Thomas Röfer

Center for Computing Technology,
FB 3 Informatik,
Universität Bremen,
Postfach 330440,
28334 Bremen, Germany

Hans-Dieter Burkhard
Uwe Düffert
Jan Hoffmann
Daniel Göhring
Matthias Jüngel
Martin Lötzsch

Institut für Informatik,
LFG Künstliche Intelligenz,
Humboldt-Universität zu Berlin,
Rudower Chaussee 25,
12489 Berlin, Germany

Oskar von Stryk
Ronnie Brunn
Martin Kallnik
Michael Kunz
Sebastian Petters
Max Risler
Maximilian Stelzer

Fachgebiet Simulation und Systemoptimierung,
FB 20 Informatik,
Technische Universität Darmstadt,
Alexanderstrasse 10,
64283 Darmstadt, Germany

Ingo Dahm
Michael Wachter
Kai Engel
André Osterhues
Carsten Schumann
Jens Ziegler

Computer Engineering Institute,
Chair of Systemanalysis,
University of Dortmund,
Otto-Hahn-Strasse 4,
44221 Dortmund, Germany

Abstract

The GermanTeam is a joint project of several German universities in the Sony Legged Robot League. This report describes the software developed for the RoboCup 2003 in Padova. It presents the software architecture of the system as well as the methods that were developed to tackle the problems of motion, image processing, object recognition, self-localization, and robot behavior. The approaches for both playing robot soccer and mastering the challenges are presented. In addition to the software actually running on the robots, this document will also give an overview of the tools the GermanTeam used to support the development process.

In an extensive appendix, several topics are described in detail, namely the installation of the software, how it is used, the implementation of inter-process communication, streams, and debugging mechanisms, and the approach of the GermanTeam to model the behavior of the robots.

Contents

1 Introduction **1**
1.1 History . . . 1
1.2 Scientific Goals . . . 1
1.2.1 Humboldt-Universität zu Berlin . . . 1
1.2.2 Technische Universität Darmstadt . . . 2
1.2.3 Universität Bremen . . . 3
1.2.4 Universität Dortmund . . . 3
1.3 Contributing Team Members . . . 4
1.3.1 Aibo Team Humboldt (Humboldt-Universität zu Berlin) . . . 4
1.3.2 Darmstadt Dribbling Dackels (Technische Universität Darmstadt) . . . 4
1.3.3 Bremen Byters (Universität Bremen) . . . 4
1.3.4 Microsoft Hellhounds (Universität Dortmund) . . . 5
1.4 Structure of this Document . . . 5

2 Architecture **6**
2.1 Platform-Independence . . . 6
2.1.1 Motivation . . . 6
2.1.2 Realization . . . 7
2.1.3 Supported Platforms . . . 8
2.1.4 Math Library . . . 8
2.1.4.1 Provided Data Types . . . 8
2.2 Multiple Team Support . . . 9
2.2.1 Tasks . . . 9
2.2.2 Debugging Support . . . 11
2.2.3 Process-Layouts . . . 11
2.2.3.1 Communication between Processes . . . 11
2.2.3.2 Different Layouts . . . 13
2.2.4 Make Engine . . . 13
2.2.4.1 Dependencies . . . 13
2.2.4.2 Realization . . . 14
2.2.4.3 Debugging and Optimization . . . 14
2.2.4.4 Automation and Integration . . . 14

3 Modules in GT2003 **16**
3.1 Body Sensor Processing . . . 18
3.2 Vision . . . 19
3.2.1 Using a Horizon-Aligned Grid . . . 20
3.2.2 Detecting Points on Edges . . . 22
3.2.3 Detecting the Ball . . . 22
3.2.4 Detecting Flags . . . 24
3.2.5 Detecting Goals . . . 24
3.2.6 Detecting Robots . . . 24
3.2.7 Detecting Obstacles . . . 25
3.3 Self-Localization . . . 26
3.3.1 Single Landmark Self-Locator . . . 26
3.3.1.1 Approach . . . 26
3.3.1.2 Results . . . 29
3.3.2 Monte-Carlo Self-Locator . . . 30
3.3.2.1 Motion Model . . . 30
3.3.2.2 Observation Model . . . 31
3.3.2.3 Resampling . . . 32
3.3.2.4 Estimating the Pose of the Robot . . . 33
3.3.2.5 Results . . . 35
3.3.3 Lines Self-Locator . . . 35
3.3.3.1 Observation Model . . . 36
3.3.3.2 Drawing from Observations . . . 38
3.3.3.3 Correcting the Posture Based on Measurements . . . 39
3.3.3.4 Experiments . . . 39
3.4 Ball Modeling . . . 41
3.4.1 Ball Position and Ball Speed . . . 41
3.4.2 Communicated Information About the Ball . . . 41
3.5 Obstacle Model . . . 42
3.5.1 Updating the Model with new Sensor Data . . . 43
3.5.2 Updating the Model Using Odometry . . . 43
3.6 Collision Detector . . . 44
3.7 Player Modeling . . . 46
3.7.1 Determining Robot Positions from Distributions . . . 46
3.7.2 Integration of Team Messages . . . 46
3.8 Behavior Control . . . 47
3.8.1 The Extensible Agent Behavior Specification Language XABSL . . . 48
3.8.1.1 The Architecture behind XABSL . . . 48
3.8.1.2 The XML Specification . . . 49
3.8.2 The Behaviors of the GermanTeam . . . 51
3.8.3 Continuous Basic Behaviors . . . 52
3.9 Motion . . . 53
3.9.1 Walking . . . 54

3.9.1.1 Approach . . . 55
3.9.1.2 Parameters . . . 55
3.9.1.3 Odometry correction values . . . 58
3.9.1.4 Inverse kinematics . . . 58
3.9.2 Special Actions . . . 62
3.9.3 Head Motion Control . . . 63
3.9.3.1 Head Control Modes . . . 63
3.9.3.2 HeadControl State Machine . . . 64
3.9.3.3 Head Path Planner . . . 64
3.9.3.4 Joint Protection . . . 64

4 Challenges **66**
4.1 Black And White Ball . . . 66
4.1.1 Detection of the ball . . . 66
4.1.2 Behavior for the Ball Challenge . . . 67
4.1.3 Results . . . 68
4.2 Localization . . . 68
4.2.1 Behavior Control . . . 68
4.2.2 Head Control . . . 69
4.2.3 Results . . . 70
4.3 Obstacle Avoidance . . . 70

5 Tools **72**
5.1 SimGT2003 . . . 72
5.1.1 Simulation Kernel . . . 73
5.1.2 User Interface . . . 74
5.1.3 Controller . . . 76
5.2 RobotControl . . . 76
5.3 Router . . . 78
5.4 Motion Net Code Generator . . . 80
5.5 Emon Log Parser . . . 81

6 Conclusions and Outlook **82**
6.1 The Competitions in Padova . . . 82
6.2 Future Work . . . 83
6.2.1 Humboldt-Universität zu Berlin . . . 83
6.2.2 Technische Universität Darmstadt . . . 84
6.2.3 Universität Bremen . . . 85
6.2.4 Universität Dortmund . . . 86

7 Acknowledgments **87**

A Installation **89**
A.1 Required Software . . . 89
A.2 Source Code . . . 89
A.2.1 Robot Code . . . 90
A.2.2 Tools Code . . . 91
A.3 The Developer Studio Workspace GT2003.dsw . . . 91

B Getting Started **93**
B.1 First Steps with RobotControl . . . 93
B.1.1 Looking at Images . . . 93
B.1.2 Discover the Simulator . . . 94
B.2 Playing Soccer with the GermanTeam . . . 94
B.2.1 Preparing Memory Sticks . . . 94
B.2.2 Establishing a WLAN Connection . . . 95
B.2.3 Operate the Robots . . . 95
B.3 Explore the Possibilities of the Robot . . . 96
B.3.1 Send Images from the Robot and Create a Color Table . . . 96
B.3.2 Create Own Kicks . . . 96
B.3.3 Test simple behaviors . . . 97
B.3.3.1 Test Basic Behaviors . . . 97
B.3.3.2 Test Options . . . 97
B.4 Configuration Files . . . 97
B.4.1 location.cfg . . . 98
B.4.2 coltable.cfg . . . 98
B.4.3 camera.cfg . . . 98
B.4.4 player.cfg . . . 99
B.4.5 robot.cfg . . . 99
B.4.6 wlanconf.txt . . . 100

C Processes, Senders, and Receivers **101**
C.1 Motivation . . . 101
C.2 Creating a Process . . . 101
C.3 Communication . . . 103
C.3.1 Packages . . . 103
C.3.2 Senders . . . 104
C.3.3 Receivers . . . 105

D Streams **107**
D.1 Motivation . . . 107
D.2 The Classes Provided . . . 107
D.3 Streaming Data . . . 109
D.4 Making Classes Streamable . . . 110
D.4.1 Streaming Operators . . . 110

D.4.2 Streaming using *read()* and *write()* 112
D.5 Implementing New Streams 113

E Debugging Mechanisms 115
E.1 Message Queues 115
E.2 Generic Debug Data 117
E.3 Debug Keys 119
E.4 Debug Macros 120
E.5 Debug Drawings 120
E.6 Modules and Solutions 121
E.7 Stopwatch 122

F XABSL Language Reference 123
F.1 Modularity 123
F.2 Symbol Definitions 125
F.3 Basic Behavior Prototypes 128
F.4 Prototypes for Options 130
F.5 Options 131
F.6 States 133
F.7 Decision Trees 135
F.8 Boolean Expressions 136
F.9 Decimal Expressions 137
F.10 Agents 140

G XABSL Tools 143
G.1 Adopting the Makefile 144
G.2 Using the Makefile 144

H The Xabsl2Engine Class Library 146
H.1 Files of the Xabsl2Engine 146
H.2 Running the Xabsl2Engine on a Specific Target Platform 147
H.3 Creating a New Engine 149
H.4 Registering Symbols 149
H.5 Registering Basic Behaviors 152
H.6 Creating the Option Graph 153
H.7 Executing the Engine 153
H.8 Debugging Interfaces 153

I SimGT2003 Usage 156
I.1 Introduction 156
I.2 Getting Started 157
I.3 Views 157
I.3.1 Scene View 157

I.3.2 Robot View . . . 158
I.3.3 Information Views . . . 158
I.3.3.1 Image Views . . . 158
I.3.3.2 Field Views . . . 159
I.3.3.3 Xabsl2 Views . . . 160
I.4 Scene Description Files . . . 160
I.5 Console Commands . . . 161
I.5.1 Initialization Commands . . . 161
I.5.2 Global Commands . . . 162
I.5.3 Robot Commands . . . 163
I.6 Examples . . . 165
I.6.1 Recording a Log File . . . 165
I.6.2 Replaying a Log File . . . 166
I.6.3 Remote Control . . . 167

J RobotControl Usage **169**
J.1 Starting RobotControl . . . 169
J.2 Application Framework . . . 169
J.2.1 The Debug Keys Toolbar . . . 169
J.2.2 The Configuration Toolbar . . . 171
J.2.3 The Settings Dialog . . . 171
J.2.4 The Log Player Toolbar . . . 172
J.2.5 WLan Toolbar . . . 172
J.2.6 Game Toolbar . . . 172
J.3 Visualization . . . 173
J.3.1 Image Viewer and Large Image Viewer . . . 173
J.3.2 Field View and Radar Viewer . . . 174
J.3.3 Radar Viewer 3D . . . 175
J.3.4 Color Space Dialog . . . 175
J.3.5 Value History Dialog . . . 175
J.3.6 Time Diagram Dialog . . . 176
J.4 The Simulator . . . 176
J.5 Debug Interfaces for Modules . . . 178
J.5.1 Xabsl2 Behavior Tester . . . 178
J.5.2 Motion Tester Dialog . . . 179
J.5.3 Head Motion Tester Dialog . . . 180
J.5.4 Mof Tester Dialog . . . 181
J.5.5 Joystick Motion Tester Dialog . . . 181
J.6 Color Calibration . . . 182
J.6.1 The Color Table Dialog . . . 182
J.6.2 HSI Tool Dialog . . . 183
J.6.3 The TSL Color Segmentation Dialog . . . 186
J.6.4 Camera Toolbar . . . 187

J.7 Other Tools 187
J.7.1 Debug Message Generator Dialog 187

Chapter 1

Introduction

1.1 History

The GermanTeam is the successor of the Humboldt Heroes who already participated in the Sony Legged League competitions in 1999 and 2000. Because of the strong interest of other German universities, in March 2001, the GermanTeam was founded. It consists of students and researchers of five universities: Humboldt-Universität zu Berlin, Universität Bremen, Technische Universität Darmstadt, Universität Dortmund and Freie Universität Berlin. For the RoboCup 2001, the Humboldt Heroes only had reinforcements from Bremen and Darmstadt during the last two or three month before the world championship in Seattle took place.

Since 2002, only the Freie Universität Berlin has not provided active team members. Therefore, the system presented in this document is the result of the work of the team members from the other four universities. Each of these four groups has its own team in the Sony Legged RoboCup League, but they only participated separately in the German Open 2002 and 2003 in Paderborn and formed a single national team in Fukuoka and Padova. The four teams are the *Aibo Team Humboldt* (Berlin), the *Bremen Byters*, the *Darmstadt Dribbling Dackels*, and the *Microsoft Hellhounds* (Dortmund).

1.2 Scientific Goals

All the universities participating have special research interests, which they try to carry out in the GermanTeam's software.

1.2.1 Humboldt-Universität zu Berlin

The main interests of Humboldt University's researchers are robotic architectures for autonomous robots based on mental models and the development of complex behavior control architectures. For robots with complex tasks in natural environments, it is necessary to equip them with behavior control architectures that allow planning based on incomplete information

about the environment, cooperation with and without communication, and actions directed to different goals. These architectures have to integrate both long-term plans and short-term reactive behaviors. On the one hand, they should not stick to dead end decisions, but on the other hand, they should also not change their intentions too frequently for a goal to be reached.

The behavior architecture first developed in 2001 was improved and an XML dialect ("XABSL") for describing behaviors was added in 2002 and largely improved in 2003. It will ultimately be joined with the case based reasoning approach developed in our Simulation League team giving us more flexibility in testing behaviors and allowing for knowledge transfer between the different leagues.

In contrast to other RoboCup leagues, elemental "skills" of the robot are still of integral importance in the Sony league (mainly due to the many degrees of freedom). By skills we mean low level behaviors and actions such as locomotion, ball kicking, searching for the ball, and other perception related tasks. These cannot be viewed as only being sensory processes or simple sequences of motor commands but instead as complex control loops that are hard to model in a high level sense-think-act-architecture. Therefore we plan to follow a more holistic, nature-inspired approach to modeling such processes (see chapter 6.2.1 for details).

1.2.2 Technische Universität Darmstadt

The long-term goals of the team in Darmstadt are conceptual and algorithmic contributions to all sub-problems involved for a successful autonomous team of soccer playing legged robots (perception, localization, locomotion, behavior control).

The group in Darmstadt is developing tools for an efficient kinetic modeling and simulation of legged robot dynamics taking into account masses and inertias of each robot link as well as motor, gear and controller models of each controlled joint. Based on these nonlinear dynamic models computational methods for simulation, dynamic off-line optimization and on-line stabilization and control of dynamic walking and running gaits are developed and applied. These methods currently are applied to develop and implement new, fast, and stable locomotion for legged robots, namely the Sony four-legged robots and a new humanoid robot under development. Until June 2002 a complete three-dimensional dynamical model of the Sony four-legged robot has been implemented based on the kinematic and kinetic data provided by Sony. This model has been used to optimize a fast trot gait in simulation using numerical optimal control methods. In May 2003 the simulation results have been validated through experiments as well as the model has been refined to achieve better agreement to the real motor data and to avoid slipping of the robot's feet on ground. Currently other gait patterns with less symmetry in the feet's relative phases, for which optimization is more demanding but which might lead to faster gaits, are considered. Our goal is to have a large variety of different gait patterns for optimal gaits.

However, new methods for planning and controlling legged locomotion of an *autonomous* robot cannot be investigated independently from the limited hard- and software resources which must be shared with other modules under real-time constraints. Thus, for the competitions in 2003, a fast self-localization algorithm combining Monte-Carlo and single landmark approaches, low-level behavior algorithms using potential fields and an improved walking engine have been

developed in addition to the modules contributed by the other German universities. The new modules have been tested successfully during the RoboCup German Open in April.

1.2.3 Universität Bremen

The main research interest of the group in Bremen is the automatic recognition of the plans of other agents, in particular, of the opposing team in RoboCup. A new challenge in the development of autonomous physical agents is to model the environment in an adequate way. In this context, modeling of other active entities is of crucial importance. It has to be examined, how actions of other mobile agents can be identified and classified. Their behavior patterns and tactics should be detected from generic actions and action sequences. From these patterns, future actions should be predicted, and thus it is possible to select adequate reactions to the activities of the opponents. Within this scenario, the other physical agents should not to be regarded individually. Rather it should be assumed that they form a self-organizing group with a common goal, which contradicts the agent's own target. In consequence, an action of the group of other agents is also a threat against the own plans, goals, and preferred actions, and must be considered accordingly. Acting under the consideration of the actions of others presupposes a high degree of adaptability and the capability to learn from previous situations. Thus these research areas will also be emphasized in the project.

The research project focuses on plan recognition detection of agents in general. However, the RoboCup is an ideal test-bed for the methods to be developed. The technology of plan recognition is of large interest in two research areas: on the one hand, the quality of forecasting the actions of physical agents can be increased, which plays an important role in the context of controlling autonomous robots, on the other hand, it can be employed to increase the robustness and security of electronic markets. This project is also part of the priority program "Cooperating teams of mobile robots in dynamic environments" funded by the Deutsche Forschungsgemeinschaft (German Research Foundation).

In the Sony Legged Robot League, it is the goal of the group from Bremen to establish a robust and stable world model that will allow techniques for opponent modeling developed in the simulation league to be applied to a league with real robots.

1.2.4 Universität Dortmund

The team of the University of Dortmund focuses its research interests on the problems of both sensor fusion and collective behavior.

The former is the combination of different sensor data information in order to build a sensor-fusion based world model. Therefore, efficient techniques to communicate and merge the different world models of the robots are implemented and analyzed. In order to reduce the influence of noisy sensor information, a common problem in the domain of real robots, an increased reliability of object classification is desired. Due to the fact that visual input is the main sensor information in the four-legged league, the focus of activities in this field lies in increasing the quality of the team's object recognition system.

First of all, a novel color classification scheme is introduced, which is adapted to the special needs of robot soccer, e.g. to the quality of the camera images or to the available processing power. In this new TSL representation, all relevant[1] colors are represented by transforming the YUV-chrominance space into a new chroma space *automatically*. Evolutionary algorithms, a powerful optimization technique mimicking Darwinian evolution, have successfully been used to optimize this transformation to robot soccer needs by means of Evolutionary Algorithms.

The term collective behavior addresses the problem of controlling a team of multiple soccer robots coherently. We developed a method for controlling a team of robots based on merged sensor information and the corresponding extracted reliability-information.

1.3 Contributing Team Members

At the four universities providing active team members, many people contributed to the GermanTeam:

1.3.1 Aibo Team Humboldt (Humboldt-Universität zu Berlin)

"Diplom" Students. Uwe Düffert, Daniel Göhring, Matthias Jüngel, Martin Lötzsch.

PhD Student. Jan Hoffmann (Aibo Team Humboldt team leader).

Professor. Hans-Dieter Burkhard.

1.3.2 Darmstadt Dribbling Dackels (Technische Universität Darmstadt)

"Diplom" Students. Ronnie Brunn, Marc Dassler, Martin Kallnik, Michael Kunz, Sebastian Petters, Max Risler, Dirk Thomas.

PhD Student. Max Stelzer.

Professor. Oskar von Stryk (Darmstadt Dribbling Dackels team leader).

1.3.3 Bremen Byters (Universität Bremen)

"Diplom" Students. Tim Laue, Tim Riemenschneider.

Assistant Professor. Thomas Röfer (GermanTeam speaker, Bremen Byters team leader).

[1] Only ten different colors are used, all other colors can be neglected.

1.3.4 Microsoft Hellhounds (Universität Dortmund)

"Diplom" Students. Arthur Cesarz, Sebastian Deutsch, Thomas Dickhöfer, Wenchuan Ding, Kai Engel, Matthias Hebbel, Peter Kudlacik, Andre Osterhues, Jan Prünte, Andreas Reiss, Sebastian Schmidt, Carsten Schumann, Christian Thiel, Michael Wachter.

PhD Students. Ingo Dahm, Jens Ziegler (Microsoft Hellhounds team leaders), Christoph Richter.

1.4 Structure of this Document

This document gives a complete survey over the software of the GermanTeam, i. e. it does not just describe the differences between last year's version [3] and the actual one. Thus new teams only have to read this year's documentation. However, people who already read the report from the year ago will find many repetitions.

Chapter 2 describes the software architecture implemented by the GermanTeam. It is motivated by the special needs of a national team, i.e. a "team of teams" from different universities in one country that compete against each other in national contests, but that will jointly line up at the international RoboCup championship. In this architecture, the problem of a robot playing soccer is divided into several tasks. Each task is solved by a *module*. The implementations of these modules for the soccer competition are described in chapter 3. Chapter 4 describes the solutions for the three so-called challenges.

Only half of the approximately 250,000 lines of code that were written by the GermanTeam for the RoboCup 2003 are actually running on the robots. The other half was invested in powerful tools that provide sophisticated debugging possibilities including the first 3-D simulator for the Sony Legged Robot League. These tools are presented in chapter 5.

The main part of this report is finished by concluding the results achieved in 2003 and giving an outlook on the future perspectives of the GermanTeam in 2004 in chapter 6.

In the appendix, several issues are described in more detail. It starts with an installation guide in appendix A. Appendix B is a quick guide how to setup the robots of the GermanTeam to play soccer and how to use the tools. Then, the GermanTeam's abstraction of *processes*, *senders*, and *receivers* is presented in appendix C, followed by appendix D on *streams* and appendix E on the debugging support. The appendices F, G, and H contain a detailed documentation of the behavior engine used by the GermanTeam in Padova. Finally, the appendices I and J describe the usage of SimGT2003 and RobotControl, the two main tools of the GermanTeam.

Chapter 2

Architecture

The GermanTeam is an example of a national team. The members participated as separate teams in the German Open 2002 and 2003, but formed a single team at the RoboCup in Fukuoka and Padova. Obviously, the results of the team would not have been very good if the members developed separately until the middle of April, and then tried to integrate their code to a single team in only two months. Therefore, an architecture for robot control programs was developed that allows implementing different solutions for the tasks involved in playing robot soccer. The solutions are exchangeable, compatible to each other, and they can even be distributed over a variable number of concurrent processes. The approach will be described in section 2.2. Before that, section 2.1 will motivate why the robot control programs are implemented in a platform-independent way, and how this is achieved.

2.1 Platform-Independence

One of the basic goals of the architecture of the GermanTeam was *platform-independence*, i. e. the code shall be able to run in different environments, e. g. on real robots, in a simulation, or—parts of it—in different RoboCup leagues.

2.1.1 Motivation

There are several reasons to enforce this approach:

Using a Simulation. A Simulation can speed up the development of a robot team significantly. On the one hand, it allows writing and testing code without using a real robot—at least for a while. When developing on real robots, a lot of time is wasted with transferring updated programs to the robot via a memory stick, booting the robot, charging and replacing batteries, etc. In addition, simulations allow a program to be tested systematically, because the robots can automatically be placed at certain locations, and information that is available in the simulation, e. g. the robot poses, can be compared to the data estimated by the robot control programs.

Sharing Code between the Leagues. Some of the universities in the GermanTeam are also involved in other RoboCup leagues. Therefore, it is desirable to share code between the leagues, e. g. the behavior control architecture between the Sony Legged Robot League and the Simulation League.

Non Disclosure Agreement. Until RoboCup 2002, only the participants in the Sony Legged Robot League got access to internal information about the software running on the Sony AIBO robot. Therefore, the universities of all members of the league signed a non disclosure agreement to protect this secret information. As a result and in contrast to other leagues, the code used to control the robots during the championship was only made available to the other teams in the league, but not to the public. This has changed in June 2002, when Open-R became publicly available, but already before, the GermanTeam wanted to be able to publish a version of the system without violating the NDA between the universities and Sony by encapsulating the NDA-relevant code and by the means of the simulator (cf. Sect. 5.1). Although the Open-R SDK is now publicly available, there is no reason for the GermanTeam to remove the platform-independent encapsulation from their code.

2.1.2 Realization

It turned out that platform-dependent parts are only required in the following cases:

Initialization of the Robot. Most robots require a certain kind of setup, e. g., the sensors and the motors have to be initialized. Most parameters set during this phase are not changed again later. Therefore, these initializations can be put together in one or two functions. In a simulation, the setup is most often performed by the simulator itself; therefore, such initialization functions can be left empty.

Communication between Processes. As most robot control programs will employ the advantages of concurrent execution, an abstract interface for concurrent processes and the communication between them has to be provided on each platform. The communication scheme introduced by the GermanTeam 2002 is illustrated in section 2.2.3.1 and in appendix C.

Reading Sensor Data and Sending Motor Commands. During runtime, the data to be exchanged with the robot and the robot's operating system is limited to sensor readings and actuator commands. In case of the software developed by the GermanTeam in 2002, it was possible to encapsulate this part as a communication between processes, i. e. there is no difference between exchanging data between individual processes of the robot control program and between parts of the control program and the operating system of the robot.

File System Access. Typically, the robot control program will load some configuration files during the initialization of the system. In case of the system of the GermanTeam, information as the color of robot's team (red or blue), the robot's initial role (e. g. the goalie), and several tables

(e. g. the mapping from camera image colors to the so-called color classes) are loaded during startup.

2.1.3 Supported Platforms

Currently, the architecture has been implemented on three different platforms:

Sony AIBO Robots. The specialty of the Sony Legged Robot League is that all teams use the same robots, and it is not allowed changing them. This allows teams to run the code of other teams, similar to the simulation league. However, this only works if one uses the complete source code of another team. It is normally not possible to combine the code of different teams, at least not without changing it. Therefore, to be able to share the source code in the GermanTeam, the architecture described above was implemented on the Sony AIBO robots. The implementation is based on the techniques provided by Open-R that form the operation system that natively runs on the robots.

Microsoft Windows. The platform independent environment was also implemented on Microsoft Windows as a part of a special controller in *SimRobot* (cf. Sect. 5.1) and, sharing the same code, in the general development support tool *RobotControl* (cf. Sect. 5.2). Under Windows, the processes are modeled as threads, i. e. all processes share the same address space. This caused some problems with global variables, because they are not shared on the real robots, but they are under Windows. As there is only a small amount of global variables in the code, the problem was solved “manually” by converting them into arrays, and by maintaining unique indices to address these arrays for all threads.

Open-R Emulator under Cygwin. The environment was also implemented on the so-called Open-R emulator that allows parts of the robot software to be compiled and run under Linux and Cygwin. The *router* (section 5.3) that functions as a mediator between the robots and RobotControl was implemented on this platform.

2.1.4 Math Library

To have common access to frequently used mathematical data types, a math library was implemented that encapsulates these data types. It provides data types for vectors (two and three dimensional), matrices (three dimensional), rotation matrices, and translation matrices (two and three dimensional).

2.1.4.1 Provided Data Types

Vector2<T> and Vector3<T> are template classes for vectors with two or three elements, respectively. They provide operators for the inner product and the cross product (^ operator) of two vectors (cross product only for *Vector3<T>*), and functions for the Euclidean

length, transposition, normalization, and the angle between the vector and the x-axis (only *Vector2<T>*).

Matrix3x3<T> is a template class for 3×3-matrices. It provides operators to add and multiply two matrices and operators to multiply a matrix and a vector.

RotationMatrix is a matrix especially for rotations. *RotationMatrix* has various functions: functions to rotate the matrix around all axes, functions returning the actual rotation around all axes, and a function to invert the rotation matrix.

Pose2D and Pose3D are transformation matrices in two and three dimensions. They can be multiplied with vectors and *Pose2D* or *Pose3D*, respectively. In addition they can be rotated by angles and translated by vectors.

2.2 Multiple Team Support

The major goal of the architecture presented in this chapter is the ability to support the collaboration between the university-teams in the German national team. Some tasks may be solved only once for the whole team, so any team can use them. Others will be implemented differently by each team, e. g. the behavior control. A specific solution for a certain task is called a *module*. To be able to share modules, interfaces were defined for all tasks required for playing robot soccer in the Sony Legged League. These tasks will be summarized in the next section. To be able to easily compare the performance of different solutions for same task, it is possible to switch between them at runtime. The mechanisms that support this kind of development are described in section 2.2.2 and in appendix E. However, a common software interface cannot hide the fact that some implementations will need more processing time than others. To compensate for these differences, each team can use its own *process layout*, i. e. it can group together modules to processes that are running concurrently (cf. Sect. 2.2.3).

2.2.1 Tasks

Figure 2.1 depicts the tasks that were identified by the GermanTeam for playing soccer in the Sony Legged Robot League. They can be structured into five levels:

Perception. On this level, the current states of the joints are analyzed to determine the point the camera is looking at. The camera image is searched for objects that are known to exist on the field, i. e. landmarks (goals and flags), field lines, other players, the ball, and general obstacles such as the referees. The sensor readings that were associated to objects are called *percepts*. In addition, further sensors can be employed to determine whether the robot has been picked up, or whether it felt down.

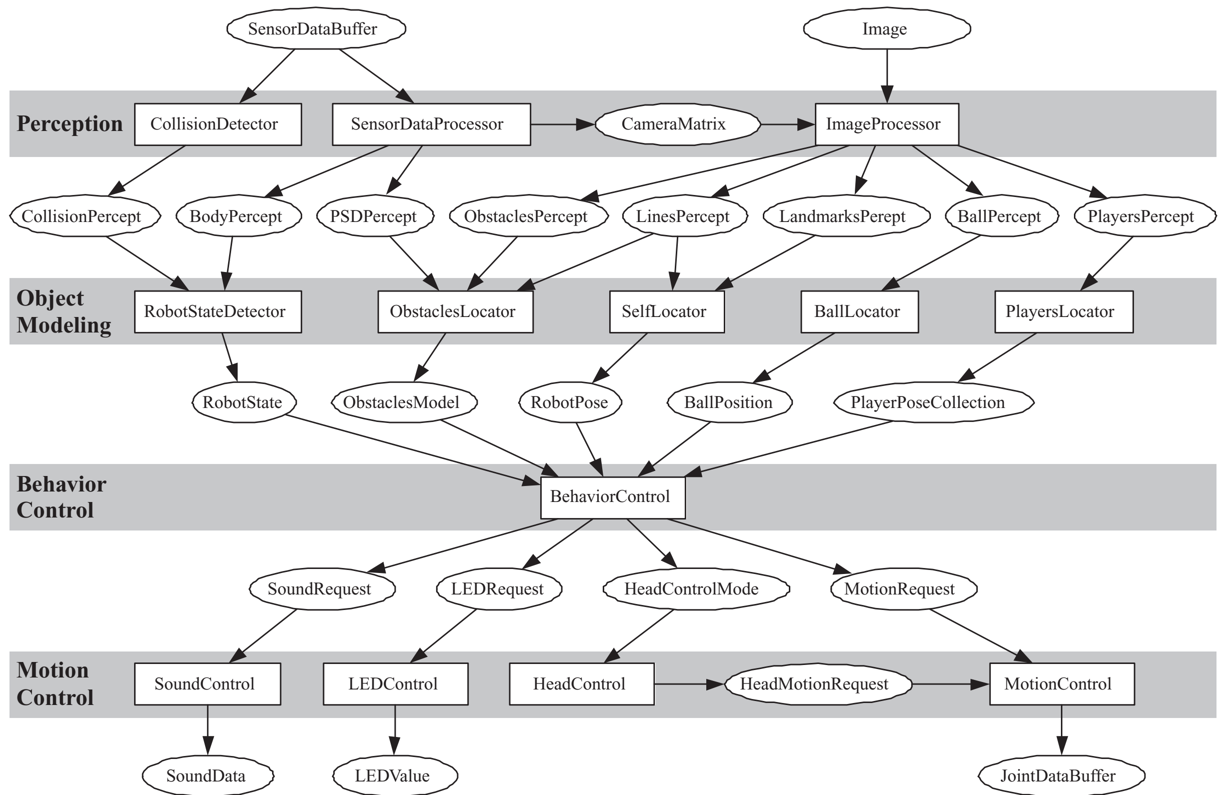

Figure 2.1: The tasks identified by the GermanTeam 2003 for playing soccer.

Object Modeling. Percepts immediately result from the current sensor readings. However, most objects are not continuously visible, and noise in the sensor readings may even result in a misrecognition of an object. Therefore, the positions of the dynamic objects on the field have to be modeled, i. e. the location of the robot itself, the poses of the other robots, the positions of further obstacles, and the position of the ball. The result of this level is the estimated *world state*.

Behavior Control. Based on the world state, the role of the robot, and the current score, the third level generates the behavior of the robot. This can either be performed very reactively, or deliberative components may be involved. The behavior level sends requests to the fourth level to perform the selected motions.

Motion Control. The final level performs the motions requested by the behavior level. It distinguishes between motions of the head and of the body (i. e. walking). When walking or standing, the head is controlled autonomously, e. g., to find the ball or to look for landmarks, but when a kick is performed, the movement of the head is part of the whole motion. The motion module also performs dead reckoning and provides this information to other modules.

This grouping is not strict; it is still possible to implement modules that handle more than a single task, such as the *SensorBehaviorControl* that includes the first three layers in a single module. However, it was not used in the competitions; instead it is mostly used for teaching.

2.2.2 Debugging Support

One of the basic ideas of the architecture is that multiple solutions exist for a single task, and that the developer can switch between them at runtime. In addition, it is also possible to include additional switches into the code that can also be triggered at runtime. The realization is an extension of the debugging techniques already implemented in the code of the GermanTeam 2001 [2]: *debug requests* and *solution requests*. The system manages two sets of information, the current state of all *debug keys*, and the currently active solutions. Debug keys work similar to C++ preprocessor defines, but they can be toggled at runtime (cf. Sect. E.3). A special infrastructure called *message queues* (cf. Sect. E.1) is employed to transmit requests to all processes on a robot to change this information at runtime, i. e. to activate and to deactivate debug keys and to switch between different solutions. The message queues are also used to transmit other kinds of data between the robot(s) and the debugging tool on the PC (cf. Sect. 5.2). For example, motion requests can directly be sent to the robot, images, text messages, and even drawings (cf. Sect. E.5) can be sent to the PC. This allows visualizing the state of a certain module, textually and even graphically. These techniques work both on the real robots and on the simulated ones (cf. Sect. 5.1).

2.2.3 Process-Layouts

As already mentioned, each team can group its modules together to processes of their own choice. Such an arrangement is called a *process layout*. The GermanTeam 2002 has developed its own model for processes and the communication between them:

2.2.3.1 Communication between Processes

In the robot control program developed by the GermanTeam 2001 for the championship in Seattle, the different processes exchanged their data through a shared memory [2], i. e., a blackboard architecture [9] was employed. This approach lacked of a simple concept how to exchange data in a safe and coordinated way. The locking mechanism employed wasted a lot of computing power and it only guaranteed consistence during a single access, but the entries in the shared memory could still change from one access to another. Therefore, an additional scheme had to be implemented, as, e. g., making copies of all entries in the shared memory at the beginning of a certain calculation step to keep them consistent. In addition, the use of a shared memory is not compatible to the ability of the Sony AIBO robots to exchange data between processes via a wireless network.

The communication scheme introduced in 2002 addresses these issues. It uses standard operating system mechanisms to communicate between processes, and therefore it also works via

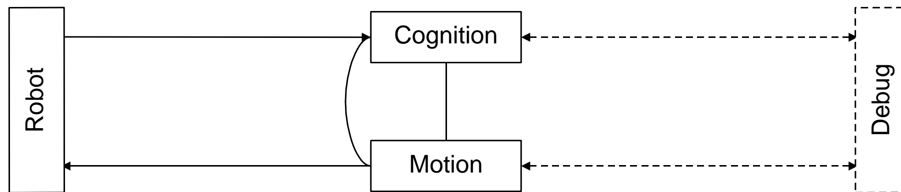

Figure 2.2: The process layout of Humboldt 2002, since RoboCup 2002 used by the whole GermanTeam.

the wireless network. In the approach, no difference exists between inter-process communication and exchanging data with the operating system. Only three lines of code are sufficient to establish a communication link. A predefined scheme separates the processing time into two communication phases and a calculation phase.

The inter-object communication is performed by *senders* and *receivers* exchanging *packages*. A sender contains a single instance of a package. After it was instructed to send the package, it will automatically transfer it to all receivers as soon as they have requested the package. Each receiver also contains an instance of a package. The communication scheme is performed by continuously repeating three phases for each process:

1. All receivers of a process receive all packages that are currently available.

2. The process performs its normal calculations, e. g. image processing, planning, etc. During this, packages can already be sent.

3. All senders that were directed to transmit their package and have not done it yet will send it to the corresponding receivers if they are ready to accept it.

Note that the communication does not involve any queuing. A process can miss to receive a certain package if it is too slow, i. e., its computation in phase 2 takes too much time. In this aspect, the communication scheme resembles the shared memory approach. Whenever a process enters phase 2, it is equipped with the most current data available.

Both senders and receivers can either be blocking or non-blocking objects. Blocking objects prevent a process from entering phase 2 until they were able to send or receive their package, respectively. For instance, a process performing image segmentation will have a blocking receiver for images to avoid that it segments the same image several times. On the other hand, a process generating actuator commands will have a blocking sender for these commands, because it is necessary to compute new ones only if they were requested for. In that case, the ability to immediately send packages in phase 2 becomes useful: the process can pre-calculate the next set of actuator commands, and it can send them instantly after they have been asked for, and afterwards it pre-calculates the next ones.

The whole communication is performed automatically; only the connections between senders and receivers have to be specified. In fact, the command to send a package is the only one that has to be called explicitly. This significantly eases the implementation of new processes.

2.2.3.2 Different Layouts

Since RoboCup 2002, the GermanTeam uses a simple process layout (cf. Figure 2.2) that was originally introduced by Humboldt 2002, consisting of only three modules. More complex layouts developed by the Bremen Byters and the Darmstadt Dribbling Dackels turned out to have more disadvantages than advantages in timing measurements. The first three levels of the architecture are all integrated into the process *Cognition*, because all of them only work with up to date sensor data. The process *Motion* is separate, because sending motor commands always has to work with full frame rate, even if image processing takes too much time. The process *Debug* collects and distributes messages sent through message queues from and to the other processes and the PC. It is only used during the development, and it is inactive in actual RoboCup games.

2.2.4 Make Engine

Using different process layouts requires a quite sophisticated engine to compile the source code. As it is desirable that each process only contains the code that it needs, complex dependencies exist between compilation targets and the source files. For the code that is compiled for Microsoft Windows, process layouts can be represented easily by different project configurations. In addition, it is not required to determine the source code relevant for each process, because under Windows, processes are implemented as threads, and these threads are all part of the same program.

However, on the AIBO, each process is a different binary file, and because memory consumption is crucial, processes should be as small as possible, i. e. only the object files required by a process should be linked together.

2.2.4.1 Dependencies

The directory structure of the source code of the GermanTeam does not reflect which source file belongs to which binary, because the source files have to be grouped based on the selected process layout. In one layout, e. g., several files share the same process while they are distributed over multiple processes in another layout.

Generating dependencies, creating object files, and linking them together is quite time consuming, especially in huge projects that require permanent modifications, expansions, testing, and fine-tuning. Therefore, one major goal of implementing a *make engine* was to execute only the steps absolutely necessary to get a complete build without missing any modifications in source code.

Therefore, a fast and flexible way to generate dependencies between source files and binaries was required. In 2002, the compiler (e. g. the *gcc*) was used to generate object dependencies. This turned out to be very time consuming and required an additional mechanism (not working in all cases) to find out which object dependencies had to be rebuild when certain source files changed. For this year's competition, a simple speed optimized pre-processor called *Depend* (*GT2003/Src/Depend*) written in *C* was developed to speed up the generation of dependencies and to make them more reliable.

2.2.4.2 Realization

For each combination of the chosen process layout, build variant and compiler used, the make engine uses a separate build directory (*$PDIR*, located in *GT2003/Build*) to avoid conflicts between different builds as well as the compulsion for a complete rebuild after changing the process layout, build variant, or compiler. Each such *$PDIR* has two subdirectories: *bin* contains the binaries and *obj* contains objects in the same subdirectory structure as the source code. All these directories will be (re)generated with each start of the build process to be sure not to miss any structural changes.

Then *Depend* is used to completely generate all dependencies for the chosen build target in *GT2003/Build/*/*/depends.incl* each time. Even with several hundreds of source files, this takes only a few seconds. *Depend* checks all includes in all source files taking *defines* and *if[n]defs* into account. Based on this information, it creates a list of all directly or indirectly included files for each **.cpp* file. Thus, all object dependencies for every object possibly needed are available.

After that, all object files required for a certain binary can easily be determined by *Depend* if the code follows some usual conventions such as the existence of header files (**.h*) and implementation files (**.cpp*), and if all items declared in a header file are either implemented by an implementation file with the same base name, or by the header file itself. This results in a list of all object files needed to be linked together for every binary / process of the chosen process layout. So a compiler will never have to touch a source file that is not needed to be linked with one of the binaries, because there is no dependency to it.

2.2.4.3 Debugging and Optimization

Compilers and linkers can be forced to output as many useful warnings as possible, to optimize the code for speed and a certain target architecture such as MIPS R4300 or to simplify debugging e. g. by add debugging symbols. The make engine uses all those options according to the chosen build variant to maximize speed or debuggability or minimize compile time as much as possible.

2.2.4.4 Automation and Integration

In 2002, the project files of Visual Studio had to be updated manually each time the structure of the source code tree changed or files were added or removed. The capability to generate all dependencies and therefore a list of all files used with *Depend* in a short time allows it to generate Visual Studio project files easily from these dependencies. This simplifies maintaining the consistency between the source code tree and the project files. All scripts necessary for that can be found in *GT2003/Src/Depend/*.

It is possible to update or completely rebuild a certain process layout (e. g. CMD) in a special build variant (e. g. Debug) with a single command, either from the command line, e. g. with *./GT.bash CMD Debug*, or from the Microsoft Visual Studio, e. g. by selecting *Rebuild* or *Rebuild All*. All important messages produced by commands in the build process, e. g. error messages of the compiler are converted immediately to a format that is understood by the Visual Studio. Thus, the list of errors and warnings can be browsed by the usual commands, presenting the source files

the messages refer to. This is done with several kinds of source files: not only with source code (**.cpp* and **.h*), but also with motion descriptions (**.mof*).

Chapter 3

Modules in GT2003

The GermanTeam has split the robot's information processing into *modules* (cf. Sect. 2.1). Each module has a specific task and well-defined interfaces. Different exchangeable *solutions* exist for many of the modules. This allows the four universities in the team to test different approaches for each task. In addition, existing and working module solutions can remain in the source code while new solutions can be developed in parallel. Only if the new version is better than the existing ones (which can be tested at runtime), it becomes the *default solution*. Mechanisms for declaring modules and for switching solutions at runtime are described in section E.6.

This chapter describes most of the modules that were implemented. For some solutions only the chosen approach is figured out in detail. The following table lists all modules with all solutions in the code release. The *default solution* is the solution that was used by the GermanTeam for the RoboCup competitions in Padova (marked underlined).

module / task	solutions	developed by
SensorDataProcessor: Processing of body sensor data.	*DefaultSensorDataProcessor*: Calculates the offset and rotation of the camera relative to the ground and detects pressed switches (cf. Sect. 3.1).	Darmstadt
ImageProcessor: Perceiving visual percepts from images.	*GT2003ImageProcessor*: Scans regions near the horizon for interesting objects using a grid. (cf. Sect. 3.2).	Berlin Bremen Darmstadt
	GridImageProcessor2: Used for auto adaptation experiments. (cf. [10]).	Berlin
	GridImageProcessorTSL: Used for experiments with TSL color space. (cf. [5]).	Dortmund
	GridImageProcessor3: Used for recognition of the black and white ball. (cf. Sect. 4.1).	Dortmund
	CheckerboardDetector: Used to recognize a pattern for motion calibration.	Berlin

BallLocator: Modeling of ball position	*PIDSmoothedBallLocator*: Smooths the ball percepts (cf. Sect. 3.4).	Berlin
	BallChallengeBallLocator: Was used for the ball challenge.	Dortmund
ObstaclesLocator: Modeling of the free parts of the field around the robot.	*DefaultObstaclesLocator*: Was used in the games and for challenge 3 (cf. Sect. 3.5).	Berlin Darmstadt
CollisionDetector: Recognizes collisions.	*DefaultCollisionDetector*: Detects collisions by comparison of motor commands and actual movement (as sensed by the servo's position sensor). Not used in the games. (cf. Sect. 3.6).	Berlin
PlayersLocator: Modeling of player positions.	*GO2003PlayersLocator*: Player modeling using a grid from the GermanTeam's 2001 project. Not used in the games (cf. Sect. 3.7).	Darmstadt
SelfLocator: Estimation of the robot's pose.	*Fusion2003SelfLocator*: Combination of GT2003 and SingleLandmark approaches that uses flags, goal posts, and field lines.	Bremen Darmstadt
	FusionSelfLocator: Combination of MonteCarlo and SingleLandmark approaches that uses flags and goal posts.	Bremen Darmstadt
	GT2003SelfLocator: Monte-Carlo-based approach that uses flags, goal posts, and field lines.	Bremen
	LinesSelfLocator: Monte-Carlo-based approach that uses the field lines (cf. Sect. 3.3.3).	Bremen
	MonteCarloSelfLocator: Monte-Carlo-based approach that uses flags and goal posts (cf. Sect. 3.3.2).	Bremen
	SingleLandmarkSelfLocator: Uses detection of a single flag or goal to update position (cf. Sect. 3.3.1).	Darmstadt
RobotStateDetector: Modeling of body sensor events.	*DefaultRobotStateDetector*: Detects fall downs and measures, for how long switches were pressed.	Darmstadt
BehaviorControl: Decision making. (cf. Sect. 3.8	*GT2003BehaviorControl*: An Behavior modules inherited from Xabsl2BehaviorControl that contains state machines that are organized in a tree and formalized in the XML language XABSL2 (cf. Sect. 3.8.1). Continuous basic behaviors realized by simple potential fields are included. Since 2002 all universities of the GermanTeam use this behavior formalization.	Berlin Darmstadt Dortmund

HeadControl: Control of head movement.	*GT2003HeadControl*: A better structured re-implementation of the most promising head control approaches (cf. Sect. 3.9.3).	Berlin Bremen Darmstadt
LEDControl: Sets the robot's LEDs.	*DefaultLEDControl*: Processes the requests from the behavior control.	Darmstadt
SpecialActions: Kicks and other moves.	*GT2003MotionNetSpecialActions*: Executes special actions that were described in a high level motion language. (cf. Sect. 3.9.2)	Berlin Darmstadt Dortmund
WalkingEngine: Walking	*InvKinWalkingEngine*: Parameterized walking engine using inverse kinematics (cf. Sect. 3.9.1).	Darmstadt
MotionControl: Setting of the joint values.	*DefaultMotionControl*: Integrates head motions, walk motions, special actions, and LED values (cf. Sect. 3.9).	Darmstadt
	DebugMotionControl: A tool for developing new motions. Works together with the *MofTester Dialog* in the RobotControl application (cf. Sect. J.5.4).	Darmstadt
SoundControl: Generation and sending of audio data.	*DefaultSoundControl*: Plays wave files for debugging purposes.	Dortmund
SensorBehaviorControl: Sensorimotor coupling.	*ObstacleAvoiderOnGreenField*: Demo. The Aibo avoids everything that is not green.	Berlin

3.1 Body Sensor Processing

The task of the *SensorDataProcessor* is to take the data provided by all sensors except the camera, and to store them, marked with a time stamp, in a buffer. This buffer is used to calculate average sensor values over the last n ticks, or to pick up the sensor values for a given point in time (usually the arrival of a new camera image).

The measurements of the acceleration sensors are used to calculate the tilt and roll of the robot's body. By comparing long term averages and short term averages of these measurements, it is possible to determine whether the robot has been lifted up or whether is has fallen down.

For every incoming image, the *SensorDataProcessor* calculates a matrix that represents the pose of the camera relative to the robot's body origin. This allows the coordinates of objects detected in camera images to be transformed into the robot's system of coordinates.

This transformation is composed of the following sub-transformations:

1. translation along the positive z-axis by the height of the robots neck

2. counterclockwise rotation about the x-axis by the roll angle of the body

3. counterclockwise rotation about the y-axis by the sum of the tilt angle of body and head

4. translation along the positive z-axis by the distance between the neck and the center of pan rotation

5. counterclockwise rotation about the z-axis by the pan angle of the head

6. counterclockwise rotation about the x-axis by the roll angle of the head

7. translation along the positive x-axis by the distance between the center of pan rotation and the camera

The camera matrix is calculated by multiplying the matrices describing these sub-transformations. It is calculated for each 8 ms frame. It is also used to determine the *PSD percept*, a transformation of the PSD distance measurement into robot-centric three-dimensional world coordinates.

3.2 Vision

The vision module works on the images provided by the robot's camera. The output of the vision module fills the data structure *PerceptCollection*. A percept collection contains information about the relative position of the ball, the field lines, the goals, the flags, the other players, and the obstacles. Positions and angles in the percept collection are stored relative to the robot.

Goals and flags are represented by four angles. These describe the bounding rectangle of the landmark (top, bottom, left, and right edge) with respect to the robot. When calculating these angles, the robot's pose (i.e. the position of the camera relative to the body) is taken into account. If a pixel used for the bounding box was on the border of the image, this information is also stored.

Field lines are represented by a set of points (2-D coordinates) on a line. The ball position and also the other players' positions are represented in 2-D coordinates. The orientations of other robots are not calculated.

The free space around the robot is represented in the *obstacles percept*. It consists of a set of lines described by a *near point* and a *far point* on the ground, relative to the robot. The lines describe green segments in the projection of the camera's image to the ground. In addition, for each far point a marking describes whether the corresponding point in the image lies on the border of the image or not. The lines usually start at the image bottom and end where the green of the ground ends or where the image ends (cf. Fig. 3.1b). If the part of the projection of the image that is close to the robot is not green, both points are identical and lie on the rim of the projection. The meaning of the lines is:

- Behind the far point there is an obstacle (if the far point is not marked as *on border*).
- Between the near and the far point there is no obstacle.
- It is unknown whether there is an obstacle before the near point.

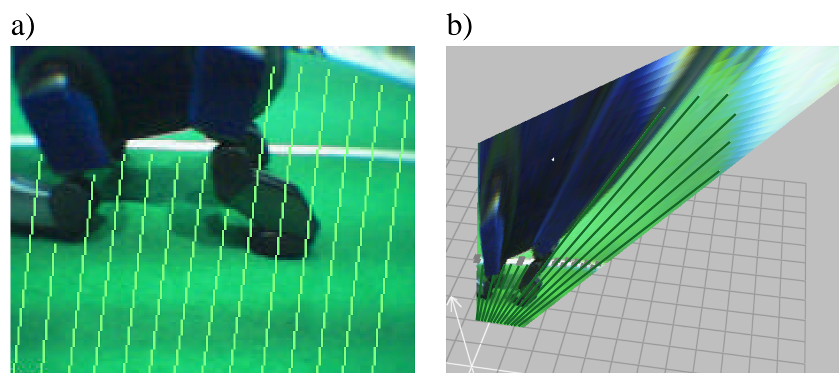

Figure 3.1: Obstacles Percept. a) An image with an obstacle. Green lines: projection of the obstacles percept to the image. b) The projection of the image to the ground. Green lines: obstacles percept.

The images are processed using the high resolution of 176×144 pixels, but looking only at a grid of less pixels. The idea is that for feature extraction, a high resolution is only needed for small or far away objects. In addition to being smaller, such objects are also closer to the horizon. Thus only regions near the horizon need to be scanned at high resolution, while the rest of the image can be scanning using a relatively wide spaced grid.

When calculating the percepts, the robot's pose, i. e. its body tilt and head rotation at the time the image was acquired, is taken into account.

3.2.1 Using a Horizon-Aligned Grid

Calculation of the Horizon. First the position of the horizon in the image is calculated. The robot's lens projects the object from the real world onto the CCD chip. This process can be described as a projection onto a virtual projection plane arranged perpendicular to the optical axis with the center of projection C at the focal point of the lens. As all objects at eye level lie at the horizon, the horizon line in the image is the line of intersection between the projection plane P and a plane H parallel to the ground at height of the camera (cf. Fig. 3.2). The position of the horizon in the image only depends on the rotation of the camera and not on the position of the camera on the field or the camera's height.

For each image the rotation of the robot's camera relative to its body is stored in a rotation matrix. Such a matrix describes how to convert a given vector from the robot's system of coordinates to the one of the camera. Both systems of coordinates share their origin at the center of projection C. The system of coordinates of the robot is described by the x-axis pointing parallel to the ground forward, the y-axis pointing parallel to the ground to the left, and the z-axis pointing perpendicular to the ground upward. The system of coordinates of the camera is described by the x-axis pointing along the optical axis of the lens outward, the y-axis pointing parallel to the horizontal scan lines of the image, and the z-axis pointing parallel to the vertical edges of the image.

To calculate the position of the horizon in the image, it is sufficient to calculate the coordinates of the intersection points h_l and h_r of the horizon and the left and the right edges of the

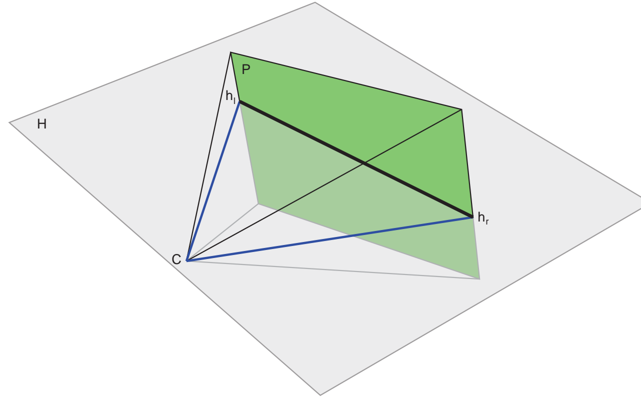

Figure 3.2: Construction of the horizon

image in the system of coordinates of the camera. Let s be the half of the horizontal resolution of the image, α be the half of the horizontal opening angle of the camera. Then

$$h_l = \begin{pmatrix} \frac{s}{\tan\alpha} \\ s \\ z_l \end{pmatrix}, h_r = \begin{pmatrix} \frac{s}{\tan\alpha} \\ -s \\ z_r \end{pmatrix} \tag{3.1}$$

with only z_l and z_r unknown. Let

$$i = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \tag{3.2}$$

be the coordinates of h_l in the system of coordinates of the robot. Solving the equation that describes the transformation between the two systems of coordinates

$$R \cdot i = h_l \tag{3.3}$$

with the rotation matrix

$$R = \begin{pmatrix} r_{11} r_{12} r_{13} \\ r_{21} r_{22} r_{23} \\ r_{31} r_{32} r_{33} \end{pmatrix} \tag{3.4}$$

leads to

$$z_l = -\frac{r_{32}s + r_{31}s \cdot \cot\alpha}{r_{33}}. \tag{3.5}$$

In the same way follows

$$z_r = -\frac{-r_{32}s + r_{31}s \cdot \cot\alpha}{r_{33}}. \tag{3.6}$$

Grid Construction and Scanning. The grid is constructed based on the horizon line, to which grid lines are perpendicular and in parallel. The area near the horizon has a high density of grid lines, whereas the grid lines are coarser in the rest of the image (cf. Fig. 3.3b).

Each grid line is scanned pixel by pixel from top to bottom. During the scan each pixel is classified by color. A characteristic series of colors or a pattern of colors is an indication of an object of interest, e. g., a sequence of some orange pixels is an indication of a ball; a sequence or an interrupted sequence of pink pixels followed by a green, sky-blue, yellow, or white pixel is an indication of a flag; an (interrupted) sequence of sky-blue or yellow pixels followed by a green pixel is an indication of a goal, a sequence of white to green or green to white is an indication of an edge between the field and the border or a field line, and a sequence of red or blue pixels is an indication of a player. All this scanning is done using a kind of state machine; mostly counting the number of pixels of a certain color class and the number of pixels since a certain color class was detected last. That way, beginning and end of certain object types can still be determined although some pixels of the wrong class are detected in between.

3.2.2 Detecting Points on Edges

As a first step toward a more color table independent classification, points on edges are only searched at pixels with a big difference of the y-channel of the adjacent pixels. An increase in the y-channel followed by a decrease is an indication of an edge. If the color above the decrease in the y-channel is sky-blue or yellow, the pixel lies on an edge between a goal and the field. The differentiation between a field line and the border is a bit more complicated. In most of the cases the border has a bigger size in the image than a field line. But a far distant border might be smaller than a very close field line. For that reason the pixel where the decrease in the y-channel was found is assumed to lie on the ground. With the known height and rotation of the camera the distance to that point is calculated. The distance leads to expected sizes of the border and the field line in the image. For the classification these sizes are compared to the distance between the increase and the decrease of the y-channel in the image. The projection of the pixels on the field plane is also used to determine their relative position to the robot.

All vertical scan lines are searched for edges between the goals and the field, because these objects are small and often occluded by robots. In contrast, only every fourth vertical scan line is searched for edge points on the border or on the field lines, but also a few horizontal scan lines are searched for these edge types, because they can appear in vertical direction in the image. As the scanning algorithm assumes to find the border before the field, which is not always true for horizontal scanning, the horizontal scan lines are scanned either from left to right or from right to left by random. If less then three points of a certain edge type are detected in the image, these points are ignored to reduce noise. If many more points on field lines than on the border are detected, the points on the border are dropped, because they are assumed to be misclassified[1].

3.2.3 Detecting the Ball

For balls, upper and lower points on their boundaries are detected during scanning. Points on the border of the image are ignored. During scanning, red pixels below a reasonable number

[1] If a scan line crosses a field line nearly in parallel, the white of the field line appears to be very long on the scan line, which erroneously is an indication for a border.

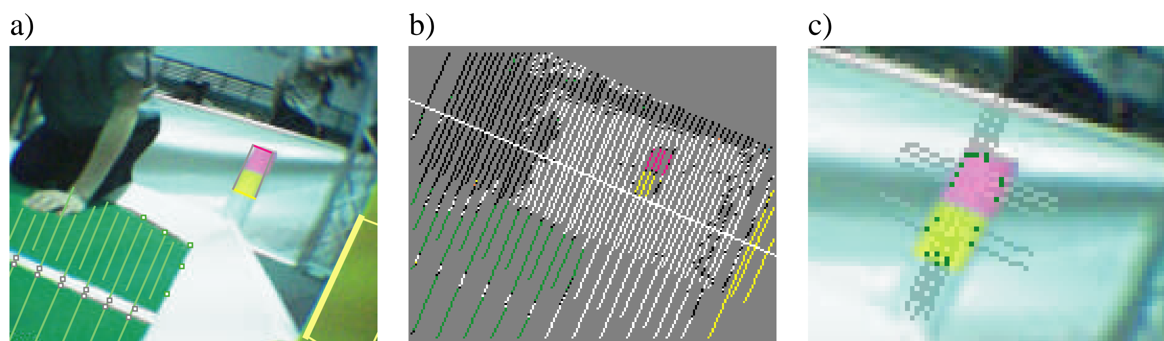

Figure 3.3: Percepts. a) An image and the recognized objects. Dots: pixels on a field line or a border, Flag: pink above yellow, Goal: One edge inside the image and three edges that intersect with the image border. Green lines: obstacles percept. b) The used scan lines. c) Recognition of the flag. Only the gray pixels have been *touched* by the flag specialist. The green pixels mark the edges recognized.

of orange pixels are also treated as orange pixels, because shaded orange often appears as red. Although only a single ball exists in the game, the points are clustered before the actual ball position is detected, because some of them may be outliers on the tricot of a red robot. To remove outliers in vertical direction, upper points are ignored if they are below many other lower points in the same cluster, and lower points are ignored if they are above many other upper points in the same cluster.

The actual calculation of the position of the ball depends on the number of points detected:

- If at least three points have been detected, these can be used to calculate the center of the ball by intersecting the middle perpendiculars. So, three points are selected to construct two middle perpendiculars. Points that are close to green are preferred, because it is assumed that they are actually located on the border of the ball. In contrast, points close to white can result from a high spot on the ball, but also from the real border of a ball in front of the border of the field. First, two point are selected that have the largest distance from each other. Then a third point is chosen that is furthest away from the two other points. If the three points do not lie on a straight line, the center of ball in the image can be calculated, even if the ball is only partially visible. If the ball is not below the horizon or if the camera matrix is not valid because the robot is currently kicking, the distance to the ball is determined from its radius. Otherwise, the distance is determined from the intersection of the ray that starts in the camera and points to the center of the ball with a plane that is parallel to the field, but on the height of the ball center.

- If there is at least a single point on the border of the ball, it is assumed to either be the highest or the lowest point of the ball. The point is projected to the field plane and the distance to the ball is determined from this projection.

- If all orange points lie on the border of the image, it is assumed that the ball fills the whole image and the middle between all orange border points is assumed to be center of the ball.

The position on the field is again determined by intersecting the view ray with the field plane in the height of the ball center.

Finally, the position of the ball in field coordinates is projected back into the image, and a disk around this position is sampled for orange pixels. If enough orange pixels are found, the ball is assumed to be valid.

3.2.4 Detecting Flags

All indications for flags found during scanning the grid are clustered. In each cluster there can actually be indications for different flags, but only if one flag got more indications than the others, it is actually used[2]. The center of a cluster is used as a starting point for the *flag specialist*. It measures the height and the width of a flag. From the initialization pixel the image is scanned for the border of the flag to the top, right, down, and left where top/down means perpendicular to the horizon and left/right means parallel to the horizon (cf. Fig. 3.3c). This leads to a first approximation of the size of the flag. Two more horizontal lines are scanned in the pink part and if the flag has a yellow or a sky-blue part, two more horizontal lines are also scanned there. The width of the green part of the pink/green flags is not used, because it is not always possible to distinguish it from the background. To determine the height of the flag, three additional vertical lines are scanned. The leftmost, rightmost, topmost, and lowest points found by these scans determine the size of the flag. The angles to the four edges of the flag are written into the *PerceptCollection*.

To find the border of a flag, the flag specialist searches the last pixel having one of the colors of the current flag. Smaller gaps with no color are accepted. This requires the color table to be very accurate for pink, yellow, and sky-blue.

3.2.5 Detecting Goals

A goal *specialist* measures the height and the width of a goal. The image is scanned for the borders of the goal from the left to the right and from the top bottom, where again top/down means perpendicular to the horizon and left/right parallel to the horizon.

To find the border of the goal the specialist searches the last pixel having the color of the goal. Smaller gaps with unclassified color are accepted. The maximal size in each direction determines the size of the goal. The angles to the four edges of the goal and the information whether the end of the goal is outside the image are written to the *PerceptCollection*.

3.2.6 Detecting Robots

To determine the indications for other robots, the scan lines are searched for the colors of the tricots of the robots. If a reasonably number of pixels with such a color is found on a scan line, it is distinguished between two cases:

[2]The borders of green flags sometimes appear in sky-blue, and vice versa. The clustering removes such outliers.

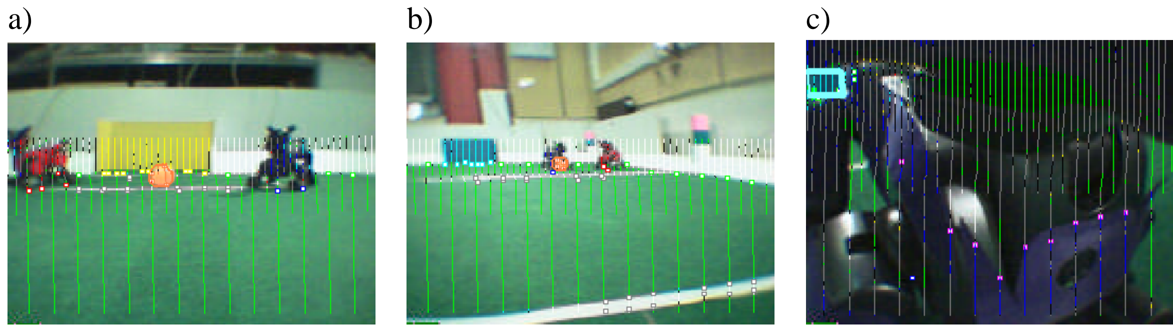

Figure 3.4: Recognition of other robots. a) Several foot points for a single robot are clustered (shown in red and blue). b) Distant robots are still recognized. c) Close robots are recognized based on the upper border of their tricot (shown in pink).

- If the number of pixels in tricot color found on a scan line is above a certain threshold, it is assumed that the other robot is close. In that case, the upper border of its tricot (ignoring the head) is used to determine the distance to that robot (cf. Fig. 3.4c). As with many other percepts, this is achieved by intersecting the view ray through this pixel with a plane that is parallel to the field, but on the "typical" height of a robot tricot. As the other robot is close, a misjudgment of the "typical" tricot height does not change the result of the distance calculation very much. As a result, the distance to close robots can be determined.

- If the number of pixels in tricot color found is smaller, it is assumed that the other robot is further away. In that case, the scan lines are followed until the green of the field appears (cf. Fig. 3.4a, b). Thus the *foot points* of the robot are detected. From these foot points, the distance to the robot can be determined by intersecting the view ray with the field plane. As not all foot points will actually be below the robot's feet (some will be below the body), they are clustered and the smallest distance is used.

3.2.7 Detecting Obstacles

While the scan lines are scanned from top to bottom, a state machine determines the last begin of a green section. If this green section meets the bottom of the image, the begin and the end points of the section are transformed to coordinates relative to the robot and written to the obstacles percept; else or if there is no green on that scan line, the point at the bottom of the line is transformed and the near and the far point of the percept are identical. Inside a green segment, an interruption of the green that has the size of $4 \cdot width_{fieldline}$ is accepted to assure that field lines are not misinterpreted as obstacles ($width_{fieldline}$ is the expected width of a field line in the image depending on the camera rotation and the position in the image).

3.3 Self-Localization

In 2002, the GermanTeam implemented three different methods to solve the problem of self-localization: the *single landmark self-locator* that is based on the approach used by the team from UNSW in 2000, the *Monte-Carlo self-locator* that uses the well known technique of the same name to localize employing the beacons and the goals, and the *lines self-locator*, i. e. a first approach to localization using edges between different objects on the field. Until the German Open 2003, the lines self-locator was completed to work in actual RoboCup games. It was used by the Bremen Byters and for the goalie of the Aibo Team Humboldt. After the German Open, the GT2003 self-locator was created, a combination of the Monte-Carlo self-locator and the lines self-locator. It was able to use both landmarks and edges for localization. After that, the GT2003 self-locator was also combined with the single landmark self-locator to generate candidate postures, resulting in the Fusion2003 self-locator. The combination allowed the number of samples required by the Monte-Carlo localization to be reduced to only 30, i. e. the Fusion2003 self-locator integrates the advantages of all three methods for self-localization developed by the GermanTeam into a single module.

This section will first present the single landmark self-locator developed in 2002, and then the two Monte-Carlo-based approaches, which only differ in the observation model used.

3.3.1 Single Landmark Self-Locator

For the RoboCup 2001 in Seattle, the GermanTeam implemented a Monte-Carlo localization method. The approach proved to be of high accuracy in computing the position of the robot on the field. Sadly this accuracy is achieved by using a quite high amount of CPU-time because the position of the robot is modeled as distribution of particles, and numerous operations have to be performed for each particle (cf. Sect. 3.3.2). Therefore, the Darmstadt Dribbling Dackels tried to develop a localization method that is a lot faster.

3.3.1.1 Approach

To calculate the exact position of the robot on the field one needs at least two very accurate pairs of angle and distance to different landmarks. If more measurements are available, their accuracy can also be lower. However due to the limitations of the hardware, in most situations it is not possible to obtain such measurements with the necessary accuracy or frequency while maintaining the attention on the game. Normally, a single image from the robot's camera contains only usable measurements of at most two landmarks.

Inspired by the approach of UNSW from 2000, a method for self-localization was developed that uses only the two landmarks recognized best within a single image, and only if they are at least measured with a certain minimum quality. In addition to the approach of UNSW, the quality of the measurements is incorporated in the calculation process.

The single landmark self-locator uses information on flags, goals, and odometry to determine the location of the robot. The latter comprises the robot's position, its orientation on the field, and the reliability of the localization. The information on flags and goals consists of the distance

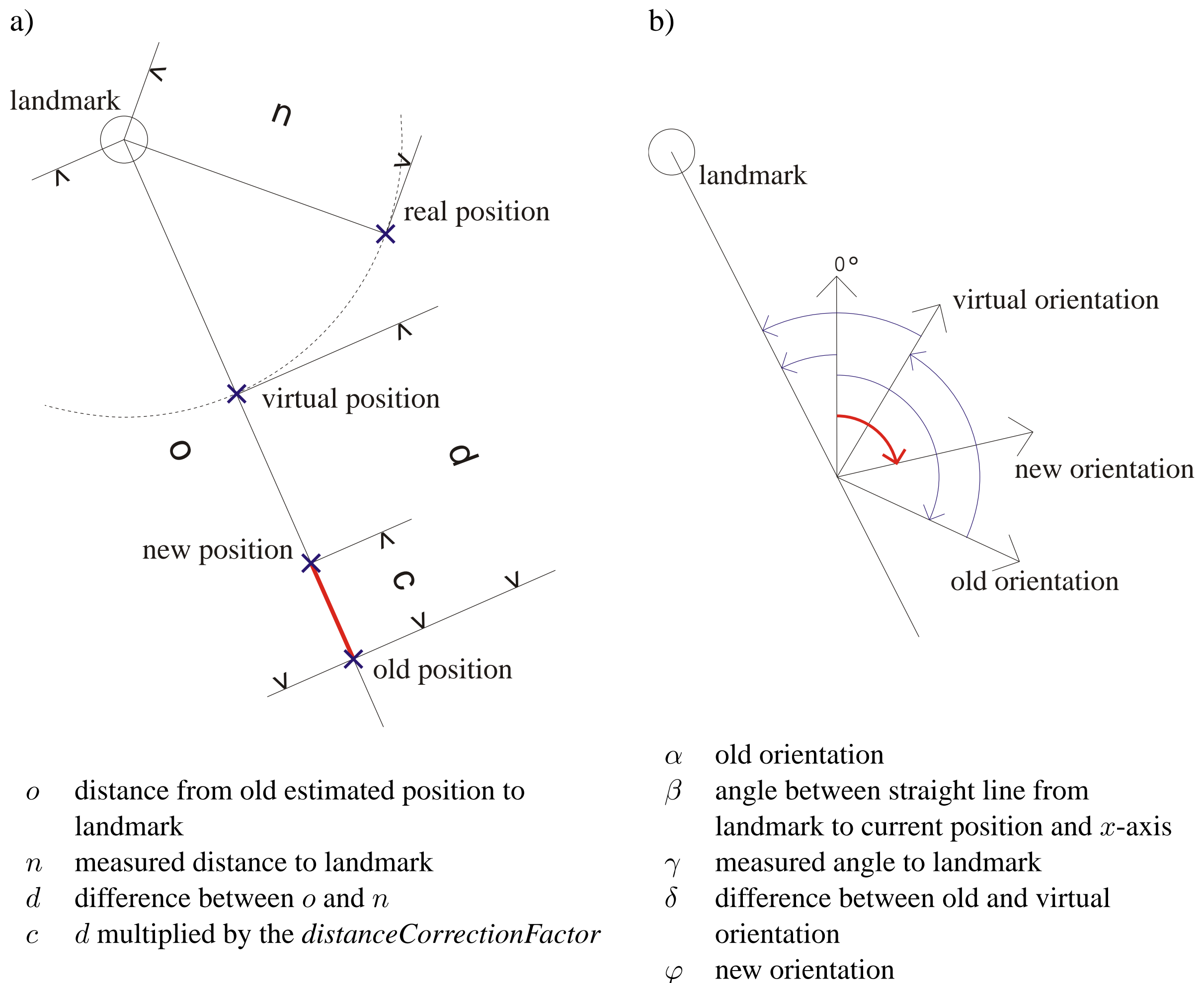

Figure 3.5: One iteration of the single landmark self-locator. a) Position update. b) Orientation update.

of the landmark, the reliability of the distance measurement, the bearing of the landmark, and the reliability of that bearing.

Basic Functionality. The self-localization is performed in an iterative process. For each new set of percepts, the previous pair of position and orientation is updated by the odometry data and their reliability values are decreased. If available, the two best landmark percepts and a goal percept are selected. They will be separately used to compute the new position.

In the following, the algorithm will be described by showing how the estimation of the position is improved with a single landmark measurement. At first the new position is estimated and then the orientation of the robot. The newly measured distance to a landmark is used to compute a virtual position. It is set on the straight line from the last estimated position to the landmark at the measured distance from that landmark. According to the relation between the reliability of the last position and the quality of the measurement, i. e. the *distanceCorrectionFactor*, the new position is interpolated between the last position and the virtual position (cf. Fig. 3.5a).

The reliability of the new position depends on the quality of the measured distance. The relative orientation to the landmark is computed in a similar fashion (cf. Fig. 3.5b). The orientation is interpolated between the old orientation and the newly measured orientation relative to the landmark. The interpolation utilizes the quality of the measured angle and the reliability of the last orientation. The reliability of the new orientation depends on the reliability of the measured angle.

Basically the goal is treated as a landmark with its position at the center of the goal line. Since measurements of distances to goals are usually very imprecise, these measurements have a very low quality and therefore play a minor role in the correction of the estimated position. On the other hand the angle to the goal is considered to be of a good quality and of a great importance. The orientation of the robot relative to the goal is among the most important information in RoboCup, and therefore the goal is taken into account after the landmarks. Thus possible false orientations obtained from the landmarks are overruled by the perception of the goal.

At the end of each cycle the computed position, orientation, and reliability values are stored for the next iteration, and the localization data is provided to other modules. This data includes the reliability that is calculated as the product of the reliabilities of orientation and position.

Expected Behavior. The algorithm described cannot compute an exact position and even makes quite huge errors when the robot sees only the same landmark over a certain period of time. As a matter of fact in such a situation the position can only be corrected along the straight line from the landmark to the old position (cf. Fig. 3.6a). On the other hand, such situations are rare and if the robot can see different landmarks in a series of images, the position will be corrected along changing straight lines, and it will settle down to an fair estimation of the true position (cf. Fig. 3.6b).

Optimization. To enhance the stability of the localization the parameter *progressiveness* was introduced. It controls the rate by which the estimated position approaches the virtual position. Depending on the value of this parameter the algorithm stalls (zero) or keeps working as before (one). A compromise has to be found between stability and the ability to adapt to changing positions very quickly. With values between 0.10 and 0.15 the localization was more stable and according to the speed of the robot it converged near the true position or was slightly behind if the robot was very fast. The results were further improved by changing the method of decrementing the reliability of the old position at the beginning of each cycle. It now depends on how far the robot walked according to the odometry data. Thus the reliability of the old position is lower if the robot walked farther and keeps higher if it stands still.

A reliability that is directly determined from the reliabilities of the input data is not really useful to judge the quality of the current estimate of the robot's position. Therefore, the reliability of a new position and a new orientation integrates the difference between the old values and the virtual values for position and orientation. Thus the reliability decreases if there is a big difference between these values. Thus a single set of unfitting data will not significantly diminish the trust in the localization but continuing conflicting measurements lead to a dropping reliability value and thereby a faster re-localization. If the robot detects continuously inconsistent landmarks or

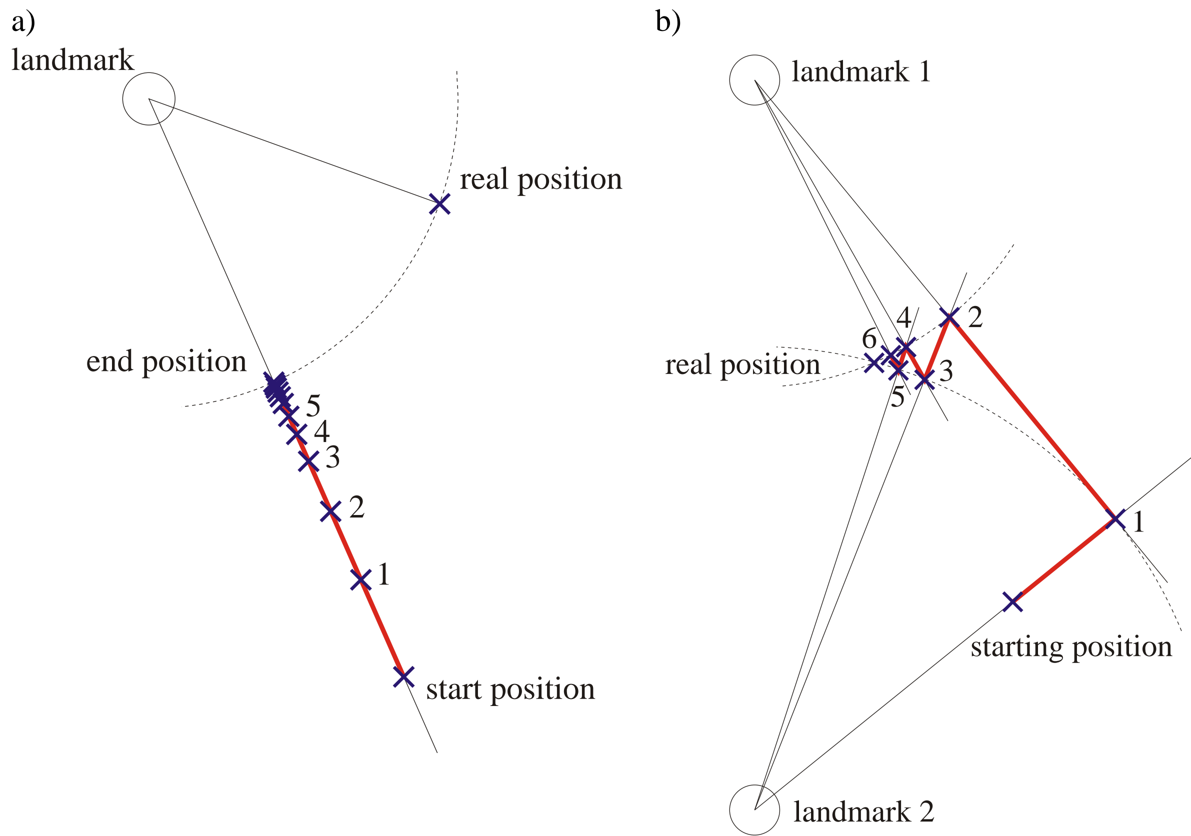

Figure 3.6: a) Problematic situation. The position can only be updated along the straight line to the seen landmark. b) Normal situation. Different landmarks are seen interchangingly and the estimated position settles down near the real position.

no landmarks at all, the reliability value will drop and remain low. Thus, the behavior control can perform the actions required to improve the localization, e. g., to let the head search for landmarks.

3.3.1.2 Results

At the RoboCup German Open 2002 in Paderborn the *Darmstadt Dribbling Dackels* used the single landmark self-locator in all games, while the teams from the other universities in the GermanTeam used the Monte-Carlo self-locator (cf. Sect. 3.3.2). The robots of the *Darmstadt Dribbling Dackels* seemed always to be well-localized on the field. In 2003, the single landmark self-locator was combined with the Monte-Carlo self-locator, using the single landmark self-locator for sensor resetting (Fusion self-locator). This allowed the number of samples in the Monte-Carlo approach to be reduced to 30. At the German Open 2003, Fusion self-locator was used by the goalie and single landmark self-locator was used by the field players. Again, localization worked quite well, and in fact, the *Darmstadt Dribbling Dackels* actually won both the German Open 2002 and 2003.

3.3.2 Monte-Carlo Self-Locator

The *Monte-Carlo Self-Locator* implements a Markov-localization method employing the so-called Monte-Carlo approach [7]. It is a probabilistic approach, in which the current location of the robot is modeled as the density of a set of particles (cf. Fig. 3.8a). Each particle can be seen as the hypothesis of the robot being located at this posture. Therefore, such particles mainly consist of a robot pose, i. e. a vector representing the robot's x/y-coordinates in millimeters and its rotation θ in radians:

$$pose = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} \tag{3.7}$$

A Markov-localization method requires both an observation model and a motion model. The observation model describes the probability for taking certain measurements at certain locations. The motion model expresses the probability for certain actions to move the robot to certain relative postures.

The localization approach works as follows: first, all particles are moved according to the motion model of the previous action of the robot. Then, the probabilities for all particles are determined on the basis of the observation model for the current sensor readings, i. e. bearings on landmarks calculated from the actual camera image. Based on these probabilities, the so-called *resampling* is performed, i. e. moving more particles to the locations of samples with a high probability. Afterwards, the average of the probability distribution is determined, representing the best estimation of the current robot pose. Finally, the process repeats from the beginning.

3.3.2.1 Motion Model

The motion model determines the odometry offset $\Delta_{odometry}$ since the last localization from the odometry value delivered by the motion module (cf. Sect. 3.9) to represent the effects of the actions on the robot's pose. In addition, a random error Δ_{error} is assumed, according to the following definition:

$$\Delta_{error} = \begin{pmatrix} 0.1d \times \text{random}(-1 \ldots 1) \\ 0.02d \times \text{random}(-1 \ldots 1) \\ (0.002d + 0.2\alpha) \times \text{random}(-1 \ldots 1) \end{pmatrix} \tag{3.8}$$

In equation (3.8), d is the length of the odometry offset, i. e. the distance the robot walked, α is the angle the robot turned.

For each sample, the new pose is determined as

$$pose_{new} = pose_{old} + \Delta_{odometry} + \Delta_{error} \tag{3.9}$$

Note that the operation $+$ involves coordinate transformations based on the rotational components of the poses.

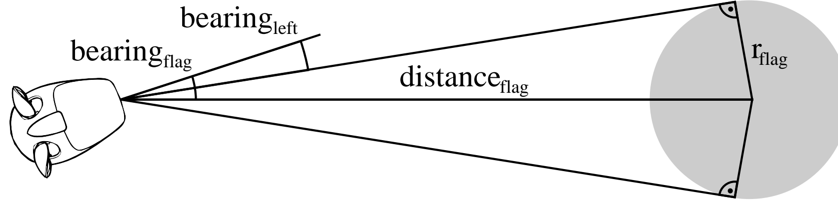

Figure 3.7: Calculating the angle to an edge of a flag.

3.3.2.2 Observation Model

The observation model relates real sensor measurements to measurements as they would be taken if the robot were at a certain location. Instead of using the distances and directions to the landmarks in the environment, i. e. the flags and the goals, this localization approach only uses the directions to the vertical edges of the landmarks. The advantage of using the edges for orientation is that one can still use the visible edge of a landmark that is partially hidden by the image border. Therefore, more points of reference can be used per image, which can potentially improve the self-localization.

As the utilized percepts delivered by the flag/goal specialist (cf. Sect. 3.2.4 and 3.2.5) are bearings on the edges of flags and goals, these have to be related to the assumed bearings from hypothetical postures. To determine the expected bearings, the camera position has to be determined for each particle first, because the real measurements are not taken from the robot's body posture, but from the location of the camera. Note that this is only required for the translational components of the camera pose, because the rotational components were already normalized during earlier processing steps. From these hypothetical camera locations, the bearings on the edges are calculated. It must be distinguished between the edges of flags and the edges of goals:

Flags. The calculation of the bearing on the center of a flag is straightforward. However, to determine the angle to the left or right edge, the bearing on the center $bearing_{flag}$, the distance between the assumed camera pose and the center of the flag $distance_{flag}$, and the radius of the flag r_{flag} are required (cf. Fig. 3.7):

$$bearing_{left/right} = bearing_{flag} \pm \arcsin(r_{flag}/distance_{flag}) \tag{3.10}$$

Goals. The front posts of the goals are used as points of reference. As the goals are colored on the inside, but white on the outside, the left and right edges of a color blob representing a goal even correlate to the posts if the goal is seen from the outside.

Probabilities. The observation model only takes into account the bearings on the edges that are actually seen, i. e., it is ignored if the robot has *not* seen a certain edge that it should have seen according to its hypothetical posture and the camera pose. Therefore, the probabilities of the particles are only calculated from the similarities of the measured angles to the expected angles. Each similarity s is determined from the measured angle $\omega_{measured}$ and the expected

angle $\omega_{expected}$ for a certain pose by applying a sigmoid function to the difference of both angles:

$$s(\omega_{measured}, \omega_{expected}) = \begin{cases} e^{-50d^2} & \text{if } d < 1 \\ e^{-50(2-d)^2} & \text{otherwise} \end{cases} \quad \text{where } d = \frac{|\omega_{measured} - \omega_{expected}|}{\pi} \tag{3.11}$$

The probability p of a certain particle is the product of these similarities:

$$p = \prod_{\omega_{measured}} s(\omega_{measured}, \omega_{expected}) \tag{3.12}$$

3.3.2.3 Resampling

In the resampling step, the samples are moved according to their probabilities. There is a trade-off between quickly reacting to unmodeled movements, e. g., when the referee displaces the robot, and stability against misreadings, resulting either from image processing problems or from the bad synchronization between receiving an image and the corresponding joint angles of the head. Therefore, resampling must be performed carefully. One possibility would be to move only a few samples, but this would require a large number of particles to always have a sufficiently large population of samples at the current posture of the robot. The better solution is to limit the change of the probability of each sample to a certain maximum. Thus misreadings will not immediately affect the probability distribution. Instead, several readings are required to lower the probability, resulting in a higher stability of the distribution. However, if the posture of the robot was changed externally, the measurements will constantly be inconsistent with the current distribution of the samples, and therefore the probabilities will fall rapidly, and resampling will take place.

The filtered probability p' is calculated as

$$p'_{new} = \begin{cases} p'_{old} + 0.1 & \text{if } p > p'_{old} + 0.1 \\ p'_{old} - 0.05 & \text{if } p < p'_{old} - 0.05 \\ p & \text{otherwise.} \end{cases} \tag{3.13}$$

Resampling is done in three steps:

Importance Resampling. First, the samples are copied from the old distribution to a new distribution. Their frequency in the new distribution depends on the probability p'_i of each sample, so more probable samples are copied more often than less probable ones, and improbable samples are removed.

Drawing from Observations. In a second step, some samples are replaced by so-called candidate postures. This approach follows the *sensor resetting* idea of Lenser and Veloso [11], and it can be seen as the small-scale version of the Mixture MCL by Thrun *et al.* [18]: on the RoboCup field, it is often possible to directly determine the posture of the robot from sensor measurements, i. e. the percepts. The only problem is that these postures are not always correct, because

of misreadings and noise. However, if a calculated posture is inserted into the distribution and it is correct, it will get high probabilities during the next observation steps and the distribution will cluster around that posture. In contrast, if it is wrong, it will get low probabilities and will be removed very soon. Therefore, calculated postures are only hypotheses, but they have the potential to speed up the localization of the robot.

Two methods were implemented to calculate possible robot postures. They are used to fill a buffer of *position templates*:

1. The first one uses a short term memory for the bearings on the last three flags seen. Estimated distances to these landmarks and odometry are used to update the bearings on these memorized flags when the robot moves. Bearings on goal posts are not inserted into the buffer, because their distance information is not reliable enough to be used to compensate for the motion of the robot. However, the calculation of the current posture also integrates the goal posts, but only the ones actually seen. So from the buffer and the bearings on goal posts, all combinations of three bearings are used to determine robot postures by triangulation.

2. The second method only employs the current percepts. It uses all combinations of a landmark with reliable distance information, i. e. a flag, and a bearing on a goal post or a flag to determine the current posture. For each combination, one or two possible postures can be calculated.

The samples in the distribution are replaced by postures from the template buffer with a probability of $1 - p'_i$. Each template is only inserted once into the distribution. If more templates are required than have been calculated, random samples are employed.

Probabilistic Search. In a third step that is in fact part of the next motion update, the particles are moved locally according to their probability. The more probable a sample is, the less it is moved. This can be seen as a probabilistic random search for the best position, because the samples that are randomly moved closer to the real position of the robot will be rewarded by better probabilities during the next observation update steps, and they will therefore be more frequent in future distributions.

The samples are moved according to the following equation:

$$pose_{new} = pose_{old} + \begin{pmatrix} 100(1-p') \times \text{random}(-1\ldots1) \\ 100(1-p') \times \text{random}(-1\ldots1) \\ 0.5(1-p') \times \text{random}(-1\ldots1) \end{pmatrix} \tag{3.14}$$

3.3.2.4 Estimating the Pose of the Robot

The pose of the robot is calculated from the sample distribution in two steps: first, the largest cluster is determined, and then the current pose is calculated as the average of all samples belonging to that cluster.

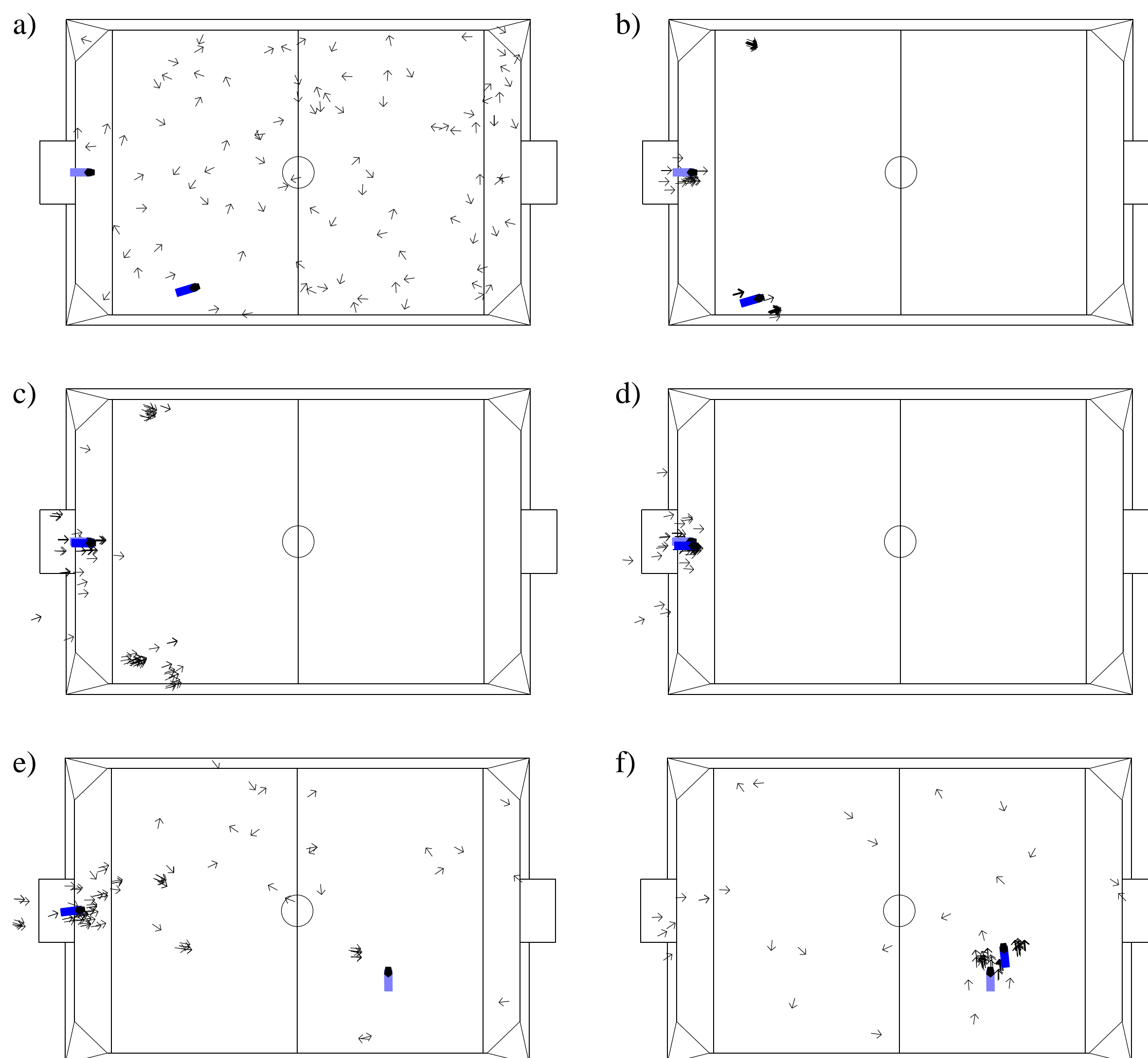

Figure 3.8: Distribution of the samples during the Monte-Carlo localization while turning the head. The bright robot body marks the real position of the robot, the darker body marks the estimated location. a) After the first image processed (40 ms). b) After eight images processed (320 ms). c) After 14 images (560 ms). d) After 40 images (1600 ms). e) Robot manually moved to another position. f) 13 images (520 ms) later.

Finding the Largest Cluster. To calculate the largest cluster, all samples are assigned to a grid that discretizes the x-, y-, and θ-space into $10 \times 10 \times 10$ cells. Then, it is searched for the $2 \times 2 \times 2$ sub-cube that contains the maximum number of samples.

Calculating the Average. All m samples belonging to that sub-cube are used to estimate the current pose of the robot. Whereas the mean x- and y-components can directly be determined, averaging the angles is not straightforward, because of their circularity. Instead, the mean angle

θ_{robot} is calculated as:

$$\theta_{robot} = \text{atan2}(\sum_i \sin\theta_i, \sum_i \cos\theta_i) \tag{3.15}$$

Certainty. The certainty c of the position estimate is determined by multiplying the ratio between the number of the samples in the winning sub-cube m and the overall number of samples n by the average probability in the winning sub-cube:

$$c = \frac{m}{n} \cdot \frac{1}{m} \sum_i p'_i = \frac{1}{n} \sum_i p'_i \tag{3.16}$$

This value is interpreted by other modules to determine the appropriate behavior, e. g., to look at landmarks to improve the certainty of the position estimate.

3.3.2.5 Results

Figure 3.8 depicts some examples for the performance of the approach using 100 samples. The experiments shown were conducted with SimGT2003 (cf. Sect. 5.1) using the *simulation time mode*, i. e. each image taken by the simulated camera is processed. The results show how fast the approach is able to localize and re-localize the robot. At the competitions in Fukuoka and Padova, the method also proved to work on real robots. The GermanTeam was the only team that supported all features of the RoboCup Game Manager that allows the referee to give instructions to the robots. This includes automatic positioning on the field, e. g. for kickoff. For instance, the robots of the GermanTeam were just started somewhere on the field, and then—while still many people were working on the field—they autonomously walked to their initial positions. In addition, the self-localization worked very well on fields without an outer barrier, e. g. on the practice field.

3.3.3 Lines Self-Locator

The previous two approaches use the colored beacons and the goals for self-localization. However, there are no beacons on a real soccer field, and as it is the goal of the RoboCup initiative to compete with the human world champion in 2050, it seems to be a natural thing to develop techniques for self-localization that do not depend on artificial clues. Therefore, the GermanTeam already started in 2002 working on a method to use the field lines to determine the robot's location on the field [15]. It was finished for the German Open 2003, and it was used there by the Bremen Byters and the goalie of the Aibo Team Humboldt [16]. In Padova, it was employed in the localization challenge, and it was used as part of the Fusion2003 self-locator in the soccer competition.

The approach only differs in two aspects from the landmark-based Monte-Carlo self-locator: in the observation model used, and in the way, how candidate postures are drawn from observations (sensor-resetting).

3.3.3.1 Observation Model

The localization is based on the points on edges determined by the image-processing system (cf. Sect. 3.2). Each pixel has an edge type (field, border, yellow goal, or blue goal), and by projecting it on the field, a relative offset from the body center of the robot is determined. Note that the calculation of the offsets is prone to errors because the pose of the camera cannot be determined precisely. In fact, the farther away a point is, the less precise the distance can be determined. However, the precision of the direction to a certain point is not dependent on the distance of that point.

Information Provided by Edge Points. The four edge types provide very different information: *The field lines* are mostly oriented across the field. As lines only provide localization information perpendicular to their orientation, the field lines can only help the robot to find its position along the field. The field lines are seen less often than the border. *The border* is surrounding the field. Therefore it provides information in both Cartesian directions, but it is often quite far away from the robot. Therefore, the distance information is less precise than the one provided by the field lines. The border is seen from nearly any location on the field. *Goals* are the only means to determine the orientation on the field, because the field lines and the border are mirror symmetric. The goals are seen only rarely.

If the probability distribution for the pose of the robot had been modeled by a large set of particles, the fact that different edges provide different information and that they are seen in different frequency would not be a problem. However, to reach real-time performance on an Aibo robot, only a small set of samples can be employed to approximate the probability distribution. In such a small set, the samples sometimes behave more like individuals than as a part of joint distribution. To clarify this issue, let us assume the following situation: as the field is mirror symmetric, only the recognition of the goals can determine the correct orientation on the field. Many samples will be located at the actual location of the robot, but several others are placed at the mirror symmetric variant, because only the recognition of the goals can discriminate between the two possibilities. For a longer period of time, no goal is detected, but the border and the field lines are seen. Under these conditions, it is possible that the samples on the wrong side of the field better match the measurements of the border and the field lines than the correctly located ones, resulting in a higher probability for the wrong position. So the estimated pose of the robot will flip from one orientation alternative to the other without ever seeing a goal. This is not the desired behavior, and it would be quite risky in actual soccer games.

To avoid this problem, separate probabilities for field lines, borders, and goals are maintained for each particle.

Closest Model Points. The projections of the pixels are used to determine the three probabilities of each sample in the Monte-Carlo distribution. As the positions of the samples on the field are known, it can be determined for each measurement and each sample, where the measured points would be located on the field if the position of the sample was correct. For each of these measured points in field coordinates, it can be calculated, where the closest point on a real field line of the corresponding type is located. Then, the horizontal and vertical angles from the camera

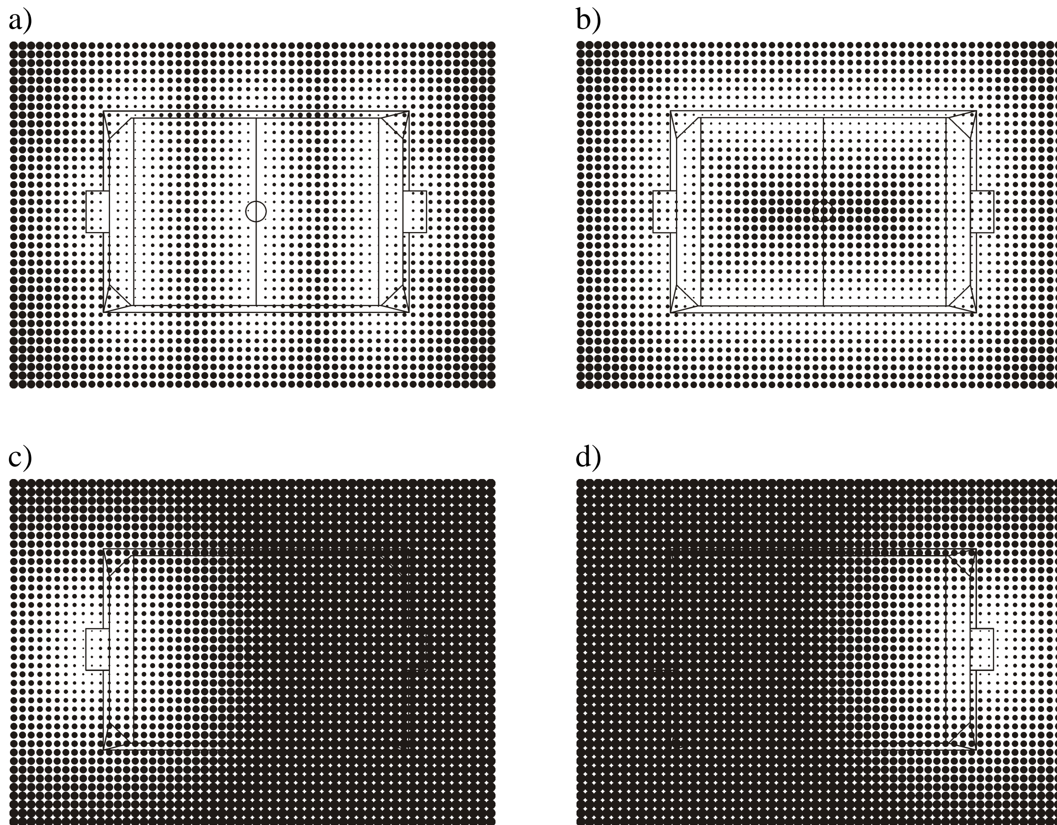

Figure 3.9: Distances from edges. Distance is visualized as thickness of dots. a) Field lines. b) Border. c) One goal. d) The other goal.

to this model point are determined. These two angles of the model point are compared to the two angles of the measured point. The smaller the deviations between the model point and the measured point from a hypothetic position are, the more probable the robot is really located at that position. Deviations in the vertical angle (i. e. distance) are judged less rigidly than deviations in the horizontal angle (i. e. direction).

Calculating the closest point on an edge in the field model for a small number of measured points is still an expensive operation if it has to be performed for, e. g., 100 samples. Therefore, the model points are pre-calculated for each edge type and stored in two-dimensional lookup tables with a resolution of 2.5 cm. That way, the closest point on an edge of the corresponding type can be determined by a simple table lookup. Figure 3.9 visualizes the distances of measured points to the closest model point for the four different edge types.

Probabilities. The observation model only takes the bearings on the edges into account that are actually seen, i. e., it is ignored whether the robot has *not* seen a certain edge that it should have seen according to its hypothetical position and the camera pose. Therefore, the probabilities of the particles are only calculated from the similarities of the measured angles to the expected angles. Each similarity s is determined from the measured angle ω_{seen} and the expected angle

ω_{exp} for a certain pose by applying a sigmoid function to the difference of both angles weighted by a constant σ:

$$s(\omega_{seen}, \omega_{exp}, \sigma) = \mathrm{e}^{-\sigma(\omega_{seen}-\omega_{exp})^2} \tag{3.17}$$

If α_{seen} and α_{exp} are vertical angles and β_{seen} and β_{exp} are horizontal angles, the overall similarity of a sample for a certain edge type is calculated as:

$$q = s(\alpha_{seen}, \beta_{seen}, \alpha_{exp}, \beta_{exp}) = s(\alpha_{seen}, \alpha_{exp}, 10 - 9\frac{|v|}{200}) \cdot s(\beta_{seen}, \beta_{exp}, 100) \tag{3.18}$$

For the similarity of the vertical angles, the probability depends on the robot's speed v (in mm/s), because the faster the robot walks, the more its head shakes, and the less precise the measured angles are.

Calculating the probability for all points on edges found and for all samples in the Monte-Carlo distribution would be a costly operation. Therefore, only a single point of each edge type (if detected) is selected per image by random. To achieve stability against misreadings, resulting either from image processing problems or from the bad synchronization between receiving an image and the corresponding joint angles of the head, the change of the probability of each sample for each edge type is limited to a certain maximum. Thus misreadings will not immediately affect the probability distribution. Instead, several readings are required to lower the probability, resulting in a higher stability of the distribution. However, if the position of the robot was changed externally, the measurements will constantly be inconsistent with the current distribution of the samples, and therefore the probabilities will fall rapidly, and resampling will take place.

The filtered probability q' for a certain edge type is updated ($q'_{old} \rightarrow q'_{new}$) for each point of that type:

$$q'_{new} = \begin{cases} q'_{old} + 0.01 & \text{if } q > q'_{old} + 0.01 \\ q'_{old} - 0.005 & \text{if } q < q'_{old} - 0.005 \\ q & \text{otherwise.} \end{cases} \tag{3.19}$$

The probability p of a certain particle is the product of the three separate probabilities for edges of field lines, the border, and goals:

$$p = q'_{field\ lines} \cdot q'_{border} \cdot q'_{goals} \tag{3.20}$$

3.3.3.2 Drawing from Observations

So far, the observation of edge points has only been used to determine the probability of the robot for being at a certain location. However, observations can also be used to generate candidate positions for the localization, i. e. to place samples at certain positions on the field. As a single observation cannot uniquely determine the location of the robot, candidate positions are drawn from all locations from which a certain measurement could have been made. To realize this, the robot is equipped with a table for each edge type that contains a large number of poses on the field indexed by the distance to the edge of the corresponding type that would be measured from that location in forward direction. Thus for each measurement, a candidate position can be

drawn in constant time from a set of locations that would all provide similar measurements. As all entries in the table only assume measurements in forward direction, the resulting poses have to be rotated to compensate for the direction of the actual measurement.

Such candidate positions are used to replace samples with a low probability. Whether a sample j is replaced or not is also drawn, based on the probability of that sample in relation to the average probability of all samples, i. e. if the following condition is satisfied:

$$\frac{rnd}{n} \sum_{i}^{n} p_i > p_j \tag{3.21}$$

In this case, rnd provides a random number between 0 and 1. If a sample is replaced, the new sample has probabilities q' that are a little bit below the average. Therefore, they have to be acknowledged by further measurements before they are seen as real candidates for the position of the robot.

3.3.3.3 Correcting the Posture Based on Measurements

In contrast to the approach described in section 3.3.2, the measurements used for edge-based localization are distances, and the differences between measured distances and distances to model points give a metric deviation of the robot's posture. This can be used to correct the postures of the samples perpendicularly to the closest edges in the model, accelerating the search of the samples for the real posture of the robot. Of course, the adaptation has to be performed slowly, because the readings are noisy, and this noise again depends on the speed of the robot.

3.3.3.4 Experiments

To judge the performance of the localization approach, two different experiments were conducted. The first one measures the localization error when the robot is continuously moving. The second one evaluates the precision in reaching certain goal points.

Experimental Setup. To be able to evaluate the precision of an approach for self-localization, a reference method for localization is required. Gutmann and Fox [8] have analyzed different localization approaches using the Aibo by manually controlling the robot around using a joystick, and whenever it reached a position that was previously marked, they stored the position of that marker and the position as calculated by the robot in a log file. They also stored all perceptions of the robot, allowing them to test different localization approaches based on the same data.

The setup used for the experiments presented in this paper is a little bit different. To be able to continuously track the position of the robot, a laser range finder was placed at the border of the field. Within its opening angle of 180°, it measured distances in a height of 35 cm, i. e. above the goals. The robot used for the experiment was equipped with a paper tube on its back that was high enough to be detected by the laser range finder (cf. Fig. 3.10). This way, the position of the robot could easily be determined by searching for an area that was significantly closer to the laser scanner than the neighboring areas. The shortest distance within that area plus the radius

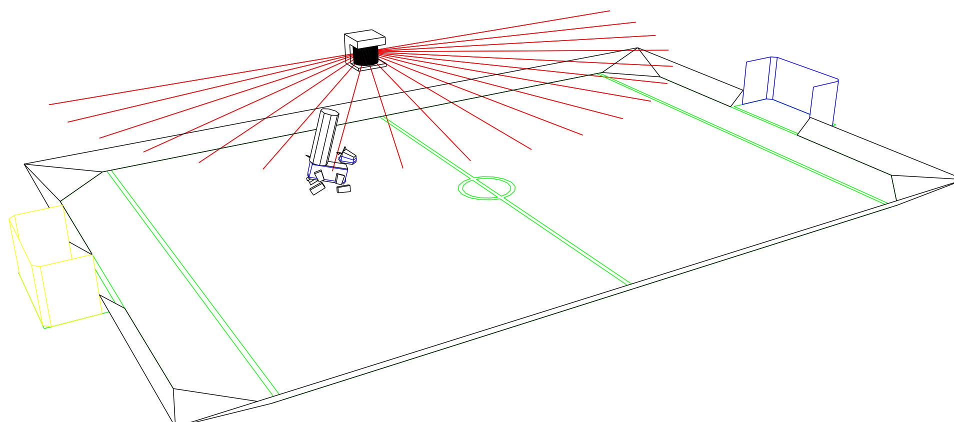

Figure 3.10: The experimental setup. The laser scanner is fixed to the border of the field. The robot carries a vertical paper tube on its back that is measured by the laser sensor.

of the tube was used as distance to the robot. Together with the angle under which the robot was measured, the exact location of the robot was determined.

In both experiments, the robot was continuously turning its head from left to right and vice versa. The Monte-Carlo localization method used 100 samples.

Experiment 1. The goal of the first experiment was to judge the precision of the localization approach when the robot is continuously moving. To accomplish this, the robot was randomly moved around on the field with a maximum speed of 15 cm/s using a joystick. The positions of the robot as calculated by the robot itself and as measured by the laser scanner were stored in a file. The experiment took about 18 minutes, resulting in approximately 5300 measurements.

The result was an average error of 10.5 cm, i. e. less than 4% of the width of the soccer field and less than 2.2% of its length. 60% of the measurements had an error less than this average. Figure 3.11a shows the path traveled and the errors made. Please note that this outcome is similar to the results presented in [8], with the two exceptions that Gutmann and Fox used color marks for localization, and that they performed their experiments on a small 3m$\times$2m field. In addition, they worked on a log file, allowing them to optimally adjust the parameters of their algorithms, e. g. the Monte-Carlo localization approach used needed only 30 samples.

Experiment 2. The goal of the second experiment was to evaluate the precision in reaching certain goal points. In this experiment, random goal positions were given to the robot. The system then performed the so-called *go-to-point* skill to reach the specified location. When the robot did not move anymore, the coordinates of the goal position, the position calculated by the robot, and the position measured by the laser scanner were stored in a file. In the experiment, 68 positions had to be reached.

There were two results: the average error between the goal position and the position reported by the laser scanner was 9.4 cm. 66% of the goals were reached with smaller deviations. However, the go-to-point skill does not reach the goal position precisely. It often stops one or two cm too

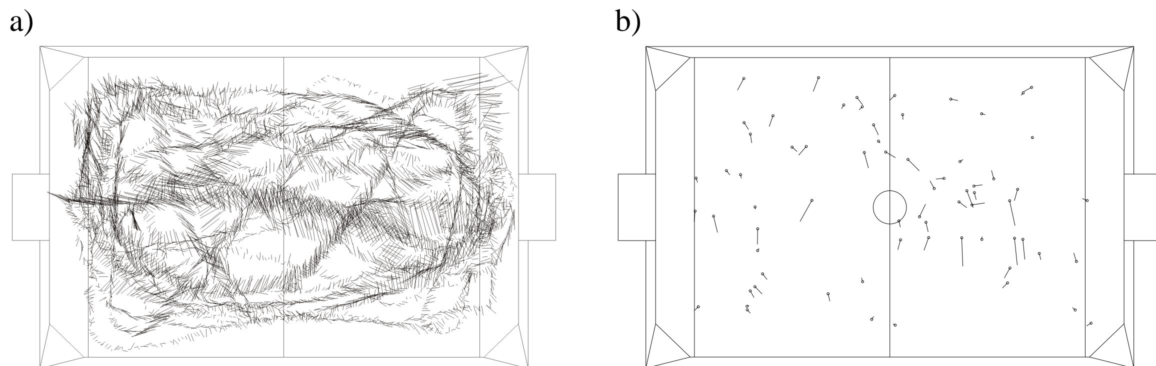

Figure 3.11: Experimental results. Each line connects a position calculated by the robot with one determined by the laser scanner. a) First experiment. b) Second experiment.

early. Therefore, the average error between the position measured by the robot and the position measured by the laser sensor is smaller, namely 8.4 cm. 60% of the goals were even reached with a smaller error. Figure 3.11b shows the 68 goal positions and the positions reached by the robot.

3.4 Ball Modeling

3.4.1 Ball Position and Ball Speed

It is of great importance for all players to keep track of the position of the ball even if they are not able to see it from where they are. Therefore, a model of the ball is created including the ball's position and speed.

The ball's position is derived geometrically from the "ball percept" taking into account the robot's pose. The ball speed was calculated from the current and the last ball position perceived. Both values are smoothed to lessen the effect of noise using a floating average.

3.4.2 Communicated Information About the Ball

In addition to this information, meta data is stored that describes whether or not the robot actually saw the ball or if the ball's position was communicated to it by other robots. This distinction is important because of two reasons:

- The way the robot moves its head: it performs a periodic left-right scanning motion, scanning its surroundings for the ball, other players, and landmarks. Due to the small opening angle of the robot's camera, the ball cannot be seen by the robot during some intervals of the scanning motion even if the robot is relatively close to the ball.

- The different errors of the ball measurements: while a robot is able to perceive where the ball is with sufficient accuracy (ball position in coordinates relative to the robot), communicating the ball's position from one robot to another requires the use of a global system of

coordinates. Since the robots are only localized within a certain accuracy, the localization errors of both robots accumulate and deteriorate the quality of the information communicated.

If the robot sees the ball (or has *recently* detected the ball in the camera image), this information is used. If the robot was unable to see the ball for some time, the ball position is derived from where other robots perceived the ball using the "team ball locator". This means that three different situations need to be differentiated:

Ball Was Seen. The ball was seen and detected in the camera image (e.g. when the robot is directly looking at the ball). If the ball was perceived (i. e. the percept collection contains a ball percept) the position of the ball is determined from the offset stored in the percept and the actual position of the robot yielding a global position of the ball.

Ball Was Not Perceived for a Short Period of Time. This happens, e. g., if the position of the ball makes it difficult to process the image and to detect the ball in some images but not in all. This was also introduced to make the ball model more robust against errors in image processing. When the robot is looking at the ball, image processing does not necessarily detect the ball in all sequential images. This is due to motion blur, temporary obstruction of the ball and special cases in which the image processing algorithm does not yield good results. To describe the situation where the robot sees the ball most of the time (but not necessarily in every single image), a time called "consecutive time ball seen" was introduced. Odometry is used to correct the ball position in the cases, in which it is not seen.

Ball Was Not Seen for Some Time. This is the case when the ball is completely obscured from where the robot is standing or the robot is simply looking the other way. If the ball was not seen for some time (i.e. no ball percept was generated by image processing for a number of seconds), the ball position communicated by other robots will be used.

3.5 Obstacle Model

In the obstacles model, a radar-like view of the surroundings of the robot is created. To achieve this, the surroundings are divided into 90 (micro-) sectors. For each of the sectors the free distance to the next obstacle is stored (see Fig. 3.12). In addition to the distance, the actual measurement that resulted in the distance is also stored (in x,y-coordinates relative to the robot). These are called representatives. Each sector has one representative.

For most applications, the minimum distance in the direction of a single sector is not of interest but rather the minimum value in a number of sectors. Usually, a sector of a certain width is sought-after, e. g. to find a desirable direction free of obstacles for shooting the ball. Therefore, the obstacle model features a number of analysis functions (implemented as member functions) that address special needs for obstacle avoidance and ball handling. One of the most frequently used functions calculates the free distance in a corridor of a given width in a given direction. This

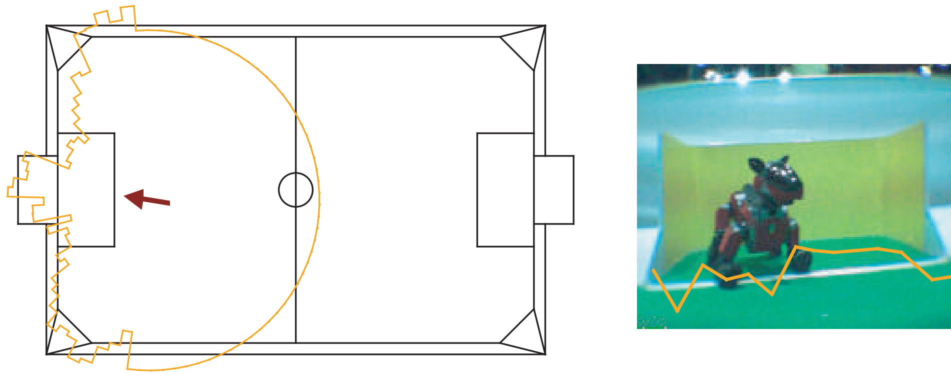

Figure 3.12: The obstacle model as seen from above and projected into the camera image. The robot is in front of the opponent goal.

can be used to check if there are any obstacles in the direction the robot is moving in and also if there's enough room for the robot to pass through.

3.5.1 Updating the Model with new Sensor Data

The robot performs a scanning motion with the camera. The sectors which are within opening angle of the camera can be updated. Image processing can yield two points, the first corresponding to the lower image boundary and the second corresponding to either the distance of an detected obstacle or the upper boundary of the image (if now obstacle was detected). This is necessary because the image only gives information about a certain distance range (due to the vertical opening angle, see fig. 3.13).

Earlier versions of the obstacle model also used the PSD distance sensor of the robot. This was not used in the competition because image processing yielded better, more detailed data. However, most of what has been said can also be applied to the case when only the PSD is used which is extremely useful in domains other then RoboCup where there may be little or no knowledge about the surface available.

3.5.2 Updating the Model Using Odometry

The distances stored in the sectors are adjusted according to how the robot moves. To do this, the representatives are translated and rotated by the robot odometry. The odometry corrected representatives are then used to re-calculate the distances stored in the sectors.

This method has one drawback: it occurs frequently that after moving the representatives by the odometry, two or more representatives are placed in one new sector. In this case, the one with the smallest distance to the robot is used; the other, further away representatives are discarded. This, however, results in the total number of representatives being smaller than the total number of sectors, which results in sectors of unknown distance. This is acceptable for most applications

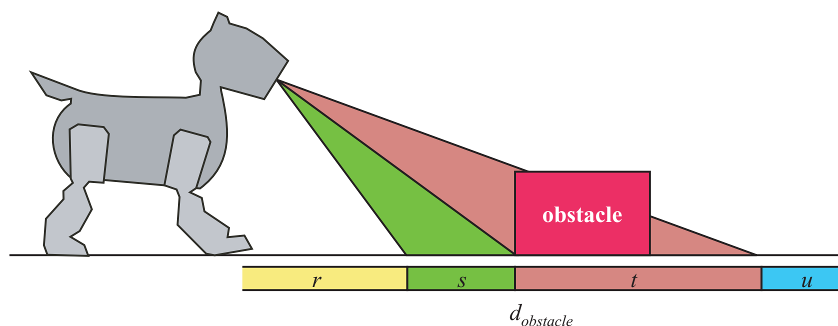

Figure 3.13: The above diagram depicts the robot looking at an obstacle. The robot detects some free space in front of it (s) and some space that is obscured by the obstacle (t). The obstacle model is updated according to the diagram (in this case the distance in the sector is set to $d_{obstacle}$ unless the distance value stored lies in r).

since usually a single sector is not of interest.

Obstacle avoidance based on the obstacle model described here was used in the RoboCup competition in Padova for a number of applications. It did, however, prove to be difficult to make good use of the information. One example to illustrate this is the case of two opposing robots going for the ball: in this case, obstacle avoidance is not desirable and would cause the robot to let the other one move forward. Many such situations are imaginable which resulted in a very limited use of the model so far. Future work will investigate ways of using obstacle avoidance, collision detection, and - ultimately - path planning in more thorough, extensive fashion.

See also section 4.3 for how the model was used successfully for moving around static obstacles swiftly in the obstacle avoidance challenge.

3.6 Collision Detector

A method for collision detection was implemented. Knowledge about whether or not a robot is running into something can obviously be used to have the robot act accordingly. In addition, collision detection can be employed to improve self-localization by adding a validity measure to the odometry data.

Since the Aibo is not equipped with sensors to directly perceive the contact to obstacles, ways of detecting collisions using the sensor readings from the servo motors of the robot's legs were investigated. It was found that under laboratory conditions, comparison of motor commands and actual movement (as sensed by the servo's position sensor)—after having compensated for the phase shift between the two signals—yields good results, i.e. collisions and obstructions are detected reliably. When the concept was applied to the RoboCup environment, it had to be extended to cope with arbitrary movements and accelerations produced by the behavioral layers of the agent.

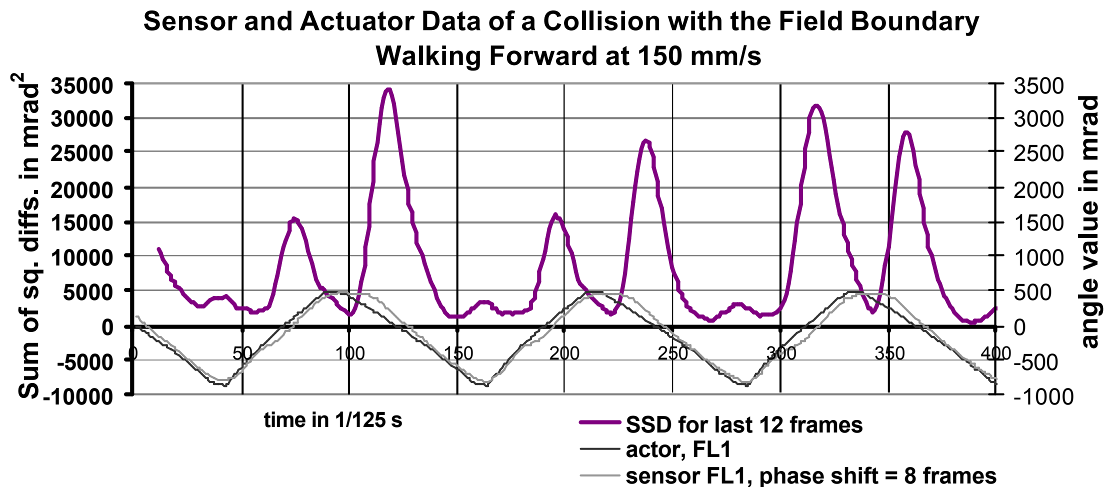

Figure 3.14: The graph shows the actuator and sensor curves and the sum of the squared differences (SSD). Peaks in the SSD curve correspond to collisions.

In an ideal world, the actuator commands and the servo motor's direction sensor readings should be congruent. If this is the case, collisions can be detected by calculating the differences between the two signals and comparing them against a threshold value (see fig. 3.14).

In the actual implementation, two things had to be considered:

- In order to make the method robust against sensor noise, not only one pair of actuator/sensor signals was compared but a sum over the last 12 squared differences (SSD) was used.

- A phase shift of variable length between the signals was observed. This phase shift is due to various reasons such as the amplitude of the signal, whether or not the robot's feet are touching the ground, and others. To compensate for this phase shift, the sum of squared differences is being calculated for a range of possible phase shifts. The smallest value of the set of SSDs was used for comparison against the threshold.

The threshold value depends on the speed of the robot. Threshold values are stored in a table and need to be calibrated for a given gait.

Using this approach we were able to detect robot collisions under laboratory conditions, i. e. if the robot was moving in a straight line or rotating at a given speed. If, however, the signals were investigated in actual game situations, quick, abrupt changes in motor commands made it impossible to detect collisions. The motor commands need to be filtered to differentiate those signals that are caused by (external) collisions from those caused by extreme changes in the motor commands.

A sample behavior was developed for the agent in which the robot would turn away from obstacles when a collision was detected. First steps towards using obstacle avoidance were taken but in the case of obstacle avoidance, finding an "appropriate" behavior once a collision is de-

tected is not a trivial task in a competitive environment. Future work will explore possibilities of finding appropriate behaviors and using collision detection to improve localization.

3.7 Player Modeling

The knowledge of other robots positions is important for avoiding collisions and for tactical planning. The locator for other players performs the calculation of these positions based on players percepts. In addition, positions of teammates received via the wireless network communication are integrated.

3.7.1 Determining Robot Positions from Distributions

The positions of percepts of other robots are relative to the position of the observing robot. In a first step, they are converted to absolute positions on the field. In a second step, it is tested, whether the absolute positions of the percepts are outside the field. In this case, they are projected to the border along an imaginary line which connects the robot with the absolute position of the player percept. The resulting positions of the percepts are stored in a list for about two seconds.

The soccer field is discretized as a grid. The positions of the percepts are converted into grid points, and distributions in x and y directions are created. Then, the maxima in these distributions are determined. A maximum results from a high density of perceived robots at a certain location in the grid. The maxima are sorted by their distinctiveness in descending order. If a maximum is above a certain threshold, a robot is assumed to be located at the corresponding point. The point in the grid is converted to an absolute position on the soccer field. Finally, this position is added to the *PlayersCollection* that contains the positions of all players recognized.

The process described above is done separately for the opponents and for the teammates.

3.7.2 Integration of Team Messages

The positions of the teammates are communicated between the robots via the wireless network. In a first approach these positions are also used for the localization of other robots. It is assumed that a position sent by a teammate is often more precise than a position calculated from the percept showing that teammate. Therefore, positions communicated by teammates are used by the players locator.

The positions resulting from percepts are replaced by the transmitted ones. If the robot has not received the positions from all teammates, or if the last position received is too old, the positions calculated from percepts are kept. To avoid representing a teammate twice, a position calculated from percepts must have a minimum distance to all positions received from teammates.

3.8 Behavior Control

The module *BehaviorControl* is responsible for decision making based on the world state, the game control data received from the *RoboCup Game Manager*, the motion request that is currently being executed by the motion modules, and the team messages from other robots. It has no access to lower layers of information processing.

It outputs the following:

- A *motion request* that specifies the next motion of the robot,
- a *head motion request* that specifies the mode how the robot's head is moved,
- a *LED request* that sets the states of the LEDs,
- a *sound request* that selects a sound file to be played by the robot's loudspeaker,
- a *behavior team message* that is sent to other players by wireless communication.

For behavior control the German Team uses the *Extensible Agent Behavior Specification Language* XABSL [13] since 2002 and improved it largely in 2003. Section 3.8.1 gives an introduction into this architecture.

For the German Open 2003, each of the four universities of the GermanTeam used XABSL for behavior control and continued the behaviors that were developed by the German Team for Fukuoka. They all followed different approaches:

The Darmstadt Dribbling Dackels from the Technische Universität Darmstadt won the German Open 2003. They implemented a dynamic role assignment. They introduced *continuous basic behaviors* (cf. 3.8.3) approach for the low level skills.

The Aibo Team Humboldt from the Humboldt-Universität zu Berlin reached the second place at the German Open 2003. They mainly focused on obstacle avoidance, team strategy, and sensor-actuator coupling.

The Microsoft Hellhounds from the University of Dortmund developed a trainer and a strategical database on top of XABSL.

The Bremen Byters from the Universität Bremen merged the existing XABSL behaviors with their potential field approach.

After the German Open 2003 the behaviors of the teams could be easily merged into a common solution that was continued until the RoboCup 2003 in Padova. Section 3.8.2 describes the strategies and behaviors of the GermanTeam.

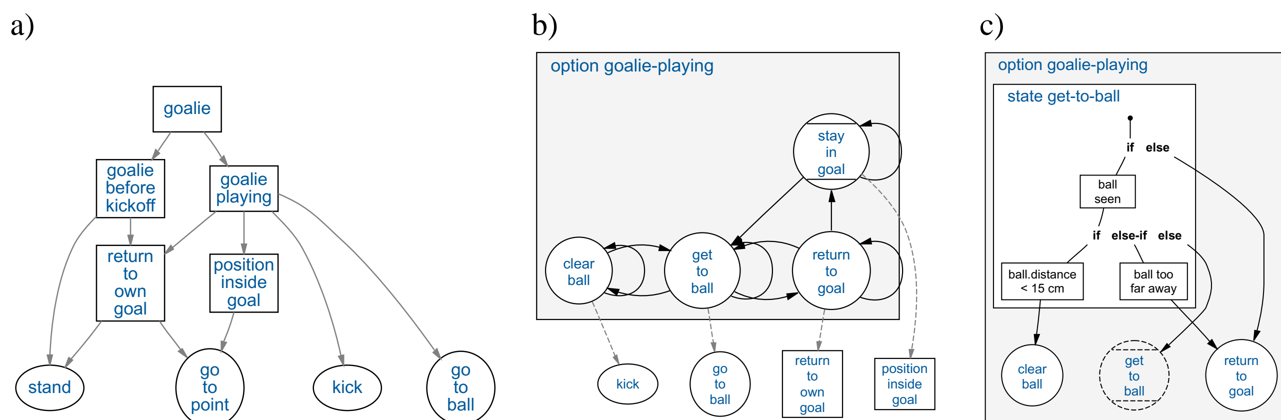

Figure 3.15: a) The option graph of a simple goalie behavior. Boxes denote options, ellipses denote basic behaviors. The edges show which other option or basic behavior can be activated from within an option. b) The internal state machine of the option *"goalie-playing"*. Circles denote states, the circle with the two horizontal lines denotes the initial state. An edge between two states indicates that there is at least one transition from one state to the other. The dashed edges show which other option or basic behavior becomes activated when the corresponding state is active. c) the decision tree of the state *"get-to-ball"*. The leaves of the tree are transitions to other states. The dashed circle denotes a transition to the own state.

3.8.1 The Extensible Agent Behavior Specification Language XABSL

The *Extensible Agent Behavior Specification Language* XABSL is an XML based behavior description language. XABSL can be used to describe behaviors of autonomous agents. The runtime system XabslEngine executes the behaviors on a target platform.

Specific behavior description languages prove to be suitable replacements to native programming language like C++ when the number and complexity of behavior patterns of an agent increases. XABSL simplifies the process of specifying complex behaviors and supports the design of both very reactive and long term oriented behaviors. XABSL uses hierarchies of behavior modules called options that contain state machines for decision making.

XABSL should be applied whenever there is a behavior control problem that is too complex to be written in C++. XABSL can be employed on any robotic platform for that a C++ compiler exists. The XML Schemas, the tools and the runtime system XabslEngine can be downloaded for free from the XABSL web site [12].

The language is documented in detail in appendix F. The validation and transformation tools are described in appendix G. The runtime system XabslEngine is explained in appendix H. The XABSL web site [12] provides XABSL examples.

The GermanTeam integrated an XABSL debug tool into the RobotControl application, which is described in section J.5.1.

3.8.1.1 The Architecture behind XABSL

In XABSL, an agent consists of a number of behavior modules called *options*. The options are ordered in a rooted directed acyclic graph, the *option graph* (cf. 3.15a). The terminal nodes of

that graph are called basic behaviors. They generate the actions of the agent and are associated with basic skills.

The task of the option graph is to activate and parameterize one of the basic behaviors, which is then executed. Beginning from the root option, each active option has to activate and parameterize another option on a lower level in the graph or a basic behavior.

Within options, the activation of behaviors on lower levels is done by state machines (cf. 3.15b). Each state has a subsequent option or a subsequent basic behavior. Note that there can be several states that have the same subsequent option or basic behavior.

Each option has an initial state. This state becomes activated when the option was not active during the last execution of the option graph. Additionally, states can be declared as *target states*. In the options above it can be queried if the subsequent option reached such a target state. This helps to check if a behavior was successful.

Additionally, each state can set special requests (*output symbols*), that influence the information processing besides the actions that are generated from the basic behaviors.

Each state has a *decision tree* (cf. 3.15c) with transitions to other states at the leaves. For the decisions the agent's world state, other sensory information and messages from other agents can be used. As timing is often important, the time how long the state is already active and the time how long the option is already active can be taken into account.

The execution of the option graph starts from the root option of the agent. For each option the state machine is carried out one times, the decision tree of the active state is executed to determine the next active state. This is continued for the subsequent option of the active state and so on until a basic behavior is reached and executed.

3.8.1.2 The XML Specification

Agents following this layered state machine architecture can be completely described in XABSL. There are language elements for options, their states, and their decision trees. Boolean logic ($||$, $\&\&$, $!$, $==$, $!=$, $<$, $<=$, $>$ and $>=$) and simple arithmetic operators ($+$, $-$, $*$, $/$ and $\%$) can be used for conditional expressions. Custom arithmetic functions (e.g. $distance - to(x, y)$) that are not part of the language can be easily defined and used in instance documents.

Symbols are defined in XABSL instance documents to formalize the interaction with the software environment. Interaction means access to input functions and variables (e. g. from the world state) and to output functions (e. g. to set requests for other parts of the information processing). For each variable or function that shall be used for conditions a symbol has to be defined. This makes the XABSL framework independent from specific software environments and platforms.

As the basic behaviors are written in C++, prototypes and parameter definitions have to be specified in an XABSL document so that states can reference them.

Below is an XABSL example source code of the option *"goalie-playing"* (cf. 3.15b).

```
<option name="goalie-playing" initial-state="stay-in-goal"
             description="goalie playing behavior">
  ...
  <state name="get-to-ball">
```

```
    <subsequent-basic-behavior ref="go-to-ball"/>
    <set-output-symbol ref="head-control-mode"
                       value="search-for-ball"/>
    <decision-tree>
      <if>
        <condition description="ball seen">
          <less-than>
            <decimal-input-symbol-ref
               ref="ball.time-since-last-seen"/>
            <decimal-value value="2000"/>
          </less-than>
        </condition>
        <if>
          <condition description="ball kickable">
            <less-than>
              <decimal-input-symbol-ref ref="ball.distance"/>
              <decimal-value value="150"/>
            </less-than>
          </condition>
          <transition-to-state ref="clear-ball"/>
        </if>
        <else-if>
          <condition description="ball too far away">
            <greater-than>
              <decimal-input-symbol-ref ref="ball.distance"/>
              <decimal-value value="900">
            </greater-than>
          </condition>
          <transition-to-state ref="return-to-goal"/>
        </else-if>
        <else>
          <transition-to-state ref="get-to-ball"/>
        </else>
      </if>
      <else>
        <transition-to-state ref="return-to-goal"/>
      </else>
    </decision-tree>
  </state>
  ...
</option>
```

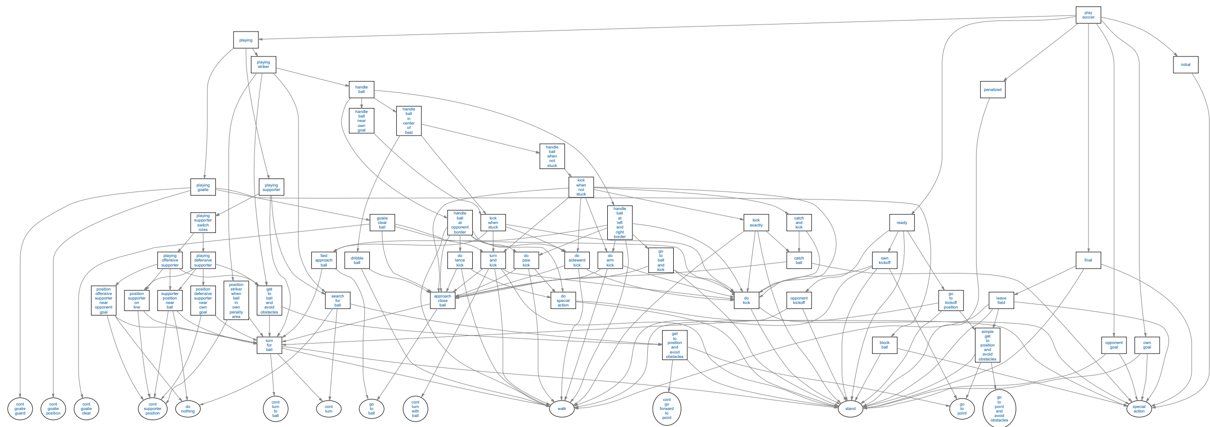

Figure 3.16: The option graph of the soccer related behaviors of the GermanTeam.

3.8.2 The Behaviors of the GermanTeam

A extensive generated HTML documentation of the GermanTeam behaviors can be found under *http://www.ki.informatik.hu-berlin.de/XABSL/examples/gt2003/index.html*.

Top Level Behaviors. The GermanTeam supports the RoboCup Game Manager to minimize human interaction during the games. The robots walk to their kickoff positions and start/stop the game autonomously. The top level behaviors are implement this.

Negotiations and Dynamic Role Assignment. The three field players negotiate, which of them is the striker, the offensive supporter, or the defensive supporter. This is done depending on the estimated time that each robot needs to approach the ball. This time is influenced by the distance to the ball, the angle to the ball, the time since the ball was seen last and the obstacle model.

Positioning. At it is disadvantageous if three field players are in the near of the ball, the robots distribute over the whole field. Only two robots are in the same half of the field at the same time. One supporter always tries to position in the near of the striker. If the striker plays near the opponent goal, the supporter positions on the other side of the goal. This helps, if the kick of the strike does not hit the goal.

Obstacle Avoidance. The GermanTeam uses the obstacle avoidance that was shown in the obstacle avoidance challenge during the whole games. It is used during positioning, ball searching and ball approaching. Only in the near of the ball (less than 70 cm) it is switched off.

Ball Handling. Although the GermanTeam also implemented a dribbling behavior, most of the time the robots try to kick the ball. The kick is chosen dependent on the obstacle model, the

position of the robot, the angle to the seen free part of the opponent goal and the position of teammates.

If there are many other robots in the near of the robot, the striker does not try to grab the ball and kick it exactly. It chooses a fast and possibly not that accurate kick. If there is time (which did not happen very often), it grabs the ball, searches for the free part of the opponent goal, rotates to the angle and kicks.

If the free part of the opponent goal is not seen, the obstacles model provides a good kick angle. The robots try to kick around obstacles. If there is a teammate in a free area, the striker tries to kick the ball there.

Ball Loss. If a robot looses a ball (does not see it anymore), the ball is mostly beside the robot. Because of that, it first heads back for about 20cm and mostly redetects the ball then. If not, the robot starts turning. If the ball is not seen by any robot of the team for a while, the robots begin to search the ball at role-dependent positions.

Communicated Ball Positions. Two ball positions are distinguished. The *"seen"* position is modeled from own observations. The *known* position is estimated from observations by other teammates. As the *known* position is mostly not that exact, it is used only for behaviors where no exact position to the ball is needed (supporter positioning, approaching far away balls). Only the *"seen"* position is used for ball handling.

Goalie. Because of the usage of field lines for localization, the goalie was mostly well localized. It does not leave the own penalty area. If the ball is inside the own penalty area, the robot tries to approach the ball as fast as possible, avoiding the edges of the own goal. The obstacles model provides an angle where to kick the ball. This angle was used to select one out of three fast kicks.

If the goalie loses the ball (doesn't see it anymore), it goes back into the center of the own goal. This helps when the ball is beside the goalie on the ground line. When the goalie reaches the center of the own goal, it will see the ball again and can clear it.

Cheering and Artistry. The Sony Four Legged League is the most interesting league for the audience, because the robots behave so cute. To make the games look more amusing some spectacular kicks and a lot of cheering behaviors were implemented. After each goal the robots rejoice (head stands, nodding, walking on back, etc.) or are annoyed (scratching head, shake head, etc). The robots leave the field autonomously after each game.

3.8.3 Continuous Basic Behaviors

Besides simple basic behaviors that execute a discrete action like performing a kick or walking in one direction, there are more complex behaviors that allow pursuing multiple objectives simultaneously while leading over from one action to another continuously. Such a basic behavior e. g. could move towards the ball, while at the same time it is avoiding to run into obstacles by moving away from them.

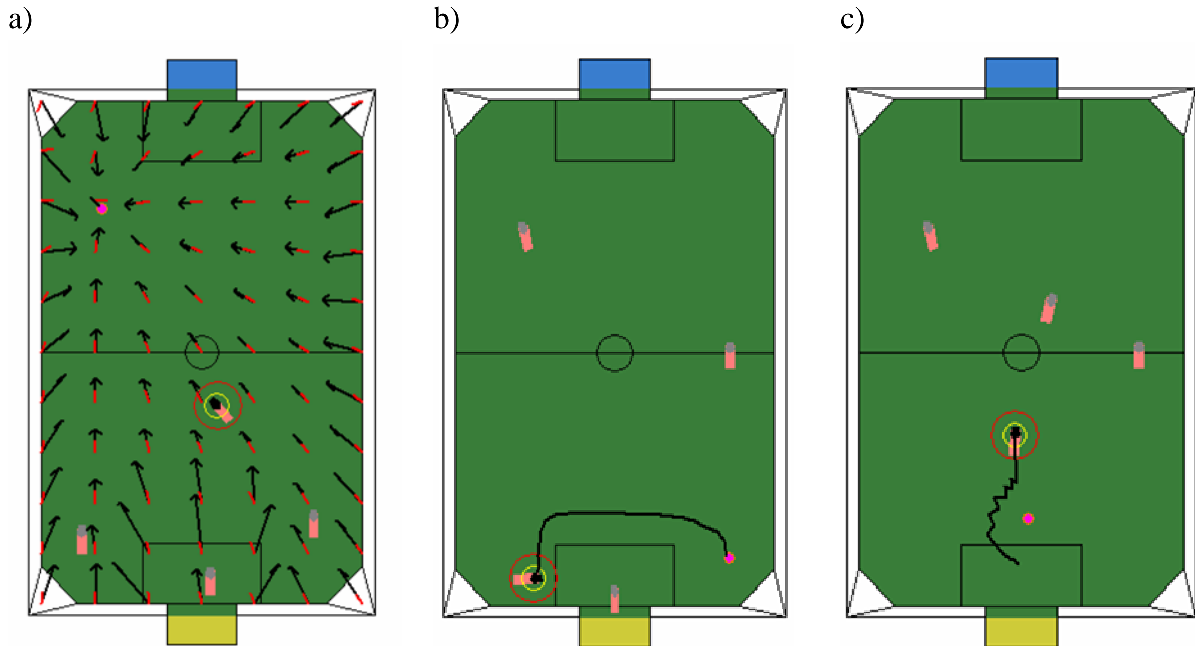

Figure 3.17: Continuous basic behaviors. a) Potential field for going to the ball while avoiding running into the own penalty area. b) Resulting walking path demonstrating the go to ball basic behavior. c) Walking path for a basic behavior going to the own goal without running over the ball.

These behaviors are called continuous basic behaviors and are implemented following a potential field approach. The potential field defining one basic behavior is configured by putting together several so-called rules, i. e. components representing single aspects of the desired behavior. Each rule is a simple potential field either attractive to a target position or repulsive from one or multiple objects. These rules may be e. g. going to the ball or avoiding running into the own penalty area.

The potential fields of all rules for one basic behavior are superposed resulting in one field which is evaluated at the position of the robot to generate the current walking direction and speed (cf. Fig. 3.17).

3.9 Motion

The module *MotionControl* generates the joint positions sent to the motors and therefore is responsible for controlling the movements of the robot.

It receives a motion request from *BehaviorControl* which is of one of four types (*walk*, *stand*, *perform special action* or *getup*). In addition, if walking is requested it contains a vector describing the speed, the direction, and the type of the walk as there are several different types of walking, such as dribbling the ball, the behavior can choose from. In case of a special action request it contains an identifier defining the requested action.

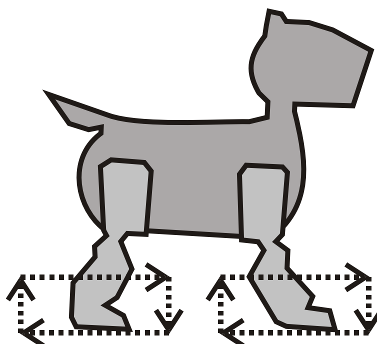

Figure 3.18: Walking by moving feet in rectangles

Furthermore *MotionControl* receives head joint values from the module *HeadControl* which is described below (cf. Sect. 3.9.3). These values are inherited by *MotionControl* but may be overridden if the current motion also requires controlling the head, e. g., for a kick with the head or dribbling the ball while holding it with the head.

Finally *MotionControl* gets current sensor data, because for some motions, sensor input is required, e. g., standing up uses acceleration sensors to detect how to stand up.

From these inputs the module produces a buffer containing joint positions and odometry data, i. e., a vector describing locomotion speed and direction, which, e. g., serves as input for self-localization.

In respect to the system's modular approach, *MotionControl* uses different modules for each of its tasks as well. There is a walking engine module for each possible walking type. Therefore each walking type can be performed by completely different walking engines as well as instances of the same engine with different sets of parameters. How the walking engine works is described below (cf. Sect. 3.9.1). The module executing special actions is described below as well (cf. Sect. 3.9.2). A getup engine module brings the robot to a standing position from everywhere as fast as possible. For standing, the walking engine for the normal walk type is executed with a speed set to zero. Thus changing from standing to walking is possible immediately as the stand position is automatically adjusted to the current walking style.

When the currently used motion module does not reflect the requested motion, the module is changed after it signals that the current motion is finished. Therefore the modules are responsible for correct transitions to other motion types, e. g., a walking engine will signal that a change to a different motion type is only possible after the current step is finished, i. e., all feet are on the ground.

3.9.1 Walking

A walking engine is a module generating joint angles in order to let the robot walk with the speed and the direction requested from behavior control. The implementation described here is called *InvKinWalkingEngine*. A main feature is that the engine and the parameters it uses are separated. The engine offers a huge set of parameters. This allows creating completely different

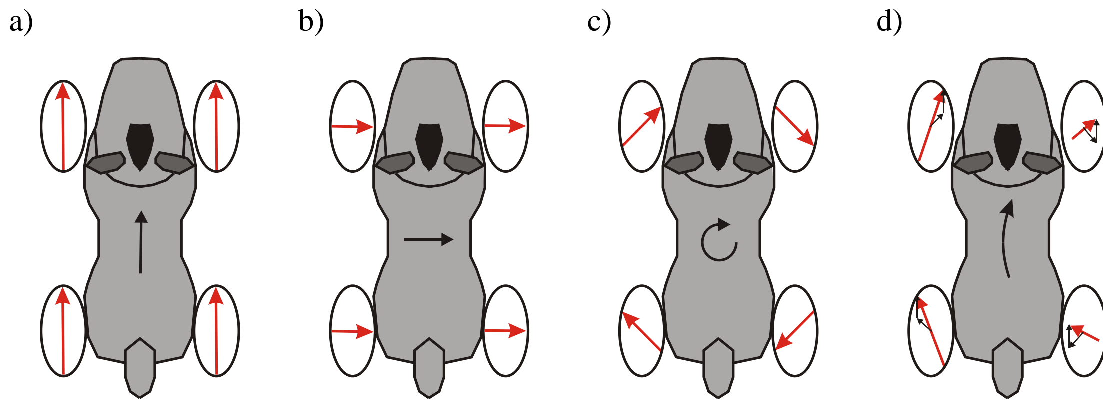

Figure 3.19: Principle of treating legs as wheels. Walking a) forwards, b) sideways. c) Turning. d) Turning while walking forward.

walks with the same engine by having different parameter values. A class containing the set of parameter values is given to the constructor of the engine. Therefore it is possible to have different instances with different parameter sets. It is even possible to transmit new parameters via the wireless network from RobotControl to test them at runtime (cf. Sect. J.7.1).

3.9.1.1 Approach

The general idea is to calculate the position of the feet relative to the body while they move in rectangles around the center position (cf. Fig. 3.18). The joint angles needed to reach the foot position are then calculated with inverse kinematics.

For the direction of walking the four legs are more or less treated as wheels. Seen from above the rectangles are rotated to the desired walking direction (cf. Fig. 3.19a, b). Turning is done by moving each leg in a different diagonal direction (cf. Fig. 3.19c). Walking and turning can also be combined resulting in curved walking. This is done by simple vector addition for each leg (cf. Fig. 3.19d).

The walking speed is defined by the size of the rectangles. The time for one step is constant but when walking faster the step length is greater.

The position and size of these rectangles and the walking gait is defined by the parameter set.

3.9.1.2 Parameters

As mentioned before the actual walking style the engine generates is mainly defined by the set of parameters applied. The parameters are the following:

footMode. This parameter selects how the feet will be moved while in the air. Besides the rectangular shape mentioned above it is also possible to have the feet move in different shapes, e. g. a semi-circle like it was the case in our walking engine for RoboCup 2001 (cf Fig. 3.20). This parameter was not used much since the rectangular shape seemed to provide best general performance. It was mainly included for increasing flexibility.

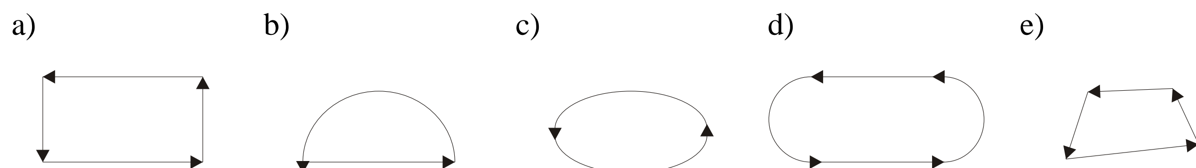

Figure 3.20: Possible modes of foot movement a) rectangle b) semi-ellipse c) ellipse d) oval e) arbitrary quadruple.

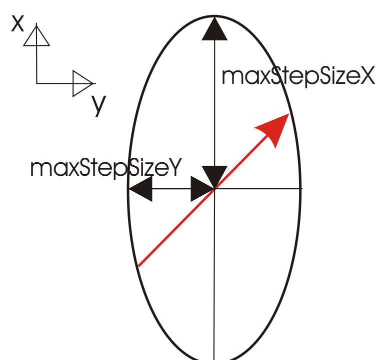

Figure 3.21: Ellipse describing possible foot target positions seen from above

foreHeight, foreWidth, foreCenterX. These values describe the center foot position of the forelegs relative to the body of the robot.

hindHeight, hindWidth, hindCenterX. The same values for the hind legs describing center foot positions.

foreFootTilt, hindFootTilt. The foot rectangles are rotated by these angles to compensate for different fore and hind walking heights.

foreFootLift, hindFootLift define the feet lifting, i. e. the height of the rectangles.

legSpeedFactorX, legSpeedFactorY, legSpeedFactorR. These values are factors between the speed of the fore and the hind legs. With these parameters it is possible to have different speeds for the fore and the hind legs. This can be useful when, e. g., the forelegs are limited to very small steps due to their position but the hind legs may still do greater steps.

maxStepSizeX, maxStepSizeY. These are the maximum step sizes that are applied when walking with full speed. They are the radii of the ellipses shown in figures 3.21 and 3.19. The rectangles are clipped to these ellipses.

maxSpeedXChange, maxSpeedYChange, maxRotationChange. By these values the acceleration of the robot is limited. A rapid change of the walking request from behavior control

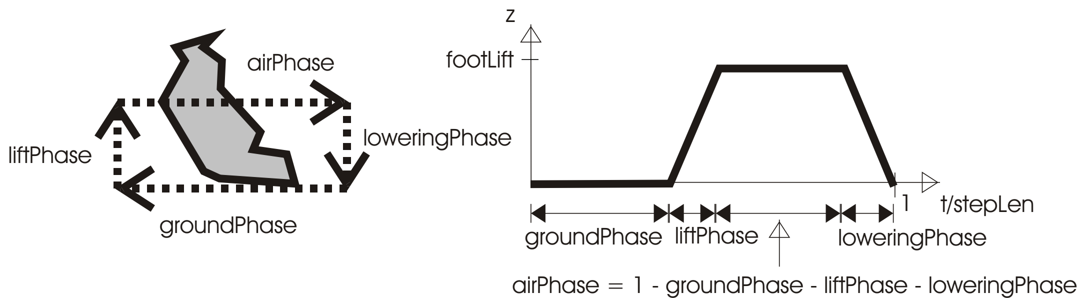

Figure 3.22: Timing of one step cycle

is applied gradually according to these limits. This prevents stumbling or even falling over when the request is changed.

counterRotation. When walking sideways, the robot sometimes tends to walk in a circle due to different contact situations or different step lengths of fore and hind legs. With this value a rotation is generated while walking sideways that compensates this unwanted effect.

stepLen. This is the time for one complete step cycle (cf. Fig. 3.22).

groundPhase, liftPhase, loweringPhase. These values define the timing of the step cycle (cf. Fig. 3.22). *groundPhase* defines how much time of the step cycle the foot will be on the ground. *liftPhase* defines how fast the foot will be lifted, *loweringPhase* how fast it will be lowered. There are two values each, one for the fore and one for the hind legs.

legPhase. These values set the relative phase offsets of each leg and therefore define the gait. For each leg there is a value which describes when, relative to the start of one step cycle, the foot is lifted.

Although the engine could employ different gaits, all currently available parameter sets use the trot gait, i. e., two diagonally opposite legs perform the same movement, while the other two legs move with a half gait phase offset. For the leg phase parameters this means the values for the left fore and the right hind leg are zero, while the values for the right fore and left hind leg are 0.5.

bodyShiftX, bodyShiftY, bodyShiftOffset. These values allow realizing a body shift away from currently lifted legs. The shift of the robot's body is performed by simply moving each foot position in the opposite direction. The values *bodyShiftX*/*bodyShiftY* define how much the body is shifted away from currently lifted feet in x/y-direction. The value *bodyShiftOffset* allows defining a time offset for the body shift so that optimally the weight is shifted away from a foot before it is lifted.

These values have no effect when using the trot gait due to the gait's symmetry, as diagonally opposite leg pairs are lifted simultaneously. Therefore these values are not used in current parameters sets but they increase the flexibility of the walking engine.

headTilt, headPan, headRoll, mouth. If these values are given they are used as angles for the head joints and the mouth. Setting these values disables the normal head motion control but can be useful, e. g. for a special walking type that holds the ball with the head.

freeFormQuadPos. These values only get evaluated when the foot mode for arbitrary quadruples is used (cf. Fig. 3.20e) and define the exact geometry of the quadruple. They contain three-dimensional coordinates for the four vertex coordinates for fore and hind legs.

3.9.1.3 Odometry correction values

Due to slippage the effective speed a walk produces differs from the calculated speed the feet have on ground and it depends quite heavily on the current underground. Therefore the maximum speed of a walk has to be measured manually. The measured speeds are stored in a file on the robot's memory stick and, they are used to correct the leg speeds so that the resulting speed of a certain gait matches the motion request.

3.9.1.4 Inverse kinematics

After the desired leg position is calculated, it is necessary to calculate the required leg joint angles to reach that position. Therefore it is necessary to determine the necessary joint angles to reach a given robot relative target position. This is called inverse kinematics problem.

In general the inverse kinematics problem is a set of non-linear equations, which can often be solved numerically only. In the given case it is possible to derive a analytical closed form solution for the inverse kinematics for one leg of the robot.

Forward kinematics solution. First a solution to the forward kinematics problem is given. This is used in solving the far more difficult inverse kinematics problem.

The forward kinematics problem is the calculation of the resulting foot position for a given set of joint angles.

The foot position relative to the shoulder joint (x, y, z) can be determined using a coordinate transformation. The origin of the local foot coordinate system is transformed into a coordinate system which origin is the shoulder joint.

In the following a simplified model of the robot's leg is applied in which this transformation is composed of the following sub-transformations:

1. clockwise rotation about the y-axis by joint angle q_1
2. counterclockwise rotation about the x-axis by joint angle q_2
3. translation along the negative z-axis by upper limb length l_1

4. clockwise rotation about the y-axis by joint angle q_3
5. translation along the negative z-axis by lower limb length l_2

In homogeneous coordinates this transformation can be described as concatenation of transformation matrices:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = Rot_y(-q_1) Rot_x(q_2) Trans \begin{pmatrix} 0 \\ 0 \\ -l_1 \end{pmatrix} Rot_y(-q_3) Trans \begin{pmatrix} 0 \\ 0 \\ -l_2 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad (3.22)$$

$Rot_{x/y}(\alpha)$ means a counterclockwise rotation around the x/y-axis of angle α and $Trans \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$ a translation of the vector (t_x, t_y, t_z).

This is equivalent to:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(q_1) & 0 & -\sin(q_1) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(q_1) & 0 & \cos(q_1) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(q_2) & -\sin(q_2) & 0 \\ 0 & \sin(q_2) & \cos(q_2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -l_1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\begin{pmatrix} \cos(q_3) & 0 & -\sin(q_3) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(q_3) & 0 & \cos(q_3) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -l_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad (3.23)$$

Matrix multiplication results in

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(q_1)\sin(q_3)l_2 + \sin(q_1)\cos(q_2)\cos(q_3)l_2 + \sin(q_1)\cos(q_2)l_1 \\ \sin(q_2)l_1 + \sin(q_2)\cos(q_3)l_2 \\ \sin(q_1)\sin(q_3)l_2 - \cos(q_1)\cos(q_2)\cos(q_3)l_2 - \cos(q_1)\cos(q_2)l_1 \\ 1 \end{pmatrix}. \quad (3.24)$$

This equation (and all of the following) is correct only for the left fore leg. But due to the symmetry of the coordinate systems of the four legs, only the signs differ in the calculation for the other legs. Thus when calculating the position of a right foot the y-coordinate has to be negated, for a hind foot the x-coordinate. Furthermore the lower limb length l_2 is slightly larger for the hind legs.

Calculation of knee joint angle q_3. To solve the inverse kinematics problem first of all the knee joint angle q_3 is calculated. As the knee joint position determines how far the leg is stretched, the angle can be calculated from the distance of the target position (x, y, z) to the shoulder joint.

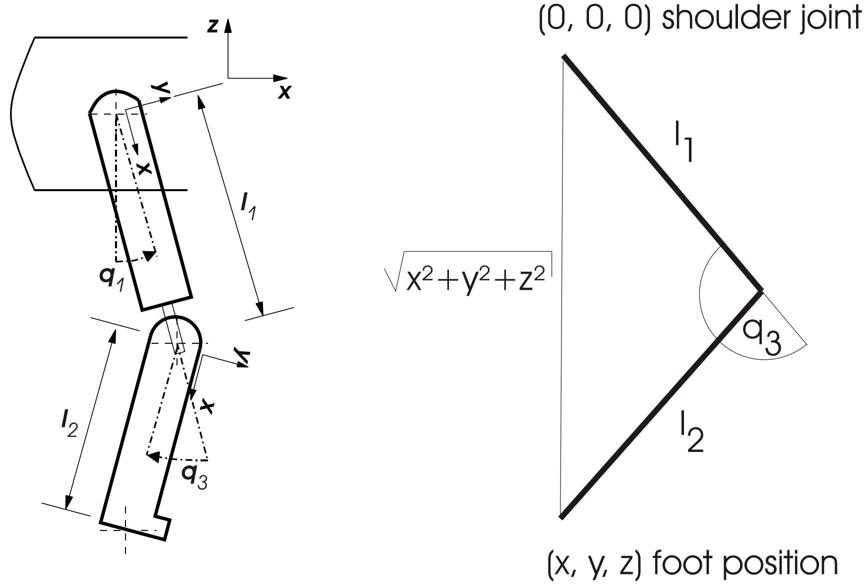

Figure 3.23: leg side view, calculation of knee joint q_3 via law of cosine

According to the law of cosine (cf. Fig. 3.23)

$$\cos\alpha = \frac{l_1^2 + l_2^2 - x^2 - y^2 - z^2}{2l_1 l_2} \tag{3.25}$$

with upper limb length l_1 and lower limb length l_2.

With

$$|q_3| = |180^\circ - \alpha| = \arccos\frac{x^2 + y^2 + z^2 - l_1^2 - l_2^2}{2l_1 l_2} \tag{3.26}$$

the absolute value of the first joint angle is calculated.

The inverse kinematics problem always has two solution, as there are two possible knee positions to reach a given target. These two solutions are selected via the sign of q_3. With respect to the joint limitations the positive value is used due to the larger freedom of movement for positive q_3.

Calculation of shoulder joint q_2. Plugging the result for q_3 into the forward kinematics solution allows determining q_2 easily. According to equation (3.24) (geometrically apparent cf. Fig. 3.24)

$$\begin{aligned} y &= \sin(q_2)l_1 + \sin(q_2)\cos(q_3)l_2 \\ &= \sin(q_2)\left[l_1 + \cos(q_3)l_2\right]. \end{aligned} \tag{3.27}$$

Consequently

$$q_2 = \arcsin\left(\frac{y}{l_2\cos(q_3) + l_1}\right). \tag{3.28}$$

Since $|q_2| < 90^\circ$ determination of q_2 via arc sine is satisfactory.

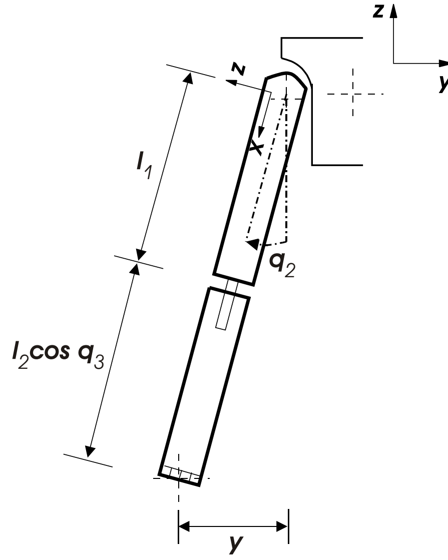

Figure 3.24: leg front view, calculation of shoulder joint q_2

Calculation of shoulder joint q_1. Finally the joint angle q_1 can be calculated. According to equation (3.24)

$$\begin{aligned} x &= \cos(q_1)\sin(q_3)l_2 + \sin(q_1)\cos(q_2)\cos(q_3)l_2 + \sin(q_1)\cos(q_2)l_1 \\ &= \cos(q_1)\sin(q_3)l_2 + \sin(q_1)\left[\cos(q_2)\cos(q_3)l_2 + \cos(q_2)l_1\right]. \end{aligned} \tag{3.29}$$

When defining

$$a := \sin(q_3)l_2, \tag{3.30}$$

$$b := -\cos(q_2)\cos(q_3)l_2 - \cos(q_2)l_1 \tag{3.31}$$

and

$$\beta := \arctan\left(\frac{a}{b}\right), \tag{3.32}$$

$$d := \sqrt{a^2+b^2} \quad \left(= \frac{b}{\sin(\beta)}\right), \tag{3.33}$$

so that

$$a = d\cos(\beta), \qquad b = d\sin(\beta), \tag{3.34}$$

equation (3.29) simplifies to

$$\begin{aligned} x &= \cos(q_1)a - \sin(q_1)b \\ &= d\left[\cos(q_1)\cos(\beta) - \sin(q_1)\sin(\beta)\right] \end{aligned} \tag{3.35}$$

which can be transformed to

$$x = d\cos(q_1 + \beta). \tag{3.36}$$

Hence

$$|q_1 + \beta| = \arccos\left(\frac{x}{d}\right). \tag{3.37}$$

The sign of $q_1 + \beta$ can be obtained by checking the z-component of equation (3.24). As in equations (3.29)-(3.36) this results in:

$$\begin{aligned} z &= \sin(q_1)\sin(q_3)l_2 - \cos(q_1)\cos(q_2)\cos(q_3)l_2 - \cos(q_1)\cos(q_2)l_1 \\ &= \sin(q_1)\sin(q_3)l_2 - \cos(q_1)\left[\cos(q_2)\cos(q_3)l_2 + \cos(q_2)l_1\right] \\ &= \sin(q_1)a + \cos(q_1)b \\ &= d\left[\sin(q_1)\cos(\beta) + \cos(q_1)sin(\beta)\right] \\ &= d\sin(q_1 + \beta). \end{aligned} \tag{3.38}$$

As $d > 0$, $q_1 + \beta$ is of the same sign as z. Hence if $z < 0$ the calculated value of $q_1 + \beta$ has to be negated.

After subtraction of β the last joint angle q_1 is computed.

3.9.2 Special Actions

Special actions are all motions of the robot that are not generated by their own algorithms but merely consist of a sequence of fixed joint positions. Currently this includes a wide variety of kicks with which it is possible to play the ball from different positions relative to the robot to various directions. The behavior is responsible for choosing the correct kick according to the position of the ball and the game situation.

The module *SpecialActions* is responsible for performing these motions. It receives the currently requested motion and produces joint angles as well as the odometry vector of the resulting movement.

The module implements a chain of nodes which is traversed every time the module is executed. These nodes either contain joint data, PID data, transitions, or jump labels.

Joint data nodes contain angles for all joints which are sent to the robot as well as timing information that state for how long these values will be sent.

Transition nodes contain a destination node and an identifier for the target special action. If the currently requested motion matches the target, the transition is followed. By this mechanism the nodes will be traversed. This ensures that the requested special action as well as the transitions from the current motion get executed. With transitions it is possible to define that another action has to be executed before the requested action, e. g. grabbing the ball before kicking.

The nodes for each special action are specified in a special description language which is compiled into C code with its own compiler which is described in section 5.4. The generated code

is part of the special action module. For each special action there is one file in the description language which contains all the necessary joint data and transition statements.

In addition, there is one special file called *extern* which serves as entry point to the module. It contains transitions to all special actions of which the correct one will be executed when the module is entered from other motion types. *extern* also serves as special transition target for leaving the special action module. If another motion type is requested, the special action module continues until a transition to *extern* is reached. By this the current special action will always be finished, avoiding, e. g., starting to walk while standing on the head.

The odometry data is calculated from the current movement and rotation speed that are taken from a table containing values for all special actions. This table can contain information about the result of completely executing a special action once, e. g. that the bicycle kick turns the robot by 180 degrees. In addition the table may contain entries giving a constant speed for a special action.

3.9.3 Head Motion Control

The module *HeadControl* determines where the robot is looking at. It receives *head control modes* from the behavior control and generates the required *head motion requests* which contain the angles of the three head joints and the mouth. These requests are sent to the module *MotionControl* that forwards them directly to the motors (cf. Sect. 3.9). Furthermore, *HeadControl* receives sensor data and the internal world model.

3.9.3.1 Head Control Modes

Since the robot can only see a small portion of its environment, it is necessary to have its head (and thus its camera) point in certain directions depending on the situation the robot is facing. A number of such situations have been identified and suitable head motions where developed, called *head control modes*. Some modes are more elemental, such as the *look-at-point* mode, whereas others utilize the basic modes (or the concepts of these modes) to achieve more complex solutions such as tracking the ball. The most interesting modes are:

Look at point. If this mode is selected, the robot will look at a certain position in 3-D space specified in its own system of coordinates. It can even look with a certain camera pixel to that position. This is quite useful to see as much as possible, e. g. by keeping the center of the ball in the lower left corner of the image you may see opponents or landmarks in the upper right corner. That feature makes calculation harder, but it is still manageable and well explained in the source code. The main idea is to look at the ray into space produced by looking with a certain camera pixel. The differences between the arcs to the destination point and the arcs to the ray are the tilt and pan angles that are looked for.

Search for Landmarks. In this mode the robot searches for landmarks, i. e. flags and goals. The head moves in a manually tuned polygon scanning the area in front of the robot. No knowledge of the world is used to guide the scanning motion.

Search for Ball. This mode looks at the ball using *Look at point* if the robot has seen it. Otherwise a manually tuned constant scan for the ball is started. In this year's approach no ball speed and no communicated ball positions are used, because they were not precise enough to base a smooth head control on.

Search auto. This mode was developed to allow the robot to track the ball successfully while maintaining its bearing on the field. To achieve this, a robust ball tracking had to be established that allows the robot to scan for landmarks occasionally. In contrast to last year's approach no world model information is used to predict where landmarks should be. Instead of only looking to the position a landmark is already believed to be it is tried to see as much as possible including landmarks, obstacles, and other robots. The robot looks at the ball until it has seen it consecutively for more then 300 ms. Then it scans for landmarks and other objects on one side of the ball. After looking at the ball again it will scan the other side. No scan will take more than 880 ms. If the ball was not seen, the robot scans for it, because this head control mode is not useful without seeing the ball. Only ball positions resulting from the robot's own perception were used to look at the ball. Speed propagations and balls seen by other robots were not exact enough to base a head control mode on. The behavior has to follow such information to bring the robot into a position were it can see the ball on its own. All together this performed much better than the approach used last year.

3.9.3.2 HeadControl State Machine

GT2003HeadControl uses an internal state machine to handle switching between different *head control modes* while maintaining internal states such as to which side of the ball the last scan for landmarks was executed or whether the ball had been seen when it was tried to look at it the last time. This made the HeadControl produce much smoother results than in the years before, even if the requested head control modes changed quite rapidly.

3.9.3.3 Head Path Planner

The head path planner was used to plan a number of way points for the head in advance. This is useful to describe a scan motion with several points and only a single overall duration. This works from every starting position of the head because the distance to the first way point is considered during the computation of the constant head speed to reach the last way point in the time given. So it was easy to fine-tune the speed of the head in certain situations by just a single parameter.

3.9.3.4 Joint Protection

The destination head joint angles, no matter whether they came from a debug request, a special action, a special walking, or a head control, will be restricted to a maximum change speed by MotionControl to avoid damaging the head joints. This way a head control does not need to care about maximum head speed, it only has to be aware of MotionControl limiting the requested

speed, e. g. it is very easy to implement a head control mode like that: look as far left as you can in 150 ms and then look back to the ball.

Chapter 4

Challenges

In 2003, the technical challenge in the Sony Legged Robot League comprised the three separate tasks “scoring a goal with a black and white ball”, “localization without the colored beacons”, and “obstacle avoidance”. The GermanTeam won this challenge. A specialty about the solutions found by the GermanTeam for the different tasks is that two of them were not only used in the challenge, but also in actual RoboCup games, namely the obstacle avoidance and the edge-based localization.

4.1 Black And White Ball

The ball challenge can be divided into two parts: the detection of the ball and a special behavior for handling the ball.

4.1.1 Detection of the ball

Different algorithms to detect a black and white ball have been evaluated. An edge detection filter followed by a Hough transform has proved to be suitable.

First of all, all vertical edges are detected by a Sobel filter. If the edge is large enough, it is tested, whether it could be an edge of the ball. Therefore, the number of green, white and black pixels left of, right of, above and below the edge pixel are counted. If the count exceeds a pre-defined threshold value, the pixel presumably belongs to a ball edge.

To detect the ball, a Hough transform is performed for all pixels on the edge of the ball. If multiple circles, each with the same radius, are constructed in a way such that their centers lie on another circle, they intersect in the center of this circle. In this case, the resulting circle is equivalent to the ball.

The image is first divided into cells of 5 x 5 pixels each. Then, for each pixel on the edge of the ball, circles with the same radius are drawn to detect the cell with the most intersections with circles. This cell could be the center of the ball.

Figure 4.1: Debug-Image. Strong edges are shown in blue. Pixels on the edge of the ball are colored green. The red circle is the result of the Hough transform

This procedure should be performed for all possible radii to find the radius with most intersections for a cell. This radius and the center of the cell are assumed to be the radius and center of the ball.

Because of limited computing time, not all possible, but only eleven different radii are tested. Thus, the calculation of the radius and the distance become less precise.

Then it is checked whether the circle really covers the ball. Therefore, 80 pixels from the inside of the circle are chosen at random. At least 50% of them have to be black or white, and at least 10% of them have to be black.

4.1.2 Behavior for the Ball Challenge

As all behaviors of the GermanTeam, the behavior control for this challenge was developed based on XABSL (Appendix F).

Because normal kicks were ineffective, there were two approaches bringing the ball into the goal. The first was to grab the ball and to walk into the goal while holding the ball between the forelegs. The results of this solution were not as good as expected, so the second solution was used. In this solution the ball was pushed into the target goal.

The behavior for the challenge works in the following way:

1. *Search for the ball.* If the ball was not seen, the robot walks towards some specific areas on the field, which are chosen in a way that the ball should be seen within a distance of less than 80cm.

2. *Move in position.* If the ball was found, the robot tries to get behind the ball. If the deviation between the straight line from the ball to the goal and the straight line from the ball to the robot is less than five degrees the robot continues with step 3.

3. *Move ball towards goal.* If the robot is in position, it moves forward with stretched out front legs and dribbles the ball towards the goal. If the robot is not in the right position behind the ball, the behavior returns to step 1 or 2.

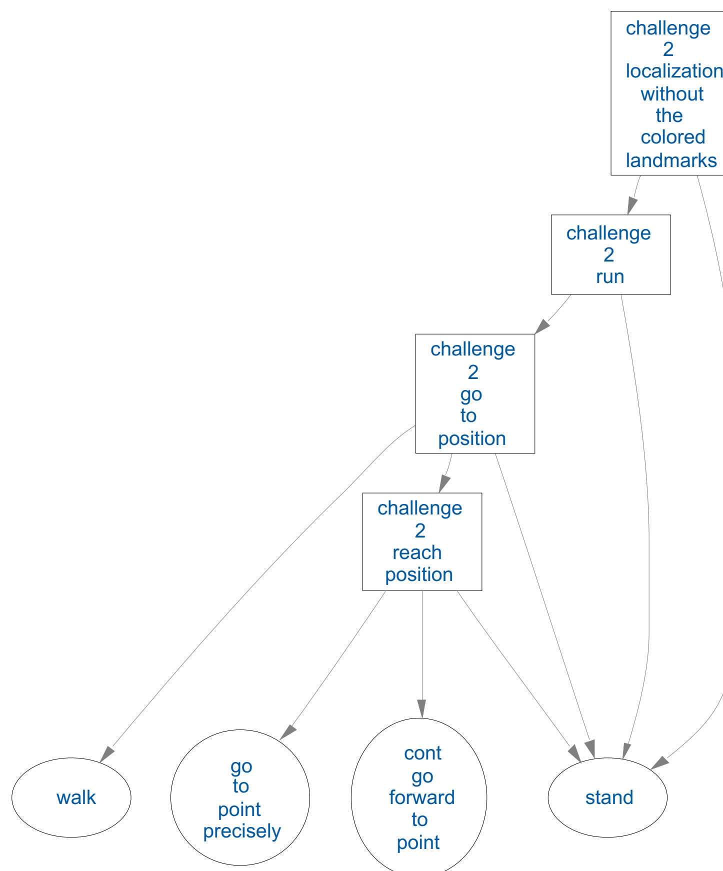

Figure 4.2: Option tree used for challenge 2.

4.1.3 Results

The ball was detected very well and the robot reached it first compared to the other teams. The robot dribbled the ball towards the goal, but the uneven lawn caused the ball to roll away towards the edge of the field. In the remaining time, the robot was unable to re-detect the ball.

4.2 Localization

The goal of the localization challenge was to reach five target positions on the field without the help of the colored beacons. The GermanTeam used the lines self-locator for this challenge that was already presented in section 3.3.3. The only two things left to explain are the behavior control used in this challenge, and the control of the robot's head.

4.2.1 Behavior Control

The behavior was modeled in XABSL2 and consisted of four options (cf. Fig. 4.2):

1. The first option just handles the back switch of the robot, i. e. if the button is pressed, the second option is executed, and if the back switch is pressed for more than one second, the behavior returns to its initial state.

2. The second option *run* cycles through the five target positions and forwards one after the other to the third option. The five positions are realized as three XABSL-symbols (for x, y and rotation) that can be parameterized by the index ($0 \ldots 4$) of the position. As a side effect of asking for the first position, a traveling salesman algorithm plans the shortest path through all five positions starting with the actual position of the robot, i. e. it is assumed that the robot is localized at this point in time. After testing some different approaches, it was decided that the rotation of the robot at the target positions is selected in a way that the robot always looks at the border of the field.

3. The third option *go-to-position* decides whether the robot is localized or not, and if it is not after a few seconds, the robot turns by $180°$ to re-localize. If it is localized, the final option is executed that actually moves the robot to a target position. If the robot has reached that position, it will wag its tail.

4. The fourth and final option *reach-position* moves the robot to the target position. It starts with a basic behavior that walks to a certain position while avoiding obstacles, just in case the goal posts will be on the planned path. If the robot is closer than 20 cm to the target position, it continues to walk without avoiding obstacles and it will also turn to the pre-computed direction. If the robot has reached the target posture with a deviation of less than 5 cm and less than $10°$, the option enters another state that terminates if the robot comes closer than 2 cm to the target location or if more than 15 seconds have passed. The latter condition ensures that the robot does not wait forever for reaching a target position.

4.2.2 Head Control

The experiments in section 3.3.3 had shown that the precision of the lines self-locator depends on the target location to reach, in fact, it depends on the distance to the closest lines in two orthogonal directions. Therefore, high precision was required to have a chance to reach the target positions sufficiently precise. In experiments it turned out that distance measurements were more precise when the tilt joint of the head remains in a constant position. Therefore, to move the robot's head a simple left to right and back scanning method was preferred over a more intelligent field line and border scanning approach (that was used by the Bremen Byters at the German Open 2003). However, while this decision was well suited for the test positions, it was very disadvantageous for the actual challenge, because two target positions were so close to the border that the robot was not able to see it anymore, and a third position was so close at the center circle that the robot had to use the border and the penalty line to localize, which were quite far away, resulting in poor precision.

4.2.3 Results

While the approach was always able to reach all five test positions, scoring at least 13 points, it nearly failed in the actual challenge. It only reached one position precisely and two others with a deviation of 7.5 cm. The remaining two positions were reached with larger errors, although the robot never lost track of its location. The final score was only five points. However, only one team scored more. The team from Washington University also scored five points, and because they scored them at the first two positions, they were faster, and the GermanTeam finally reached the third place in this challenge.

4.3 Obstacle Avoidance

The obstacle avoidance approach described here finished first in the challenge. The robot finished the course in 36 seconds, approximately twice as fast as the runner up. The robot had to move from one goal to the other with 7 other robots standing on the field in typical game-like positions.

For this challenge, a *local* obstacle avoidance method was used. This is in contrast to some approaches that used global path planning and even search algorithms. Obstacle avoidance was based on the obstacle model described in 3.5.

The fundamental idea was to have the robot always face the direction it is moving in to make sure it does not run into obstacles it is unable to see. (Therefore, the robot would never walk backwards or sideways.)

To move towards the destination, two states where used:

- *Move freely.* If no obstacles were detected in the direction of the destination (or rather if the free length of path was greater than a threshold), the robot would turn in the direction of the destination (until the robot is facing in that direction) and walk forward at the same time.

- *Avoid obstacle.* If an obstacle was detected in the robot's way, the robot would turn away from it while still walking forward. The forward speed is determined by the distance to the closest obstacle in the direction the robot is walking in. This ensures that the robot will never run into an object as long as it is able to see it.

In all cases, it was checked against the obstacle model whether the space the robot was moving to was empty. This is particularly important when the robot is passing an obstacle. At some point, it is not able to see it anymore. In test runs, the robot would often try to avoid an obstacle and then, after passing it, run into it while trying to turn to the destination.

The robot's head performs a left/right-scanning motion. This, on the one hand, has the advantage that it is possible to create a detailed obstacles model (or map) of the robot's local surroundings with enough information for the robot to find a good path. On the other hand it caused the robot to come into close vicinity of obstacles when it was facing the other way. For the robot, these obstacles would "pop up" from out of nowhere. Both the robots forward speed and the angle range of the scanning motion had to be carefully adjusted to keep the robot safe from obstacles.

A local obstacle model was used (rather than a global one) for a number of reasons: the model is independent of localization, it is easily applicable in non-RoboCup environments, and this approach is well suited for dynamic environments (such as actual RoboCup games, although the challenge's setup itself was static).

A video of the run is available for download at *http://www.aiboteamhumboldt.com*.

Chapter 5

Tools

The GermanTeam spent a lot of time on programming the tools that do not run on the AIBO platform but that helped very much in the development of the soccer software.

In section 5.1 and 5.2 two very similar programs are described: SimGT2003 and RobotControl. They both have in common:

- The complete source code that was developed for the robot is also compiled and linked into these applications. That allows algorithms to be tested and debugged very easily. New source code can be tested with the tools before compiling it for the robot and testing it on the field.
- As the interfaces of the source code to the physical robot are very narrow, the robot could be easily replaced by a simulator.
- They provide a lot of debugging and visualization tools.

Section 5.3 describes a router software for the wireless network that dispatches messages between the robot and SimGT2003/RobotControl.

The *Motion Net Code Generator* (cf. Sect. 5.4) was used by the GermanTeam for generating C code from motion description files containing, e. g., the kicks as well as for generating xml files containing a list of all motions that can be requested by the behavior.

The *Emon Log Parser* (cf. Sect. 5.5) was used to get as much information as possible out of the log files produced by the Open-R SDK Emergency Monitor.

5.1 SimGT2003

SimRobot is a kinematic robotics simulator that was developed at the Universität Bremen [14]. It is written in C++ and is distributed as public domain [1]. It consists of a portable simulation kernel and platform specific graphical user interfaces. Implementations exist for the *X Window System*, *Microsoft Windows 3.1/95/98/ME/NT/2000/XP*, and *IBM OS/2*. Currently, only the development for the 32 bit versions of Microsoft Windows is continued.

SimRobot consists of three parts: the *simulation kernel*, the *graphical user interface*, and a *controller* that is provided by the user. Already in 2002, the GermanTeam has implemented the whole simulation of up to eight robots including the inter-process communication described in appendix C as such a controller, providing the same environment to robot control programs as they will find on the real robots. In addition, an object called *the oracle* provides information to the robot control programs that is not available on the real robots, i. e. the robots' own location on the field, the poses of the teammates and the opponents, and the position of the ball. On the one hand, this allows implementing functionality that relies on such information before the corresponding modules that determine it are completely implemented. On the other hand, it can be used by the implementors of such modules to compare their results with the correct ones.

SimRobot, linked with the special controller that provides the interface to the robots and linked with the robot code is called *SimGT2003*. The following sections will give a brief overview of SimRobot, and how it is used to simulate a team of robots.

5.1.1 Simulation Kernel

The kernel of SimRobot models the environment, simulates sensor readings, and executes commands given by the controller. A simulation scene is described textually as a hierarchy of objects. Objects are bodies, emitters, sensors, and actuators. Some objects can contain other objects, e. g. the base joint of a robot arm contains the objects that make up the arm.

Emitters. SimRobot uses a very abstract model of measurable quantities. Instead of defining objects as lamps or color cameras that emit and measure light, it uses objects that emit intensities of particular *radiation classes* (emitters) and objects that measure these intensities (sensors). Hence, it is up to the user to define some of these abstract classes to represent real phenomena. In case of the Sony AIBO robots, the radiation classes 0, 1, and 2 represent the three channels of the YUV color model.

There are only two types of emitters in SimRobot: *radial emitters* send their radiation to all directions, whereas *spot emitters* have a certain opening cone. In addition, an ambient intensity can be specified for each radiation class that defines the base intensity for all surfaces in a simulation scene. Hence, the surfaces that are not reached by any of the emitters in a scene still have a sensible radiation signature. In the RoboCup simulation, only a high degree of ambient white light is used and no emitters.

Bodies. Currently, bodies can only be modeled as a collection of polygons. Each polygon has a radiation vector that defines its appearance—together with the radiation of the emitters that reaches the surface. The GermanTeam uses color tables to map the colors measured by the robot's camera onto *color classes*. To avoid having two different color tables, one for the real robot and one for the simulation, the simulation scene is automatically colored according to the actual color table. This is also the reason why no additional emitters are employed for illumination. Their influence may have changed the colors of the surfaces, resulting in a wrong mapping from colors to color classes in the image processing modules of the simulated robots.

Actuators allow the user or the *controller* to actively influence the simulation. They can be used, e. g., to move a robot or to open doors. Each actuator can contain other objects, i. e. the objects that it moves. SimRobot provides four types of actuators: rotational joints, translational joints, objects moving in space in six degrees of freedom, and vehicles with typical car kinematics, i. e. with a driving axle and a steering axle. SimRobot is only a kinematic simulator; thus it cannot directly simulate walking machines. Therefore, the motion of the simulated AIBOs is generated by a trick: the GT2003 robot control program has its own model of which kind of walk will generate a certain motion of the robot. This model is also employed for the simulation. Thus, the simulated robots will always behave as expected by their control programs—in contrast to the real robots, of course. In addition, the body tilt is simulated. This is performed on the assumption that the body roll is always zero. In each simulation step, the distance of the four feet to the ground is determined. Then the robot body is moved and rotated around the tilt axis in a way that at least one foreleg and one hind leg touches the ground. This approach only fails if the feet are not the lowest parts of the robot's body, e. g. when it performs the "getup" action.

Sensors. SimRobot provides a wide variety of sensors. However, only three types of information can be sensed:

Intensities of Radiation. There are two types of cameras that allow measuring two-dimensional arrays of intensities of radiation. The *camera* object imitates normal pinhole cameras, and is used to simulate AIBO's color camera. The *facette* simulates cameras with a spherical geometry, i. e. the angle between all adjacent pixels is constant. The sensor readings can be calculated using *flat shading*, i. e. each surface has a single combination of intensities, or with different intensity signatures for each pixel. In addition, it is possible to determine shadows.

Distances. There are several sensors that measure distances. A *whisker* can imitate the behavior of an infrared sensor. On the one hand, it is used to simulate the PSD sensor in AIBO's head. On the other hand, whiskers could be employed to implement the ground contact sensors in the feet of the robots. As these sensors are not used by the GermanTeam, this has not been implemented yet.

Collision detection. For every actuator, it can be detected whether a collision-free execution of the last command was possible. This information is not available in reality, but it is required for the simulator to suppress motions that result in collisions. However, as each robot has 20 degrees of freedom, this costly calculation is even too slow for a single robot, but it surely is for eight. Therefore, the current simulation does without collision detection.

Apart from the latter, all sensor readings can be disturbed by a selectable amount of white noise.

5.1.2 User Interface

The user interface of SimRobot includes an editor for writing the required scene definition files (cf. Fig. 5.1, upper left window). If such a file has been written and has been compiled error-free,

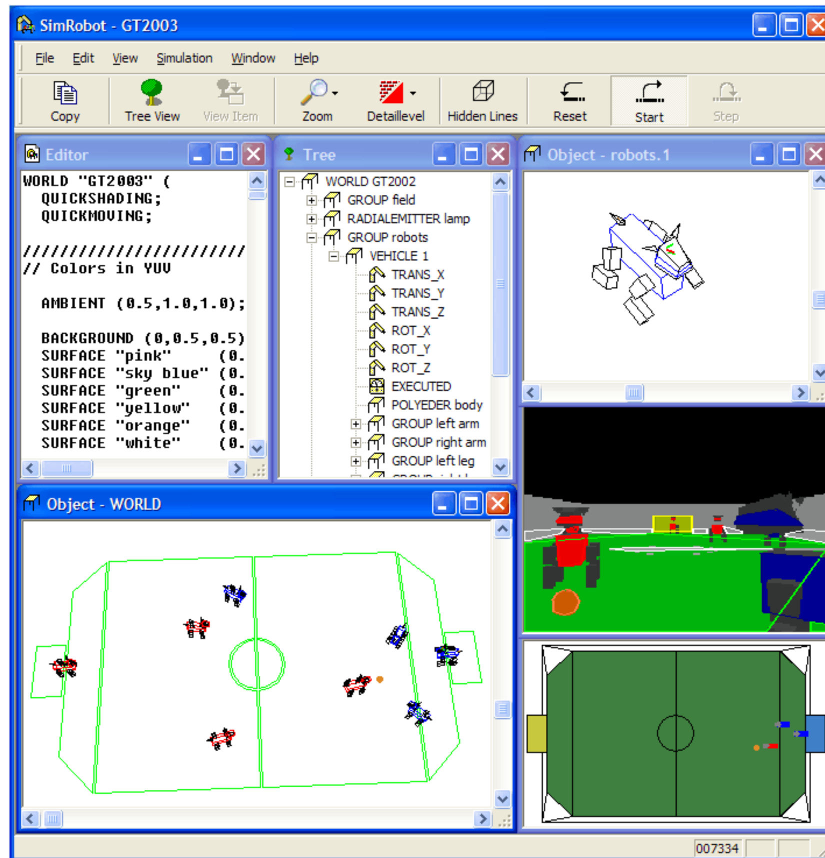

Figure 5.1: SimRobot simulating the GermanTeam 2003.

the scene can be displayed as a tree of objects (cf. Fig. 5.1, upper middle window). This tree is the starting point for opening further views. SimRobot can visualize any object and the readings of any sensor that are defined in a scene. Objects are displayed as wire-frames with or without hidden line removal (cf. Fig. 5.1, lower left and upper right window) and parts of the scene can be hidden. In case of the lower left window, the flags, the goals, and the field border are not displayed.

Sensor data can be depicted as line graphs, column graphs, monochrome images, and color images (cf. Fig. 5.1, middle right window). In addition, depth images can be visualized as single image random dots stereograms. Any of these views and a numerical representation of the sensory data can be copied to the system's clipboard for further processing, e. g., in a spreadsheet application or a word processor. The whole window layout is stored when a scene is closed and restored when SimRobot is started again with the same scene.

SimRobot also has a console window that can be used to enter text and to print some data on the screen. SimGT2003 uses this window to print text messages sent by the robot processes, and it allows the user to enter a large variety of commands. These are documented in appendix I.

5.1.3 Controller

The controller implements the sense-think-act cycle; it reads the available sensors, plans the next action, and sets the actuators to the desired states. Then, SimRobot performs a simulation step and calls the controller again. Controllers are C++ classes derived from a predefined class *CONTROLLER*. Only a single function must be defined in such a controller class that is called before each simulation step. In addition, the controller can recognize keyboard and mouse events. Thereby, the simulation supports to move around the robots and the ball.

A very powerful function is the ability to insert *views* into the scene. These are similar to sensors but in contrast to them, their value is not determined by the simulation but instead by the controller. This allows the controller to visualize, e. g., intermediate data. In fact, the middle right window in figure 5.1 is a view that contains a camera image overlaid by the so-called blob collection, i. e., colored octagonal areas detected by the robot control program's image processor. The lower right window is completely drawn by the controller: a field with the visualization of the estimations of a robot's own pose (in this case the right goalie), the locations of some other robots, and the position of the ball.

The whole environment that the processes of a robot control program will find on a real robot has been resembled as such a controller. It supports multiple robots, each robot can run multiple processes, these processes can communicate with each other, and also the communication between different robots is supported. Thus the code of a whole team of four communicating robots runs in the simulator.

5.2 RobotControl

In contrast to SimGT2003 that evolved from a pure simulator, RobotControl (cf. Fig. 5.2) was initially intended to be a general support tool that should help to increase the speed and comfort of the software development process.

First, it functions as a debugging interface to the robot. Via the wireless network or a memory stick, messages can be exchanged with the robot. Almost all internal representations of the robot (images, body sensor data, percepts, world states, sent joint data) and even internal states of modules can be visualized.

In the other direction, many intermediate representations of the robot can be set from RobotControl. For instance, one can send motion requests that are normally set by the behavior control module of the robot to test the motion modules separately.

Second, as in SimGT2003, the complete source code for the robots is compiled into RobotControl and encapsulated in "simulated robots". The debugging interfaces of RobotControl function both for the simulated and the physical robots. So it is possible to test source code without switching to a robot. The virtual robots can receive their data from a simulator (which was adapted from SimGT2003), a real robot, or a log file. The GermanTeam could develop its vision modules long before they had a wireless network connection to the robot by testing the algorithms on log files.

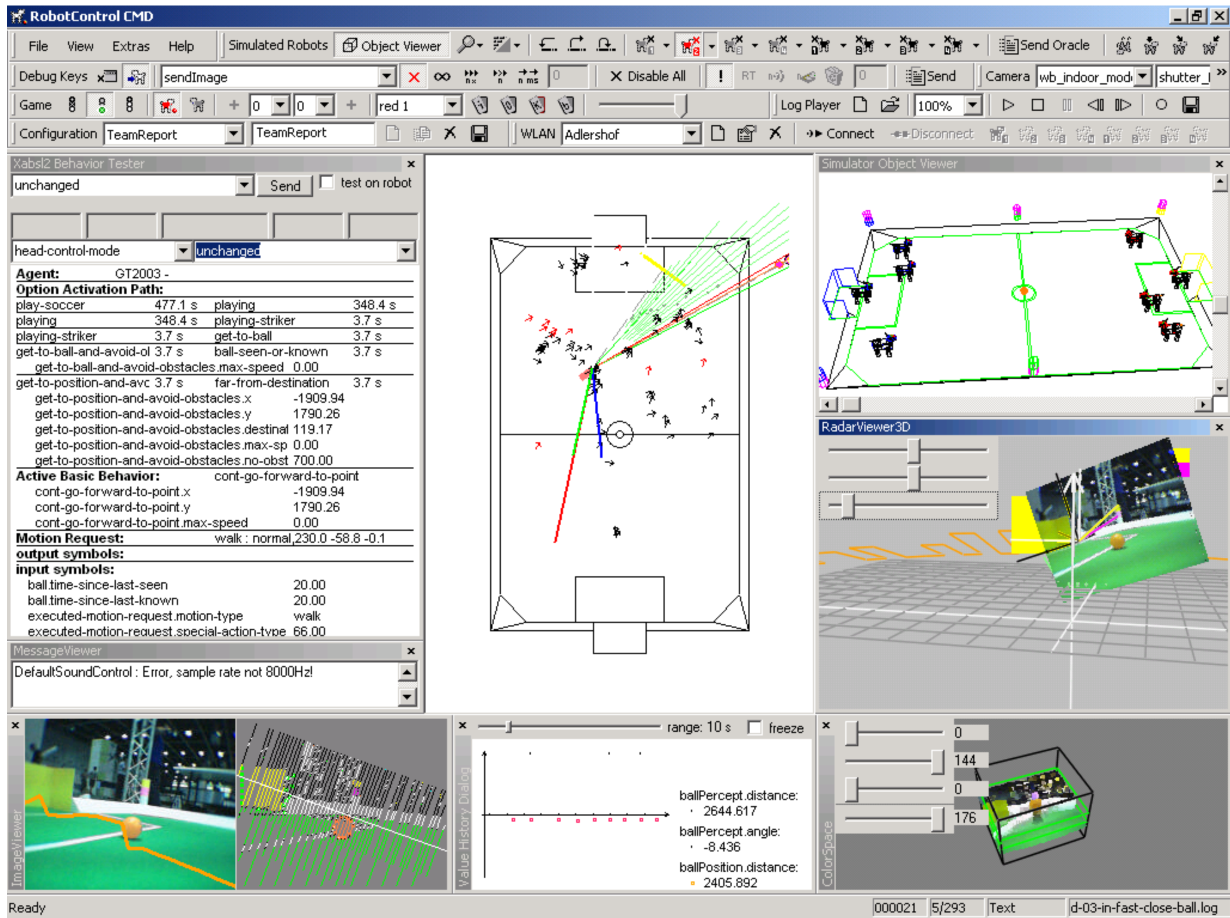

Figure 5.2: The RobotControl application

In addition, a variety of other helper tools is integrated into the application, e. g. for color calibration or for copying data to the memory sticks.

Almost all of RobotControl's functionality was programmed into toolbars and dialogs. There are simple interfaces to create and embed them in the application, so that many team members could easily program graphical user interfaces for their debugging needs.

This is also one of the two main differences between RobotControl and SimGT2003: In SimGT2003 most of the interaction with the program is done using a text console whereas in RobotControl many graphical user interfaces exist. As many tasks require a graphical user interface, e. g. creating color tables, SimGT2003 provides only a small portion of the functionality of RobotControl.

The second difference is that RobotControl can only communicate with exactly one simulated and one physical robot at the same time. Although it can simulate up to 8 robots as well as keep network connections to up to 8 real robots simultaneously, only one of each kind can be "connected" to the application. Only game manager commands can be sent to all of them simultaneously.

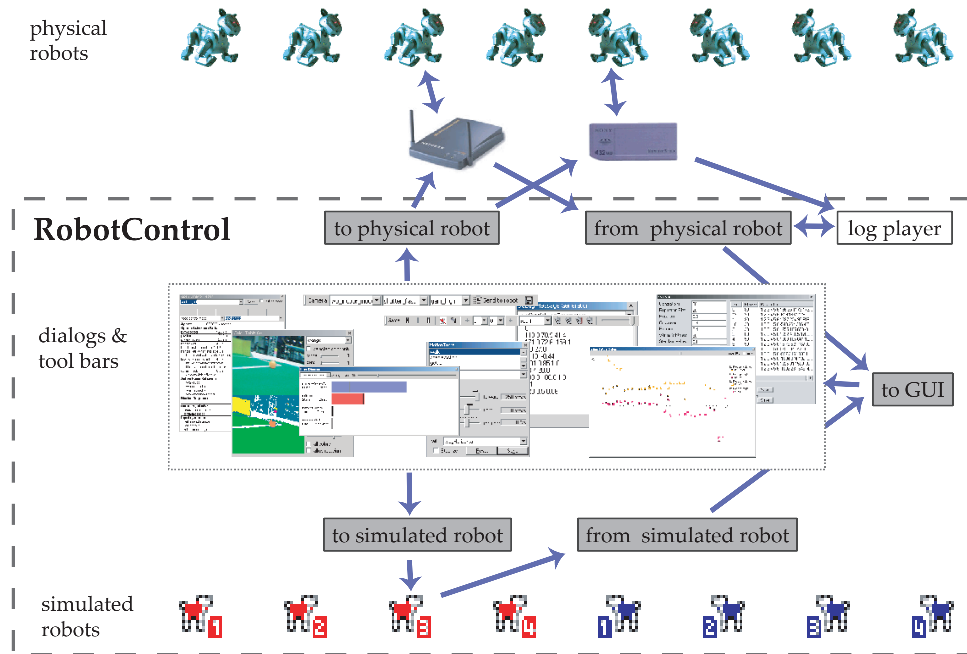

Figure 5.3: Data flow in the RobotControl application. The gray boxes denote the main message queues.

RobotControl has a very modular structure (cf. Fig. 5.3): it almost only consists of the simulated robots, a simulator, the tool bars, the dialogs, an interface to the wireless network, and message queues (cf. Sect. E.1) between these units. Dialogs and toolbars only communicate via the queues.

To send data to the robot or to the simulated robot, messages are put into the *queue to physical robot*, *queue to all physical robots*, *queue to simulated robot* or *queue to all simulated robots*. The *queue to physical robot* can be sent to the robot via the wireless network or by writing the queue onto a memory stick. The *queue to simulated robot* is sent to the simulated robot that is currently connected to the application. One can change, which of the 8 robots shall be connected. Messages from the robot can be received over the wireless network or from a log file that was written by the robot onto the memory stick. They arrive in the *queue from physical robot*. From there, some of the messages are first sent to the simulated robot *queue to simulated robot*, some directly to the *queue to GUI*. All messages from the currently connected simulated robot arrive in the *queue from simulated robot* and later in the *queue to gui*. At last, the messages in the *queue to GUI* are distributed among the dialogs and toolbars. The *log player* records messages from the *queue from physical robot* and stores them there again when playing a log file.

5.3 Router

It is desirable to exchange data between the tools running on the PC and the robots. Last year, a wireless network was introduced into the Sony Legged Robot League. On the side of the PC, only Linux and Cygwin are supported platforms for the wireless communication. However, the

GermanTeam uses tools running natively under Microsoft Windows. Therefore, they have no direct access to the new wireless communication capabilities of the robots.

That is the point where the *Router* comes into play. It functions as a mediator between the physical robots and the native Windows tools. On the one hand, it communicates with up to eight robots using the *tcpGateway*. On the other hand, it exchanges data with the Windows tools via a different IP-port for each robot. The data transferred are *message queues*, one from the PC to each robot, and a second back from each robot to the PC.

That way, RobotControl can send a message queue to the robot by sending it to the appropriate IP-port (by convention the least significant byte of the robot's IP-address plus 15000). The Router will receive the queue and forward it to the *tcpGateway* on the PC. Then the gateway will send the queue to the *tcpGateway* on the robot via the wireless network. The latter will again forward the message queue to the *Debug* process on the robot. The other way round, the robot can send a message queue to RobotControl.

start.bash. The router needs a couple of configuration files to work properly, namely *connect.cfg*, *object.cfg*, and *port.cfg*. These files are small and quiet easy to edit manually as long as one only wants to communicate with one robot using the *tcpGateway*. But they are hard to read and maintain for multiple robots, e. g. for two robot teams of four players each, complete point-to-point connections between all robots of the same team, *roboCupGameManager* control for these eight robots, and debug connections to the PC.

Because these configuration files have to be changed to adapt to different conditions (smaller number of robots, only one team, no *roboCupGameManager*), the script *GT2003\Bin\start.bash* automates that work. A message such as the following is generated by the script after its first execution, i. e. when *connect.cfg*, *object.cfg*, and *port.cfg* do not exist. It explains the usage of the script:

```
usage: start.bash [-gm] [-release] [-cmu] [ IP | ( subnet
  [ auto | ( A1 [A2 [A3 [A4 [B1 [B2 [B3 [B4]]]]]]] ) ] ) ]
  where -gm starts the RoboCupGameController
    -release only uses connections for release sticks
     (no MessageQueues for RobotControl)
    -cmu starts the last team in CMPack'02 mode
    subnet is a subnet equal for all robots followed by .Ai or .Bi
    Ai are own robot IP addresses
    Bi are opponent robot IP addresses
example1: start.bash                 //uses old $PORTCFG
example2: start.bash -gm 10.0.1.100
example3: start.bash 10.0.1  100 101
example4: start.bash 10.0.1  auto
example5: start.bash -gm -release 10.0.1  10 11 12 13   20 21 22 23
```

If the script is parameterized correctly, it performs all tasks required to start the router:

- Stop previous instances of the *ipc-daemon* and the *oobjectManager* with *stop.bash*,

- start a new instance of the *ipc-daemon*,
- start the *oobjectManager* which will start the router, and, if desired, the *roboCupGameManager*,
- and kill the *ipc-daemon* after the termination of the *oobjectManager*.

The CMPack'02 mode compensates for the lack of the ability of the current tcpGateway to send messages received from different subjects (senders) to a single observer (receiver). It also generates the configuration files required to run CMPack'02. Thus, the GermanTeam was able to perform test games against the world champion of 2002.

stop.bash. As already mentioned above, there is second script called *stop.bash* in the same directory that removes all processes started by *start.bash*, including the *ipc-daemon* and its temporary files.

5.4 Motion Net Code Generator

The *Motion Net Code Generator* parses a set of motion specifications described in a special language and compiles it into C code. The parser checks the motion set defined for consistency, i. e., it checks for missing transitions from one motion to another and for transitions to unknown motions.

With this tool it is possible to generate motions that consist only of fixed sequences of joint positions quickly and easily. This is the case for all kicks implemented by the GermanTeam as well as some other motions including, e. g., head stand and ball holding. The resulting C code is integrated into the *SpecialActions* module (cf. Sect. 3.9.2).

The *Motion Net Code Generator* also generates an xml representation of all motions allowing new motions to be used in the behavior (cf. Sect. 3.8.1) without having to specify them in more than one place.

For interactively testing motions and their transitions, RobotControl provides a dialog to request specific motions. The requested motion and the transition from the current one will be executed immediately (cf. Sect. J.5.2). Furthermore, a second dialog is provided to transmit new motion descriptions to the robot and execute them without the need to recompile anything (cf. Sect. J.5.4).

Motion Description Language. The specification for a single motion consists of the description of the desired action and the definition of a set of transitions to all other motions. This is simplified by using groups of motions, e. g., it is possible to define that the transition from motion X to any other motion always goes via the motion Y.

As most simple motions (such as kicking or standing up) can be defined by sequences of joint data vectors, a special motion description language was developed, in which all our special motions are defined. Programs in this language consist of transition definitions, jump labels, lines defining motor data, and lines defining PID data. A typical data line looks like this:

~ ~ ~ ~ ~ ~ -350 -190 1750 -350 -190 1750 -1840 -40 2500 -1840 -40 2500 1 25

The first three values represent the three head joint angles, the next three values describe the mouth and the tail angles, followed by the twelve leg joint angles, three for each leg, all angles given in milliradians. The last but one value decides whether specified joint angles will either be repeated or interpolated from the current joints angles to the given angles. The last value defines how often the values will be repeated or over how many frames the values will be interpolated, respectively. The tilde character in the first six columns means that no specific value is given, i. e. "don't care". This has special importance for the head joint angles as it allows head motion requests to be executed.

5.5 Emon Log Parser

The Perl script *emonLogParser* provided by the Open-R SDK samples was considerably extended to retrieve as much information as possible from the log files called *emon.log* generated by the Emergency Monitor.

The script *GT2003/Bin/emonLogParser.pl* uses *mipsel-linux-readelf* and *mipsel-linux-objdump* to output an assembler dump around the crashing opcode and around the caller of that routine. The crashing line is highlighted. This is especially useful if the crash happened in unoptimized binaries with debug symbols. Furthermore the call stack is analyzed to give an idea of the order of calling methods.

The script has to be called by:

```
Bin/emonLogParser.pl <emon.log> <configuration>
```

So, the path to the emon.log of the crash has to be provided as well as the name of the configuration of GT2003 that caused the crash, e. g. *Debug*, *Release* or *DebugNoDebugDrawing*. This will find the correct **.nosnap.elf* in the build directories. It can easily be modified to be used with binaries of other teams. Of course it is only useful to provide the binaries that caused the crash.

Chapter 6

Conclusions and Outlook

The GermanTeam now exists for more than two years. The general architecture developed in 2002 has proven to be sustainable, still satisfying our needs. Only a few changes were applied during the last year. Switching from the Greenhills-based environment to the gcc-based environment required only minor changes in the platform dependent part. The behavior architecture XABSL was used by all sub-teams at the German Open, and again in Padova. It has been proven to be a flexible and powerful way to describe behaviors.

In 2003, the main innovations by the GermanTeam were the introduction of using edges for self-localization, detecting and modeling obstacles to improve game play, using potential fields to implement basic behaviors, and using dynamic role assignments. Additional innovations, not used during the games this year, but that will hopefully be used next year, are the automatic color calibration, and the detection of collisions.

Despite all the problems that arise when software is developed by a group of persons distributed over different towns, we recommend to build up national teams as the GermanTeam is one. Having enough participating team members, different solutions for single tasks can be employed and compared to each other. The different scientific backgrounds of the members from different universities enriched the project very much. At last, the rivalry between the single teams results in better solutions for single tasks.

Altogether we were quite satisfied with the results we achieved, and we are continuing that work. We hope to reach even better results in the competitions next year in Lisbon.

6.1 The Competitions in Padova

This year, the GermanTeam was quite prepared when it arrived in Padova. Many new features were added to the code, in test games against the code of the previous world champion, CM-Pack'02, results of up to 7:0 were achieved, and the main parts of the challenges were already solved. However, although first test games ended as expected (10:0 against Uppsala, 7:0 against Osaka, both in 10 minutes), it turned out that there was still a lot to do. The wireless network was very unreliable in Padova, resulting in poor coordination between the robots. Many actions, such

Round Robin	
GermanTeam – Austin Villa	9:0
GermanTeam – UTS Unleashed!	2:2
GermanTeam – UPennalizers	3:1
GermanTeam – Asura	5:0
Quarter Final	
GermanTeam – CMPack’03	2:2 (x:x+1)

Table 6.1: The results of the GermanTeam in Padova

as walking to the ball and kicking or changing roles, took too long. So the code was improved from game to game.

The GermanTeam finished the round robin as winner of its group, even against the later runner-up, the UPennalizers. In the quarter final, the GermanTeam lost in a 29 minutes penalty shootout against CMPack’03. However, the GermanTeam won the RoboCup Challenge with 70 out of 72 possible points.

The results of the actual games are shown in table 6.1. Please note that the actual number of goals scored in the penalty shootout is not known.

An important point is also that members of the GermanTeam were accepted for four talks in the RoboCup Symposium. The groups involved in the GermanTeam even contributed an overall number of six talks and one poster to the Symposium, including the paper that won the scientific award.

6.2 Future Work

The GermanTeam owns a powerful code basis for the next year’s work. For the RoboCup German Open in April 2004, each of the four universities will again set-up its own team based on the shared code basis with own solutions for different tasks. From their different research interests, the teams will also focus on different topics next year.

6.2.1 Humboldt-Universität zu Berlin

An important aspect of our future work will be the application of case based reasoning to robot control architectures and machine learning. The efforts will be pursued not only in the Sony Legged League but also in the Simulation League.

A behavior architecture called the “Double Pass Architecture” [4] has already been implemented in the Simulation League and will be applied to the Sony Legged League. It provides for long term “deliberator” planning and short time “executor” reactions. The executor allows quick reactions even for the options on the higher levels in the option hierarchy. This is made possible by using the reduced search space defined prior by the deliberator. It implements a kind of bounded rationality. Therefore, the state machine concept has to be extended for the two separate passes of the deliberator and the executor (the name “Double Pass Architecture” refers to

these two passes). Many useful behaviors have been developed. Selecting the appropriate one becomes an increasingly difficult task. It becomes even more difficult if behaviors are combined to more complex ones, such as they can be described in the option hierarchy. The *Extensible Agent Behavior Specification Language* (XABSL) will be extended and adopted to that.

Another prerequisite of useful decisions is a reliable world model. In case of the Sony AIBO, knowledge about the environment is exclusively derived from the camera image. With this limited field of view, information gathering has to be optimized. First steps in this direction have been taken by actively scanning for landmarks using world model information (i. e. pointing the camera in a direction where a landmark should be according to the world model). A tighter coupling of information gathering and information processing turned out to be desirable rather then having the two run as separate processes. Active vision and attention based vision approaches will be examined. World and object modeling will be extended to make use of negative information (e. g. the ball was not seen) and to actively search for information that is needed (e. g. have the robot look for a specific landmark that is needed to clarify the robot's position on the field). Having both, complex behavior and reliable world model, the correspondence of situations and most appropriate actions have to be resolved. This will be done by methods of case based reasoning. Cases describe typical behavior in typical situations (e. g. standard situations). The recent situation is matched against the case base, and the most similar cases are analyzed for proposals of behaviors. The behaviors are adapted according to the recent situations. Problems to be solved in the next steps include description of cases, definition of useful similarity measures and adaptation methods.

In this year's RoboCup symposium we presented and auto-calibrating vision system for the RoboCup environment [10]. We were not able to use it in the competition because it did have problems in certain situations. However we feel confident to have the system running reliably to use it next year. The system uses knowledge about the robot's environment in the form of heuristics and qualitative color calibration for image processing.

The obstacle model which proved highly successful in this years RoboCup challenge (see 3.5, 4.3) will be integrated more extensively into the behaviors of the agent. This may also require a more abstract, qualitative modeling of the agent behavior. Similarly, the newly gained possibility to detect collisions of the robot with its surroundings and other robots will be reflected in agent behaviors.

Furthermore, we are trying to improve the motion modeling of the robot, the long term goal being to develop a full motion model of the robot: a model that integrates robot locomotion *and* robot (special) actions such as kicking. By this we hope to achieve smoother, better controlled, and overall quicker and more livelike robot movement.

6.2.2 Technische Universität Darmstadt

The team in Darmstadt will continue their efforts towards a complete, efficient, and validated simulation of the four-legged robot's dynamics and its use for dynamic off-line optimization, on-line stabilization, and control of dynamic walking and running gaits. For the behavior control of cooperating robots as well as for object recognition it is planned to investigate alternative approaches to the already existing ones.

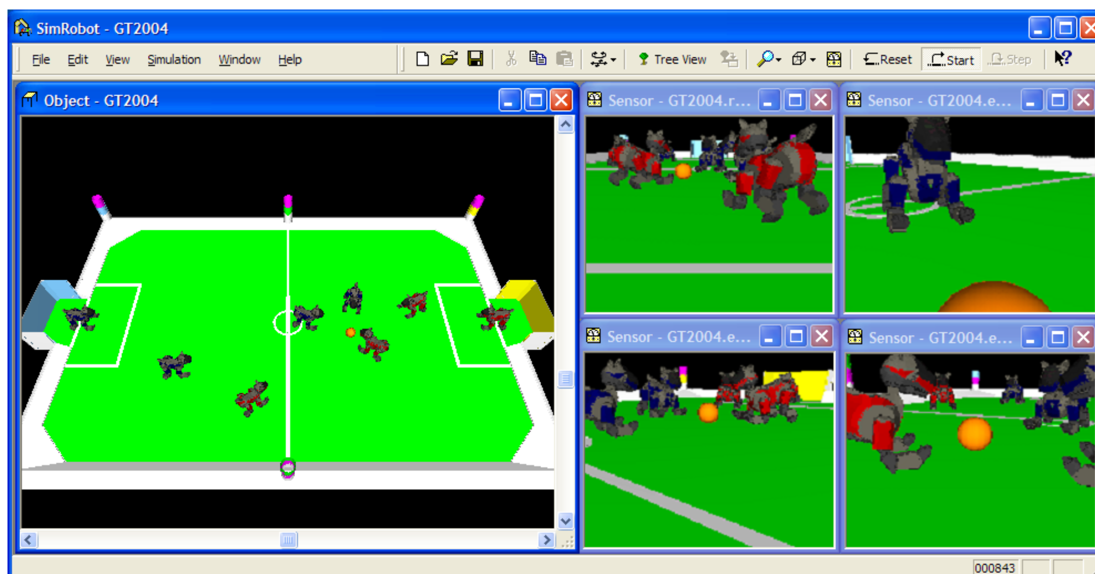

Figure 6.1: The OpenGL-based version of SimRobot. The picture shows a typical scene and the camera images of four simulated robots.

In 2001, the GermanTeam was able to measure the true positions of the robots on the field with a camera mounted at the ceiling. We propose to reintegrate such a mechanism in GT2004 at least for one half of the field. By this different localization methods can be compared on an empiric level. Besides with the exact data not only the self-localization but also a number of other algorithms can be tested and even automatically learned. This includes localization of other robots, the ball, odometry, and others.

6.2.3 Universität Bremen

As the use of the Markov-localization was very successful in the GermanTeam 2003, probabilistic approaches will also be introduced to model the location of the ball and of the opponents. In contrast to the field of self-localization, in which the sensor resetting approach by [11] can only be used if an estimation of a global pose can directly be derived from the sensor readings, the sensor resetting method seems to be a promising approach for the probabilistic modeling of the locations of the opponents and the ball in a robot-centric system of coordinates. In addition, work that was done on tracking people [17] can be integrated in such an approach. The modeling of the world state will also exploit the ability of the robots to communicate.

After such a probabilistic world model has been realized, it will be investigated, how the uncertainties can influence the behavior, or whether even a probabilistic behavior control can be implemented. The German Open 2004 will be the test-bed for such an approach.

In addition, the new, OpenGL-based version of SimRobot will replace the previous version. As it uses the hardware acceleration of modern graphics hardware, it is faster than the version currently used which results in higher frame rates in the generation of camera images, and it also prepares the GermanTeam for the simulation of the hires-camera of the new ERS-7.

6.2.4 Universität Dortmund

We will continue in developing a robust and detailed distributed world model and collective behavior. Therefore, we will enhance our world-model interchange protocol (WIP), that allows including and excluding single robots from the world-model interchange as well as from collective behavior transparently. Finally, we will build a peer-to-peer like network functionality, such that every robot can share sensor information and computing power with the rest of the team.

Chapter 7

Acknowledgments

The GermanTeam and its members from Berlin, Bremen, Darmstadt, and Dortmund gratefully acknowledge the continuous support given by the Sony Corporation and its Open-R Support Team. The GermanTeam thanks the organizers of RoboCup 2003 for travel support. The team members from Berlin and Bremen thank the Deutsche Forschungsgemeinschaft (DFG) for funding parts of their respective projects. The team members from Dortmund thank the Deutsche Arbeitsschutz Ausstellung (DASA) and the Thyssen Krupp AG for their effective cooperation. Further, we thank the Deutscher Akademischer Austauschdienst (DAAD), Lachmann & Rink GmbH, the Dortmund Project, the Freundegesellschaft der Universität Dortmund e.V., and the Faculty of Information Science for travel support.

The members of the GermanTeam 2003 also want to thank the members of the GermanTeam 2002 for creating the foundation for the continuing success of the GermanTeam and for writing the previous year's team report [3] that was the basis for this document.

The GermanTeam uses a variety of code libraries and tools and also likes to thank the authors of them:

- A code library called "Sizing Control Bars" from Cristi Posea. (http://www.datamekanix.com) is used for the dialogs in RobotControl.
- A code library for "Internet Explorer-like toolbars" from Nikolay Denisov (nick@actor.ru) is used.
- The code library "Grid Control" from Chris Maunder (cmaunder@mail.com) is used for the "Settings" dialog.
- Doxygen (http://www.doxygen.org/) is used for the source documentation.
- The "dot" tool from the GraphViz collection (http://www.graphviz.org) is used for behavior documentation purposes.
- The LibXSLT (http://xmlsoft.org/XSLT/) library is used used for behavior documentation purposes.

- This product includes software developed by the Apache Software Foundation (http://www.apache.org/).
- This product includes DOTML developed by Martin Lötzsch (http://www.martin-loetzsch.de/DOTML).
- This product includes XABSL developed by Martin Lötzsch (http://www.ki.informatik.hu-berlin.de/XABSL).
- This product includes SimRobot developed by Thomas Röfer, Uwe Siems, Christoph Herwig, and Jan Kuhlmann (http://www.informatik.uni-bremen.de/simrobot).

Appendix A

Installation

The GermanTeam uses Microsoft Windows as development platform. The package provided was used under Windows 2000 and Windows XP. The center of the development process is Microsoft Visual Studio; all parts of the system are edited and built with this software.

A.1 Required Software

- Microsoft Windows 2000/XP
- Microsoft Visual C++ 6.0 SP5 (can be installed anywhere)
- Cygwin 1.3.22 (can be installed anywhere). Version 1.5.x does not work with wireless!
- CygIPC 1.13-2 (unpack it to Cygwin-Path */*). Do not install it as a service!
- gtk+ 1.3 (unpack it to Cygwin-Path */*)
- Open-R SDK 1.1.3-r2 and MIPS developer tools with self-built gcc-3.3.1 (Cygwin-Path */usr/local/OPEN_R_SDK*)

The system path and the path of Visual Studio (extras/options/directories/executable files) must include *...\cygwin\bin;...\cygwin\lib;*.

A.2 Source Code

The source code has to be unpacked anywhere in a directory *XXX\GT2003*. Without white spaces in *XXX*. There are several subdirectories under this root:

Bin contains the binaries of the programs running on the PC after they have been compiled.

Build contains all intermediate files during compiling.

Config contains the configuration files. Most of them will also be copied to *OPEN-R/APP/CONF* on the memory stick.

Doc contains this document[1] and the documentation generated by *Doxygen* from the source files.

Make contains makefiles, batch files, and Visual C++ project files. The *GT2003.dsw* is located here, i. e. the file that has to be launched to open Visual C++.

Src contains all source files of the GermanTeam.

Util contains additional utilities, e. g. *Doxygen*.

The directory *Src* contains subdirectories that can be grouped in two categories: on the one hand, some directories contain code that runs on the robots, on the other hand, other subdirectories hold the code of the tools running on the PC.

A.2.1 Robot Code

Modules contains source files implementing the modules of the robot control program. For each module there exist an abstract base class, one or more different implementations, and, at least if there are multiple implementations, a module selector that allows switching between the different implementations.

Platform contains the platform dependent part of the robot code. There exist three subdirectories containing the platform specific implementations for *Aperios*, *Win32*, and *Linux* (Cygwin). A fourth directory *Win32Linux* contains implementations that are shared between Cygwin and Windows. *Platform* itself contains some header files that automatically include the code for the right platform.

Processes contains one subdirectory for each process layout, and each of these subdirectories contains one implementation file for each process (**.cpp*), the *object.cfg*, the *connect.cfg*, and the *.ocf*-file required for the process layout.

Representations contains source files implementing classes the main purpose of which is to store information rather than to process it. Objects of classes defined here are often communicated between different modules, processes, or even different robots.

Tools contains all that does not fit into the other categories. The mathematical library (cf. Sect. 2.1.4) can be found here, the implementation of streams (cf. App. D), and message queues. However, the code of the Windows tools such as RobotControl cannot be found here.

[1] Or at least it is a good idea to place it there.

A.2.2 Tools Code

Depend contains the code of the tool (cf. Sect. 2.2.4.1) to create the dependencies for all robot builds as well as a few scripts to create dependencies for RobotControl and SimGT2003, too, and scripts to create Visual Studio project files from these dependencies (cf. Sect. 2.2.4.4).

MotionNetCodeGenerator contains the code of the tool (cf. Sect. 5.4) to create new motions, i. e. special actions (cf. Sect. 3.9.2).

RobotControl contains the code of the tools with the same name (cf. Sect. 5.2).

Router contains the code of the mediator between the robots and the Windows-based tools such as RobotControl (cf. Sect. 5.3).

SimRob95 contains SimRobot, a kinematic robotics simulator. It is the framework for SimGT2003 (cf. Sect. 5.1) and the simulation integrated into RobotControl (cf. Sect. 5.2).

A.3 The Developer Studio Workspace GT2003.dsw

The Developer Studio Workspace GT2003.dsw contains several projects, most of them in different configurations:

Documentation. This project creates the documentation of the code using *Doxygen*. Documentation can be generated for the projects *GT2003*, *RobotControl*, and *SimGT2003*. Please note that legacy code such as *SimRobot* does not support *Doxygen* and generates no proper help files.

GT2003 creates the code for the robots. It can be selected between *Release* (optimized, no debugging information and support), *Debug* (full debugging information and support), *Debug no DebugDrawings* (debugging information and support, but no DebugDrawings to reduce performance impacts), and *Debug no WLAN* (debugging information and support, but memory stick and console are used instead of wireless lan). In addition, a process layouts (at the moment only *CMD*) can be selected.

The result of the build process can be copied to a memory stick by calling *copyfiles.bash*. Please note that this file assumes that the memory stick can be found in drive *E:* if your HOSTNAME is not handled in *copyfiles.bash*. Try *copyfiles.bash - -help* to see all options. Instead of calling *copyfiles.bash* manually, you can use the WLAN dialog of RobotControl (cf. Sect. J.2.5) to ensure that the same configuration parameters are used in RobotControl and on the memory stick.

RobotControl can be built in different configurations. The executable will be copied to *GT2003\Bin*. The *Win32 CMD Release* configuration uses statically linked MFC libraries, the Debug configurations dynamically linked ones. The configuration *Win32 Debug Optimized* uses $G6$ optimization, but does not support *Edit and Continue*. *Win32 Debug* supports it but is not optimized.

Router. The router (cf. Sect. 5.3) can only be built as a debug or a release version. The result will be copied to *GT2003\Bin*, together with several executables from the Open-R for Linux/Cygwin release. Note that only the debug version checks for errors during the initialization of the program.

SimGT2003 currently only supports a single configuration. The executable, together with the help file, will be copied to *GT2003\Bin*.

The other projects generate libraries (*GUI*, *SimRobot*, *SimRobotForRobotControl*), source code (*SpecialActions* by executing MotionNetCodeGenerator), tools (*Depend* cf. Sect. 2.2.4.1, *MotionNetCodeGenerator* cf. Sect. 5.4), or preprocessed behavior (*Xabsl2*cf. Sect. 3.8.1) required by at least one of the projects named above. There is no need to build them directly because they will be built on demand. The only exception is the configuration *Documentation* of the project *Xabsl2*. It produces a pretty good documentation of the behavior.

Appendix B

Getting Started

If you have installed the required software and the source code we suggest you to follow the introduction to GermanTeam's code given in this section. Step by step it is explained how to use the debug tool on the PC, how to play with the robots and how to let the robots play. In addition, the main configuration files used by the robots are described.

B.1 First Steps with RobotControl

This debug tool of the GermanTeam provides a lot of possibilities. In this section the most useful features and some impressive effects are presented. For a more detailed description see appendix J.

B.1.1 Looking at Images

Image and Segmented image. Open the *Image Viewer Dialog*, the *Color Table 64 Tool* and the *Color Space Dialog*. In the *Log Player Toolbar*, open a log file, e. g. *Config\Logs\padova01.log* and click on the *Step Forward* button a several times. In the bottom right corner of RobotControl the name of the current log file and the current number of the message are displayed.

All images contained in the log file are displayed in the *Image Viewer* and in the *Color Table 64 Dialog*. The *Color Table 64 Dialog* shows the segmented image. The *Color Space Dialog* shows the colors of the YUV-cube used by the current image.

With clicks with the left and the right mouse buttons into one of the images in the *Color Table 64 Dialog*, you can modify the color table. Clicking and dragging with the left mouse button in the *Color Space Dialog* changes the point of view. With the context menu you can select to show height maps of single channels of the image. Or you can display a 3D view of the current color table. The image viewer has eight areas to display images. In the context menu of these areas you can choose the image to be shown there, e. g. the raw image or a debug image such as the *imageProcessorGoals*.

Results of the Image Processing. Please make sure that in the toolbar *Simulated Robots* the state for the first red robot is set to *Active, generate images*. If you have to change the state, press the *reset* button afterwards. All images from the log files are not only displayed by the dialogs, but also put into the image processor now. Each image contains position and rotation of the camera relative to the body. This information was calculated by the *SensorDataProcessor* on the robot. Therefore the *SensorDataProcessor* on the PC has to be switched off for the image processor to work properly: in the *Settings Dialog* change to the setting *images from robot*. If that setting does not exist, create a new one by pressing the new button while the default setting is selected. Then change the local solution for the *SensorDataProcessor* from “Default” to “disabled”.

Several debug drawings illustrate how the image processor works. With the context menu the drawings can be selected. Select “percepts: ball; flags; goals”, “image processor: ball”, and “image processor: horizon” to see the horizon line in the image and how the flags, the goals, and the ball are recognized. The color table *GT2003\Config\Padua\coltable.c64* is adapted to the log files from Padova. If you change something in the color table, you can see the effect on the percepts immediately.

B.1.2 Discover the Simulator

The simulator provides the same data as a real robot would do. To see this, open the toolbar *Simulated Robots*, the *Object Viewer* (i. e. press the button on the toolbar *Simulated Robots*) and the *Large Image Viewer*. In the toolbar *Simulated Robots* press the *start* button. In the *Image Viewer* you can see the images the simulator provides.

In the *Simulator Toolbar* press the *back switch* of the robot two times for the duration of a second to start the robot. The robot is a goalie by default. If you move the ball in front of the goal, you can see how the goalie behaves.

With the *Joint Viewer* dialog, you can visualize the joint and sensor data provided by the simulated robot. In the debug keys toolbar select *Edit table for local processes*. Select the debug key “sendJointData” and choose *Always*. Then select the debug key “sendSensorData” and choose *Always*. With the context menu in the joint viewer select “Sensors: head” and “Actorics: Head”. The line graphs show the values of the joints and the sensors while the robot looks to the left and to the right alternately. Or try to select “Actorics: legFR” and deselect the values for the head. Then you can see the joint data of the front right leg while the robot moves.

B.2 Playing Soccer with the GermanTeam

B.2.1 Preparing Memory Sticks

First of all you need a complete build of the Visual Studio project GT2003. If you don’t have Visual Studio use *Make\makeForRobot.bat* The result of the build process will be in *GT2003\Build\MS*. This directory contains all binaries and system configuration files, but not our own configuration files.

Then you may want to change the player role, the team color, or the IP address, e. g. to prepare memory sticks for two different robots. You can use the *WLAN Toolbar* of *RobotControl* for that: Press the *Add connection* button and fill in the settings as they are required for your wireless environment. Then insert a memory stick in your memory stick-reader and press *cp* to copy all data for a specific player. This calls *GT2003\Make\copyfiles.bash*.

You may have to edit that file, because drive *E:* is assumed to be the memory stick by default[1]. If it is adapted to your needs you may call it by pressing one of the *cp* buttons in *WLAN Dialog*. Then you get information about the configuration on the stick (team color, player role, WLAN settings) to check whether writing was successful.

B.2.2 Establishing a WLAN Connection

After inserting a WLAN card into an AIBO, creating a memory stick as explained in section B.2.1, inserting it into an AIBO and starting that robot, you should be able to ping the robot. Its IP address was shown as the result of executing *GT2003/Make/copyfiles.bash*. The IP addresses of the robot and the computer trying to ping it have to be in the same subnet, the ESSID has to be the same as well as the (usage of) encryption. Furthermore you should use APMode 0 for ad hoc connections and APMode 1 or 2 when using an access point or a computer with Windows XP for pinging the robot.

If ping works for all robots you want to use, you should start the router (cf. Sect. 5.3) by executing *GT2003\Bin\start.bash* with the IP addresses of the robots as parameters. This will enable you to establish a debug connection to one robot as well as the opportunity for all robots of the same team to communicate with each other. The robots use point-to-point connections for that. Each connection is visualized by a green or yellow LED, so if you want to start a complete team of four robots, each should have 3 yellow and green LEDs switched on.

In some cases the Linux inter-process communication (IPC) for Cygwin is not very stable, so if you recognize strange error messages when trying to start the router, abort it. Executing *start.bash* again might help. Due to restrictions of Cygwin you will not be able to establish all connections for two complete teams including intra-team communication, *RoboCup Game Manager* control and debug channels, but one and a half team should work fine.

B.2.3 Operate the Robots

Once all the robots are started, they are in the state “initial”. The red LEDs show the role of the player:

1 red LED: Goalie

2 red LEDs: Defender

3 red LEDs: Striker 1

[1] If you use the option *–force* you should be **absolutely sure** that *copyfiles.bash* will use the proper drive, because it will format it without any question!

4 red LEDs: Striker 2

Now you can start the router on the remote computer. If all the green LEDs are on, then the robot has connections to all other robots. If not, then there are problems with the wireless network.

Then you can use the *RoboCup Game Manager* by Sony or touch the back switches of the robots to set them into the "ready" state. Both top LEDs of the robots blink alternating. Note that the robots walk to their start positions on their own. If they walk around without arriving there, you can press one of the pressure sensors on the head to stop them. If the robots arrived at their start position, the tail LED blinks quickly in the team color. For the *striker1*, you can set whether the own team does the kickoff by using the *RoboCup Game Manager* or pushing the tail to the left or to the right.

The game can be started with the *RoboCup Game Manager* or by touching the back switches of the robots.

The robots can be stopped again with the *RoboCup Game Manager* or by pressing their back switch for more than 1 second. They are in the "ready" state again then.

B.3 Explore the Possibilities of the Robot

In this section some nice "experiments" are described that demonstrate the possibilities of the robot.

B.3.1 Send Images from the Robot and Create a Color Table

To have a more relaxed robot it is suggest to start the robot without pressing the back button to keep it in *playDead* mode. In the *Debug Keys Toolbar* type "1" in the edit box and select the button *n times*. Each time you press *Send* an image is sent by the robot. Everything described in section B.1.1 for images from log files can now be done with "real" images.

You can create a color table based on these images using the *Color Table 64 Tool*. Have a look at the segmented images from the log files from Padova if *Padua\coltable.c64* is used. The segmented images from your field should look the same way with the color table for your field for the image processor to work correctly. Then overwrite *Padua\coltable.c64* using the save button.

B.3.2 Create Own Kicks

To create your own kicks use the *Mof Tester Dialog*. In the *Settings Dialog* change the solution for the Module *Motion Control* from Default to Debug.

Now you can move the legs of the robot to a desired position and they will stay as forced. If you press *Read* the sensor data of each joint is read and transmitted to the dialog. Now move the legs to the next position and press *Read* again. In this way you can record the whole kick or another motion step by step. With *Execute* you can execute the motion. For more details study section J.5.4.

B.3.3 Test simple behaviors

To test some simple behaviors, all modules have to work with the default solution. If you did some experiments before, change to the "default" setting in the *Settings Dialog*.

Open the *Xabsl2 Behavior Tester Dialog* and select "Test on robot". Now you can see the option activation path, the current basic behavior with its parameters, the current motion request and the current values of the selected input symbols. If you move the ball in front of the robot you can see how the value for "ball.seen.distance" changes.

B.3.3.1 Test Basic Behaviors

Basic Behaviors are the basic components of the behavior, here is a closer look at two of them. With the *Xabsl Behavior Tester Dialog*, you can test a basic behavior with the desired parameters. In the top most drop down box you can find the basic behaviors below the separator.

To test the basic behavior that goes to the ball select "go-to-ball" in the combo box. In the edit box for the first parameter type for example "300" and press the button *Send request*. If the robot can see the ball it goes there and stops at a distance of 300 mm (center of ball to point beneath the pan joint of the head). Now try different distances and see how the robot reacts. You can also try to change the second parameter of this skill - the maximal speed when going to the ball (mm/s).

With the basic behavior "go-to-point" you can see if the self-localization works. The first to parameters specify the x and the y position of the robot on the field. The unit of measurement is millimeter, the origin of the system of coordinates is the center of the field. The x-axis points to the opponent goal. The third parameter specifies the desired angle of the robot at the destination point. With the other combo boxes in the *Xabsl Behavior Tester Dialog* you can set the output symbol for the "head-control-mode" to different values, for example "search-for-landmarks", search for ball or "search-for-ball" and study the effect to the accuracy of the localization.

B.3.3.2 Test Options

Options represent the higher level of the behavior. They also can have parameters. You can select different options with the combo box at the top of the *Xabsl2 Behavior Tester Dialog*. The options are above the separator.

The option "go-to-ball-and-kick" is one of the most important options in the play-soccer behavior. If you select this option you can perform the following tests. Place the ball behind the robot. The robot will turn until it sees the ball then go there and kick to the direction of the opponent goal. Put the ball somewhere on the field. The robot will go to the ball.

B.4 Configuration Files

The robots of the GermanTeam are configured using several configuration files. The files that have to be adapted to be able to run the code of the GermanTeam are described in this section. On the memory stick, all configuration files are located under *MS/OPEN-R/APP/CONF*.

B.4.1 location.cfg

This text file contains the name of a subdirectory in the configuration directory. The subdirectory contains location-dependent configuration information such as camera settings, the color table, and the set of active solutions. The *location.cfg* allows the members of the GermanTeam to store several such settings in the common CVS repository (a set of settings for each sub-team), while the only file that has to be changed locally is this one. If this file is not present or it is empty, the *camera.cfg*, *coltable.c64*, and *modules.cfg* are read from the main configuration directory, i. e. *MS/OPEN-R/APP/CONF* on the robot and *GT2003\Config* on the PC. In the code release, the file contains the name "Padua", so the configuration files in that subdirectory are used.

B.4.2 coltable.cfg

The GermanTeam uses an 18-bit color table with 6 bits color depth for each of the YUV channels, i. e. the two LSBs of each channel are dropped:

```
unsigned char colorClasses[64][64][64];
```

Each of the entries in the color table is one of the following color classes:

```
enum colorClass {noColor, orange, yellow, skyblue, pink,
                 red, blue, green, gray, white, black};
```

However, the color table in the binary file *coltable.c64* is compressed. It has to be written by a routine such as

```
unsigned char* colorTable = &colorClasses[0][0][0];
unsigned char currentColorClass = colorTable[0],
int currentLength = 1;
for(int i = 1; i < sizeof(colorClasses); ++i)
  if (colorTable[i] != currentColorClass)
  {
    stream << currentLength << currentColorClass;
    currentColorClass = colorTable[i];
    currentLength = 1;
  }
  else
      ++currentLength;
stream << currentLength << currentColorClass << int(0);
```

B.4.3 camera.cfg

This file describes the camera settings. It must contain the settings that were active when the color table was created. The binary file consists of three values (each four bytes and little endian):

```
enum whiteBalance {wb_indoor_mode, wb_outdoor_mode, wb_fl_mode};
enum gain {gain_low, gain_mid, gain_high};
enum shutterSpeed {shutter_slow, shutter_mid, shutter_fast};
```

B.4.4 player.cfg

With this text file, it is possible to set the team color and the initial role of a robot. Although the robots perform dynamic role switching, all robots must have different initial roles, because the roles are used for positioning before kickoffs and as fall-back if the wireless network is not working.

```
// teamColor red | blue
teamColor blue
// playerRole goalie | defender | striker1 | striker2
playerRole striker2
```

B.4.5 robot.cfg

The vision module determines many distances to objects by intersecting a view ray with planes that are parallel to the field. At least if objects are far away, the precision of such computations depends on precision of the estimation of the pose of the camera relative to the field. It has turned out that there are some variations between different robots in the relationship between the joint angles measured and the real posture of the head. Therefore, the *robot.cfg* contains correction values for the *tilt* and the *roll* of the body of the robot[2]. The *robot.cfg* contains these corrections for all robots of the GermanTeam, indexed by the MAC-addresses of the robots, e. g.:

```
[00022D1F626B]
bodyTiltOffset 0.06
bodyRollOffset 0
```

The goal of the calibration process is that a robot located in one goal, looking at the other goal, will calculate the *horizon* parallel to the field and on the height of the camera. This can be checked by displaying camera images and the *horizon drawing* in RobotControl. Using the *Debug Message Generator Dialog* (cf. App. J.7.1) with “Body Offsets” selected, the correction values can be directly entered into the edit field and sent to the robot, e. g. “0.06 0”. If the horizon is display parallel to the field and in the vertical middle of the opponent goal (i. e. at a height of approximately 15 cm), the values are correct. However, the robot will forget them. Therefore they have to be entered manually into the *robot.cfg* under the MAC-address of that robot afterwards.

Please note that the localization capabilities of the robots using the *GT2003SelfLocator* depend on these correction values.

[2]The two values are a first approach to compensate for the deviations. More correction values are required. Hopefully, it will be possible to let the Aibos automatically calibrate themselves in the future.

B.4.6 wlanconf.txt

Don’t forget to adapt the *wlanconf.txt* located in *MS/OPEN-R/SYSTEM/CONF* to the appropriate network settings.

Appendix C

Processes, Senders, and Receivers

C.1 Motivation

In GT2001, there exist two kinds of communication between processes: on the one hand, Aperios queues are used to communicate with the operating system, on the other hand, a shared memory is employed to exchange data between the processes of the control program. In addition, Aperios messages are used to distribute the address of the shared memory. All processes use a structure that is predefined by Sony's stub generator. This approach lacks of a simple concept how to exchange data in a safe and coordinated way. The resulting code is confusing and much more complicated then it should be.

However, the internal communication using a shared memory also has its drawbacks. First of all, it is not compatible with the new ability of Aperios to exchange data between processes via a wireless network. In addition, the locking mechanism employed may waste a lot of computing power. However, the locking approach only guarantees consistence during a single access, the entries in the shared memory can change from one access to another. Therefore, an additional scheme has to be implemented, as, e. g., making copies of all entries in the shared memory at the beginning of a certain calculation step to keep them consistent.

The communication scheme introduced in GT2002 addresses these issues. It uses Aperios queues to communicate between processes, and therefore it also works via the wireless network. In the approach, no difference exists between inter-process communication and exchanging data with the operating system. Three lines of code are sufficient to establish a communication link. A predefined scheme separates the processing time into two communication phases and a calculation phase.

C.2 Creating a Process

Any new process has to be part of a special *process layout*. Process layouts group together different processes that make up a robot control program, and they are stored in subdirectories under *GT2003\Src\Processes*. Process layouts are named after the processes that exist in them, and in fact, in 2003 there was only one layout, namely *CMD* that consists of the processes *Cognit(ion)*,

Motion, and *Debug*. An Aperios process is allowed to have a name with a maximum length of eight characters. To create a new process, one has to think such a name up (in fact a *full name* and a *short name* (up to eight characters)

- insert a new line into *GT2003**Src**Processes*\processLayout*object.cfg*, following the format */MS/OPEN-R/APP/OBJS/*shortName.*bin*,
- insert a line in *GT2003**Src**Processes*\processLayout\processLayout.*ocf* starting with *# objectmapping* followed by the *short name* and the *full name*,
- create a new *object* line in the same file using the *short name*,
- create a *.cpp* file in *GT2003**Src**Processes*\processLayout with the full name,
- and, insert that source file in the GT2003 project both under *GT2003**Processes*\processLayout and *RobotControl**SharedCode**Processes*\processLayout in the Microsoft Developer Studio.

The new source file must include "Tools/Process.h", derive a new class from *class Process*, implement at least the function *Process::main()*, and must instantiate the new class with the macro *MAKE_PROCESS*. As an example, look at this little process[1]:

```
#include "Tools/Process.h"

class Example : public Process
{
  public:
    virtual int main()
    {
      printf("Hello World!\n");
      return 0;
    }
};

MAKE_PROCESS(Example);
```

The process will print "Hello World" once. If the function *main()* should be recalled after a certain period of time, it must return the number of milliseconds to wait, e. g.

```
      return 500;
```

to restart *main()* after 500 ms. However, if *main()* itself requires 100 ms of processing time, and then pauses for 500 ms before it is recalled, it will in fact be called every 600 ms. If this is not desired, *main()* must return a negative number. For instance,

[1]Note that the examples given here will not compile, because the debugging support required by *class Process* is missing. One can derive from *class PlatformProcess* instead, naming the *main*-function *processMain*.

```
        return -500;
```

will ensure a cycle time of 500 ms, as long as *main()* itself does not require more than this amount of time.

Note that if main returns 0, it will only be recalled if there is at least one blocking receiver or at least one active blocking receiver (cf. next section). Otherwise, the process will be inactive until the robot will be rebooted.

C.3 Communication

The inter-object communication is performed by *sender*s and *receiver*s exchanging *packages*. Packages are normal C++ classes that must be *streamable* (cf. the technical note on streams in appendix D). A sender contains one instance of a package and will automatically transfer it to a receiver after the receiver requested it and the sender's member function *send()* was called. The receiver also contains an instance of a package. Each data exchange will be performed after the function *main()* of a process has terminated, or immediately when the function *send()* is called. A receiver obtains a package before the function *main()* starts and will request for the next package after *main()* was finished. Both senders and receivers can either be blocking or non-blocking objects. The function *main()* will wait for all blocking objects before it starts, i. e. it waits for blocking receivers to acquire new packages, and for blocking senders to be asked to send new packages.[2] *main()* will not wait for non-blocking objects, so it is possible that a receiver contains the same package for more then one call of *main()*.

C.3.1 Packages

A package is an instance of a class that is streamable, i. e. that implements the $<<$ and $>>$ operators for the classes *Out* and *In*, respectively. So, an example of a package is

```
class NumberPackage
{
  public:
    int number;
    NumberPackage() {number = 0;}
};

Out& operator<<(Out& stream,const NumberPackage& package)
{
  return stream << package.number;
}

In& operator>>(In& stream, NumberPackage& package)
```

[2]Note that under RobotControl, a process will be continued when a single blocking event occurs. This is currently required to support debug queues.

```
{
  return stream >> package.number;
}
```

Note also that it is a good idea to provide a public default constructor.
A special case of packages are Open-R packages:

- Packages that are received from the operating system must provide a streaming operator that reads exactly the format as provided by Open-R. The packages are all defined in *<OPENR/ODataFormats.h>*. However, the data types provided there do not reflect the real size of the objects, they are only headers. Therefore, new types must be declared that have the real size of the Open-R packages. This size can be determined from their *vectorInfo.totalSize* member variable. The size is constant for each type, but it may vary between different versions of Open-R. Such data types are only required to implement the streaming operators, they are not needed elsewhere.
- Packages that are sent to the operating system require special allocation operators. Therefore, special senders (cf. next section) were implemented that allocate memory using the appropriate methods, and then use these memory blocks for the communication with the operating system.

C.3.2 Senders

Senders send packages to other processes. A process containing a sender for *NumberPackage* could look like this:

```
#include "Tools/Process.h"

class Example1 : public Process
{
  private:
    SENDER(NumberPackage);

  public:
    Example1() :
      INIT_SENDER(NumberPackage,false) {}

    virtual int main()
    {
      ++theNumberPackageSender.number;
      theNumberPackageSender.send();
      return 100;
    }
};
```

```
MAKE_PROCESS(Example1);
```

The macro *SENDER* defines a sender for a package of type *NumberPackage*. As the second argument is false, it is a non-blocking sender. Macros as *SENDER* and *RECEIVER* will always create a variable that is derived from the provided type (in this case *NumberPackage*) and that has a name of the form *the*Type*Sender* or *the*Type*Receiver*, respectively (e. g. *theNumberPackageSender*).

Packages must always explicitly be sent by calling the member function *send()*. *send()* marks the package as to be sent and will immediately send it to all receivers that have requested a package. However, each time the function *main()* has terminated, the package will be sent to all receivers that have requested it later and have not got it yet. Note that the package that will be sent has not necessarily the state it had when calling *send()*. As packages are not buffered, always the actual content of a package will be transmitted, even if it changed since the last call to *send()*.

As the communication follows a real-time approach, it is possible that a receiver misses a package if a new package is sent before the receiver has requested the previous one. The approach follows the idea that all receivers usually want to receive the most actual packages. The only possibility to ensure that a receiver will get a package is to only send it, when it already has been requested. This can be realized by either using a blocking sender, or by checking whether the sender has been requested to send a new package: *theNumberPackageSender.requestedNew()* provides this information. Note: a sender can provide a package to more than one receiver. *requestedNew()* returns true if at least one receiver requested a new package. This is different from a blocking sender: a blocking sender will wait for *all* receivers to request a new package!

C.3.3 Receivers

Receivers receive packages sent by senders. A process that reads the package provided by *Example1* could look like this:

```
#include "Tools/Process.h"

class Example2 : public Process
{
  private:
    RECEIVER(NumberPackage);

  public:
    Example2() :
      INIT_RECEIVER(NumberPackage,true) {}

    virtual int main()
    {
      printf("Number %d\n",theNumberPackageReceiver.number);
      return 0;
    }
```

```
};

MAKE_PROCESS(Example2);
```

Here, the function *main()* will wait for the *RECEIVER* (i. e. the second parameter is *true*), so it will always print out a new number.

However, one thing is missing: Aperios has to know which process wants to transfer packages to which other process. Therefore, the file *connect.cfg* has to be extended by the following line:

```
Example1.Sender.NumberPackage.S Example2.Receiver.NumberPackage.O
```

If more than one receiver is used in a process, the non-blocking receivers shall be defined first. Otherwise, the packages of the non-blocking receivers may be older than the packages of the blocking receivers. To determine whether a non-blocking receiver got a new package, call its member function *receivedNew()*.

Appendix D

Streams

D.1 Motivation

In most applications, it is necessary that data can be serialized, i. e. transformed into a sequence of bytes. While this is straightforward for data structures that already consist of a single block of memory, it is a more complex task for dynamic structures, as e. g. lists, trees, or graphs. The implementation presented in this document follows the ideas introduced by the C++ iostreams library, i. e., the operators << and >> are used to implement the process of serialization.

There are two reasons not to use the C++ iostreams library for this purpose: on the one hand, it does not guarantee that the data is streamed in a way that it can be read back without any special handling, especially when streaming into and from text files. On the other hand, the iostreams library is not fully implemented on all platforms, namely not on Aperios.

Therefore, the *Streams* library was implemented. As a convention, all classes that write data into a stream have a name starting with "Out", while classes that read data from a stream start with "In". In fact, all writing classes are derived from class *Out*, and all reading classes are derivations of class *In*.

All stream classes derived from *In* and *Out* are composed of two components: One for reading/writing the data from/to a physical medium and one for formatting the data from/to a specific format. Classes writing to physical media derive from *PhysicalOutStream*, classes for reading derive from *PhysicalInStream*. Classes for formatted writing of data derive from *StreamWriter*, classes for reading derive from *StreamReader*. The composition is done by the *OutStream* and *InStream* class templates.

D.2 The Classes Provided

Currently, the following classes are implemented:

PhysicalOutStream. Abstract class

OutFile. Writing into files

OutMemory. Writing into memory

OutSize. Determine memory size for storage

OutMessageQueue. Writing into a MessageQueue

StreamWriter. Abstract class

OutBinary. Formats data binary

OutText. Formats data as text

Out. Abstract class

OutStream<PhysicalOutStream,StreamWriter>. Abstract template class

OutBinaryFile. Writing into binary files

OutTextFile. Writing into text files

OutBinaryMemory. Writing binary into memory

OutTextMemory. Writing into memory as text

OutBinarySize. Determine memory size for binary storage

OutTextSize. Determine memory size for text storage

OutBinaryMessage. Writing binary into a MessageQueue

OutTextMessage. Writing into a MessageQueue as text

PhysicalInStream. Abstract class

InFile. Reading from files

InMemory. Reading from memory

InMessageQueue. Reading from a MessageQueue

StreamReader. Abstract class

InBinary. Binary reading

InText. Reading data as text

InConfig. Reading configuration file data from streams

In. Abstract class

InStream<PhysicalInStream,StreamReader>. Abstract class template

InBinaryFile. Reading from binary files

InTextFile. Reading from text files

InConfigFile. Reading from configuration files

InBinaryMemory. Reading binary data from memory

InTextMemory. Reading text data from memory

InConfigMemory. Reading config-file-style text data from memory

InBinaryMessage. Reading binary data from a MessageQueue

InTextMessage. Reading text data from a MessageQueue

InConfigMessage. Reading config-file-style text data from a MessageQueue

D.3 Streaming Data

To write data into a stream, *Tools/Streams/OutStreams.h* must be included, a stream must be constructed, and the data must be written into the stream. For example, to write data into a text file, the following code would be appropriate:

```
#include "Tools/Streams/OutStreams.h"
// ...
OutTextFile stream("MyFile.txt");
stream << 1 << 3.14 << "Hello Dolly" << endl << 42;
```

The file will be written into the configuration directory, e. g. *GT2003\Config\MyFile.txt* on the PC. It will look like this:

```
 1 3.14000 Hello\ Dolly
 42
```

As spaces are used to separate entries in text files, the space in the string “Hello Dolly” is escaped. The data can be read back using the following code:

```
#include "Tools/Streams/InStreams.h"
// ...
InTextFile stream("MyFile.txt");
int a,d;
double b;
char c[12];
stream >> a >> b >> c >> d;
```

It is not necessary to read the symbol *endl* here, although it would also work.

To make the streaming independent of the kind of the stream used, it could be encapsulated in functions. In this case, only the abstract base classes *In* and *Out* should be used to pass streams as parameters, because this generates the independence from the type of the streams:

```
#include "Tools/Streams/InOut.h"

void write(Out& stream)
{
  stream << 1 << 3.14 << "Hello Dolly" << endl << 42;
}
```

```
void read(In& stream)
{
  int a,d;
  double b;
  char c[12];
  stream >> a >> b >> c >> d;
}
// ...
OutTextFile stream("MyFile.txt");
write(stream);
// ...
InTextFile stream("MyFile.txt");
read(stream);
```

D.4 Making Classes Streamable

Streaming is only useful if as many classes as possible are streamable, i. e. they implement the streaming operators $<<$ and $>>$. The purpose of these operators is to write the current state of an object into a stream, or to reconstruct an object from a stream. As the current state of an object is stored in its member variables, these have to be written and restored, respectively. This task is simple if the member variables themselves are already streamable.

D.4.1 Streaming Operators

As the operators $<<$ and $>>$ cannot be members of the class that shall be streamed (because their first parameter must be a stream), it must be distinguished between two different cases: In the first case, all relevant member variables of the class are public. Then, implementing the streaming operators is straightforward:

```
#include "Tools/Streams/InOut.h"

class Sample
{
  public:
    int a,b,c,d;
};

Out& operator<<(Out& stream,const Sample& sample)
{
  return stream << sample.a << sample.b
                << sample.c << sample.d;
}
```

```
In& operator>>(In& stream,Sample& sample)
{
  return stream >> sample.a >> sample.b
                >> sample.c >> sample.d;
}
```

However, if the member variables are private, the streaming operators must be friends of the class. This can be a little bit complicated, because some compilers require the function prototypes to be already declared when they parse the *friend* declarations:

```
class Sample;
Out& operator<<(Out&,const Sample&);
In& operator>>(In&,Sample&);

class Sample
{
  private:
    int a,b,c,d;
  friend Out& operator<<(Out&,const Sample&);
  friend In& operator>>(In&,Sample&);
};
// ...
```

Another possibility to avoid these additional declarations would be to define public member functions that perform the streaming and that are called from the streaming operators. However, this would not be shorter.

If dynamic data should be streamed, the implementation of the operator $>>$ requires a little bit more attention, because it always has to replace the data already stored in an object, and thus if this is dynamic, it has to be freed to avoid memory leaks.

```
class Sample
{
  public:
    char* string;
    Sample() {string = 0;}
};

Out& operator<<(Out& stream,const Sample& sample)
{
  if(sample.string)
    return stream << strlen(sample.string) << sample.string;
  else
    return stream << 0;
}
```

```
In& operator>>(In& stream,Sample& sample)
{
  if(sample.string)
    delete[] sample.string;
  int len;
  stream >> len;
  if(len)
  {
    sample.string = new char[len+1];
    return stream >> sample.string;
  }
  else
  {
    sample.string = 0;
    return stream;
  }
}
```

D.4.2 Streaming using *read()* and *write()*

There also is a second possibility to stream an object, i. e. using the functions Out::write() and In::read() that write a memory block into, or extract one from a stream, respectively:

```
class Sample
{
  public:
    int a,b,c,d;
};

Out& operator<<(Out& stream,const Sample& sample)
{
  stream.write(sample,sizeof(Sample));
  return stream;
}

In& operator>>(In& stream,Sample& sample)
{
  stream.read(sample,sizeof(Sample));
  return stream;
}
```

This approach has its pros and cons. On the one hand, the implementations of the streaming operators need not to be changed if member variables in the streamed class are added or removed.

On the other hand, this approach does not work for dynamic members. It will corrupt pointers to virtual method tables if classes or their base classes contain virtual functions. Last but not least, the structure of an object is lost (not the data) when it is streamed to a text file, because in the file, it will look like a memory dump, which is not well readable for humans.

D.5 Implementing New Streams

Implementing a new stream is simple. If needed, a new medium can be defined by deriving new classes from *PhysicalInStream* and *PhysicalOutStream*. A new format can be introduced by deriving from *StreamWriter* and *StreamReader*. Streams that store data must be derived from class *OutStream*, giving a *PhysicalOutStream* and a *StreamWriter* derivate as template parameters, reading streams have to be derived from class *InStream*, giving a *PhysicalInStream* and a *StreamReader* derivate as template parameters.

As a simple example, the implementation of *OutBinarySize* is given here. The purpose of this stream is to determine the number of bytes that would be necessary to store the data inserted in binary format, instead of actually writing the data somewhere. For the sake of shortness, the comments are removed here.

```
class OutSize : public PhysicalOutStream
{
  private:
    unsigned size;
  public:
    void reset() {size = 0;}
    OutSize() {reset();}
    unsigned getSize() const {return size;}
  protected:
    virtual void writeToStream(const void*,int s) {size += s;}
};

class OutBinary : public StreamWriter
{
  protected:
    virtual void writeChar(char d,PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeUChar(unsigned char d,
                            PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeShort(short d,PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeUShort(unsigned short d,
```

```
                              PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeInt(int d,PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeUInt(unsigned int d,
                            PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeLong(long d,PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeULong(unsigned long d,
                             PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeFloat(float d,PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeDouble(double d,
                              PhysicalOutStream& stream)
      {stream.writeToStream(&d,sizeof(d));}

    virtual void writeString(const char *d,
                              PhysicalOutStream& stream)
    {
      int size = strlen(d);
      stream.writeToStream(&size,sizeof(size));
      stream.writeToStream(d,size);
    }

    virtual void writeEndL(PhysicalOutStream& stream) {};

    virtual void writeData(const void* p,int size,
                            PhysicalOutStream& stream)
      {stream.writeToStream(p,size);}
};

class OutBinarySize : public OutStream<OutSize,OutBinary>
{
  public:
    OutBinarySize() {}
};
```

Appendix E

Debugging Mechanisms

The software architecture of the programs and tools developed contains a rich variety of debugging mechanisms which are described in detail in the following sections.

The following section describes in detail how messages are passed and how they can be handled. The later sections describe a more high level approach and introduce macros simplify the workflow. Also it is shown how to create drawings for visualization purposes. (You can skip section E.1 to get started quickly and come back to it at a later point.)

E.1 Message Queues

Besides the package-oriented inter-object communication with senders and receivers (cf. Sect. C.3), *message queues* are used for the transport of debug messages between processes, platforms, and applications. They consist of a list of *messages*, which are stored and read using streams (cf. App. D).

Writing Data to Message Queues. As almost all data types have streaming operators, it is very easy to store them in message queues. When a message is written to a queue, a *message id* is added to identify the type x of the data. In addition, a time stamp is stored for every new message. A typical piece of code looks like this:

```
#include "Tools/MessageQueue/MessageQueue.h"
// ...
Image myImage;
myMessageQueue.out.bin << myImage;
myMessageQueue.out.finishMessage(idImage);

int numOfBalls = 3;
myMessageQueue.out.text << "found " << numOfBalls << " balls."
MyMessageQueue.out.finishMessage(idText);

int a, b, c, d;
```

```
myMessageQueue.out.bin << a << b;
myMessageQueue.out.bin << c << d;
myMessageQueue.out.bin.finishMessage(id4FunnyNumbers);
```

First an image is written in binary format to the queue. The type of the message is *idImage*. Then the text *"found 3 balls"* is written in text format to the queue. The id *idText* marks the message as unstructured text. At last, four integer values are written to the queue as a message of the type *id4FunnyNumbers*.

Transmitting Message Queues. Message queues are exchanged between processes such as all other packages. They are also used for the transmission of debug messages via the wireless network. They can be written to and read from logfiles. Messages can be moved to other queues using

```
myQueue.copyAllMessages(otherQueue);
myQueue.moveAllMessages(otherQueue);

message >> otherQueue;
```

copyAllMessages copies and *moveAllMessages* moves all messages to another queue. The third statement shows how a single message can be copied.

Distribution of Debug Messages. As all messages can be written in any order into a queue, a special mechanism for distributing the message is needed. A *message handler* does this job, e. g.:

```
class MyMessageHandler : public MessageHandler
{
  virtual bool handleMessage(InMessage& message);
};
// ...
bool MyMessageHandler::handleMessage(InMessage& message)
{
  switch (message.getMessageID()
  {
  case idImage:
    message.bin >> myImage;
    return true;
  case idText:
    message >> otherQueue;
    return true;
  case id4FunnyNumbers;
    message.bin >> a >> b >> c >> d;
```

```
    return true;
  default:
    return false;
  }
}
// ...
MyMessageHandler handler;
myQueue.handleAllMessages(handler);
```

The class *MyMessageHandler* is derived from *MessageHandler*. In the implementation, for every *message id* the data is read differently from a queue. Whereas the image is just streamed to a local member variable, the text message is copied to another queue. If a message has to be copied to two targets, the function *resetReadPosition* can be used to reset the read position of the source queue for the second read.

Instantiating Message Queues. The class *MessageQueue* has two different mechanisms to store the data into memory. On Windows and Linux platforms, a dynamic list is used. But on the Open-R platform, dynamic memory allocations are expensive. Therefore the messages are stored in a static memory buffer on that platform.

If the code containing a *MessageQueue* shall run on the Open-R platform, the size of the queue in bytes has to be set using

```
MessageQueue queue;
queue.setSize(1000000); // ignored on Win32 and Linux
```

Message Queues and Processes. The class *Process*, which is the base class for all processes, already has the members *debugIn* for incoming and *debugOut* for outgoing debug messages. In addition, *Process* is derived from *MessageHandler* so that every process can distribute debug messages.

E.2 Generic Debug Data

This is a way to quickly send data to a module, especially if you only looking for a mechanism to adjust internal parameters of a module. You can send an array of 10 double values to a module. It is necessary to give the data an ID and to include message handling in the module. The ”Test Data Generator Dialog” lets you send data easily to the module from within RobotControl.

First, you need to give create a new ID in *Src\Tools\Debugging\GernericDebugData.h*:

```
...

  enum GenericDebugDataID
```

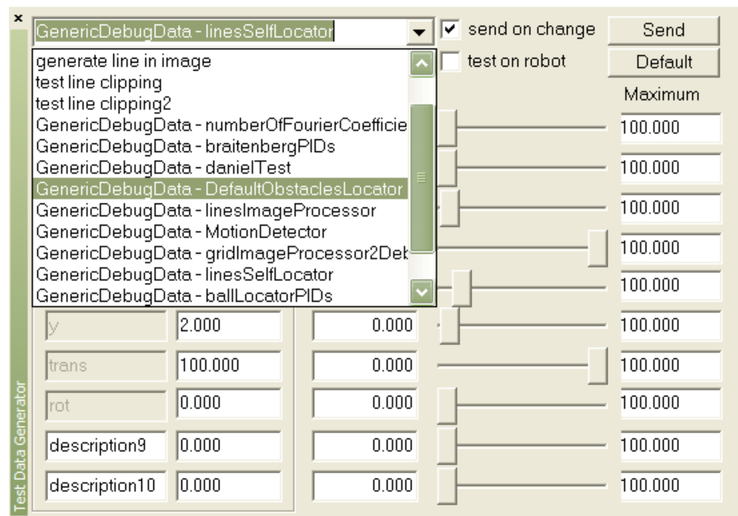

Figure E.1: The Test Data Generator Dialog for different types of generic debug data. For each of these, an array of 10 possible values is stored.

```
    {
      numberOfFourierCoefficients = 0,
      defaultObstaclesLocator,

      // insert your ID here!

      numOfGenericDebugDataIDs,
      unknown
    };

  ...
```

Further down in this include file, some additional information about the debug data can be provided which will be used by the RobotControl dialog. If this information is not provided, they will be given default names.

Message handling in a module looks like this:

```
  ...

  bool moduleName::handleMessage(InMessage& message)
  {
    bool handled = false;

    switch(message.getMessageID())
    {
    case idGenericDebugData:
```

```
    {
      GenericDebugData d;
      message.bin >> d;

      if(d.id == GenericDebugData::genericDebugDataIDforThisModule)
      {
        OUTPUT(idText,text,"generic debug message
           handled by module ModuleName");

        memberVariable = d.data[0];
        anotherMemberVariable = (int )d.data[1];
      }
      handled = true;
      break;
    }
  }
  return handled;
}
```

E.3 Debug Keys

The tools RobotControl and SimGT2003 (cf. Sect. 5) can process a wide range of messages from physical or simulated robots. As it is not possible to send all the messages at once, *debug keys* are used to toggle the output of these messages. They are also transmitted to the robot using message queues. In *GT2003\Src\Tools\Debugging\DebugKeyTable.h* a variety of keys are declared.

Each key can have one of following states:

Disabled. No output is sent.

Send always. The output is always sent.

Send n times. The output is sent a total of n times.

Send every n times. Every n-th output is actually being sent.

Send every n ms. The output is sent every n milliseconds.

The class *DebugKeyTable* has a member *isDebugKeyActive()* that determines whether the message shall be sent dependent on the state of a given key, e. g.:

```
if (myDebugKeyTable.isDebugKeyActive(DebugKeyTable::sendImages))
{
  myQueue.out.bin << image;
```

```
    myQueue.out.finishMessage(idImage);
  }
```

There are two modes associated with sending the output (queue fill requests):

Immediately. This will put the output into the queue. This makes sure that all requested outputs are being sent. (Caution: It can lead to the queue becoming too large.)

Real-Time. If an output of the same type is already in the queue and has not been sent, it will be removed from the queue and replaced by the new data. This is useful when the requested output is large and tends to fill up the queue quickly. It is particularly useful for sending images (or JPEG encoded images) from the robot to RobotControl.

E.4 Debug Macros

To simplify the access to outgoing message queues and to the appropriate debug key table, two macros are defined in *GT2003\Src\Tools\Debugging\Debugging.h*:

OUTPUT(type, format, data) stores *data* in a certain *format* and a certain message *type* in the outgoing queue of the process.

INFO(key, type, format, data) works similar to *OUTPUT*, but only, when the debug key *key* is active.

Example:

```
OUTPUT(idText,text,"Hello World");
OUTPUT(idText,text,"Found " << numberOfBalls << " balls.");
OUTPUT(idSensorData,bin,mySensorData);
INFO(sendImage,idImage,bin,myImage);
```

Both macros are ignored in *Release* configurations to save processing time.

E.5 Debug Drawings

At every location in the code a debug drawing can be drawn and sent. There exist two types of drawings: *imageDrawings* are in pixel coordinates and will be displayed in the image viewer (cf. Sect. J.3.1), whereas *fieldDrawings* are in the system of coordinates of the field and will be shown in the field view and the radar viewer (cf. Sect. J.3.2). Use the context menu to toggle the visibility of a given debug drawing in the respective dialog. Bear in mind that for the debug drawing to be displayed, the module that creates the drawing needs to be active (i. e. it needs to be selected in the settings dialog).

To generate a *DebugDrawing* the following has to be done:

- *Tools/Debugging/DebugDrawing.h* has to be included in the file from which it will be drawn.
- In *GT2003\Src\Tools\Debugging\DebugDrawing.h* a new drawingID has to be added to one of the enumeration types *FieldDrawing* or *ImageDrawing*. In the method *getDrawingName()*, a string representation for the new drawingID has to be given. In the method *getDebugKeyID()*, a debug key for requesting the drawing has to be added. This debug key has to be defined in *GT2003\Src\Tools\Debugging\DebugKeyTable.h*.
- In the file that should draw, the following has to be added, e. g. to create a drawing called "sketch":

```
...

// paint to the drawing
CIRCLE(sketch, x, y, radius, 3, 0, Drawings::orange);

...

// send the debug drawing
DEBUG_DRAWING_FINISHED(sketch);
```

A debug drawing is only painted if the corresponding request is sent. The requests are set automatically, if the drawing is selected in the context menu of the image viewer or the field view. It is possible to write to the same debug drawing from any number of modules. If you do so, make sure that the last module to write to the drawing also contains the finish-command.

E.6 Modules and Solutions

To obtain solutions for the modules that can be exchanged during runtime, there is a base class for each module, e. g. the class *ImageProcessor*. All solutions for this module (e. g. *LinesImageProcessor* and *GT2003ImageProcessor*) are derived from this base class. Each module has its own *ModuleSelector* class, e. g. the *ImageProcessorSelector*. An instance of the selector class is created in the process the module is part of. The selector class creates instances of all solutions of the module during construction, but executes only one of the solutions during runtime.

The data structure *SolutionRequest* stores what the current solution for each module is. This data structure can be modified on the PC and can be sent to the robot (cf. Sect. J.2.3). Each process contains a *SolutionHandler* which receives the solution requests. Each module selector class has a reference to the solution handler, and thus it can decide, which of the solutions has to be executed.

To add a new module, the base class, the selector class based on the template *TModuleSelector*, and the different classes for the solutions that derive from the base class have to be created.

In the selector class all solutions have to be added. In *GT2003\Src\Tools\SolutionRequest.h* the new module and the solutions have to be added to the enum data types and to the functions providing names for the modules and solutions. In one of the available processes, an instance of the selector class has to be instantiated, and its *execute()* method has to be called.

E.7 Stopwatch

To track down waste of time in the code, in *GT2003\Src\Tools\Stopwatch.h* two macros are defined:

STOP_TIME(expression) measures the system time before and after the execution of *expression* and outputs the difference as a text.

STOP_TIME_ON_REQUEST(eventID, expression). If the time keeping is requested for the *eventID*, the time is measured before and after the execution of *expression* and sent as an debug message with the id *idStopwatch*. With the time diagram dialog (cf. Sect. J.3.6) the requests can be generated and the results of the measurements are displayed.

Example:

```
STOP_TIME_ON_REQUEST(imageProcessor, pImageProcessor->execute(); );

STOP_TIME(
  for(int i = 0; i < 1000000; i++)
  {
    double x = sqrt(i);
    x *= x;
  }
);
```

Appendix F

XABSL Language Reference

The XABSL language is specified in XML Schema 1.0 [6].

Agents following the introduced layered state machine architecture (cf. section 3.8.1) can be completely described in that language.

The XABSL language design ensures:

- The interoperability with existing XML editors and tools.
- That no other compile or validation tools than standard XSLT / XML processors are needed.
- The scalability of agent behavior solutions. Agent behaviors are easy to extend.
- High validation and compile speed resulting in a short change-compile-test cycle.

F.1 Modularity

An XABSL agent behavior specification is distributed over many files. This helps to keep an overview over larger agents and to work in parallel.

Figure F.1 shows the different file types that are part of an XABSL agent behavior:

- Symbol files contain the definitions of symbols. These symbols are used in the options.
- Basic behavior files contain prototypes for basic behaviors and their parameters. They are referenced from states which have a subsequent basic behavior.
- Option files contain a single option.
- “options.xml” defines prototypes for each option and its parameters. These prototypes are needed to check in an option file, whether a referenced subsequent option exists.
- “agents.xml” includes all the option files. Agents and its root options can be defined here.

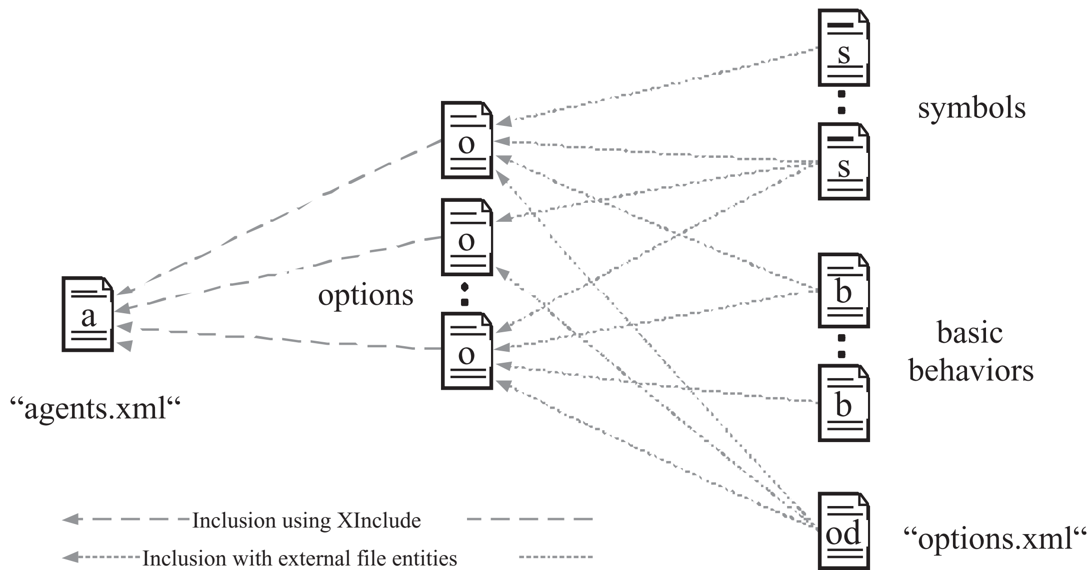

Figure F.1: Different file types of an XABSL specification

There are two include mechanisms (including one XML file into another) in use:

- *External file entities*: A code block, e. g. the file "my-symbols.xml" is defined as an external file entity inside a DTD. At the correct position in the code it is used with e. g. *&mySymbols;*. Most XML editors support this mechanism. It allows checking the validity of an option inside the XML editor.

  The disadvantage: No cascading inclusions possible.

- *XInclude* (http://www.w3.org/TR/xinclude/): A file is directly included into another one with a statement like this:

  ```
  <xinclude href="another-file.xml"/>
  ```

  An XInclude processor later resolves these includes for further processing.

  The disadvantage: Most XML editors don't resolve XInclude statements for validation.

As an agent can be distributed over many files, the schemas are also modularized:

xabsl-2.1.agent-collection.xsd	The root element of an XABSL behavior. Agent definitions.
xabsl-2.1.basic-behaviors.xsd	Prototypes for basic behaviors.
xabsl-2.1.expressions.xsd	Decimal and boolean expressions.
xabsl-2.1.option-definitions.xsd	Prototypes for options.
xabsl-2.1.option.xsd	Definition of an option.
xabsl-2.1.parameter.xsd	Parameters for options, basic behaviors and functions.
xabsl-2.1.symbols.xsd	Definition of symbols.
xinclude-1.0.xsd	A simplified scheme for the XInclude standard.
xmlbase.xsd	A simplified scheme for the XML base standard.

The exported XABSL namespace is:

```
http://www.ki.informatik.hu-berlin.de/XABSL2.1
```

F.2 Symbol Definitions

All symbols that are used inside options have to be defined before in separate symbol files. For example the file *"my-symbols.xml"* can look like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<symbols xmlns="http://www.ki.informatik.hu-berlin.de/XABSL2.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ki.informatik.hu-berlin.de/XABSL2.1
                      xabsl-2.1.symbols.xsd"
  id="my-symbols" title="My Symbols"
  description="My most used symbols">
    <boolean-input-symbol name="something-wrong"
                      description="a boolean symbol"/>
    <decimal-input-symbol name="foo"
                      description="a decimal symbol" measure="mm"/>
    <decimal-input-function name="abs"
                      description="the absolute value of a number"
                      measure="">
      <parameter name="abs.value" measure="" range="decimal"
        description="The value for that abs() is calculated"/>
    </decimal-input-function>
    <enumerated-input-symbol name="type-of-recognized-pet"
                  description="Which pet was seen by the robot">
        <enum-element name="dog"/>
        <enum-element name="cat"/>
        <enum-element name="guinea-pig"/>
```

```
    </enumerated-input-symbol>
    <enumerated-output-symbol name="op-mode"
             description="The mode how fast the robot shall act">
        <enum-element name="op-mode.slow"/>
        <enum-element name="op-mode.fast"/>
        <enum-element name="op-mode.very-fast"/>
    </enumerated-output-symbol>
    <constant name="pi" description="The value of pi"
              measure="rad" value="3.14"/>

    ...
</symbols>
```

Attributes of the element *symbols*:

- *"id"*: An id for the symbol collection. Must be identical to the file name without extension.
- *"title"*: A title needed for the documentation.
- *"description"*: A description needed for the documentation.

Figure F.2 shows the structure of the element *symbols*. There are 6 different symbol types allowed inside a symbols element:

boolean-input-symbol: A symbol for a Boolean variable or function.
Attributes:

- *"name"*: The name of the symbol.
- *"description"*: A description needed for the documentation.

decimal-input-symbol: A symbol for a decimal variable or function (the XabslEngine uses double).
Attributes:

- *"name"*: The name of the symbol.
- *"description"*: A description needed for the documentation.
- *"measure"*: The measure of the values. Needed for the documentation.

decimal-input-function: A prototype for a parameterized decimal function (cf. fig. F.3).
The parameters of the functions are defined in separate *parameter* child elements.
Attributes:

- *"name"*: The name of the function.
- *"description"*: A description needed for the documentation.
- *"measure"*: The measure of the values. Needed for the documentation.

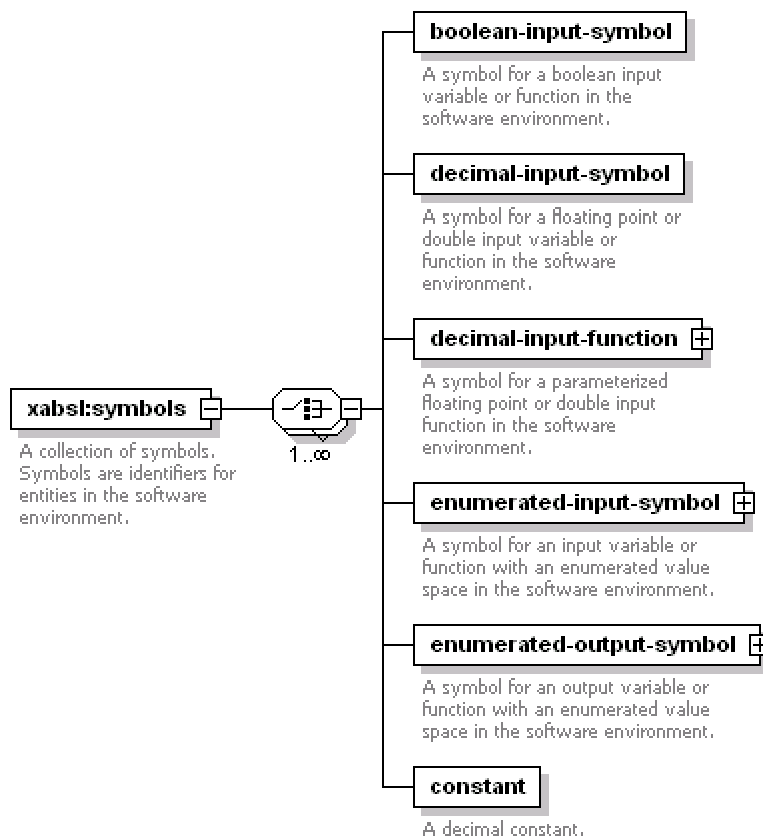

Figure F.2: The structure of the element *symbols*

enumerated-input-symbol: A symbol for an enumerated variable or function (cf. fig. F.4). Each enum element is defined in a single *enum-element* child element.
Attributes:

- *"name"*: The name of the symbol.
- *"description"*: A description needed for the documentation.

enumerated-output-symbol: Enumerated symbol can be set by states to influence the agent's software environment besides the execution of a basic behavior. As the *enumerated-input-symbol* it has *enum-element* child elements.
Attributes:

- *"name"*: The name of the symbol.
- *"description"*: A description needed for the documentation.

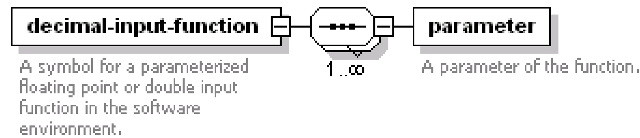

Figure F.3: The element *decimal-input-function*

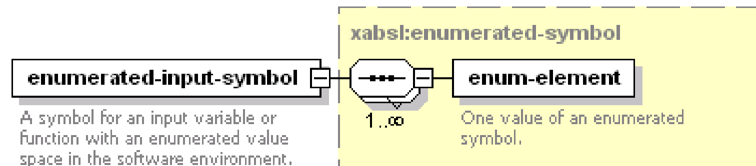

Figure F.4: The element *enumerated-input-symbol*

constant: Defines a decimal constant.
Attributes:

- *"name"*: The name of the constant.
- *"description"*: A description needed for the documentation.
- *"measure"*: The measure of the values. Needed for the documentation.
- *"value"*: The decimal value of the constant.

F.3 Basic Behavior Prototypes

For each basic behavior (they are written in C++), a prototype has to be declared. An example basic behavior file *"my-basic-behaviors.xml"* can look like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<basic-behaviors
  xmlns="http://www.ki.informatik.hu-berlin.de/XABSL2.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ki.informatik.hu-berlin.de/XABSL2.1
      xabsl-2.1.basic-behaviors.xsd"
  id="my-basic-behaviors" title="My Basic Behaviors"
  description="My common basic behaviors">
  <basic-behavior name="move-to"
                  description="Lets the agent move to a point">
    <parameter name="move-to.x" measure="mm" range="-1000..1000"
               description="X of destination position"/>
    <parameter name="move-to.y" measure="mm" range="-1000..1000"
```

```
                description="Y of destination position"/>
   </basic-behavior>
   <basic-behavior name="wait"
                   description="The agent performs no action"/>
 </basic-behaviors>
```

Attributes of the element *basic-behaviors*:

- *“id”*: An id for the basic behavior collection. Must be identical to the file name without extension.
- *“title”*: A title needed for the documentation.
- *“description”*: A description needed for the documentation.

The element *basic-behaviors* has to have at least on child element of the type *basic-behavior*. The element *basic-behavior* (cf. fig. F.5 defines a prototype for a basic behavior.

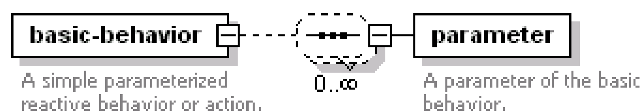

Figure F.5: The element *basic-behavior*

Attributes:

- *“name”*: The name of the basic behavior.
- *“description”*: A description needed for the documentation.

Optionally it has *parameter* child elements, which parameterize a basic behavior written in C++.
Attributes:

- *“name”*: The name of the parameter.
- *“description”*: A description needed for the documentation.
- *“measure”*: The measure of the values. Needed for the documentation.
- *“range”*: The range of possible values. Needed for the documentation.

F.4 Prototypes for Options

Every option can be encapsulated in an own file. To be able to validate a single option (e. g. the existence of a referenced subsequent option), there must be prototypes for all other options.

Therefore, in each XABSL agent behavior specification a file named “options.xml” has to exist. It should look like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<option-definitions
  xmlns="http://www.ki.informatik.hu-berlin.de/XABSL2.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ki.informatik.hu-berlin.de/XABSL2.1
                      xabsl-2.1.option-definitions.xsd">
  <option-definition name="nice-behavior"
      description="A nice behavior"/>
  <option-definition name="move-around"
      description="A behavior for randomly moving around">
    <parameter name="move-around.speed" measure="mm/s"
        range="0..500"
        description="The speed with that the robot shall move"/>
  </option-definition>
  ...
</option-definitions>
```

The element *option-definitions* (cf. fig. F.6) has no attributes. It has to have at least one *option-definition* child element.

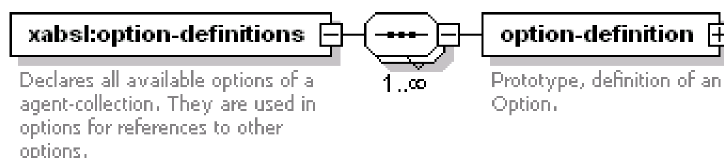

Figure F.6: The element *option-definitions*

The element *option-definition* defines a prototype for an option.
Attributes:

- *“name”*: The name of the option.
- *“description”*: A description needed for the documentation.

It can have *parameter* child elements, which allow it to parameterize an option.
Attributes:

- *“name”*: The name of the parameter.

- *“measure”*: The measure of the values. Needed for the documentation.
- *“range”*: The range of possible values. Needed for the documentation.

F.5 Options

Each option has to be defined in a separate file, e. g. *“Options/foo.xml”*:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE dummy-doc-type [
    <!ENTITY my-symbols SYSTEM "../my-symbols.xml">
    <!ENTITY my-basic-behaviors SYSTEM "../my-basic-behaviors.xml">
    <!ENTITY options SYSTEM "../options.xml">
]>
<option xmlns="http://www.ki.informatik.hu-berlin.de/XABSL2.1"
    xmlns:xi="http://www.w3.org/2001/XInclude"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ki.informatik.hu-berlin.de/XABSL2.1
                        xabsl-2.1.option.xsd"
    name="foo" initial-state="first-state">
    &my-symbols;
    &my-basic-behaviors;
    ...
    &options;
    <state name="first-state">
        ...
    </state>
    <state name="second-state">
        ...
    </state>
</option>
```

The element *option* (cf. fig. F.7) is the root element of an option file and has these attributes:

- *“name”*: The name of the option. Must be the file name without extension.
- *“initial-state”*: The name of the initial state. This state becomes activated if the option was not active during the last execution of the option graph.

First, the files for all referenced symbol definitions, all referenced basic behaviors, and the option definitions have to be included into the option. As shown in the example, this can be done by declaring all included files as external file entities in a dummy DTD at the top of the document. Inside the option element, these entities are used to include the files at the correct position.

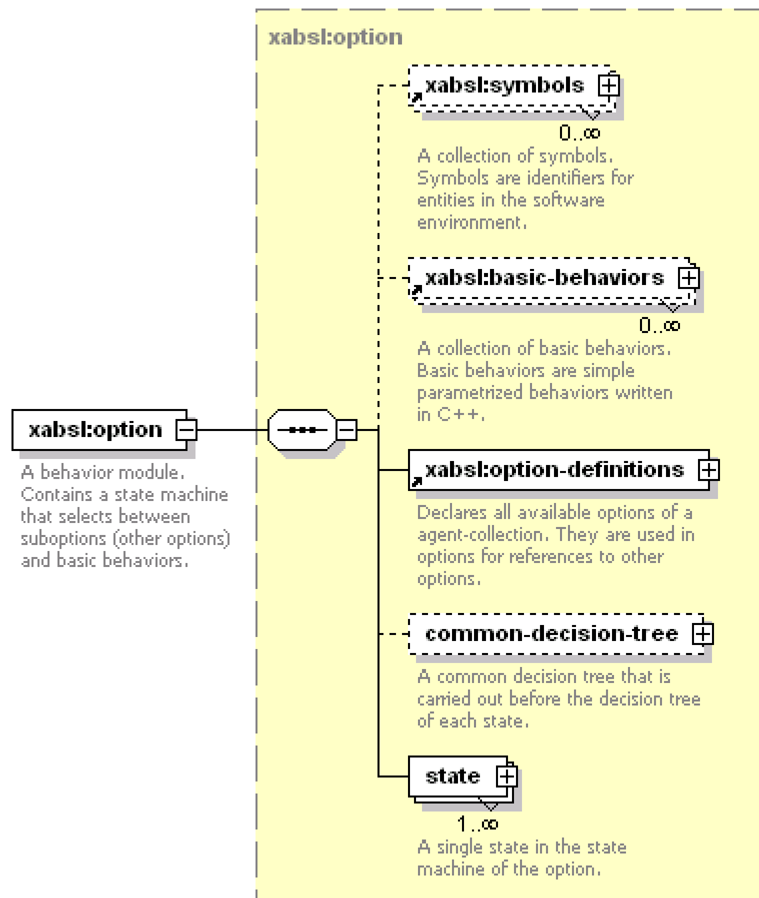

Figure F.7: The element *option*

After the included *symbols*, *basic-behaviors*, and *option-definitions* child elements there can follow a *common-decision-tree* child element (cf. fig. F.8).

If there are transitions with the same conditions in each state, these conditions can be put into this common decision tree. It is carried out before the decision tree of the active state. If no condition of the common decision tree evaluates true, the decision tree of the active state is carried out. That's also the reason why there is no *else* child element.

If the common decision tree uses expressions that are specific for a state (*time-of-state-activation*, *subsequent-option-reached-target-state*), these expressions refer to the state that is currently active.

The child elements *if* and *else-if* are the same as in the normal decision tree of a state, which is explained later in this chapter.

Besides and after the optional *common-decision-tree* each option has to have at least one state child element, which is described in the next section.

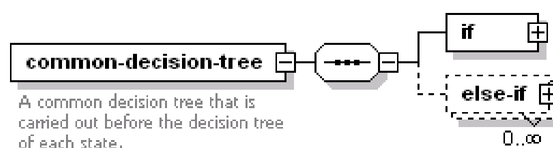

Figure F.8: The element *common-decision-tree*

F.6 States

The *state* element (cf. fig. F.9) represents a single state of an option's state machine:

```
<state name="first-state" is-target-state="true">
  <subsequent-basic-behavior ref="move-to">
    <set-parameter ref="move-to.x">
      <decimal-value value="42"/>
    </set-parameter>
  </subsequent-basic-behavior>
  <set-output-symbol ref="op-mode" value="op-mode.fast"/>
  <decision-tree>
    <if>
      <less-than>
        <decimal-input-symbol-ref ref="foo"/>
          <decimal-value value="14"/>
        <less-than>
      <transition-to-state ref="second-state"/>
    </if>
    <else>
      <transition-to-state ref="first-state"/>
    </else>
  </decision-tree>
</state>
```

Attributes:

- *"name"*: The name of the state.
- *"is-target-state"* (optional): If true, this state is marked as a "target state". In an option containing a state with this option as subsequent behavior, it can be queried if the subsequent option reached this marked target state.

Each *state* has to have either a *subsequent-option* or a *subsequent-basic-behavior* child element. This determines, which subsequent behavior becomes executed if this state is active.

Attributes for both:

- *"ref"*: The name of the referenced option or basic behavior.

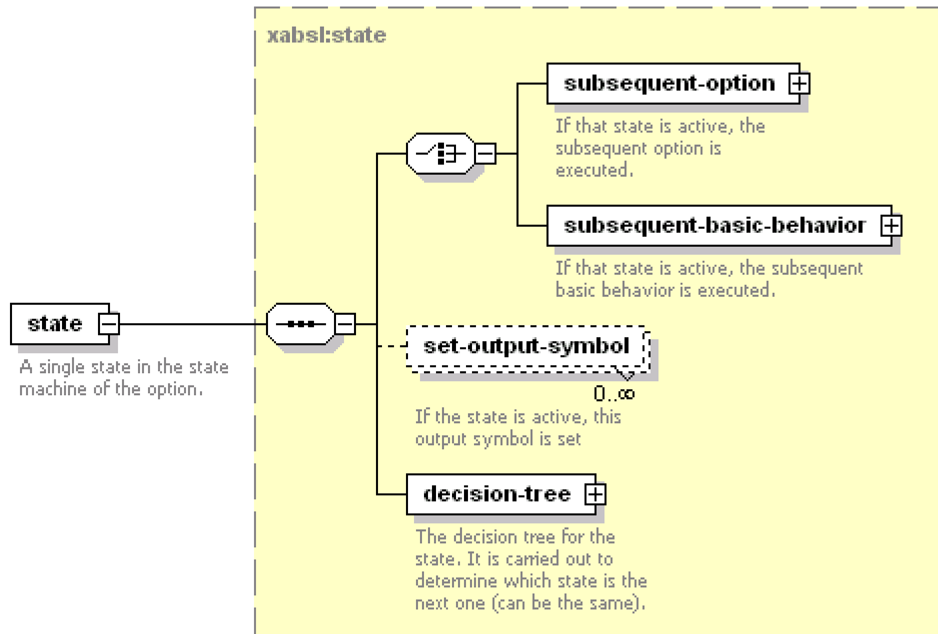

Figure F.9: The element *state*

If the referenced options or basic behaviors have parameters, these can be set with the *set-parameter* child elements (cf. fig. F.10).

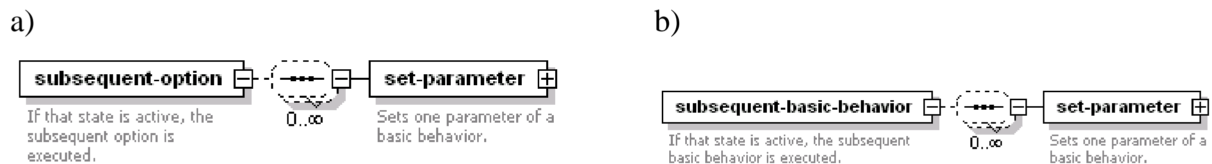

Figure F.10: a) The element *subsequent-option* b) The element *subsequent-basic-behavior*

Attributes:

- *"ref"*: The name of the referenced parameter of the subsequent behavior.

If the subsequent behavior has a parameter that is not set in the state referencing the parameter, the executing engine sets the parameter to zero.

The child element of the *set-parameter* element has to be a decimal expression. Decimal expressions are described later in this chapter.

After the definition of the subsequent behavior, output symbols can be set by inserting *set-output-symbol* child elements. The state which is active after the state machine of the option was carried out can set these symbols. It may happen that an option which becomes activated lower in the option graph overwrites an output symbol. The output symbols are applied to the software environment only when the option graph was executed completely.

Parameters:

- *"ref"*: The referenced output symbol.
- *"value"*: The value to be set. This must be one of the enum elements of the output symbol.

At last, each state has to contain a *decision-tree* child element, which is described in the next section.

F.7 Decision Trees

Each state has a decision tree. The task of this decision tree is to determine a transition to another state depending on the input symbols, which can be the same state. So the leaves of a decision tree are transitions.

The element *decision-tree* itself is a *statement* (cf. fig. F.11).

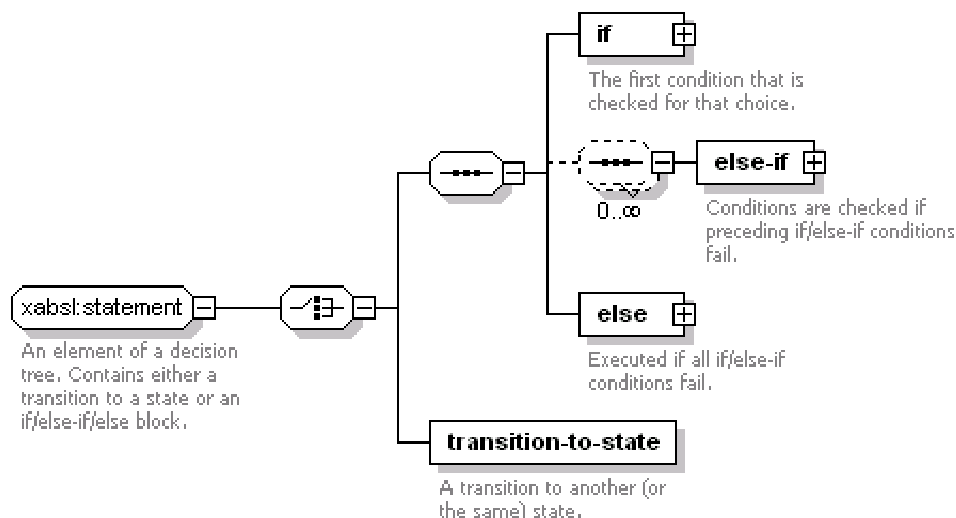

Figure F.11: The group *statement*

This element contains either an if/else-if/else block or a transition to a state.

An if/else-if/else block consists of an *if* element, optional *else-if* elements and an *else* element. The *if* and the *else-if* elements both have a *condition* child element and a statement which is executed if the condition is true. The statement itself is again either a if/else-if/else block or a transition to a state. This allows for complex nested expressions.

The *condition* element has a Boolean expression as a child element. This is explained in the next section.

Parameters:

- *"description"*: A description needed for the documentation.

The *transition-to-state* element represents a transition to another state.
It has these parameters:

- *“ref”*: The name of the referenced state.

F.8 Boolean Expressions

A *boolean-expression* can be one of these elements shown in figure F.12:

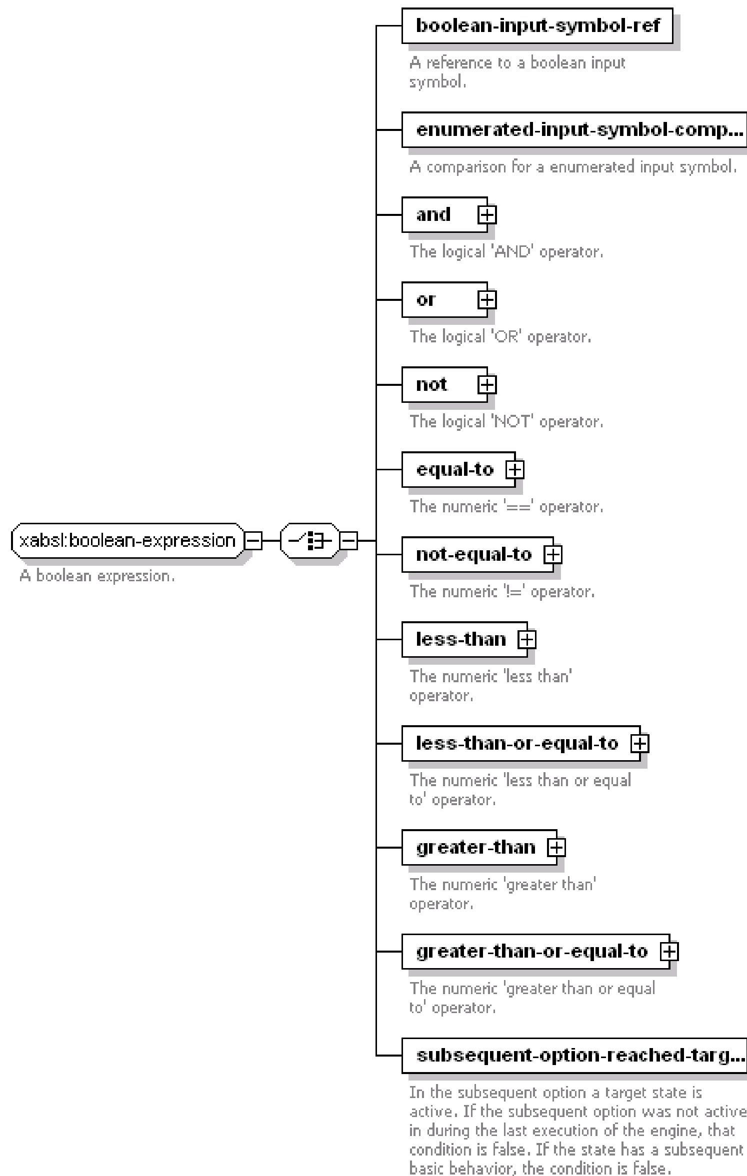

Figure F.12: The group *boolean-expression*

boolean-input-symbol-ref: A reference to a Boolean input symbol.
Parameters:

- *"ref"*: The name of the referenced symbol.

enumerated-input-symbol-comparison: Compares the value of an enumerated input symbol with a given enumerated value.
Parameters:

- *"ref"*: The name of the referenced symbol.
- *"value"*: The enum element for comparison.

and, or: The Boolean && and || operators. They have two *boolean-expression* child elements.

not: The Boolean ! operator. It has a *boolean-expression* child element.

equal-to, not-equal-to, less-than, less-than-or-equal-to, greater-than, greater-than-or-equal-to: The $==$, $!=$, $<$, $<=$, $>$ and $>=$ operator. They all have two *decimal-expression* child elements. These are described in the next section.

subsequent-option-reached-target-state: This statement becomes true, when

- the subsequent behavior of the state is an option,
- the active state of the subsequent option is marked as a target state.

Otherwise this statement is false.

F.9 Decimal Expressions

The *decimal-expression* group can be used inside some Boolean expressions and for the parameterization of subsequent behaviors. It can be one of the elements shown in figure F.13.

decimal-input-symbol-ref: A reference to a decimal input symbol. Parameters:

- *"ref"*: The name of the referenced symbol.

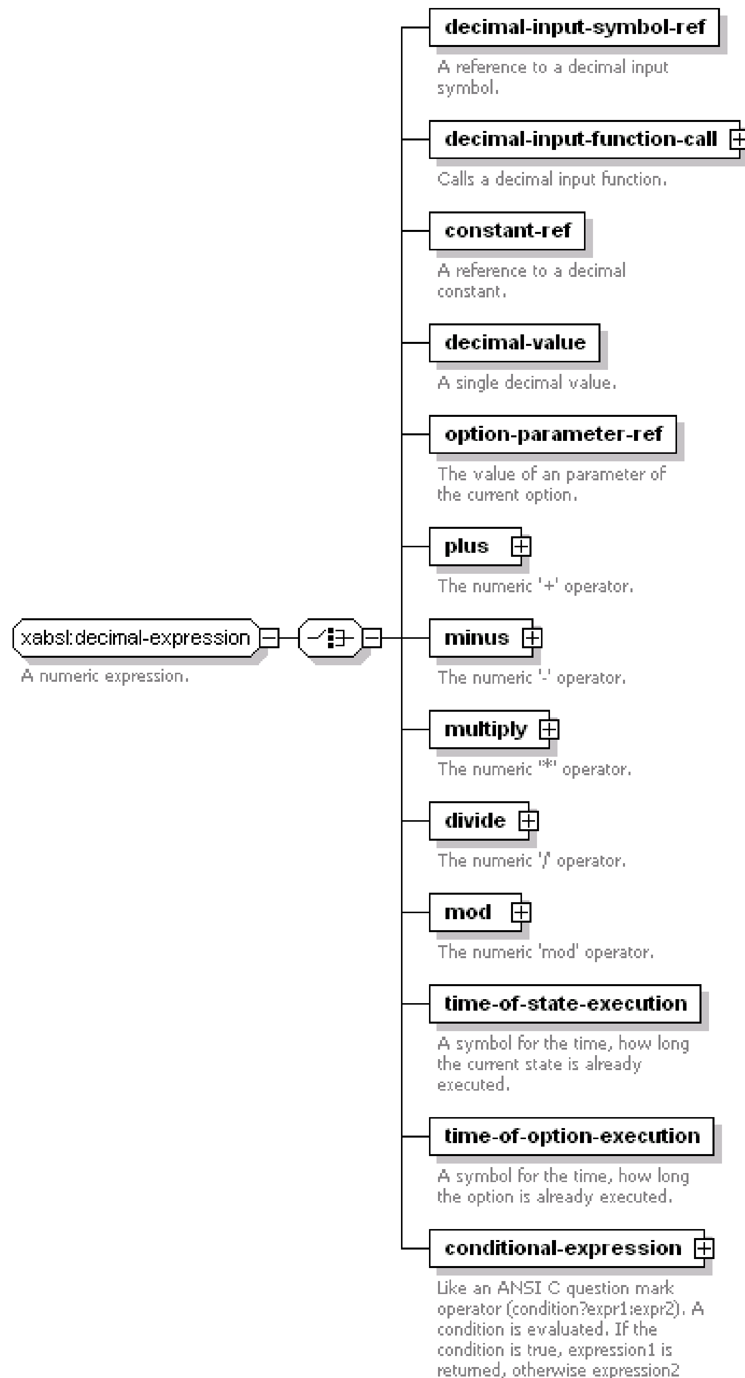

Figure F.13: The group *decimal-expression*

decimal-input-function-call: A call to a decimal input function. Parameters:

- *"ref"*: The name of the referenced function.

For each parameter of the function, a *with-parameter* element must be inserted (cf. fig. F.14). If a parameter is not set, the executing engine sets the parameter to 0. Parameters of the *with-parameter* element:

- *"ref"*: The name of the parameter of the referenced function.

The element *with-parameter* must have a *decimal-expression* child element.

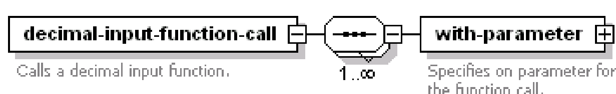

Figure F.14: The element *decimal-input-function-call*

constant-ref: A reference to a constant which was defined in a *symbols* collection. Parameters:

- *"ref"*: The referenced constant.

decimal-value: A decimal value, e. g. *"3.14"*. Parameters:

- *"value"*: The decimal value.

option-parameter-ref: References a parameter of the option. Parameters:

- *"ref"*: The referenced option parameter. Must be defined in the option definition for that option.

plus, minus, multiply, divide, mod: The arithmetic $+$, $-$, $*$, $/$ and $\%$ operators. They all have two *decimal-expression* child elements.

time-of-state-execution: The time, how long the state is already active. This time is reset when the state was not active during the last execution of the engine. Note that it may happen that the option activation path above the current option changes without this time being reset (it is only important that the option and the state were active during the last execution of the engine).

time-of-option-execution: The time, how long the option is already active. This time is reset if the option was not active during the last execution of the engine. Also here it may happen that the option activation path above the current option changes without this time being reset.

conditional-expression: This works such as an ANSI C question mark operator (cf. fig. F.15). A *condition* is checked. If the condition is true, the decimal expression *expression1* is returned, otherwise *expression2*.

The *condition* element contains a *boolean-expression* child element, the *expression1* and *expression2* elements contain *decimal-expression* child elements.

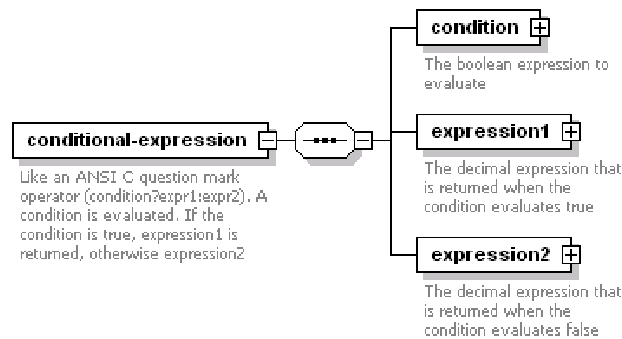

Figure F.15: The element *conditional-expression*

F.10 Agents

The file *"agents.xml"* is the root document of an XABSL behavior specification. It includes all the options and defines agents. An example *"agents.xml"* file may look like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE options [
<!ENTITY options SYSTEM "options.xml">
]>
<agent-collection
    xmlns="http://www.ki.informatik.hu-berlin.de/XABSL2.1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ki.informatik.hu-berlin.de/XABSL2.1
                        xabsl-2.1.agent-collection.xsd"
    xmlns:xi="http://www.w3.org/2001/XInclude">
  <title>My XABSL behavior application</title>
  <platform>My robot/agent platform.</platform>
  <software-environment>My software platform</software-environment>

  <agent id="default-agent" title="Default"
         description="The default agent behavior"
         root-option="foo"/>
  <agent id="test-behavior" title="Test"
         description="A test environment for the option bla"
         root-option="bla"/>
```

```
    ...
    &options;
    <options>
      <xi:include href="Options/foo.xml"/>
      <xi:include href="Options/bla.xml"/>
      ...
    </options>
  </agent-collection>
```

Figure F.16 shows the structure of the agent-collection element.

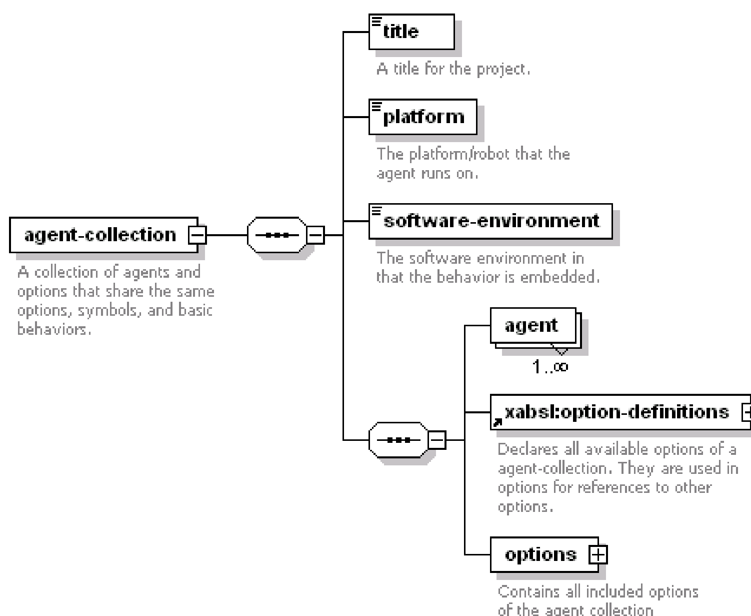

Figure F.16: The element *agent-collection*

The DTD at the top of the file declares the file *"options.xml"* as the external file entity *&options;*, which is later used to include the file at the correct position before the options element.

The *title*, *platform*, and *software-environment* elements are only used for generating the HTML documentation.

In an XABSL behavior, the option graph doesn't need to be completely connected. So it is difficult to determine a single root option of the graph. Instead a sub-graph that is spanned by an option and all it's subsequent options and basic behaviors can be declared as an agent. So an agent defines a starting point into the option graph.

There has to be at least one agent child element inside the *agent-collection* element. Attributes:

- *"id"*: The id of the agent. This id must be used to select that agent at the engine.

- *"title"*: A title needed for the documentation.

- *"description"*: A description needed for the documentation.
- *"root-option"*: The root option of the agent.

After the definition of the agents, the option prototypes are included. Although they are already included in the option files, they are included here again because they are needed for the validation of the agent elements.

At last, all options that are used by the agents and all options that are referenced from other options used have to be included inside the *options* element using XInclude.

Appendix G

XABSL Tools

From XABSL source documents, three types of documents can be generated:

- An *Intermediate Code* which is executed by the XabslEngine. Thus no XML parser is needed as on many embedded computing platforms XML parsers are not available due to resource and portability constraints.
- *Debug Symbols* containing the names of all options, states, basic behaviors, and symbols make it possible to implement platform and application dependent debugging tools for monitoring option and state activations as well as input and output symbols.
- An extensive HTML-documentation containing SVG-charts for each agent, option, and state which helps the developers to understand what their behaviors do.

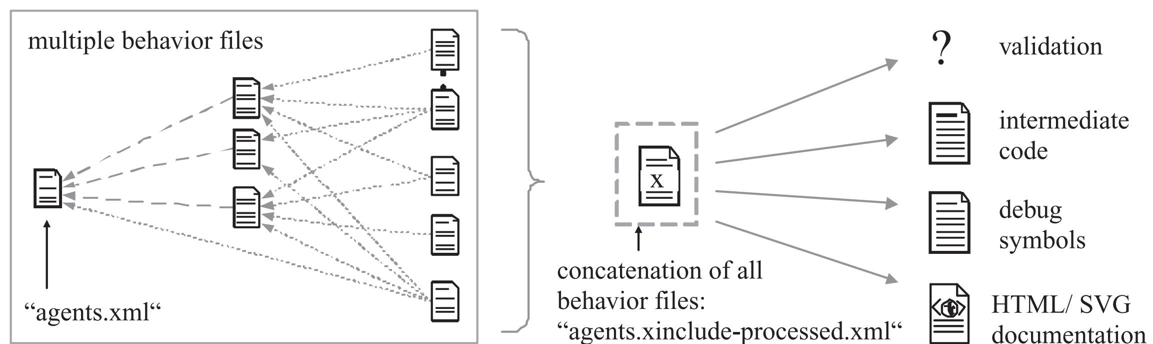

Figure G.1: Document generation in XABSL

Figure G.1 shows how these documents are generated. Because an XABSL agent behavior specification is distributed over many XML files, first all these files are concatenated into a single big file *"agents.xinclude-processed.xml"*. Then this file is validated against the XABSL schema. If that was successful, the intermediate code, the debug symbols, and the documentation are generated. The files are automatically generated using a Makefile, which is described in the next section.

G.1 Adopting the Makefile

As there are very many input files and XSLT style sheets, it is recommended to use a Makefile instead of executing the XSLT processor directly. There is a common XABSL Makefile. This file has to be included into a custom Makefile which should be located inside the directory of the source files. The custom Makefile must contain the following variables:

$XSLT$	An XSLT processor that can process XInclude statements (with necessary parameters). We recommend LibXSLT (http://xmlsoft.org/XSLT/).
DOT	Path to the dot tool (http://www.research.att.com/sw/tools/graphviz/). This is needed for the charts in the HTML documentation.
$DOTML_DIR$	Directory that contains the DotML Schemas and XSLT stylesheets (http://www.martin-loetzsch.de/DOTML/). DotML is needed to generate the charts in the HTML documentation.
$SCHEMA_VALIDATOR$	Validates an XML file against the schemas referenced in the source file taking the input from stdin. We recommend to use Xerces (http://xml.apache.org/xerces-c/index.html).
$XABSL_DIR$	Directory of the XABSL Schemas and XSLT Stylesheets relative to the source files and the custom Makefile.
$XABSL_OUTPUT_DIR$	Directory where the intermediate code and the debug symbols shall be generated.
DOC_OUTPUT_DIR	Directory for the documentation output.
$INSTANCE_DIR$	Directory containing the sources relative to the Xabsl2/xabsl-2.1 directory.
$DEBUG_SYMBOLS$	Path of the debug symbols to be generated.
$INTERMEDIATE_CODE$	The path of the intermediate code to be generated.
$SYMBOL_FILES$	All symbol files.
$BASIC_BEHAVIOR_FILES$	All basic behavior files.
$OPTION_FILES$	All option files. The have to be all in a directory "Options" inside the directory of *agents.xml*.

At last, the common Makefile must be included into the Makefile:

```
include $(XABSL_DIR)/Xabsl2Makefile
```

G.2 Using the Makefile

```
make all
```

generates *agents.xinclude-processed.xml* from *agents.xml* resolving all the XInclude statements. Then this resolved file is validated and the intermediate code and the debug symbols are generated. If the validation fails, the line number in the error message refers to *agents.xinclude-processed.xml*.

```
make DOC
```

generates the documentation. Note that single HTML pages can also be generated separately by typing

```
make file_path_and_name_of_the_HTML_page.html
```

Some shortcuts:

```
make VALID
```

Validation only.

```
make IC
```

Intermediate code only.

```
make DS
```

Debug symbols only.

```
make IC_DS
```

Intermediate code and debug symbols.

Appendix H

The Xabsl2Engine Class Library

The *Xabsl2Engine* is the XABSL runtime system. It is written in plain ANSI C++ and does not use any extensions such as the STL. It is platform and application independent and can be easily employed on any robotic platform. To run the engine in a specific software environment only two classes (for file access and error handling) have to be derived from abstract classes.

The engine parses and executes the intermediate code that was generated from XABSL documents. It links the symbols from the XML specification that are used in the options and states to the variables and functions of the agent platform. Therefore, for each symbol used an entity in the software environment has to be registered to the engine. While options and their states are represented in XML, basic behaviors are written in C++. They have to be derived from a common base class and registered to the engine. The engine provides extensive debugging interfaces for monitoring the option and state activations, the values of the symbols, and the parameters of options and basic behaviors. Instead of executing the engine from the root option, single options or basic behaviors can be tested separately.

This document is also available at [12].

H.1 Files of the Xabsl2Engine

```
Xabsl2Engine/Xabsl2Agent.cpp
Xabsl2Engine/Xabsl2Agent.h
Xabsl2Engine/Xabsl2Array.h
Xabsl2Engine/Xabsl2BasicBehavior.h
Xabsl2Engine/Xabsl2BooleanExpression.cpp
Xabsl2Engine/Xabsl2BooleanExpression.h
Xabsl2Engine/Xabsl2DecimalExpression.cpp
Xabsl2Engine/Xabsl2DecimalExpression.h
Xabsl2Engine/Xabsl2Engine.cpp
Xabsl2Engine/Xabsl2Engine.h
Xabsl2Engine/Xabsl2Option.cpp
Xabsl2Engine/Xabsl2Option.h
Xabsl2Engine/Xabsl2Symbols.cpp
```

```
Xabsl2Engine/Xabsl2Symbols.h
Xabsl2Engine/Xabsl2Tools.cpp
Xabsl2Engine/Xabsl2Tools.h
```

See the Doxygen-generated source code documentation of the Xabsl2Engine at [12] for more details.

H.2 Running the Xabsl2Engine on a Specific Target Platform

First, one has to declare a message and error handling class that is derived from *Xabsl2ErrorHandler*. This class has to implement the *printMessage()* and *printError()* function, e. g.:

```
class MyErrorHandler : public Xabsl2ErrorHandler
{
public:
  MyErrorHandler();

  virtual void printError(const char* text)
    { cout << "error: " << text << endl;}

  virtual void printMessage(const char* text)
    { cout << text << endl;}
};
```

The Boolean variable "*errorsOccurred*" can be used to determine if there occurred errors during the creation or execution of the engine.

Then, a class that gives the engine a read access to the intermediate code has to be derived from *Xabsl2InputSource*. These pure virtual functions have to be implemented:

- *open()*: opens the file containing the intermediate code. Note that the code doesn't need to be read from a file. It is also possible to read it from a memory buffer or any other stream.
- *close()*: is called by the engine after having read the data.
- *readValue()*: reads a numeric value from the file.
- *readString()*: reads a string from the file.

An example:

```
class MyFileInputSource : public Xabsl2InputSource
{
public:
  MyFileInputSource(const char* fileName) : file(0), theChar(' ')
```

```
     { strcpy(filename,fileName); }

    ~MyFileInputSource() {if (file!=0) delete file;}

    virtual bool open()
     {file = new std::ifstream(filename); return(file!=0);}

    virtual void close() {if (file!=0) delete file; file = 0;}

    virtual double readValue()
     { char buf[20]; readFromFile(buf); return atof(buf); }

    virtual bool readString(char* destination, int maxLength)
     { readFromFile(destination); return true; }

  private:
    char filename[200];
    std::ifstream* file;
    char theChar;


    void readFromFile(char* value)
    {
      while(!file->eof() && isWhitespace())
      {
        if (theChar == '/')
          while(!file->eof() && theChar != '\n')
            file->read(&theChar,1);
        file->read(&theChar,1);
      }

      while(!file->eof() && !isWhitespace())
       { *value++ = theChar;
         if(!file->eof()) file->read(&theChar,1); }
      *value = 0;
    }

    bool isWhitespace()
    { return theChar == ' ' || theChar == '/' || theChar == '\n'
                          || theChar == '\r' || theChar == '\t'; }
  };
```

Please note that the file contains comments (//...) that have to be skipped by the read functions:

```
// divide (7)
7
```

```
// multiply (6)
6
// decimal value (0): 52.5
0 52.5
// reference to decimal symbol (1) ball.y:
1 13
```

The comments have to be treated as in C++ files. (New line ends a comment.) In the example only "*7 6 0 52.5 1 13*" should be read from the file.

At last, a static function that returns the system time in milliseconds has to be defined, e.g.:

```
static unsigned long getSystemTime() {
  timeb sysTime;
  ftime(&sysTime);
  return (sysTime.time * 1000 + sysTime.millitm);
}
```

H.3 Creating a New Engine

First, an instance of the adapted *Xabsl2ErrorHandler* has to be created:

```
MyErrorHandler errorHandler;
```

Then, the engine can be created, passing a reference to the error handler and a pointer to the time function as parameters:

```
Xabsl2Engine* pMyEngine
  = new Xabsl2Engine(errorHandler,&getSystemTime);
```

Now all the symbols and basic behaviors have to be registered to the engine. Note that this has to be done before the option graph is created.

H.4 Registering Symbols

As the behaviors written in XABSL use symbols to interact with the agent's software environment, for each of these symbols the corresponding variable or function have to be registered to the engine. The following example registers the variable *aDoubleVariable* to the symbol "*a-decimal-symbol*" which was defined in the XABSL agent:

```
pMyEngine->registerDecimalInputSymbol("a-decimal-symbol",
                                      &aDoubleVariable);
```

If the value for the symbol is not represented by a variable but by a function, this function has to be registered to the engine. Note that this function has do be defined inside a class which is derived from *Xabsl2FunctionProvider*:

```
class MySymbols : public Xabsl2FunctionProvider
{
public:
  double doubleReturningFunction() { return 3.7; }
};

...

MySymbols mySymbols;

pMyEngine->registerDecimalInputSymbol("a-decimal-symbol",
  &mySymbols, (double (Xabsl2FunctionProvider::*)())
  &MySymbols::doubleReturningFunction);
```

The registration of boolean symbols works in a similar way:

```
pMyEngine->registerBooleanInputSymbol("a-boolean-symbol",
  &aBooleanVariable);
```

Or:

```
class MySymbols : public Xabsl2FunctionProvider
{
public:
  bool booleanReturningFunction() { return false; }
};

...

MySymbols mySymbols;

pMyEngine->registerBooleanInputSymbol("a-boolean-symbol",
  &mySymbols, (bool (Xabsl2FunctionProvider::*)())
  &MySymbols::booleanReturningFunction);
```

Enumerated input symbols have to be registered that way:

```
class MySymbols : public Xabsl2FunctionProvider
{
public:
  enum MyEnum { element1, element2, element3 } anEnumVariable;
  MyEnum enumReturningFunction() { return MySymbols::element3; }
```

```
};

...

MySymbols mySymbols;

pMyEngine->registerEnumeratedInputSymbol("an-enumerated-symbol",
  (int*)&mySymbols.anEnumVariable);
```

Or:

```
pMyEngine->registerEnumeratedInputSymbol("an-enumerated-symbol",
  &mySymbols, (int (Xabsl2FunctionProvider::*)())
  &MySymbols::enumReturningFunction);
```

After that, for each enum element that was defined in the XABSL agent the corresponding value has to be registered:

```
pMyEngine->registerEnumeratedInputSymbolEnumElement(
  "an-enumerated-symbol","element1",MySymbols::element1);
pMyEngine->registerEnumeratedInputSymbolEnumElement(
  "an-enumerated-symbol","element3",MySymbols::element3);
```

The registration of enumerated output symbols works in a very similar way using these functions:

```
void registerEnumeratedOutputSymbol(char *name, int *pVariable);

void registerEnumeratedOutputSymbol(char *name,
  Xabsl2FunctionProvider *pInstance,
  void(Xabsl2FunctionProvider::*pFunction)(int));

void registerEnumeratedOutputSymbolEnumElement(
  const char *symbolName, const char *name, int value);
```

At last, decimal input functions have to be registered similar to decimal input symbols. In addition, for each parameter of the function a variable has to be declared and registered:

```
class MySymbols : public Xabsl2FunctionProvider
{
public:
  double parameter1, parameter2;
  double myFunction() { return (parameter1 + parameter2) / 2; }
};

...
```

```
MySymbols mySymbols;

pMyEngine->registerDecimalInputFunction(
  "a-decimal-input-function", &mySymbols,
  (double (Xabsl2FunctionProvider::*)())&MySymbols::myFunction);
pMyEngine->registerDecimalInputFunctionParameter(
  "a-decimal-input-function","parameter1",mySymbols.parameter1);
pMyEngine->registerDecimalInputFunctionParameter(
  "a-decimal-input-function","parameter2",mySymbols.parameter2);
```

H.5 Registering Basic Behaviors

All basic behaviors have to be derived from the class *Xabsl2BasicBehavior* and have to implement the pure virtual function *execute()*. The name of the basic behavior has to be passed to the constructor of the base class. The parameters of the basic behavior have to be declared as members of the class and registered using *registerParameter(..)*:

```
class MyBasicBehavior : public Xabsl2BasicBehavior
{
public:
  double parameter1, parameter2;

  MyBasicBehavior(Xabsl2ErrorHandler& errorHandler)
      : Xabsl2BasicBehavior("a-basic-behavior",errorHandler)
  {
    registerParameter("parameter1",parameter1);
    registerParameter("parameter2",parameter2);
  }

  virtual void execute()
  {
    // do the requested action using parameter1 and parameter2
  }
};
```

Then, for each basic behavior class an instance has to be registered to the engine:

```
MyBasicBehavior myBasicBehavior(errorHandler);

pMyEngine->registerBasicBehavior(myBasicBehavior);
```

H.6 Creating the Option Graph

After the registration of all symbols and basic behaviors, the intermediate code can be parsed:

```
MyFileInputSource input("path_to_the_intermediate_code.dat");

pMyEngine->createOptionGraph(input);
```

If the engine detects an error during the execution of the option graph, the error handler is invoked. This can happen when the intermediate code contains a symbol or a basic behavior that was not registered before. Whether the option graph was created successfully or not can be checked like this:

```
if (errorHandler.errorsOccurred)
{
  // do some backup behavior
  delete pMyEngine;
}
```

H.7 Executing the Engine

If no errors occurred during the creation, the engine can be executed this way:

```
pMyEngine->execute();
```

This function executes the option graph only a single time. Starting from the selected root option, the state machine of each option is carried out to determine the next active state. Then for the subsequent option of this state the state machine becomes carried out and so on until the subsequent behavior is a basic behavior, which is executed then, too. After that the output symbols that were set during the execution of the option graph become applied to the software environment.

In the *execute()* function the execution starts from the selected root option, which is in the beginning the root option of the first agent. The agent can be switched using this function:

```
pMyEngine->setSelectedAgent("name-of-the-agent");
```

H.8 Debugging Interfaces

Instead of executing the option graph with *execute()*, the function *executeSelectedBasicBehavior()* can be called. This is useful to test a single basic behavior. Before that, with

```
pMyEngine->setSelectedBasicBehavior("name-of-the-basic-behavior");
```

a basic behavior must be selected for execution. With

```
pMyEngine->setBasicBehaviorParameter("name-of-the-basic-behavior",
  "name-of-the-parameter",42);
```

the parameters can be set.
For testing single options, the root option can be changed and parameterized:

```
pMyEngine->setRootOption("another-option");

pMyEngine->setOptionParameter("another-option",
  "parameter-name",23);
```

There is a number of functions to trace the current state of the option graph, the option activation path, the option parameters, and the selected basic behavior:

```
const Xabsl2Option* getRootOption () const;

const char* getSelectedAgentName ();

const Xabsl2BasicBehavior* getSelectedBasicBehavior ();
```

The member functions of the *Xabsl2Option* and *Xabsl2BasicBehavior* objects returned can be used to retrieve this information.
For tracing the values of symbols, the engine provides access to the symbols stored:

```
Xabsl2DecimalInputSymbol* getDecimalInputSymbol(
  const char *name);

Xabsl2BooleanInputSymbol* getBooleanInputSymbol(
  const char *name);

Xabsl2EnumeratedInputSymbol* getEnumeratedInputSymbol(
  const char *name);

Xabsl2EnumeratedOutputSymbol* getEnumeratedOutputSymbol(
  const char *name);
```

Note that these functions crash if the symbol requested does not exist. The existence of symbols can be checked using these methods:

```
bool existsDecimalInputSymbol(const char *name);
bool existsBooleanInputSymbol(const char *name);
```

```
bool existsEnumeratedInputSymbol(const char *name);
bool existsEnumeratedOutputSymbol(const char *name);
```

Enumerated output symbols can also be set manually for testing purposes. Note that this has to be done after the option graph was executed. The changes are committed to the software environment using this function:

```
pMyEngine->setOutputSymbols();
```

Appendix I

SimGT2003 Usage

I.1 Introduction

SimGT2003 is based on SimRobot [1], a kinematic robotics simulator. In fact, only a so-called controller has been added to SimRobot that provides the same environment to robot control code that it will also find on the real robots. Therefore, SimGT2003 shares the user interface with SimRobot. This user interface is documented in the online help file that comes with SimRobot. In addition, the scene description language that is used to model the simulation scenes is explained in the help file. Hence, these descriptions are not repeated here.

SimGT2003 is the second Windows tool of the GermanTeam besides RobotControl. While RobotControl focuses on *interaction*, SimGT2003 has its strength in *automation*. The main input channel of SimGT2003 is a console window that is much harder to use than the mouse-enabled interface of RobotControl, but the text based approach to command input also provides the possibility to use script files, which is the key feature to automate a lot of processes. Therefore, SimGT2003 can speed up the development, because—once configured—no further user intervention is required after the start of the program. Therefore, there is no waste of time for opening log files, setting debug keys, switching solutions, and connecting to robots. Besides, SimGT2003 still seems to be more stable than RobotControl, mainly because of its simpler concept. Each process layout has its own set of views, and message handling is not dependent on Windows idle time. The approach requires a lot less synchronization, which also makes SimGT2003 faster than RobotControl. On the other hand, there are a lot of things that cannot be done with SimGT2003, e. g. creating color tables, setting camera parameters, all kinds of OpenGL visualizations, etc. And, in fact, if very different tasks have to be performed in a row as, e. g., during a contest, the mouse-enabled interface of RobotControl is much more comfortable.

I.2 Getting Started

SimGT2003 can either be started directly from the Windows Explorer (from *GT2003\Bin*), from Microsoft Developer Studio, or by starting a scene description file[1]. In the first case, a scene description file has to be opened manually, whereas it will already be loaded in the latter two cases. When a simulation is started for the first time and no layout has been patched into the Windows registry, only the editor window will show up in the main window. Select *Simulation|Start* to run the simulation. The *Tree View* will appear. A *Scene View* showing the soccer field can be opened by double-clicking *WORLD GT2003*. However, the scale of the display will not be appropriate. After selecting *View|Zoom|4x* and *View|Perspective Distortion|Level 1* the field will fit into the window. In addition, the presentation can be simplified by reducing the *View|Detail Level*. Please note that there also exist keyboard shortcuts and toolbar buttons for most commands.

After starting a simulation, a script file may automatically be executed, setting up the robots as desired. The name of the script file is part of the scene description file. Together with the ability of SimRobot to store the window layout, the software can be configured to always start with a setup suitable for a certain task.

Although any object in the *Tree View* can be opened, only displaying certain entries in the object tree makes sense, namely the *WORLD*, the objects in the group *robots*, and all *VALUE*s of objects starting with *VIEW*.

I.3 Views

I.3.1 Scene View

The *Scene View* appears if the *WORLD* is opened from the *Tree View*. As stated above, a 4x-zoom and a level 1 perspective distortion are ideal for displaying the field. The view can be rotated around two axes, and it supports several mouse operations:

- If a robot is clicked between its forelegs (on the field plane), it can be dragged to another position.

- If a robot is clicked between its hind legs, it can be rotated around its body center, i. e. the middle between its forelegs.

- If an active robot (see below) is double-clicked, it is the currently selected robot, i. e. the robot console commands are sent to.

- The ball can be dragged around. Note that its *click position* is on the field plane.

[1]This will only work if SimGT2003 was started at least once before.

I.3.2 Robot View

A *Scene View* containing a single robot can be opened by double-clicking a *VEHICLE* in the sub-tree *robots* of the *Tree View*. In such a view, the robot is displayed centered, and it can be zoomed to fill the entire window. This allows seeing more details of the robot, e. g. the state of its LEDs. The view supports four different mouse actions:

- If the back of the robot is clicked, this simulates an activated back switch of the robot. A second click will deactivate the switch.
- If the back area of the top surface of the head is clicked, this simulates an activated back head switch. Again, a second click will deactivate the switch.
- If the front area of the top surface of the head is clicked, this simulates an activated front head switch. It is deactivated by a second click.
- A double-click in the window throws the robot on its side. Note that this can only be seen in the global *Scene View*. In the robot view, the fact that the robot felt down is visualized by hiding its tricot. A second double-click will bring the robot back on its feet.

I.3.3 Information Views

In SimGT2003, *information views* are used to display debug drawings. These are generated by the robot control program, and they are sent to SimGT2003 via *message queues*. In SimGT2003, the views are defined in the source code. They are instantiated separately for each robot. All views in the current code are defined in *GT2003\Src\Platform\Win32\SimRobot\RobotConsole.cpp*. There are three kinds of views related to information received from robots: *image views*, *field views*, and *Xabsl2 views*. Field and image views display debug drawings received from the robot, whereas the Xabsl2 views print text information sent by the Xabsl2 behavior control on the robot.

I.3.3.1 Image Views

An image view displays information in the system of coordinates of a camera image. It is defined by giving it a name and by listing the debug drawings that will be part of the view. The identifiers of all debug drawings are defined in class *Drawings*. Two special elements can be part of an image view that are not debug drawings: *image* and *colorClassImage*. They either show the camera image or the segmented camera image, respectively, and must be the first entries in the list, because they will occlude anything drawn before.

Note that only information can be drawn that is actually sent by the robot, i. e. the corresponding debug requests must have been set. To receive images, either the debug keys *sendImage* or *sendJPEGImage* must have been activated. To display a certain debug drawing *XYZ*, the debug key *send_XYZ_drawing* must be set.

For instance, the view *image* is defined as:

```
IMAGE_VIEW(image)
  Drawings::image,
  Drawings::imageProcessor_horizon,
  Drawings::imageProcessor_scanLines,
  Drawings::perceptCollection,
  Drawings::selfLocator
END_VIEW(image)
```

To display all this information, console commands (cf. Sect. I.5) such as the following are also required:

```
dk sendJPEGImage every 100 ms
dk sendPercepts every 100 ms
dk send_imageProcessor_horizon_drawing every 100 ms
dk_send_imageProcessor_scanLines_drawing every 100 ms
dk send_selfLocator_drawing every 100 ms
```

I.3.3.2 Field Views

A field view displays information in the system of coordinates of the soccer field. It is defined similar to image views. Two special elements can be part of a field view that are not debug drawings: *fieldPolygons* and *fieldLines*. The field polygons are green, sky-blue and yellow areas visualizing the field and goal areas. The field lines are the field boundary and all lines. If used, the field polygons must be the first entry in the list of drawings, because they will occlude anything drawn before.

For instance, the view *worldState* is defined as:

```
FIELD_VIEW(worldState)
  Drawings::fieldPolygons,
  Drawings::fieldLines,
  Drawings::selfLocatorField,
  Drawings::worldState,
  Drawings::percepts_ballFlagsGoalsField
END_VIEW(worldState)
```

To display all this information, console commands (cf. Sect. I.5) such as the following are also required:

```
dk send_selfLocatorField_drawing every 500 ms
dk sendPercepts on
dk sendWorldState on
```

Please note that the Monte-Carlo drawing is sent less often, because it is pretty large.

I.3.3.3 Xabsl2 Views

A single Xabsl2 view is part of each set of views. The information displayed is configured by the console commands *xis* and *xos* (cf. Sect. I.5.3). In addition, the debug key *sendXabsl2DebugMessages* must have been set, and a Xabsl2 behavior that matches the one loaded by the console command *xlb* (cf. Sect. I.5.3) must be active on the robot.

```
# set behavior control solution to GermanTeam 2003
sr BehaviorControl GT2003-soccer

# load behavior of the GermanTeam 2003
xlb gt03

# request Xabsl2 debug messages
dk sendXabsl2DebugMessages on

# show some symbols
xis ball.seen.distance on
xis ball.time-since-last-seen on
xos head-control-mode on

# set output symbol
xos head-control-mode head-control-mode.search-for-ball
```

I.4 Scene Description Files

The language of scene description files is documented in the online help file of SimRobot. However, there are some facts that are special in SimGT2003:

- At the top of a scene description, just below *WORLD*, the instruction *REMARK* can be used to specify the name of the script that will be executed when the simulator is started. A script file contains commands as specified below, one command per line. The default location for scripts is *GT2003\Config\Scenes*, their default extension is *.con*. If no file is specified, SimGT2003 will use *GT2003\Config\Scenes\console.con* if it exists.

- Near the end of a scene description file, there is a group called *robots*. It contains all *active* robots, i. e. robots for which processes will be created.

- Below the group *robots*, there is the group *extras*. It contains *passive* robots, i. e. robots which just stand around, but which are not controlled by a program. Passive robots can be activated by moving their definition to the group *robots*.

- Below that, there is the group *balls*. It contains the balls, i. e. normally a single ball, but it can also contain more of them if necessary, e. g. for the ball challenge in 2002.

A lot of scene description files can be found in *GT2003\Config\Scenes*. Please note that there are two types of scene description files: the ones required to simulate one or more robots (about 20 KB in size), and the ones that are sufficient to connect to a physical robot or to replay a log file (about 1 KB in size).

I.5 Console Commands

Console commands can either be directly typed into the console window or they can be executed from a script file. There exist three different kinds of commands. First, two commands can only be used in a script file that is executed when the simulation is started. Second, *global commands* change the state of the whole simulation, or they are always sent to all robots. Third, *robot commands* only have an impact on the set of currently *selected robots*.

I.5.1 Initialization Commands

sc [**gameManager**] **(** <**a.b.c.d**> | <**a.b.c**> <**d**> {<**d**>} **).** Starts a wireless connection to real robots. The syntax is very similar to the one of the start-script of the router). The command will start the router in the background and will display its messages in the console window. It should only be used once. It will add new robots to the list of available robots (named by the least significant byte of their IP-addresses), and for each of these robots, a full set of views is added to the *Tree View*. Please note that physical robots only send debug drawings on demand, so the views will remain empty until the drawings are requested by the appropriate debug keys. When the simulation is reset or SimGT2003 is exited, the router will be terminated.

sl <**name**> <**file**>**.** Replays a log file. The command will instantiate a complete set of processes and views. The processes will be fed with the content of the log file. The first parameter of the command defines the name of the virtual robot. This name can be used in the *robot* command (see below), and all views of this particular virtual robot will be identified by this name in the *Tree View*. The second parameter specifies the name and path of the log file. If no path is given, *GT2003\Config\Logs* is used as default. Otherwise, the full path is used. *.log* is the default extension of log files. It will be automatically added if no extension is given.

Please note that the backslash character has to be doubled to be recognized by the system, e. g. write *sl AIBO1 c:\\logs\\hallo* to load the log file *c:\logs\hallo.log*.

When replaying a log file, the replay can only be stopped by halting the simulation, i. e. by pressing the *start/stop* button. To avoid the loss of log data during the replay, select the *simulation time mode*, i. e. execute the command *st on* (see below).

I.5.2 Global Commands

call <file>. Executes a script file. A script file contains commands as specified here, one command per line. The default location for scripts is *GT2003\Config\Scenes*, their default extension is *.con*.

cls. Clears the console window.

echo <text>. Print text into the console window. The command is useful in script files to print commands that can later be activated manually by pressing the *enter* key.

gc reset | ready | playing | final | kickOff (blue | red) [<blueScore> <redScore>]. Game control. The command is sent to all robots. The *kickOff*-command is interpreted according to the team color of each robot. *gc reset* resets the score counters.

help | ?. Displays a help text.

jbc <button> <command>. Sets a joystick button command. The first parameter specifies the joystick button by its number between 1 and 32. Any text after this first parameter is part of the second parameter. The second parameter can contain any legal script command. The command will be executed when the corresponding button is pressed. While a joystick button is pressed, no changes in the walking direction of the robot will be accepted. A typical command to be assigned to a button is the executing of a special action, e. g. *jbc 1 mr unswBashOptimized* will try to kick the ball when button 1 is pressed.

jhc tilt | pan | roll. Set head axis to be controlled by the accelerator lever of the joystick. The other two axes will be controlled by the coolie head. By default, the pan axis is controlled by the accelerator lever.

robot ? | all | <name> {<name>}. Connects the console window to a set of *selected robots*. All commands in next section are only sent to the selected robots. The command *robot ?* displays a list of all robot names. To select a single simulated robot, it can also be double-clicked in the *Scene View*. To select them all, type *robot all*.

st off | on. Switches the simulation of time on or off. Without the simulation of time, all calls to *SystemCall::getCurrentSystemTime()* will return the real time of the Windows PC. However, as the simulator runs slower than real-time, the simulated robots will receive less sensor readings than the real ones. If the simulation of time is switched on, each step of the simulator will advance the time by 8 ms. Thus, *SimGT2003* simulates real-time, but it is running slower. By default this option is switched off.

 # <text>. Comment. Useful in script files.

I.5.3 Robot Commands

ci off | **on.** Switches the calculation of images on or off. The simulation of the robot's camera image costs a lot of time, especially if multiple robots are simulated. In some development situations, it is a better solution to switch off all low level processing of the robots and to work with *oracled world states*, i. e. world states that are directly delivered by the simulator. In such a case there is no need to waste processing power by calculating camera images. Therefore, it can be switched off. However, by default this option is switched on. Note that this command only has an effect on simulated robots.

dk ? | **(** <**key**> **off** | **on** | <**number**> | **every** <**number**> [**ms**] **).** Sets a debug key. The GermanTeam uses so-called debug keys to switch several options on or off at runtime. Type *dk ?* to get a list of all available debug keys. Debug keys can be activated permanently, for a certain number of times, or with a certain frequency, either on a counter basis or on time. All debug keys are switched off by default.

hcm ? | <**mode**>**.** Sets the head control mode. Type *hcm ?* to get a list of all available head control modes.

hmr <**tilt**> <**pan**> <**roll**> <**mouth**>**.** Sends a head motion request, i. e. it sets the joint angles of the three axes of the head and the opening angle of the mouth. This will only work if the actual head control mode is *none*. The angles have to be specified in degrees.

log start | **stop** | **clear** | **save** <**file**>**.** Records a log file. *log start* starts or continues recording all data received from the robot. *log stop* stops the recording. *log clear* removes all recorded data from memory. *log save* stores the data recorded to the log file with the name specified. If the file already exists, it will be replaced. If no path is given, *GT2003\Config\Logs* is used as default. Otherwise, the full path is used. *.log* is the default extension of log files. It will be automatically added if no extension is given.

mr ? | <**type**> [<**x**> <**y**> <**r**>]**.** Sends a motion request. This will only work if no *behavior control* is active. Type *mr ?* to get a list of all available motion requests. Walk motions also have to be parameterized by the motion speeds in forward/backward, left/right, and clockwise/counterclockwise directions. Translational speeds are specified in millimeters per second; the rotational speed has to be given in degrees per second.

msg off | **on.** Switches the output of text messages on or off. All processes can send text messages via their debug queues to the console window. As this can disturb entering text into the console window, it can be switched off. However, by default text messages are printed.

pr continue | **illegalDefender** | **obstruction** | **keeperCharged** | **ballHolding.** Penalize robot. The command sends one of the four penalties to all selected robots, or it signals them to continue with the game after a penalty.

qfr queue | **replace** | **reject** | **collect** <**seconds**> | **save** <**seconds**>**.** Send queue fill request. This request defines the mode how the message queue from the debug process to the PC is handled.

queue is the default mode. It will insert all messages received by the debug process from other processes into the queue, and send it as soon as possible to the PC. If more messages are received than can be sent to the PC, the queue will overflow[2].

replace. If the mode is set to *replace*, only the newest message of each type is preserved in the queue[3]. On the one hand, the queue cannot overflow, on the other hand, messages are lost, e. g. it is not possible to receive 25 images per second from the robot.

reject will not enter any messages into the queue to the PC. Therefore, the PC will not receive any messages.

collect <seconds>. This mode sends messages to the PC for the specified number of seconds. After that period of time, no further messages will be sent until another queue fill request is sent.

save <seconds>. This mode collects messages for the specified number of seconds, and it will afterwards store them on the memory stick as a log file under *OPEN-R/APP/CONF/LOGFILE.LOG*. No messages will be sent to the PC until another queue fill request is sent.

sg ? | <id> {<num>}. Sends generic debug data. Generic debug data consists of an *id* and up to ten decimal numbers. Type *sg ?* to list all generic debug data ids.

so off | on. Switch sending of *oracled world states* on or off. *Oracled world states* are normally sent to all processes. This allows the modules calculating the world state to be switched off without a failure of the robot. However, the option can produce confusing results if parts of the world state are only sometimes calculated by the robot. Then, the world state sometimes results from the robot's own calculations and sometimes from the simulator. Therefore, sending oracled world states to the robots can be switched off. By default, it is switched on. Note that this command only has an effect on simulated robots.

sr ? | <task> (? | <solution>). Sends a solution request. This command allows switching the solutions for a certain task. Type *sr ?* to get a list of all tasks. To get the solutions for a certain task, type *sr <task> ?*.

tr ? | <type>. Sends a tail request. Type *tr ?* to see all available tail requests.

xbb ? | unchanged | <behavior> {<num>}. Selects a Xabsl2 basic behavior. The command suppresses the basic behavior currently selected by the Xabsl2 engine and replaces it with the behavior specified by this command. Type *xbb ?* to list all available Xabsl2 basic behaviors. Some basic behaviors can be parameterized by a list of decimal numbers, e. g. *xbb go-to-point 1600 0 0* to walk to position (1600 mm, 0 mm, 0°). Use *xbb unchanged* to switch back to the basic behavior currently selected by the Xabsl2 engine. The command *xbb* only works if a Xabsl2 behavior was loaded with the command *xlb* (see below).

[2]Currently, the robot crashes if the queue overflows.

[3]Currently, this mode does not reasonably work together with debug drawings, because newer drawing commands replace the older ones.

xis ? | <inputSymbol> (on | off). Switches the visualization of a Xabsl2 input symbol in the *Xabsl2 View* on or off. Type *xis ?* to list all available Xabsl2 input symbols. The command *xis* only works if a Xabsl2 behavior was loaded with the command *xlb* (see below).

xlb ? | <name>. Load a Xabsl2 behavior. The command loads the symbols for the specified behavior and will send the compiled version of the behavior to the robot. The command must be executed before any other Xabsl2 command and the *Xabsl2 View* will work. Type *xlb ?* to list all available behaviors. Please note that the behavior loaded has to match the solution for *behavior control* selected on the robot. To use the *Xabsl2 View*, the corresponding debug key has to be set, i. e. *dk sendXabsl2DebugMessages on*.

xo ? | unchanged | <option> {<num>}. Selects a Xabsl2 option. The command suppresses the option currently selected by the Xabsl2 engine and replaces it with the option specified by this command. Some options can be parameterized by a list of decimal numbers, e. g. *xo go-to-kickoff-position 2000 0* to walk to position (2000 mm, 0 mm). Type *xo ?* to list all available Xabsl2 options. Use *xo unchanged* to switch back to the option currently selected by the Xabsl2 engine. The command *xo* only works if a Xabsl2 behavior was loaded with the command *xlb* (see above).

xos ? | <outputSymbol> (on | off | ? | unchanged | <value>). Show or set a Xabsl2 output symbol. The command can either switch the visualization of a Xabsl2 output symbol in the *Xabsl2 View* on or off, or it can suppress the state of an output symbol currently set by the Xabsl2 engine and replace it with the value specified by this command. Type *xos ?* to list all available Xabsl2 output symbols. To get the available states for a certain output symbol, type *sr <outputSymbol> ?*. Use *xos <outputSymbol> unchanged* to switch back to the state currently set by the Xabsl2 engine. The command *xos* only works if a Xabsl2 behavior was loaded with the command *xlb* (see above).

I.6 Examples

This section presents some examples of script files to automate various tasks:

I.6.1 Recording a Log File

To record a log file, the robot shall send images including the camera matrix and odometry data. The script connects to a robot and configures it to do so. In addition, it prints several useful commands into the console window, so they can be executed by simply setting the caret in the corresponding line and pressing the *enter* key. As these lines will be printed before the messages coming from the router, one has to scroll to the beginning of the console window to use them. Note that the file name behind the line *log save* is missing. Therefore, a name has to be provided to successfully execute this command.

```
# connect to a robot
```

```
sc 172.21.3.201

# suppress messages
msg off

# disable everything but sensor data processor and head control
sr SensorDataProcessor Default
sr ImageProcessor disabled
sr SelfLocator disabled
sr BallLocator disabled
sr PlayersLocator disabled
sr RobotStateDetector disabled
sr BehaviorControl disabled
sr HeadControl GT2003

# stop motion
mr normal 0 0 0
hcm none
hmr 0 0 0 0

# queue real-time mode, send JPEG images and odometry
qfr replace
dk sendJPEGImage on
dk sendOdometryData on

# print some useful commands
echo hcm searchForLandmarks
echo hcm searchForBall
echo hcm none
echo hmr 0 0 0 0
echo log start
echo log stop
echo log save
echo log clear
```

I.6.2 Replaying a Log File

The example script shown was used to test the LinesImageProcessor2/LinesSelfLocator pair. It instantiates a robot named *LOG1* that is fed by the data stored in the log file *GT2003\Config\Logs\myLogFile.log*.

```
# replay a log file
sl LOG1 myLogFile
```

```
# suppress messages
msg off

# simulation time on, otherwise log data may be skipped
st on

# configure modules. Important: sensor data processor disabled
sr SensorDataProcessor disabled
sr ImageProcessor GT2003
sr SelfLocator GT2003
sr BallLocator PIDSmoothed
sr PlayersLocator GO2003
sr RobotStateDetector disabled
sr BehaviorControl disabled
sr HeadControl disabled

# request some drawings
dk send_imageProcessor_horizon_drawing on
dk send_imageProcessor_scanLines_drawing on
dk send_selfLocator_drawing on
dk send_selfLocatorField_drawing on
```

I.6.3 Remote Control

This script demonstrates joystick remote control of the robot.

```
# connect to a robot
sc 172.21.3.201

# suppress messages
msg off

# switch off everything but motion
sr ImageProcessor disabled
sr SelfLocator disabled
sr BallLocator disabled
sr PlayersLocator disabled
sr BehaviorControl disabled

# stop motion
mr normal 0 0 0
hcm none
hmr 0 0 0 0
tr noTailWag
```

```
# queue real-time mode, send JPEG images
qfr replace
dk sendJPEGImage on

# use accelerator lever to control head pan
jhc pan

# assign actions to joystick buttons
jbc 1 mr unswBashOptimized
jbc 2 mr leftHeadKick
jbc 3 mr rightHeadKick
jbc 4 mr sit
jbc 5 mr scratchHead
jbc 6 tr wagHorizontalFast
jbc 7 tr noTailWag
```

Appendix J

RobotControl Usage

This chapter describes how to use the RobotControl application. As RobotControl is a very complex system, not all features will be described. But it will help to get an overview about the capabilities of the program.

J.1 Starting RobotControl

Requirements. RobotControl needs at least version 4.0 of the Microsoft Internet Explorer installed on the system to work properly. In addition, it is important that RobotControl is started from a directory under *GT2003*.

After the First Start The application looks a little bit strange. No child windows appear and all tool bars are pushed together. But the toolbars can be moved with the mouse to be distributed over more rows. To switch toolbars on or off, right-click on the toolbar area and select the visible toolbars from the pop-up menu. Dialogs can be opened by using the “View” menu or, for some of them, by using the “Views” toolbar. Note that the window layout will not be restored during the next start of the application unless it is saved using the “configuration” toolbar (cf. Sect. J.2.2).

J.2 Application Framework

The following toolbars and dialogs form the framework of the application.

J.2.1 The Debug Keys Toolbar

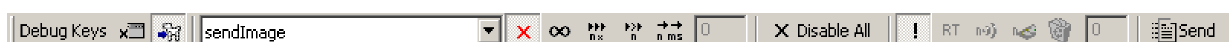

Figure J.1: The Debug Keys Toolbar

The *Debug Keys Toolbar* (cf. Fig. J.1) is used to switch debug keys on or off (cf. App. E.3). Each debug key can be parameterized in four different ways describing how often and in which frequency it will be enabled.

The combo box contains all available debug keys. To edit the properties of a debug key, select the key from the list and use on of these buttons:

Disabled. The debug key is disabled.

Always. The debug key is always enabled.

n **times.** The debug key is enabled for n times, i. e., it will return *true* the next n frames, and *false* afterwards. n has to be entered into the edit control before the button is pressed.

Every n **times.** The debug key is enabled every n-th frame, i. e., it will return *true* every n-th call, and *false* in between.

Every n **ms.** The debug key is enabled every n milliseconds, i. e., it will return *false* until at least n milliseconds passed since the last time it returned *true*.

Disable All. Disables all debug keys.

There are five buttons to select how the outgoing message queue is treated on the robot:

Immediately. All outgoing messages are sent immediately via the wireless network.

Realtime. This mode allows dropping messages if there is not enough time to transmit. This is useful, e. g., for sending as many images as possible without slowing down the robot.

Send after n **seconds.** The transmission of outgoing messages is delayed for n seconds. The value of n is set with the attached edit field.

Save to stick after n **seconds.** Instead of transmitting outgoing messages via the wireless network, they are stored on the robot's memory stick after a delay of n seconds. A log file is created on the memory stick which afterwards can be replayed using the *Log Player Toolbar* (cf. Sect. J.2.4). The value of n is set with the attached edit field.

Reject all. All outgoing messages are dropped.

There are two debug key tables in RobotControl, one for a physical robot connected via the wireless network, and one for the selected simulated one. With the buttons *Edit table for robot* and *Edit table for local processes* one can select which of them is edited. *Send* sends both debug key tables to the proper destinations by putting them into message queues, i. e. nothing will change as long as *Send* has not been pressed.

Figure J.2: The Configuration Toolbar

J.2.2 The Configuration Toolbar

The *Configuration Toolbar* (cf. Fig. J.2) is intended to manage different configurations of RobotControl. Up to now, it only works for window layouts.

As it is sometimes very difficult to arrange the dialogs and toolbars in the main window for efficient use, the *Configuration Toolbar* allows saving window layouts to the Windows registry. With the *New* button, new layouts can be created from the current one, *Rename* renames a layout, and *Delete* deletes a layout. Note that changes in the layout are not automatically saved. Instead, this has to be done manually by pressing the *Save* button. One can select between different layouts by selecting a different entry in the combo box.

J.2.3 The Settings Dialog

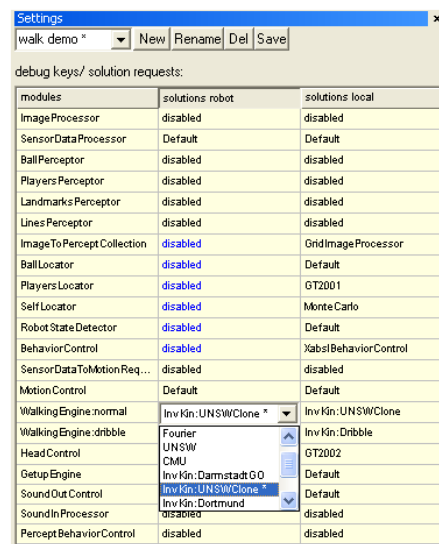

Figure J.3: The Settings Dialog: RobotControl's most often used dialog

With the *Settings Dialog* (cf. Fig. J.3), solutions for modules (cf. Sect. 3) running on the robot or on the PC can be switched. A certain combination of solutions is called a setting and can be stored. All settings are stored in *GT2003\Config\Settings*. The default solution for each module is marked with an asterisk.

Figure J.4: The Log Player Toolbar

J.2.4 The Log Player Toolbar

In RobotControl, log files can store a set of messages of any kind used to communicate between modules or between dialogs or toolbars and modules, e. g. it is possible to record pictures sent by a robot in a log file, and then play that log file several times to test different kinds of image processing with exactly the same input data.

The *Log Player Toolbar* (cf. Fig. J.4) is used to record, play, and modify such log files. Its buttons should be known from other players, e. g. CD-players. *Play*ing a log file sends all messages in it to all running modules in RobotControl as well as to all dialogs that can handle that type of message. *Step forward* and *Step backward* do the same with single messages.

*Record*ing appends all messages sent from a robot via the wireless network to the actual log file (in memory) that can be *save*d afterwards.

J.2.5 WLan Toolbar

Figure J.5: The WLan Toolbar

The *WLan Toolbar* (cf. Fig. J.5) is used to create, edit, and switch between different wireless network configurations. It also allows connecting to (and of course disconnecting from) all enabled robots in the current wireless network configuration.

Creating new and modifying existing wireless network configurations opens the dialog shown in figure J.6. Connections to robots are established via the router (cf. Sect. 5.3). The dialog allows all relevant parameters to be specified: the IP addresses of the robots as well as the IP address of the router and its base port that is used to map between port numbers on the router and robot IP addresses to be routed to.

Furthermore settings can be edited such as the wireless network *ESSID*, the *Netmask*, the *APMode*, the *WepKey*, and the *Channel*. Each robot IP has a *cp*-button. Pushing that button calls *copyfiles.bash* with all parameters visible in the dialog. So using this dialog it is possible to keep the settings written to memory sticks and the settings used to connect to robots consistent.

J.2.6 Game Toolbar

With the *Game Toolbar* (cf. Fig. J.7) the game control data can be generated and sent. So Sony's RoboCup Game Manager is not needed to test behaviors.

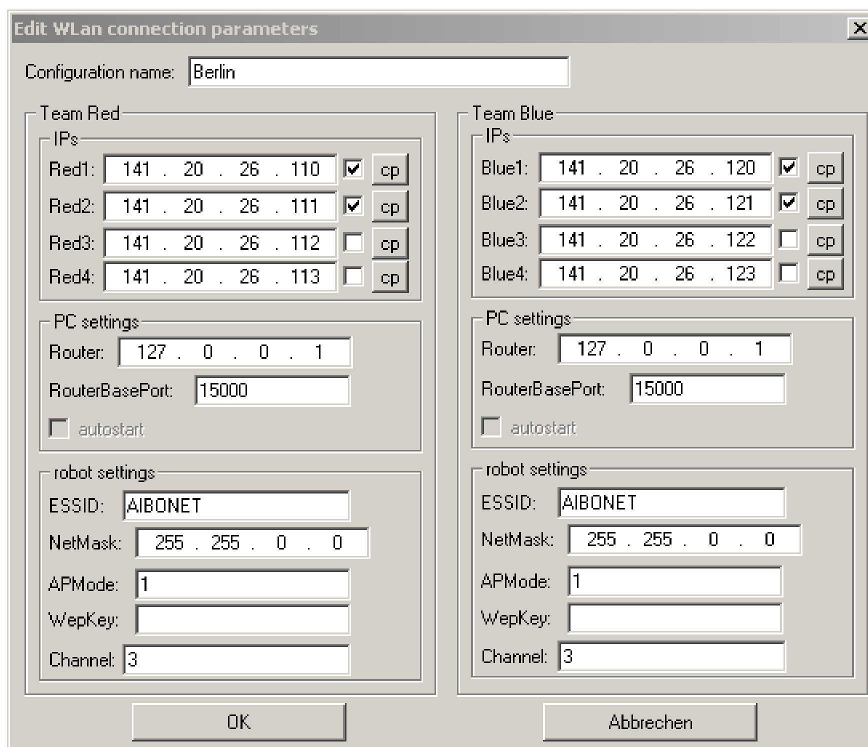

Figure J.6: The WLan Connection Parameters Dialog

Figure J.7: The Game Toolbar

Penalties are set by selecting the player from the combo box and pressing one of the four penalty buttons to the right. A penalty is released by pushing the *playing* button. In the *ready* state, all penalties are reset.

The slider at the right adjusts the *game speed* (from 0.1 to 1). This allows for testing behaviors in “slow motion”. The value is multiplied to the translation vector of the motion request processed by the *MotionControl* module.

J.3 Visualization

J.3.1 Image Viewer and Large Image Viewer

The two image viewer dialogs (cf. Fig. J.8) display images and debug drawings from the *queue-ToGUI* (cf. Fig. 5.3) so images from the robot, the log player, or the simulator are displayed. The *Image Viewer* has space for eight images with a fixed size. The *Large Image Viewer* shows only one image and is sizable. With the context menu different types of images and different debug drawings can be selected. The context menu also contains *Copy drawings* and *Copy image and*

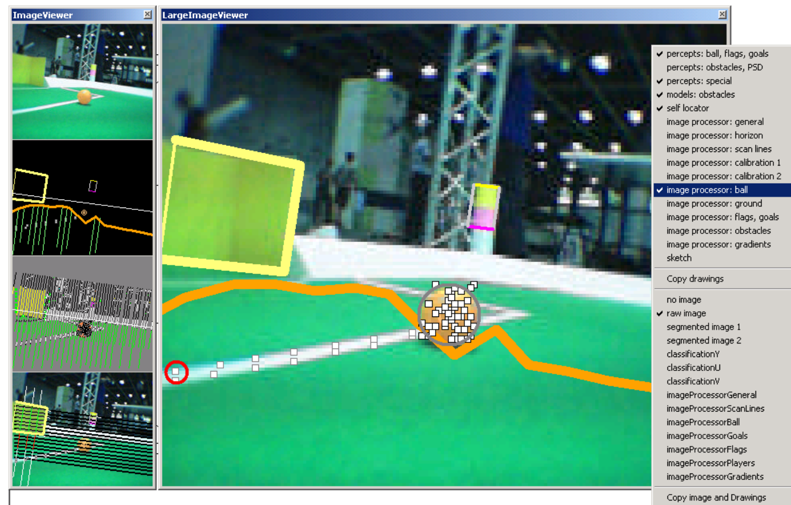

Figure J.8: Image Viewer and Large Image Viewer Dialog

drawings which copies only the debug drawings as a vector graphic or the image with the drawings as a bitmap to the clipboard. Important: to see a debug drawing created in a special solution of some module (cf. Sect. 3), this solution has to be selected (cf. Sect. J.2.3).

J.3.2 Field View and Radar Viewer

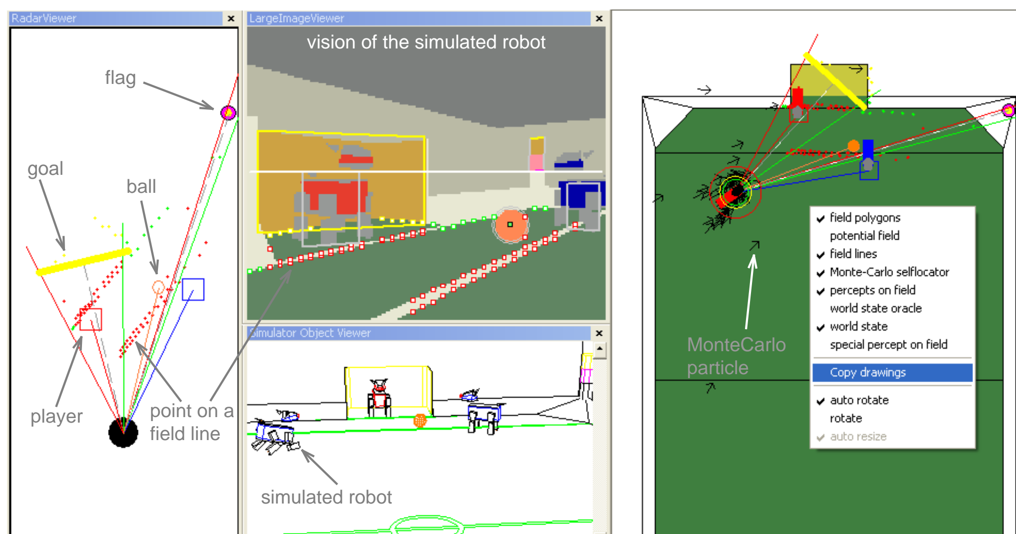

Figure J.9: RobotControl's Field View and the Radar Viewer Dialog

The *Field View* and the *Radar Viewer* (cf. Fig. J.9) both display percept drawings. The *Radar Viewer* display the percepts relative to the robot. The *Field View* draws the percepts based on the robot's localization on the field. The field view is also used for world state and localization

drawings. In both dialogs the drawings can be selected with the context menu. *Copy drawings* copies the drawings as a vector graphic to the clipboard.

J.3.3 Radar Viewer 3D

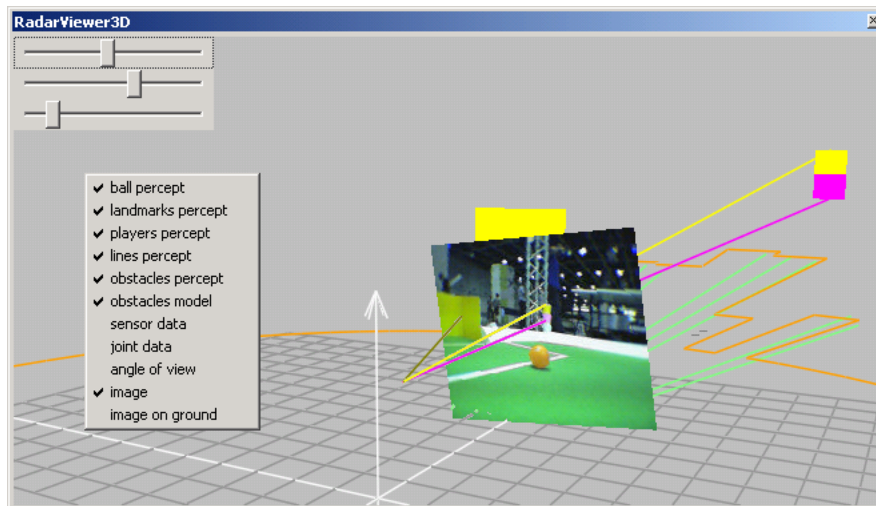

Figure J.10: The Radar Viewer 3D

The *Radar Viewer 3D* (cf. Fig. J.10) provides a 3D visualization for percepts, but also for the image. The percepts are displayed in the 3D space after a transformation of the percepts into the robot's system coordinates of using the camera matrix. The sliders can be used to adjust the position of the camera.

J.3.4 Color Space Dialog

The *Color Space Dialog* (cf. Fig. J.11) visualizes how an image uses the YUV color space, and it displays either the y, u, or v channel as a height map. By dragging with the left mouse button the 3-D scene can be rotated. With the context menu the type of view can be selected.

J.3.5 Value History Dialog

The *Value History Dialog* (cf. Fig. J.12) allows displaying different values over time. This helps, e. g., to check the stability of a ball modeling algorithm. The values that shall be traced can be selected from the context menu. The time range that is displayed can be changed using the slider at the top of the dialog.

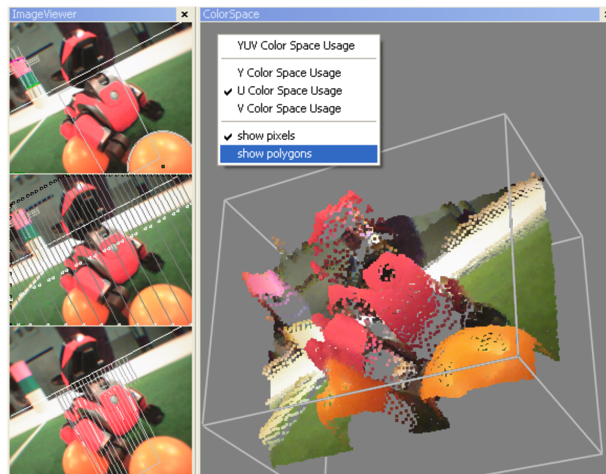

Figure J.11: The Color Space Dialog showing the u-channel of an image as a height map.

J.3.6 Time Diagram Dialog

The *Time Diagram Dialog* (cf. Fig. J.13) visualizes the times which different modules need for their execution in terms of bars. The values next to each bar show the measured time in milliseconds and the frequency (in times per second).

Times can be measured on the robot by selecting *stop times on robot*. If the simulator is used (cf. Sect. J.4), the times can be measured on the computer by selecting *stop times local*. The option *view log files* displays the measured times of recorded log files. Since the times for the execution of the modules can vary very much from one measurement to the next one, the motion of the time-indicators can be smoothed by using average values. The average can be chosen between 2 and 500 measurements. Clicking the right mouse button in the dialog opens a context menu in which the modules of interest can be selected. This menu also offers the option to export the values to a file in a comma-separated format.

The design of the dialog varies, depending on its size and the number of the selected modules.

J.4 The Simulator

The simulator is a very powerful extension for RobotControl. As SimGT2003 (cf. Sect. 5.1), it is based on *SimRobot*. The simulator offers a lot of possibilities to develop, test, and debug new algorithms or alternative solutions for modules without using a robot.

As shown in Figure J.15, all relevant objects for robot-soccer are included in the simulation: the field (including landmarks, goals, lines, etc.), players, and the ball. Other objects (e. g. for challenges) can be added with ease. The image created depends on the position of the robot and also on the current angles of the head and the leg joints.

For developing modules which result in movement of the robots, e. g. behavior, the object viewer (cf. Fig. J.15) shows the complete field with all simulated objects. It can be activated by the button *Object Viewer* of the simulator toolbar (cf. Fig. J.14). The vantage point of the observer

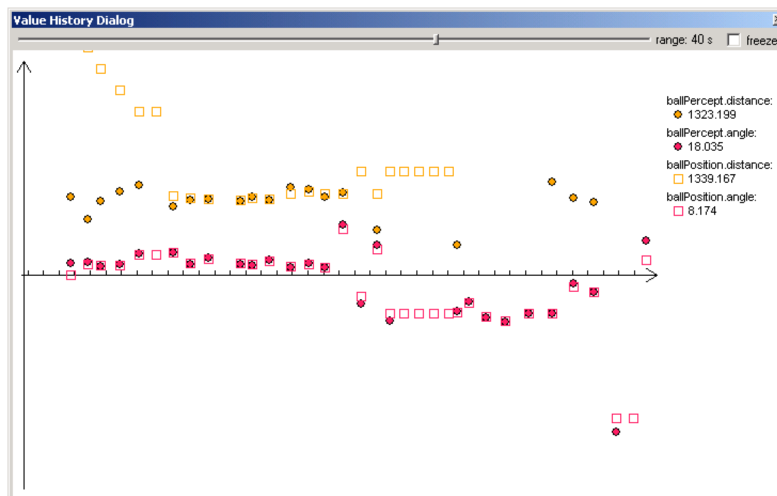

Figure J.12: The Value History Dialog

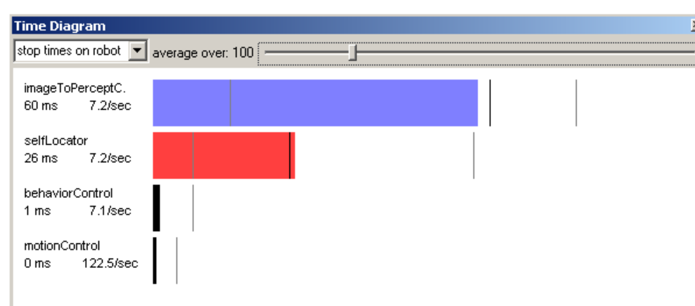

Figure J.13: The Time Diagram Dialog

is variable and can be changed by moving the bars under and beneath the scene displayed. The zooming level, detail level, and the perspective distortion can be adjusted by using the appropriate buttons in the toolbar. Besides these options, the toolbar contains buttons to start and reset the simulation, and to force a step-by-step mode. The touch sensors at the back and the head of the robot can be "virtually pressed" by the buttons marked with an arrow at the according position. One of those buttons pretends the robot to be fallen aside.

A very helpful feature of the simulator is the oracle. It lets the robot know everything of its environment exactly. This can help to develop modules without being dependent on other modules. For example a behavior can be implemented and tested without a self-locator. The button *send oracle* activates this function.

Up to four robots of one team can be simulated at the same time. To send commands or receive information from a robot, connect to it by choosing it out of the list at *Robots* in the menu bar. This menu also includes the option to generate images for the connected robot or all simulated robots. Be aware, that generating images for all robots needs a lot of computing power. An entry in the status line of RobotControl shows the currently connected robot.

Figure J.14: Toolbar of the Simulator

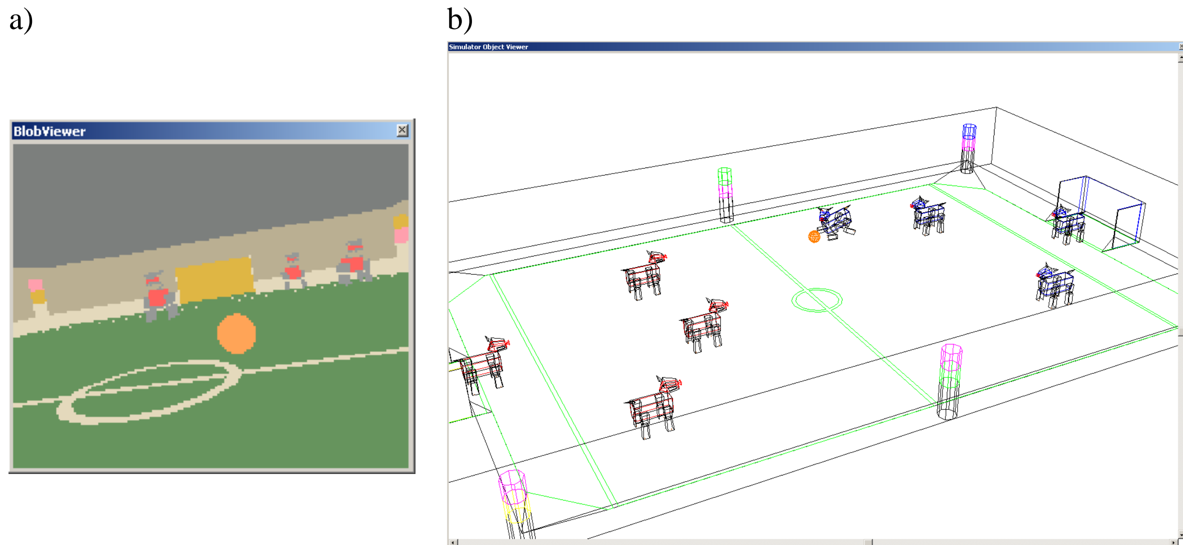

Figure J.15: a) Simulated image and b) Object Viewer of the simulator

J.5 Debug Interfaces for Modules

For some of the modules specific debug interfaces were developed.

J.5.1 Xabsl2 Behavior Tester

The *Xabsl2 Behavior Tester* is a debugging interface to the behavior modules derived from *Xabsl2BehaviorControl* (cf. Sect. 3.8.1). One can view almost all internal states of the engine. *Options* and *Basic Behaviors* and *Output Symbols* can be selected manually for separate testing.

With the check box *test on robot* in the right upper corner one can set whether the behavior shall be tested on the robot or in the simulator.

In the topmost combo box an option or basic behavior can be selected. If an option is selected it is used as the root option. The execution of the option tree starts from that option. If a basic behavior is selected only this behavior gets executed without any option tree being traversed. This allows testing single options and basic behaviors separately. If the option or basic behavior selected has parameters these can be entered into the edit fields below.

The following two combo boxes allow altering output symbols manually. The left box selects an output symbol while the right one sets its value. If this is done the value generated through the engine by traversing the option activation path is ignored and overwritten with the set value. This is useful for testing output symbols separately.

At the top of the white area *Agent* shows the name of the active Xabsl2 agent. Next the *Option Activation Path* is displayed. The first column shows all the activated options. The second column

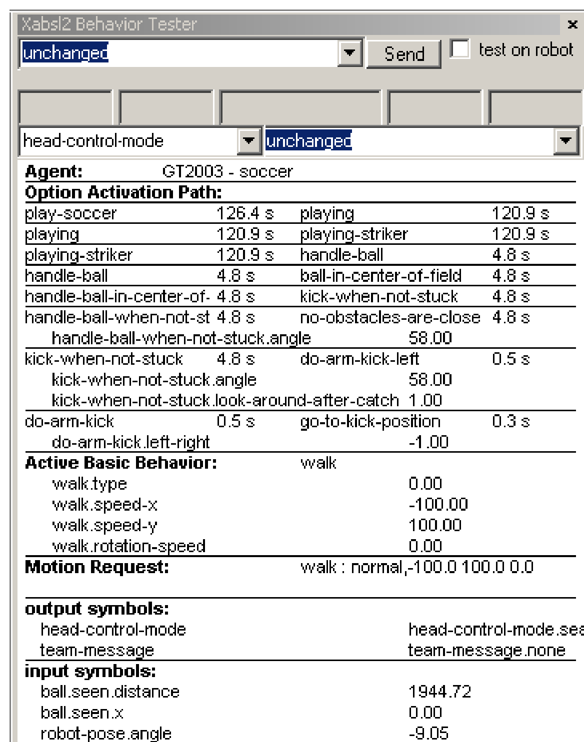

Figure J.16: The Xabsl2 Behavior Tester Dialog

shows the time how long these options are already activated. Then in the third column the active state of each option, and in the fourth column the time how long the state is already active, are displayed. If an activated option is parameterized its current parameter values are shown below.

Then the *Active Basic Behavior* shows, which basic behavior is currently activated along with its parameter values. The *motion request* shows the motion request that resulted from the execution of the basic behavior.

The *Input Symbols* section shows the current values of the input symbols selected. The selection, which symbols shall be displayed, can be done with the context menu of the dialog. The same applies to *Output Symbols*.

At last, in the context menu exists an entry *Reload Files*. The dialog rereads the debugging symbols and sends the intermediate code to the robot and to the local processes, where the engine is newly created. That allows the testing of changes in the behavior without rebooting the robot or restarting *RobotControl*.

J.5.2 Motion Tester Dialog

With the *Motion Tester Dialog* (cf. Fig. J.17) it is possible to send *MotionRequests* from *RobotControl* to the robot.

In the upper area of the dialog the different modes stand, getup, walk, or special action can be chosen. In walk mode the velocities in x and y direction and the rotation speed can easily be

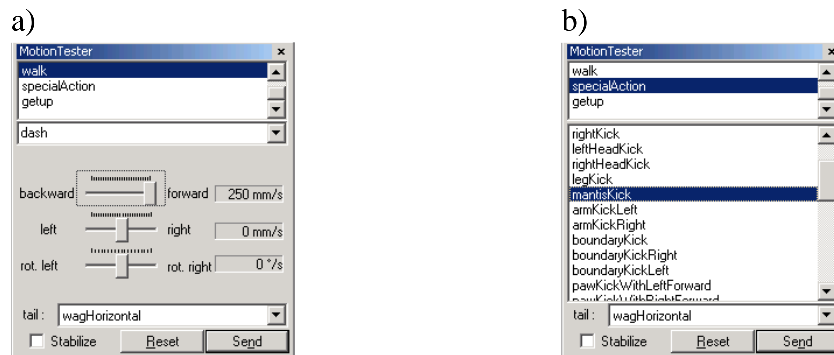

Figure J.17: The Motion Tester Dialog with a) walking and b) special actions

set by sliders. In special action mode the different special actions (i. e. kicks or joy dance) can be chosen from the select box. In the lower area of the dialog the movement of the tail can be set. Possible settings are, e. g., wag horizontally or in parallel to the ground (based on the evaluation of the gravity sensors).

To send the *MotionRequest* to the robot, you have to push the *send* button.

For another way to control the robots motion, the *Joystick Motion Tester* (cf. Sect. J.5.5) is available.

J.5.3 Head Motion Tester Dialog

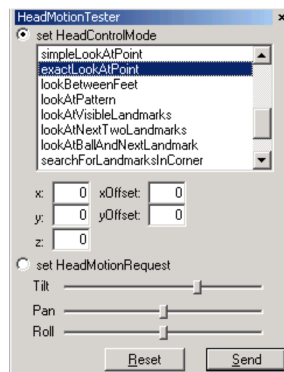

Figure J.18: The Head Motion Tester Dialog

The *Head Motion Tester Dialog* (cf. Fig. J.18) is handy to test the head control module, it is possible to send *HeadMotionRequests* or to set the *HeadControlMode* from *RobotControl*.

The desired *HeadControlMode* is selectable from a list of all available modes. Some modes require additional parameters (i. e. the coordinates of a point to look at). These can be set in the appearing input elements.

In *HeadMotionRequest* mode, the desired joint values can be set by sliders.

J.5.4 Mof Tester Dialog

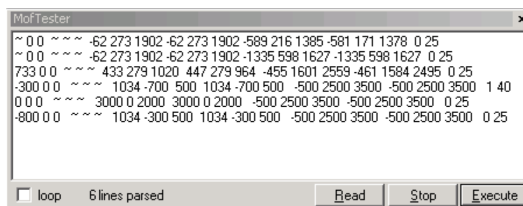

Figure J.19: The Mof Tester Dialog

The *Mof Tester Dialog* (cf. Fig. J.19) is used to write and test new motions. Motions are specified in a description language (cf. Sect. 5.4). Joint data lines from these descriptions may be entered into the input field of the dialog and can be sent to the robot via the wireless network at runtime.

For this dialog to work it is necessary that the module *DebugMotionControl* is running on the robot instead of the default motion control module. The debug module will not execute normal motion requests from behavior control but rather wait for debug messages sent from the *MofTester* dialog. It is activated on the robot by switching module solutions with the *Settings Dialog* (cf. Sect. J.2.3).

The *execute* button parses the input field for lines containing joint data information and sends the sequence to the robot. If the *loop* check box is activated when pressing *execute* the sequence will be executed repeatedly.

The *stop* button stops any sequence currently being executed.

The *read* button provides a very handy tool when creating new motions. It reads the robot's current joint angles from sensor data and puts them into a new line in the input field. This is extremely useful in combination with the stay-as-forced motion mode the *DebugMotionControl* module provides while not executing joint data sequences. In stay-as-forced mode all motors are controlled with feedback from sensor input, and therefore they always maintain their current position. In this mode joints may be moved manually and the resulting joint angles can be read into the *Mof Tester Dialog*.

J.5.5 Joystick Motion Tester Dialog

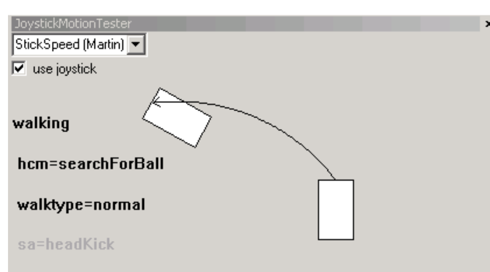

Figure J.20: The Joystick Motion Tester Dialog

The *Joystick Motion Tester Dialog* (cf. Fig. J.20) is used to control a robot that is connected via the wireless network by joystick. Using such an analog input device is, e. g., useful for testing new walking engines or parameters or just walking stability. The primary aim is not a complete remote control, but only moving the robot around (and optionally moving its head) while all other modules keep running autonomously.

This dialog is only tested with an MS Sidewinder Precision 2 USB joystick, but should work with any joystick providing three axes, an accelerator, and eight buttons. After the joystick has been attached, the *use joystick* box has to be checked to start using this dialog. If no joystick seems to be connected, the dialog will refuse to work.

The joystick controls the walking engine of a robot connected with RobotControl via the wireless network. The direction the robot should move to according to the state of the joystick is visualized in the dialog. Red text in *Joystick Motion Tester Dialog* signals that current changes have not been executed yet, black text shows the actual states or commands, and gray text visualizes states or commands that were sent last but are inactive at the moment.

The buttons 1 to 4 can be used for different kicks, button 7 to start the *getup* motion and button 5 to switch from walking control to head control. As long as button 5 is pressed, the joystick will control the head instead of the walking. That will be visualized by the dialog, too.

To ease the use of the dialog for different people, different schemes for joystick control were implemented. One is called *StickSpeed*, in which the walking speed is completely controlled by the three axes of the joystick, the accelerator is used to switch between head control modes, the walking type can be changed with button 8, and all special actions can be generated by holding button 6 pressed and moving the accelerator.

Another scheme for *Joystick Motion Tester Dialog* is called *AcceleratorSpeed*: the accelerator is used to control the forward walking speed, the axes are only used for sideways speed and direction, and walk types can be changed with button 6.

Commands will only be sent via the wireless network if they differ from the previous command and at most every 300 ms, because the router throughput and especially its response times do not allow much more. So whirling around the joystick will definitely not encourage the robot to do the same.

J.6 Color Calibration

J.6.1 The Color Table Dialog

The *Color Table Dialog* (cf. Fig. J.21) is used to create color tables for image processing. The current image from the simulator, a logfile, or the robot's camera (via WLAN) is shown top right (cf. Fig. J.21). Beneath the same image can be seen, segmented with the current color table. Choosing a color class and clicking with the left button on a pixel in one of the two images will assign the color class to the $4 \times 4 \times 4$ cube in YUV color space according to the color value of the pixel. The result takes effect immediately and the segmented image will be updated. Clicking with the right button on a pixel will remove the according color class assignment. There are also

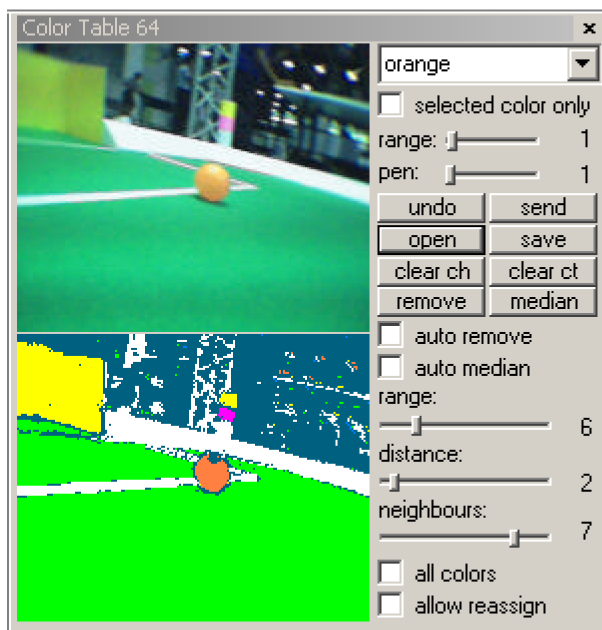

Figure J.21: The Color Table Dialog

functions to undo the last assignment, to clear all assignments for a whole color class and to reset the complete table.

To speed up the process of creating a color table, pixels with no neighbors of the same color class can be removed by pressing the *remove* button. For large areas of the same color class, it is useful to add pixels with at least a given count of neighbors and a maximum distance in the color space. This can be done by pressing the *median* button. Both optimizations can work with the selected color class or with all colors (*all colors* checkbox selected). If existing assignments should not be overridden, *allow reassign* checkbox should not be selected to prevent unwanted changes.

With *auto-remove/median* enabled, the optimizations are triggered with every incoming image. This is useful if logfiles with specific colors have been recorded. The color table can be created without many manual assignments.

After having assigned all colors needed, the table can be sent to the robot via WLAN or saved to a file. The GT2003 vision modules need a file named *coltable.c64*. It is also possible to load an existing file and to modify it.

J.6.2 HSI Tool Dialog

This tool uses the HSI color space to create a color table. It allows the user to specify the minimum and maximum values for hue, saturation, and intensity for each color class using sliders, and it gives an instant feedback by real time color classification of four images at the same time. These images are directly taken from the memory of *RobotControl* and can be held as long as needed. It is also possible to update an image simultaneously with the one in the memory of

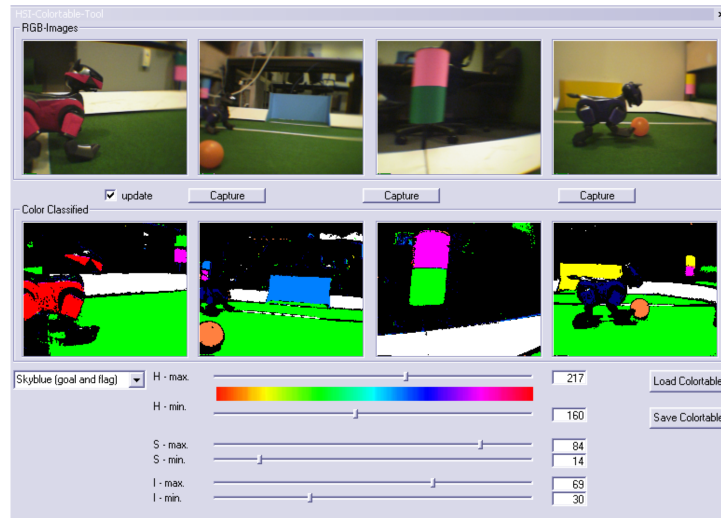

Figure J.22: The HSI Tool Dialog

RobotControl. The tool saves the color table both in HSI and YUV format. By selecting an image there is another dialog with a zoomed view (cf. Fig. J.23) of it and the belonging color classified image. In this dialog the color class can simply be edited by selecting single pixels. There is also an undo function for several steps.

This HSI approach leads to good results very quickly by defining big sectors in the color space and it is more tolerant against changing light conditions. The tool can be combined with the other color table tool using YUV color space. First, the HSI tool is used to quickly generate a color table and the YUV tool can be employed for fine tuning if required.

Dialog Structure. The main dialog (cf. Fig. J.22) consists of two parts. In the upper half are spaces for four RGB images and below them the corresponding color classified images will be displayed. Below each space there are buttons for capturing an image, and at the left is a check box for an automatic update of the image above it.

In the lower half of the dialog most of the controls are located. There is a combo box with entries for all color classes used by the image processing module, a button for loading HSI color tables, a button for saving the HSI and the corresponding YUV color table, and six sliders for determining the range of the selected color class in the HSI color system. An HSI color class consists of a minimum and a maximum value for hue (H), saturation (S), and intensity (I).

Main Dialog. With the *capture* buttons in the upper half of the main dialog (cf. Fig. J.22) the actual image that *RobotControl* has in memory can be saved. It can be done for four images. When the box before *update* below the left place is checked, this image is updated automatically, when *RobotControl* gets a new image from the robot or a log file. The color classified image will be updated automatically when you change the range of a color class.

With the combo box at the lower left of the dialog the color class to be edited can be selected. The sliders show the minimum and maximum values of this color class. The ranges for H, S,

and I can be modified by changing the positions of the sliders. While moving a slider the color classified image is permanently updated.

The range of values for the hue of a color class goes from 0 to 360, and for saturation and for intensity from 0 to 100. Because the value for hue in HSI color system lies on a circle, it is possible that the maximum value for this range is smaller than the minimum value. For a red color, e. g., the minimum of the hue range could be 350 and the maximum could be 15. For saturation and intensity the minimum value should be below the maximum value.

The color table can be saved by pressing the *Save* button. It appears a file dialog where the destination and the name of the color table can be selected. The suffix *.hsi* will be added automatically. A YUV color table converted from the HSI color table will also be saved with the same name and the suffix *.c64*.

An HSI color table can be load by pressing the *Load* button. It can be selected in the appearing dialog. It is only possible to load HSI color tables. YUV color tables are not supported yet by the HSI tool.

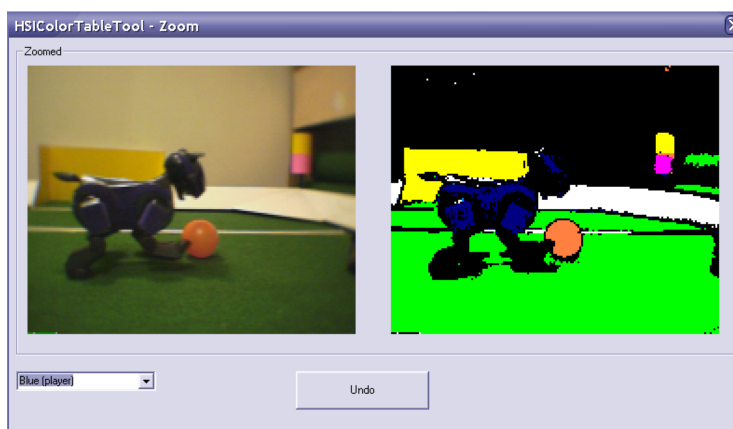

Figure J.23: The HSI Tool Zoom Dialog

Zoom Dialog. By performing a click with the left mouse button on one of the RGB images, another dialog window with a zoomed view (cf. Fig. J.23) of the selected image and the belonging color classified image will appear. In this dialog it is possible to improve the precision of the ranges for color classes by selecting single pixels. At the lower left the combo box with the color classes to edit is located. There is also an *Undo* button for the last six changes.

By pressing the left or the right mouse button on a pixel in one of the zoomed images the selected color class will be modified. If the left mouse button has been pressed and the color of the selected pixel lies outside the color class, the range will be enlarged until the values for hue, saturation, and intensity are inside this range. If the right mouse button has been pressed and the selected pixel lies inside the range of the color class, the range will be reduced until the values for hue, saturation, and intensity are outside of it.

By selecting single pixels, it is possible to determine which color should belong or not belong to the chosen color class. Thus precision of the class can be improved.

J.6.3 The TSL Color Segmentation Dialog

The main idea of color segmentation is to partition the chrominance space into subspaces, where each subspace represents exactly one color. The algorithmic performance of color classification depends on the chosen chrominance space. The reason for that is, that the complexity of color segmentation strongly depends on the shape of the subspaces. If the normal vectors of all bounding hyperplanes are parallel to the unit vectors of the chrominance space coordinate system, then the color assignment can be done at highest efficiency. Therefore, we define a new chrominance space where all relevant colors can be extracted by using a set of thresholds $t_1^c, \ldots, t_6^c$ per color c. For our purpose, a classification is needed to distinguish between the main nine RoboCup colors (green, sky-blue, yellow, orange, pink, dark blue, red, black and white). The so-called TSL* chroma-space based color-segmentation is more robust against luminance variation than similar YUV-based algorithms.

The TSL Dialog is a powerful tool for creating a set of thresholds for a robust color classification. On the left side of the dialogbox, two images are displayed. The upper images contains the raw data (either camera data or simulator output). The lower images shows the result of the color classification.

The user can select a color (e.g. ”yellow” in the *Select color* list in Fig. J.24) and a corresponding region in the image by clicking on the upper left and lower right corners of a rectangle. Then, a histogram of each color component (T’, S’, and L’) is calculated in real-time and displayed immediately. By pressing the *Auto* button, threshold values are set automatically. Using the sliders, the user can adjust and optimize the thresholds (in Fig. J.24: $t_1^{yellow}, \ldots, t_6^{yellow}$) manually. The influence of parameter values on the color classification can be seen in the lower left image. The robustness of the classification can be checked by adding noise using the *Noise* slider to the original image.

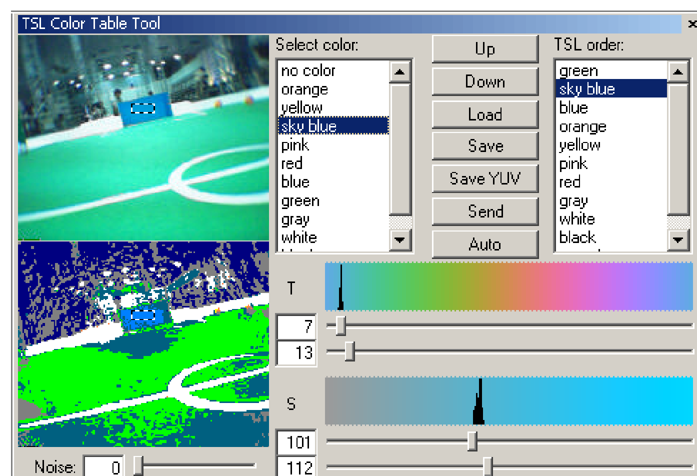

Figure J.24: The TSL Color Table Tool.

If areas overlap, a classification is not unique. Thus, the order of color classifications is important. With the *Up* and *Down* buttons, the priority of a selected color can be changed. The

priority is displayed in the *TSL Order* list. Settings can be saved and reloaded by using the *Load* and *Save* buttons. The *Save YUV* button stores the settings in the conventional YUV color table format. A set of threshold values can be send to a connected robot by clicking the *Send* button.

J.6.4 Camera Toolbar

Figure J.25: The Camera Toolbar

The *Camera Toolbar* (cf. Fig. J.25) is used to set the parameters that are provided by the robot's camera. These parameters are set with the combo boxes. White balance may be set to indoor, outdoor, or FL mode. Shutter speed may be selected from slow, medium, or fast. Finally low, medium, or high camera gain can be chosen.

The *send to robot* button sends the selected parameters via the wireless network to the robot. The new settings are applied immediately. When viewing the camera pictures with the image viewer (cf. Sect. J.3.1) the effects of different camera settings can be observed.

The *save* button writes the settings to a file named *camera.cfg*. This file is loaded at the start of RobotControl to initialize the *Camera Toolbar* with its contents. But more important this file is copied to the memory stick and loaded when booting the robot. The settings from this file are used on the robot unless different parameters are sent with the toolbar.

J.7 Other Tools

J.7.1 Debug Message Generator Dialog

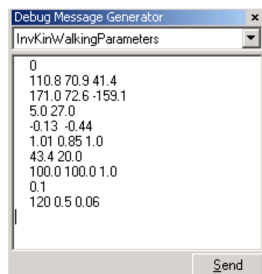

Figure J.26: The Debug Message Generator Dialog

In addition to the *Test Data Generator* described in E.2, the *Debug Message Generator Dialog* (cf. Fig. J.26) can be used to generate less common debug messages for which no special dialog exists. The *Test Data Generator* is usually easier to use and offers more features. The need to use the *Debug Message Generator Dialog* might arise if more than 10 parameters of a module

need to be updated in one go. The combo box is used to select what type of debug message is generated. When pressing the *send* button a message will be generated by parsing the input in the text field. The dialog may be extended easily by adding the code to parse the text input into a debug message. Therefore it allows generating new debug messages with *RobotControl* without having to create a new dialog. It is used, e. g., to send a new set of parameters to the walking engine. By this it is possible to change the walking style at runtime and therefore develop new walks very quickly. Another application is to test playing acoustic messages on the robot by sending the corresponding debug message.

References

[1] Simrobot homepage. http://www.tzi.de/simrobot.

[2] R. Brunn, U. Düffert, M. Jüngel, T. Laue, M. Lötzsch, S. Petters, M. Risler, T. Röfer, K. Spiess, and A. Sztybryc. Germanteam 2001. In *RoboCup 2001*, number 2377 in Lecture Notes in Artificial Intelligence, pages 705–708. Springer, 2002.

[3] H.-D. Burkhard, U. Düffert, J. Hoffmann, M. Jüngel, M. Lötzsch, R. Brunn, M. Kallnik, N. Kuntze, M. Kunz, S. Petters, M. Risler, O. v. Stryk, N. Koschmieder, T. Laue, T. Röfer, K. Spiess, A. Cesarz, I. Dahm, M. Hebbel, W. Nowak, and J. Ziegler. GermanTeam 2002, 2002. Only available online: http://www.tzi.de/kogrob/papers/GermanTeam2002.pdf.

[4] H.D. Burkhard, J. Bach, R. Berger, B. Brunswieck, and M. Gollin. Mental models for robot control. In M. Beetz et al., editor, *Plan Based Control of Robotic Agents*, number 2466 in Lecture Notes in Artificial Intelligence. Springer, 2002. To appear.

[5] Ingo Dahm, Sebastian Deutsch, Matthias Hebbel, and André Osterhues. Robust color classification for robot soccer. In *7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2004. to appear.

[6] David C. Fallside. W3C recommendation: XML schema part 0: Primer. 2001. http://www.w3.org/TR/xmlschema-0/.

[7] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte Carlo localization: Efficient position estimation for mobile robots. In *Proc. of the National Conference on Artificial Intelligence*, 1999.

[8] D. Gutmann, J.-S. Fox. An experimental comparison of localization methods continued. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Lausanne, Switzerland, 2002. EPFL.

[9] V. Jagannathan, R. Dodhiawala, and L. Baum. *Blackboard Architectures and Applications*. Academic Press, Inc., 1989.

[10] Matthias Jüngel, Jan Hoffmann, and Martin Lötzsch. A real-time auto-adjusting vision system for robotic soccer. In *7th International Workshop on RoboCup 2003 (Robot World*

Cup Soccer Games and Conferences), Lecture Notes in Artificial Intelligence. Springer, 2004. to appear.

[11] S. Lenser and M. Veloso. Sensor resetting localization for poorly modeled mobile robots. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, 2000.

[12] Martin Lötzsch. XABSL web site. http://www.ki.informatik.hu-berlin.de/XABSL.

[13] Martin Lötzsch, Joscha Bach, Hans-Dieter Burkhard, and Matthias Jüngel. Designing agent behavior with the extensible agent behavior specification language XABSL. In *7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2004. to appear.

[14] T. Röfer. Strategies for using a simulation in the development of the Bremen Autonomous Wheelchair. In R. Zobel and D. Moeller, editors, *Simulation-Past, Present and Future*, pages 460–464. Society for Computer Simulation International, 1998.

[15] T. Röfer and M. Jüngel. Vision-based fast and reactive monte-carlo localization. In *IEEE International Conference on Robotics and Automation*, pages 856–861, Taipei, Taiwan, 2003. IEEE.

[16] T. Röfer and M. Jüngel. Fast and robust edge-based localization in the sony four-legged robot league. In *7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence, Padova, Italy, 2004.

[17] D. Schulz, W. Burgard, D. Fox, and A.B. Cremers. Tracking multiple moving targets with a mobile robot using particle filters and statistical data association. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, 2001.

[18] S. Thrun, D. Fox, and W. Burgard. Monte carlo localization with mixture proposal distribution. In *Proc. of the National Conference on Artificial Intelligence*, pages 859–865, 2000.