuen, Sylvia Wong, Evan Woo, Rowan Gronlund, Toby Collett, Félix-Étienne Trépanier, and Geoff Biggs. Robot programming environments. In *ENZCon2003 10th Electronics New Zealand Conference*, University of Waikato, Hamilton, 1–2 September 2003.

[77] Douglas C. MacKenzie, Ronald C. Arkin, and Jonathan M. Cameron. Multiagent mission specification and execution. *Autonomous Robots*, 4(1):29–52, March 1997.

[78] Maja J. Mataric. Behaviour-based control: Examples from navigation, learning, and group behavior. *Journal of Experimental and Theoretical Artificial Intelligence*, 9, 1997.

[79] Gerd Mayer, Ulrich Kaufmann, Gerhard K. Kraetzschmar, and Günther Palm. Neural robot detection in RoboCup. In *27th German Conference on Artificial Intelligence (KI2004), NeuroBotics Workshop*, University of Ulm, 2004. to appear in the book "Biomimetic neural learning for intelligent robots".

[80] Gerd Mayer, Jonas Melchert, Hans Utz, Gerhard Kraetzschmar, and Günther Palm. Neural object classification and tracking. In *Proceedings of 4th Chapter Conference on Applied Cybernetics*, London, United Kingdom, September 2005. IEEE Systems, Man and Cybernetics Society.

[81] Gerd Mayer, Hans Utz, and Gerhard K. Kraetzschmar. Towards autonomous vision self-calibration for soccer robots. In *Proceeding of the 2002 International Conference on Intelligent Robots and Systems (IROS 2002)*, volume 1, pages 214–219, Lausanne, Switzerland, September-October 2002. IEEE/RSJ.

[82] Gerd Mayer, Hans Utz, and Gerhard K. Kraetzschmar. Playing robot soccer under natural light: A case study. In Polani et al. [96], pages 238–249.

[83] Nik Melchior and William D. Smart. Autonomic systems for mobile robots. In *Proceedings of the International Conference on Autonomic Computing (ICAC 2004)*, pages 280–281, 2004.

[84] Microsoft. Overview of the windows registry. Available via `http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/regedit_overview.mspx`.

[85] Sun Microsystems. Java, preferences class. Available via `http://java.sun.com/j2se/1.5.0/docs/api/java/util/prefs/Preferences.html`.

[86] Dave L. Mills. Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI. Network Working Group Request for Comments: 2030, October 1996.

[87] Robin R. Murphy. *An Introduction to AI Robotics*. MIT Press, November 2000.

[88] John Murray, Stephan Wermter, and Harry Erwin. Auditory robotic tracking of sound sources using hybrid cross-correlation and recurrent network. In *Proceedings of the 2005 International Conference on Intelligent Robots and Systems (IROS 2005)* [55]. – Accepted.

[89] Karen Myers. A procedural knowledge approach to task-level control. In *Third International Conference on AI Planning System*. AAAI Press, 1996.

[90] D. Nardi, Riedmiller, Sammut M., and J. C., Santos-Victor, editors. *RoboCup 2004: Robot Soccer World Cup VIII*, volume 3276 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, Germany, 2005.

[91] Issa Nesnas, Anne Wright, Max Bajracharya, Reid Simmons, and Tara Estlin. CLARAty and challenges of developing interoperable robotic software. In *Proceedings of the 2003 International Conference on Intelligent Robots and Systems (IROS 2003)* [54].

[92] Anders Orebäck. *A Component Framework for Autonomous Mobile Robots*. PhD thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, 2004.

[93] Anders Orebäck and Henrik I. Christensen. Evaluation of architectures for mobile robotics. *Autonomous Robots*, 14(1):33–49, January 2003.

[94] Carlos O'Ryan, Fred Kuhns, Douglas C. Schmidt, and Jeff Parsons. Applying patterns to develop a pluggable protocols framework for ORB middleware. In ed. Linda Rising, editor, *Design Patterns in Communications*. Cambridge University Press, 2000.

[95] Günther Palm and Gerhard K. Kraetzschmar. Adaptivity and learning in teams of cooperating mobile robots, 2005. Proposal and Research Plan, Ulm Project of DFG SPP-1125.

[96] D. Polani, B. Browning, A. Bonarini, and K. Yoshida, editors. *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, Germany, 2004.

[97] A. Pope. *The CORBA Reference Guide*. Addison-Wesley, Reading, MA, 1997.

[98] Real World Interface, 32 Fitzgerald Drive, PO Box 375, Jeffrey, New Hampshire 03452. *Mobility 1.1, Robot Integration Software, User's Guide*, 1999.

[99] Hans P. Reiser, Franz J. Hauck, Rüdiger Kapitza, and Andreas I. Schmied. Integrating fragmented objects into a CORBA environment. In *Proceedings of the Net.ObjectDays*, Erfurt, Germany, September 2003.

[100] Zack Rusin. Using KConfig XT. Available via `http://developer.kde.org/documentation/tutorials/kconfigxt/kconfigxt.html`.

[101] Stefan Sablatnög, Stefan Enderle, Mark Dettinger, Thomas Boß, Mohammed Livani, Michael Dietz, Jan Giebel, Urban Meis, Heiko Folkerts, Alexander Neubeck, Peter Schaeffer, Marcus Ritter, Hans Braxmeier, Dominik Maschke, Gerhard Kraetzschmar, Jörg Kaiser, and Günther Palm. The Ulm Sparrows 99. In Veloso et al. [145].

[102] Stefan Sablatnög. *Region-Based Representation of Spatiotemporal Concepts*. PhD thesis, University of Ulm, Neuroinformatics, Ulm, Germany, October 2001.

[103] A. Saffiotti, E. H. Ruspini, and K. Konolige. Blending reactivity and goal-directedness in a fuzzy controller. In *Second IEEE International Conference on Fuzzy Systems*, pages 134–139, Silver Spring, MD, March–April 1993. IEEE Computer Society Press.

[104] Alessandro Saffiotti, Kurt Konolige, and Enrique H. Ruspini. A multivalued logic approach to integrating planning and control. *Artificial Intelligence*, February 1995.

[105] Christian Schlegel. *Navigation and Execution for Mobile Robots in Dynamic Environments - An Integrated Approach*. PhD thesis, University of Ulm, 2004.

[106] Christian Schlegel and Robert Wörz. Interfacing different layers of a multilayer architecture for sensorimotor systems using the object-oriented framework smartsoft. In *Third European Workshop on Advanced Mobile Robots (Eurobot 1999)*, Zürich, Switzerland, September 1999.

[107] Christian Schlegel and Robert Wörz. The software framework smartsoft for implementing sensorimotor systems. In *Proceedings of the 1999 International Conference on Intelligent Robots and Systems (IROS 1999)*, volume 3, pages 1610–1616, Kyongju, Korea, October 1999. IEEE/RSJ.

[108] Douglas C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In *Proceedings of the 1st Pattern Languages of Programs Conference*, August 1994.

[109] Douglas C. Schmidt and Charles D. Cranor. Half-sync/half-async – an architectural pattern for efficient and well-structured concurrent I/O. In *Proceedings of the Second Pattern Languages of Programs Conference*, Monticello, Illinois, September 1995.

[110] Douglas C. Schmidt, Mayur Deshpande, and Carlos O'Ryan. Operating system performance in support of real-time middleware. In *Proceeding of the 7th IEEE International Workshop on Object-oriented Real-time Dependable Systems*, San Diego, California, January 2002. IEEE.

[111] Douglas C. Schmidt, Andy Gokhale, Tim Harrison, and Guru Parulkar. A high-performance endsystem architecture for real-time CORBA. *IEEE Comm. Magazine*, 14(2), 1997.

[112] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming*, volume 2. Addison-Wesley Longman, December 2002.

[113] Douglas C. Schmidt and Fred Kuhns. An overview of the real-time CORBA specification. *Computer*, 33(6):56–63, 2000.

[114] Douglas C. Schmidt, Carlos O'Ryan, Ossama Othman, Fred Kuhns, and Jeff Parsons. *Design Patterns in Communications*, chapter Applying patterns to develop a pluggable protocols framework for ORB middleware, pages 439–494. Cambridge University Press, 2001.

[115] Douglas C. Schmidt and Steve Vinoski. Object interconnections: The CORBA component model: Part 1, evolving towards component middleware. *C/C++ Users Journal*, February 2004.

[116] Douglas C. Schmidt and Steven Vinoski. An overview of the OMG CORBA messaging quality of service (QoS) framework. *C++ Report*, 12(3), March 2000.

[117] Douglas C. Schmidt and Steven Vinoski. Standard C++ and the OMG C++ mapping. *C/C++ Users Journal*, January 2001.

[118] T. Schmitt, R. Hanek, M. Beetz, S. Buck, and B. Radig. Cooperative probabilistic state estimation for vision-based autonomous mobile robots. *IEEE Trans. on Robotics and Automation*, 18(5):670–684, 10 2002.

[119] Reid Simmons and David Apfelbaum. A task description language for robot control. In *International Conference on Intelligent Robots and Systems*, 1998.

[120] Reid Simmons and Dale James. *Inter Process Communication*. CMU, 3.4 edition, February 2001.

[121] imlib3d. Available via `http://imlib3d.sourceforge.net`.

[122] Lti-lib. Available via `http://ltilib.sourceforge.net/doc/homepage/index.shtml`.

[123] Vxl. Available via `http://vxl.sourceforge.net`.

[124] M. T. J. Spaan and F. C. A. Groen. Team coordination among robotic soccer players. In G. Kaminka, P. U. Lima, and R. Rojas, editors, *Proceedings of RoboCup International Symposium 2002*, 2002.

[125] Gerald Steinbauer, Michael Faschinger, Gordon Fraser, Arndt Mühlenfeld, Stefan Richter, Gernot Wöber, and Jürgen Wolf. Mostly harmless team description. In Polani et al. [96].

[126] Gerald Steinbauer, Gordon Fraser, Arndt Mühlenfeld, and Franz Wotawa. A modular architecture for a multi-purpose mobile robot. In *Proceedings of the 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, 2004.

[127] Gerald Steinbauer, Martin Mörth, and Franz Wotawa. Real-time diagnosis and repair of faults of robot control software. In *RoboCup 2005: Robot Soccer World Cup IX*, Lecture Notes in Artificial Intelligence, Berlin, Heidelberg, Germany, 2006. Springer-Verlag. (to appear).

[128] Peter Stone and Manuela Veloso. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 110(2):241–273, June 1999.

[129] H. Tamura, S. Sakane, F. Tomita, and N. Yokoya. Design and implementation of spider – a transportable image processing package. *Computer Vision, Graphics and Image Processing*, 23(3):273–294, 1983.

[130] A. Tews and G.F Wyeth. Thinking as one: Coordination of multiple mobile robots by shared representations. In *Proceeding of the 2002 International Conference on Intelligent Robots and Systems (IROS 2000)*, volume 2, pages 1391–1396. IEEE/RSJ, 2000.

[131] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. A probabilistic approach to concurrent mapping and localization for mobile robots. *Machine Learning*, 31(1-3):29–53, 1998.

[132] Scott E. Umbaugh. *COMPUTER VISION and IMAGE PROCESSING: A Practical Approach Using CVIPtools*. Prentice Hall, June 1998.

[133] Hans Utz. *Miro Manual*. University of Ulm, Ulm, Germany, 1999. `http://smart.informatik.uni-ulm.de/Miro/`.

[134] Hans Utz. Quo vadis? Robust, hierarchical navigation for autonomous mobile robots. Master's thesis, University of Ulm, Neuroinformatics, Ulm, Germany, October 2000. (in German).

[135] Hans Utz, Ulrich Kaufmann, and Gerd Mayer. Advanced video image processing on autonomous mobile robots. 19th International Joint Conference on Artificial Intelligence (IJCAI), August 2005. Workshop on Agents in Real-time and Dynamic Environments.

[136] Hans Utz, Gerhard K. Kraetzschmar, Gerd Mayer, and Günther Palm. Hierarchical behavior organization. In *Proceedings of the 2005 International Conference on Intelligent Robots and Systems (IROS 2005)* [55].

[137] Hans Utz, Gerd Mayer, and Gerhard K. Kraetzschmar. Middleware logging facilities for experimentation and evaluation in robotics. 27th German Conference on Artificial Intelligence (KI2004), September 2004. Workshop on Methods and Technology for Empirical Evaluation of Multiagent Systems and Multirobot Teams.

[138] Hans Utz, Gerd Mayer, Dominik Maschke, Alexander Neubeck, Peter Schaeffer, Philipp Baer, Ingmar Baetge, Jan Fischer, Roland Holzer, Markus Lauer, Alexander Reisser, Florian Sterk, Günther Palm, and Gerhard K. Kraetzschmar. The Ulm Sparrows 2001. In Birk et al. [9], pages 677–680.

[139] Hans Utz, Alexander Neubeck, Gerd Mayer, and Gerhard K. Kraetzschmar. Improving vision-based self-localization. In Gal Kaminka, Pedro Lima, and Raul Rojas, editors, *RoboCup 2002: Robot Soccer World Cup VI*, volume 2752 of *Lecture Notes in Artificial Intelligence*, Berlin, Heidelberg, Germany, 2003. Springer-Verlag.

[140] Hans Utz, Stefan Sablatnög, Stefan Enderle, and Gerhard K. Kraetzschmar. Miro – middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures*, 18(4):493–497, August 2002.

[141] Hans Utz, Freek Stulp, and Arndt Mühlenfeld. Sharing belief in teams of heterogeneous robots. In Nardi et al. [90], pages 508–515.

[142] Ingo van Lil. Anpassung und Erweiterung der Middleware für mobile Roboter (Miro) für die mobile Plattform Pioneer 2 AT. Master's thesis, Technical University of Chemnitz, Chemnitz, Germany, 2005. in German.

[143] Richard T. Vaughan, Brian P. Gerkey, and Andrew Howard. On device abstractions for portable, reusable robot code. In *Proceedings of the 2003 International Conference on Intelligent Robots and Systems (IROS 2003)* [54], pages 212–2427.

[144] Manuela Veloso, Hioaki Kitano, Enrico Pagello, Gerhard K. Kraetzschmar, Peter Stone, Tucker Balch, Minoru Asada, Silvia Coradeschi, Lars Karlsson, and Masahiro Fujita. Overview of RoboCup-99. In Veloso et al. [145], pages 1–34.

[145] Manuela Veloso, Hiroaki Kitano, and Enrico Pagello, editors. *RoboCup-99: Robot Soccer World Cup III*, volume 1856 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, Germany, 2000.

[146] Richard Volpe, Issa Nesnas, Tara Estlin, Darren Mutz, Richard Petras, and Hari Das. The CLARAty architecture for robotic autonomy. In *Proceedings of the 2001 IEEE Aerospace Conference*, Big Sky Montana, March 2001. IEEE.

[147] Cornelius Weber, Stephan Wermter, and Alexandros Zochios. Robot docking with neural vision and reinforcement. *Knowledge Based Systems*, 17:165–172, 2004.

[148] Middleware. `http://en.wikipedia.org/wiki/Middleware`.

[149] Wikipedia. `http://en.wikipedia.org`.

[150] Evan Woo, Bruce A. MacDonald, and Félix Trépanier. Distributed mobile robot application infrastructure. In *Proceedings of the 2003 International Conference on Intelligent Robots and Systems (IROS 2003)* [54], pages 1475–1480.