Thomas Röfer
Tim Laue
Michael Weber

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

Hans-Dieter Burkhard
Matthias Jüngel
Daniel Göhring
Jan Hoffmann
Benjamin Altmeyer
Thomas Krause
Michael Spranger

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

Oskar von Stryk
Ronnie Brunn
Marc Dassler
Michael Kunz
Tobias Oberlies
Max Risler

Fachgebiet Simulation und Systemoptimierung,
FB 20 Informatik,
Technische Universität Darmstadt,
Hochschulstr. 10,
64289 Darmstadt, Germany

Uwe Schwiegelshohn
Matthias Hebbel
Walter Nisticó
Stefan Czarnetzki
Thorsten Kerkhof
Matthias Meyer
Carsten Rohde
Bastian Schmitz
Michael Wachter
Tobias Wegner
Christine Zarges

Institute for Robot Research,
Information Technology Section,
Dortmund University,
Otto-Hahn-Strasse 8,
44221 Dortmund, Germany

Abstract

The GermanTeam is a joint project of four German universities in the Four-Legged League. This report describes the software developed for the RoboCup 2005 in Osaka. 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.

The report serves as detailed documentation of the work that has been done and aims at enabling other researchers to make use of it. 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 2
  - 1.2.1 Humboldt-Universität zu Berlin 2
  - 1.2.2 Technische Universität Darmstadt 2
  - 1.2.3 Universität Bremen 3
  - 1.2.4 Universität Dortmund 4
- 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) 5
  - 1.3.4 Microsoft Hellhounds (Universität Dortmund) 5
- 1.4 Structure of this Document 5
- 1.5 Innovations in 2005 6

2 Architecture **9**
- 2.1 Platform-Independence 9
  - 2.1.1 Motivation 9
  - 2.1.2 Realization 10
  - 2.1.3 Supported Platforms 11
  - 2.1.4 Math Library 11
    - 2.1.4.1 Provided Data Types 11
- 2.2 Multiple Team Support 12
  - 2.2.1 Tasks 12
  - 2.2.2 Debugging Support 14
  - 2.2.3 Process-Layouts 14
    - 2.2.3.1 Communication between Processes 14
    - 2.2.3.2 Team Communication 16
    - 2.2.3.3 Different Layouts 18
  - 2.2.4 Make Engine 18
    - 2.2.4.1 Dependencies 18
    - 2.2.4.2 Realization 19
    - 2.2.4.3 Debugging and Optimization 19

2.2.4.4 Automation and Integration . . . 19

3 Modules in GT2005 21

3.1 Body Sensor Processing . . . 21
3.2 Vision . . . 22
3.2.1 Using a Horizon-Aligned Grid . . . 23
3.2.2 Color Table Generalization . . . 25
3.2.3 Camera Calibration . . . 25
3.2.4 Detecting Field Lines and the Center Circle . . . 28
3.2.5 Detecting the Ball . . . 29
3.2.6 Detecting Beacons . . . 31
3.2.7 Detecting Goals . . . 32
3.2.8 Detecting Robots . . . 36
3.2.9 Detecting Obstacles . . . 36
3.2.10 Motion Compensation . . . 37
3.3 Self-Localization . . . 37
3.3.1 Motion Model . . . 38
3.3.2 Observation Model . . . 39
3.3.2.1 Beacons . . . 39
3.3.2.2 Goals . . . 39
3.3.2.3 Edge Points . . . 40
3.3.2.4 Line Crossings and the Center Circle . . . 42
3.3.2.5 Probabilities for percepts and the overall probability . . . 43
3.3.3 Resampling . . . 45
3.3.3.1 Importance Resampling . . . 45
3.3.3.2 Drawing from Observations . . . 45
3.3.3.3 Probabilistic Search . . . 46
3.3.4 Estimating the Pose of the Robot . . . 46
3.3.4.1 Finding the Largest Cluster . . . 46
3.3.4.2 Calculating the Average . . . 46
3.3.4.3 Certainty . . . 47
3.3.5 Results . . . 47
3.4 Ball Modeling . . . 47
3.4.1 Filtering of Ball Percepts using a Particle Filter . . . 49
3.4.2 Communicated Information about the Ball . . . 51
3.4.3 Team Ball Locator . . . 52
3.4.3.1 Calculation of representative particles . . . 52
3.4.3.2 Reaction to loss of information . . . 53
3.5 Obstacle Model . . . 54
3.5.0.3 Updating the Model with new Sensor Data . . . 54
3.5.0.4 Updating the Model Using Odometry . . . 55
3.6 Collision Detector . . . 56
3.7 Player Modeling . . . 57

3.8 Behavior Control 58
3.8.1 Ball Handling 60
3.8.1.1 Approaching 60
3.8.1.2 Dribbling 62
3.8.1.3 Grabbing 64
3.8.1.4 Carry Ball 65
3.8.1.5 Pushing Backwards 66
3.8.1.6 Kicking 67
3.8.1.7 Zones for Ball Handling 68
3.8.1.8 Transitions Between Ball Handling Behaviors 70
3.8.2 Navigation and Obstacle Avoidance 71
3.8.2.1 Walking to a Position 71
3.8.2.2 Walking to a Far Away Ball 72
3.8.2.3 Positioning 72
3.8.3 Player Roles 74
3.8.3.1 Striker 74
3.8.3.2 Supporters 74
3.8.3.3 Goalie 78
3.8.3.4 Dynamic Role Assignments 80
3.8.4 Game Control 81
3.8.5 Cheering and Artistry 83
3.9 Motion 84
3.9.1 Walking 86
3.9.1.1 Approach 86
3.9.1.2 Parameters defining a gait 86
3.9.1.3 Combining several optimized parameter sets 87
3.9.1.4 Odometry correction 88
3.9.1.5 Inverse kinematics 89
3.9.1.6 Gait Evolution 93
3.9.2 Special Actions 94
3.9.3 Head Motion Control 96
3.9.3.1 Geometric Considerations 96
3.9.3.2 Head Path Planner 99
3.9.3.3 Landmark State 99
3.9.3.4 State Machine 99
3.9.3.5 Basic Behaviors 101

4 Challenges **103**
4.1 almost SLAM Challenge 103
4.1.1 The Challenge Rules 103
4.1.2 Problem Analysis 104
4.1.3 Our Approach to the almost SLAM Challenge 104
4.2 Variable Lighting Challenge 105

4.2.1 The Challenge Rules . . . 105
4.2.2 Problem Analysis . . . 106
4.2.3 Our Approach to the Variable Lighting Challenge . . . 106

5 Tools **109**
5.1 RobotControl . . . 109
5.2 Simulator . . . 111
5.2.1 Simulation Kernel . . . 111
5.2.2 User Interface . . . 113
5.2.3 Controller . . . 115
5.3 Ceiling Camera . . . 115
5.4 MakeStick . . . 117
5.4.1 Installation . . . 118
5.4.2 Usage . . . 118
5.4.2.1 Actions . . . 118
5.4.2.2 Copy Options . . . 119
5.4.2.3 Player Role . . . 119
5.4.2.4 Team . . . 119
5.4.2.5 Location . . . 120
5.4.2.6 WLAN . . . 120
5.4.2.7 GT Cam . . . 121
5.5 Universal Resource Compiler . . . 121
5.5.1 Motion Description Language . . . 121
5.6 Depend . . . 123
5.7 Emon Log Parser . . . 124

6 Conclusions and Outlook **125**
6.1 The Competitions in Osaka . . . 126
6.2 Future Work . . . 126
6.2.1 Humboldt-Universität zu Berlin . . . 126
6.2.2 Technische Universität Darmstadt . . . 127
6.2.3 Universität Bremen . . . 128
6.2.4 Universität Dortmund . . . 128

7 Acknowledgments **131**

A Installation **133**
A.1 Required Software . . . 133
A.2 Source Code . . . 133
A.2.1 Robot Code . . . 134
A.2.2 Tools Code . . . 135
A.3 The Developer Studio Workspace GT2005.dsw/.sln . . . 135

B Getting Started **137**
B.1 Configuration Files . . . 137
B.1.1 location.cfg . . . 137
B.1.2 coltable.cfg . . . 137
B.1.3 camera.cfg . . . 138
B.1.4 player.cfg . . . 138
B.1.5 robot.cfg . . . 139
B.1.6 wlanconf.txt . . . 139
B.1.7 coeff.c$\{u,v,y\}$ . . . 139

C Simulator Usage **141**
C.1 Introduction . . . 141
C.2 Getting Started . . . 142
C.3 Scene View . . . 142
C.4 Information Views . . . 143
C.4.1 Image Views . . . 143
C.4.2 Color Space Views . . . 145
C.4.3 Field Views . . . 146
C.4.4 Xabsl Views . . . 146
C.4.5 Sensor Data View . . . 147
C.4.6 Timing View . . . 148
C.5 Scene Description Files . . . 148
C.6 Console Commands . . . 149
C.6.1 Initialization Commands . . . 149
C.6.2 Global Commands . . . 150
C.6.3 Robot Commands . . . 151
C.7 Examples . . . 155
C.7.1 Recording a Log File . . . 155
C.7.2 Replaying a Log File . . . 156
C.7.3 Remote Control . . . 157

D Extensible Agent Behavior Specification Language **159**
D.1 Hierarchies of Finite State Machines . . . 159
D.1.1 The Option Graph . . . 159
D.1.2 State Machines . . . 161
D.1.3 Interaction with the Environment . . . 164
D.1.4 The Execution of the Option Hierarchy . . . 164
D.2 Behavior Specification in XML . . . 165
D.3 The XABSL Language . . . 168
D.3.1 Symbols, Basic Behaviors, and Option Definitions . . . 168
D.3.2 Options and States . . . 169
D.3.3 Boolean and Decimal Expressions . . . 171
D.3.4 Agents . . . 172

D.4 Mechanisms and Tools 175
D.4.1 File Types and Inclusions 175
D.4.2 Document Processing 176
D.5 The XabslEngine Class Library 177
D.5.1 Running the Xabsl2Engine on a Specific Target Platform 178
D.5.2 Registering Symbols and Basic Behaviors 178
D.5.3 Creating the Option Graph and Executing the Engine 179
D.5.4 Debugging Interfaces 180
D.6 Discussion 182
D.7 YABSL 184

E Processes, Senders, and Receivers **187**
E.1 Motivation 187
E.2 Creating a Process 187
E.3 Communication 189
E.3.1 Packages 189
E.3.2 Senders 190
E.3.3 Receivers 191

F Streams **193**
F.1 Motivation 193
F.2 The Classes Provided 194
F.3 Streaming Data 195
F.4 Making Classes Streamable 196
F.4.1 Streaming Operators 197
F.4.2 Streamable 199
F.4.3 Streaming Protocols 199
F.4.4 Streaming using *read()* and *write()* 200
F.5 Implementing New Streams 201

G Debugging Mechanisms **205**
G.1 Exchanging Messages Between Robots and PC 205
G.1.1 Message Queues 205
G.1.2 Distribution of Debug Messages 207
G.1.3 Sending Messages 208
G.2 Message Queues and Processes 208
G.2.1 Message Handling 209
G.2.2 The Process Debug 210
G.3 Common Debug Mechanisms 211
G.3.1 Debug Requests 211
G.3.1.1 Macros 211
G.3.1.2 Debug Request Architecture 212
G.3.1.3 Polling 212

G.3.2 Debug Images . . . 213
G.3.2.1 Macros. . . . 213
G.3.3 Debug Drawings . . . 214
G.3.3.1 Macros . . . 214
G.3.3.2 Drawing Manager . . . 215
G.3.4 Debug Data . . . 215
G.3.5 Stopwatch . . . 216

H Mechanisms for Modules and Solutions 217
H.1 Division of Information Processing into Tasks . . . 217
H.2 Defining Modules and Solutions . . . 219
H.2.1 Class Module . . . 219
H.2.2 Interface Classes . . . 220
H.2.3 Base Classes For Modules . . . 221
H.2.4 Selecting Solutions . . . 222
H.2.5 Administration of Modules . . . 224
H.3 Modules and Processes . . . 225
H.3.1 Embedding Modules into Processes . . . 225
H.3.2 Representations in Processes . . . 226

I Programming RobotControl 227
I.1 The Application . . . 227
I.2 General Structure . . . 229
I.3 Message Handling . . . 229
I.4 Manager . . . 230
I.5 User Controls . . . 230

Chapter 1

Introduction

1.1 History

The GermanTeam is the successor of the Humboldt Heroes who 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 four universities: Humboldt-Universität zu Berlin, Universität Bremen, Technische Universität Darmstadt, and Universität Dortmund. For the RoboCup 2001, the Humboldt Heroes were only actively joined by Bremen and Darmstadt in the last two to three months before the world championship in Seattle.

In 2002 the Universität Dortmund also joined in. The system presented in this document is the result of the work of the team members of all four universities. Each of these four groups participated individually in the German Open 2002, 2003, 2004, and 2005 in Paderborn. In 2005, the Microsoft Hellhounds (from Dortmund) also competed in the Robogames in San Francisco, and against the US Open champion for the US/Germany Championship in Atlanta. The national team, the “GermanTeam”, is formed each year after the German Open and participated in the RoboCup World Championships in Fukuoka (2002), Padova (2003), Lisbon (2004) and Osaka (2005).

The four teams are the *Aibo Team Humboldt* (Berlin, winner of the GermanOpen 2001 and 2004), the *Bremen Byters*, the *Darmstadt Dribbling Dackels* (winner of the GermanOpen 2002 and 2003), and the *Microsoft Hellhounds* (Dortmund, winner of the GermanOpen 2005, Robogames 2005 and US/Germany Championship 2005).

The GermanTeam won the RoboCup Soccer World Championship and reached the 3rd place in the Technical Challenge competition in 2005. Previously, it won the RoboCup Soccer World Championship 2004, made it to the quarter finals in 2002 and 2003 and won the 2003 Technical Challenge.

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

A main interest of the researchers at Humboldt-Universität zu Berlin (Aibo Team Humboldt) are robotic architectures for autonomous robots based on mental models and the development of complex behavior control architectures. At the moment there coexist two different architectures for behavior control that were developed in Berlin. These architectures proofed to be very successful during the RoboCup competitions in the Sony and in the Simulation League. The benefits of both architectures are to be integrated into an unified architecture. It will be investigated, how the capability to learn long term behaviors can be added to this architecture. The implemented behaviors are just as important as the behavior architecture. The team in Berlin wants to analyze existing behaviors and their impact on the team play to find out which models and which kinds of communication are essential for effective cooperation. For a better environment modeling and self localization we developed techniques which are capable to gain knowledge from negative information. The improvement of interacting and handling objects of the agent's environment was another key part for a better ball handling. Furthermore methods of machine learning are to be applied for optimizing ball handling behaviors.

A second focus of the research activities in Berlin is perception. We want to develop an architecture for robot vision that enables robots to recognize their environment independent of the present lighting conditions and that integrates knowledge about the environment that is collected during the robot operates. This architecture will contain image processing methods, modeling techniques, and ways to control the camera that are inter-coordinated. Collision detection techniques will help the robot recognize disturbances in its intended motions.

For better development and testing operations we created a new debugging tool as well as new debugging mechanisms which enable the programmer to quicker testing and better monitoring the robot.

1.2.2 Technische Universität Darmstadt

The RoboCup scenario of soccer playing, cooperating, autonomous robots represents an extraordinary challenge for the design, control and stability of *legged* robots. In a game, fast, goal-oriented motions must be planned autonomously and implemented online which not only preserve the robot's stability but can also be adapted in real-time to the quickly changing enviroment. Existing design and control strategies for quadrupedal and especially humanoid robots with many degrees of freedom and many actuated joints can only meet these challenges to a small extent. A long-term research goal of the team at TU Darmstadt is to consider the highly nonlinear physical dynamical effects of legged robots on all levels of a reactive-deliberative control architecture realizing autonomous robot behaviour. Many subproblems in autonomous locomotion and behavior control which can be decoupled for wheeled robots cannot be considered independently

for legged robots. E.g., in autonomous navigation localization with an actively movable camera cannot be considered indepently from motion control of legs and arms which moves the robot towards a goal position while maintaining stability of the gait. The problem of generating and maintaining a wide variety of fast, statically or dynamically stable legged locomotions is predominant for all types of motions during a soccer game and even more important for humanoid robots than for four-legged robots. Since its origin in 2001 the Darmstadt Dribbling Dackels participated in and contributed to the GermanTeam in the 4-Legged-League. Since 2004 the Darmstadt Dribblers participate in the Humanoid-League.

The group in Darmstadt is developing tools for an efficient kinetical modeling and simulation of four-legged and humanoid robot dynamics in three dimensions 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 haven been applied to investigate and implement new, fast, and stable locomotion in upright position for the ERS-210 model where the kinematical and kinetical data had been provided by Sony (see CLAWAR 2003), as well as for a 80 cm high humanoid robot prototype (see IEEE ICRA 2003, IEEE/RAS Humanoids 2003). Starting from the experience in the Four-Legged-League the group in Darmstadt is investigating a software architecture as well as navigation and behavior control methods for soccer-playing, autonomous humanoid robots but also for solving complex tasks quite different from a soccer scenario by teams of different types of autonomous robots (e.g. wheeled and humanoid).

Contributions of the Darmstadt Dribbling Dackels to the GermanTeam code include new algorithms for the recognition and modeling of other players, further improvements on the existing algorithms of image processing and self-localization, and a global vision system using a ceiling camera.

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 only be regarded individually. Rather it should be assumed that they form a self-organizing group with common goals, which contradict the agent's own aims. 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 of agents in general. However, the RoboCup is an ideal test-bed for the methods to be developed. 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 Four-Legged 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 Dortmund University in 2006 will keep its focus on the research fields of learning algorithms, modeling, and behavior control with applications in robot soccer.

We plan to further improve our ceiling cam global vision and use it for automated behavior learning tasks. As it has been agreed that next year the league won’t undergo any major rule changes, we feel that strategical analysis and team-play will have a major role in distinguishing the perfomance of one team from the others.

From the situation classification achieved with the help of such an external tool, the robots must be able to perform a matching with their internal world representation, so it is crucial to be able to model the noisy sensor data in an effective way. Thus, we will continue to refine our sensor models for localization, ball and robot tracking.

1.3 Contributing Team Members

At the four universities providing active team members, many people contributed to the German-Team:

1.3.1 Aibo Team Humboldt (Humboldt-Universität zu Berlin)

“Diplom” Students. Benjamin Altmeyer, Thomas Krause, Michael Spranger.

PhD Students. Matthias Jüngel (Aibo Team Humboldt team leader), Daniel Göhring, Jan Hoffmann.

Professor. Hans-Dieter Burkhard.

1.3.2 Darmstadt Dribbling Dackels (Technische Universität Darmstadt)

“Diplom” Students. Marko Borazio, Ronnie Brunn, Marc Dassler, Michael Kunz, Tobias Oberlies, Marcus Schobbe, Dorian Scholz, Patrick Stamm, Patrick Winkler.

PhD Student. Max Risler.

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

1.3.3 Bremen Byters (Universität Bremen)

"Diplom" Students. Jan Aschmann, Eike Carls, Jan Carstens, Oliver Herdin, Heng Jiang, Meike Klose, Torsten Kohlmann, Tim Laurinat, Marc Messing, Judith Müller, Sven Oesau, Oliver Sanz, Boris Schwetzler, Mark Siska, and Michael Weber.

PhD Student. Tim Laue.

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

1.3.4 Microsoft Hellhounds (Universität Dortmund)

"Diplom" Students. Markus Broszeit, Stefan Czarnetzki, Timo Diekmann, Denis Fissler, Jörn Hamerla, Thorsten Kerkhof, Thomas Kindler, Matthias Meyer, Carsten Rohde, Bastian Schmitz, Carsten Schumann, Xuan-Anh Tran, Michael Wachter, Tobias Wegner, Judith Winter, Christine Zarges.

PhD Students. Ingo Dahm, Matthias Hebbel, Walter Nisticò.

Professor. Uwe Schwiegelshohn.

1.4 Structure of this Document

This document gives a complete survey over the software of the GermanTeam. It does not only describe last year's innovations but the entire system. This is due to the fact that this report also serves as documentation for new members of the GermanTeam.

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 used in the *variable lighting challenge* and the *almost SLAM challenge*.

Only 60% of the approximately 360,000 lines of code that were written by the GermanTeam for the RoboCup 2005 are actually running on the robots. The other 40% were invested in powerful tools that provide sophisticated debugging possibilities including a 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 2005 and giving an outlook on the future perspectives of the GermanTeam in 2006 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 appendices C describes the usage of the simulator of the GermanTeam. Appendix D contains a detailed documentation of the behavior engine used by the GermanTeam in Osaka. Afterwards, the GermanTeam's abstraction of *processes*, *senders*, and *receivers* is presented in Appendix E, followed by Appendix F on *streams*, Appendix G on the debugging support, and Appendix H on the way how the GermanTeam supports different implementations for a single task in parallel. The final appendix describes how the main debugging tool *RobotControl* works.

1.5 Innovations in 2005

For those who already read our team report from 2004 [48], pointers to the main innovations achieved in 2005 are given here.

Image Processing. The image processor is now able to detect real lines instead of only points on lines (cf. Sect. 3.2.4). In addition, the center circle is recognized as separate feature, and different kinds of line crossings (X, T, L) are detected. The image processor also recognizes more than a single ball percept (cf. Sect. 3.2.5), allowing the world modeling to rule out possible misdetections (e. g. outside the field) later. In general, a reliability is determined for each ball percept. The goal recognizer is now shape-based, i. e. it exploits the fact that the goal is a rectangular structure (cf. Sect. 3.2.7). There now also exists a robot recognizer (cf. Sect. 3.2.8).

World Modeling. The self-locator now employs line crossings and the center circle (cf. Sect. 3.3), improving the precision of the localization, especially in the corners. The ball locator now uses a particle filter, both for the per-robot ball model and the team-wide ball model (cf. Sect. 3.4). The latter is realized by drawing particles from the individual ball models of each robot. Teammates and opponents are modeled using a grid (cf. Sect. 3.7).

Behavior Control. The XML-based behavior modeling language XABSL now has a front end called YABSL, that allows behaviors to be described in a much more compact and readable way (cf. App. D.7).

Motion Control. The new walking engine supports more degrees of freedom for the generation of the trajectories of the feet (cf. Sect. 3.9.1). It is able to continuously interpolate between specialized parameter sets for different walking directions to improve omni-directional walking. The parameter sets used in Osaka were developed using Evolutionary Algorithms and a ceiling camera providing ground truth. In addition, a three-dimensional lookup table for precise odometry was also learned using the ceiling camera.

Infrastructure. The remote debugging tool RobotControl was streamlined and completely rewritten in C# (cf. Sect. 5.1). Thereby, new debugging techniques were established, e. g. the ability to monitor and modify arbitrary data structures on the robots (cf. App. G). The robots now communicate with each other using UDP broadcasts (cf. Sect. 2.2.3.2). UDP broadcasts can also be used to monitor the internal state of the robots. Two systems for establishing ground truth using ceiling cameras were developed. The one that is used in the European League is described in Sect. 5.3.

Chapter 2

Architecture

The GermanTeam is an example of a national team. The members participated as separate teams in the German Open 2002, 2003, 2004, and 2005 but formed a single team at the RoboCup in Fukuoka, Padova, Lisbon, and Osaka. 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 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. 5.2). 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 E.

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. 5.2). 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. Since it is not used anymore, it is not part of the current software.

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 spezialisations and n-dimensional), matrices (three dimensional spezialiation and n-dimensional), rotation matrices, and translation matrices (two and three dimensional). The math library also provides Datatypes for dealing with histograms, geometric objects and PID smoothing.

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.

Vector_n<T, N> is a template class for `N`-dimensional vectors of type `T`. It provides functions for addition, scalar multiplication and Euclidian length. This class can be used with functions of the *Matrix_nxn* class.

Matrix_nxn<T, N> is a template class for `N` $\times$ `N`-matrices of type `T`. It provides functions to add, multiply and invert matrices, to multiply matrices with vectors and to solve linear equations.

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 G. 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 which 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 four levels:

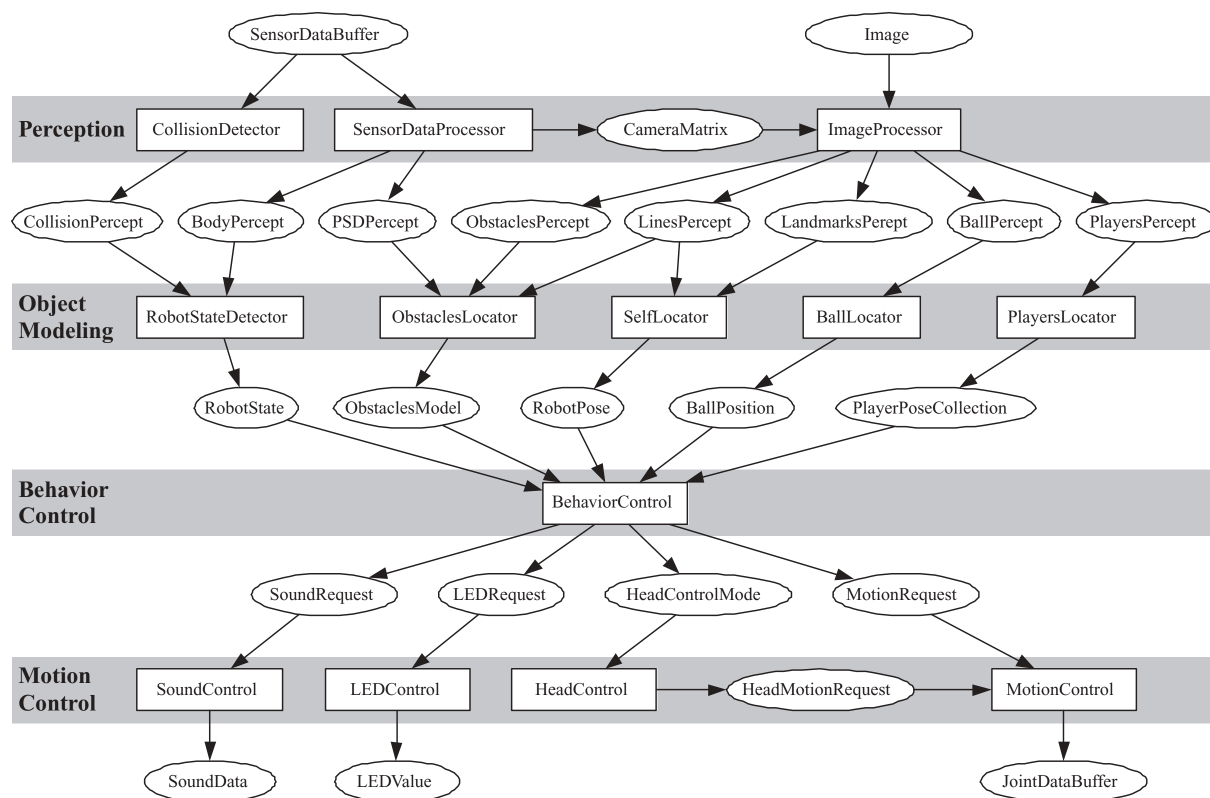


Figure 2.1: The tasks identified by the GermanTeam 2005 for playing soccer.

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 fell down.

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 developers can switch between them at runtime. In addition, it is 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 [9]: *debug requests* and *solution requests*. The debug request architecture has been completly rewritten, compared to the code implemented in GermanTeam 2004 [48]. The system manages two sets of information, the current state of *debug requests*, and the currently active solutions. Debug request work similar to C++ preprocessor defines, but they can be toggled at runtime. A special infrastructure called *message queues* (cf. Sect. G.1.1) 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 requests 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. 5.1). For example, motion requests can directly be sent to the robot, images, text messages, and even drawings 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. 5.2).

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 2005 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 [9], i. e., a blackboard architecture [29] 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 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 Team Communication

In the last years the communication between the robots had to be done via the *TCPGateway* provided by Sony. Because of the bad latency of the *TCPGateway* the league committee decided to allow direct communication between the robots.

So we implemented an UDP based protocol for the team communication and a TCP based protocol for the debug-communication to RobotControl.

For both of these protocols the process-framework had to be extended to handle this communication.

Putting IP-communication into the process-framework. The functionality for IP based communication in OPEN-R is provided by the *ANT-library* [16]. The basic concept of this library is the "endpoint" which provides functionality for sending and receiving data over the network interface. There are different endpoints for TCP-Streams and UDP-Packages and also for higher level protocols like HTTP.

The ANT-library is used by sending some messages for connecting, disconnecting, sending and receiving to the endpoint. This can be done either blocking or non-blocking, which means that they return after doing their work or they return at once and raise an event by calling an ntry-point of the process object.

All this functionality is encapsulated in the process-framework by the *IPEndpoint*, *TCPEndpoint* and *UDPEndpoint* classes.

Debug-communication to RobotControl. For debug-communication to RobotControl a TCP-based protocol is used. The data sent is the same we sent last year by using the *TCPGateway* and the *router* [47], but now these two programs are not used any more.

To minimize the necessary changes on the code, *NetSenders* and *NetReceivers* are introduced. They have the same interface as the normal *senders* and *receivers* used for inter-process communication (cf. App. E.3), but send the data over the network by use of a protocol handler. So nothing had to be changed in debug messages generation on the robot or the handling of the debug messages in *RobotControl*. Actually, with the use of some macros the communication between RobotControl and its internal simulator is still the same.

In the cognition process the debug messages are collected in a *MessageQueue* (cf. App. G.1.1) which is sent by normal inter-process communication to the debug process where it is merged with the *MessageQueue* from the motion-process. Now, as the outgoing message queue is also derived from NetSender, the content of the MessageQueue is streamed into a memory buffer by the NetSender. The NetSender then calls the *TCPHandler*. This class is derived from *TCPEndpoint* and handles the actual protocol for sending the memory buffer over the network by first sending the size of the memory buffer and after that the memory buffer itself.

When there is incoming data on the connection, Aperios calls the entry-point for receiving data in the process-framework. This entry-point indirectly calls *onReceive()* in the *TCPHandler* where first the size is read and after that the incoming-data is collected into a memory buffer. If all data is received, the NetReceiver is called which is using a streaming operator to read this data back into the incoming message queue.

Team communication between the robots. In our approach for team communication every robot sends information about its world model and its behavior related data to every other robot of its team. There are two ways to spread this information among the team. The first way is an UDP-broadcast which sends the data to all other robots and computers on the same network, but in this case also the opponent could use this data. The second way is to send single-cast packets. In this solution every robot sends out a separate packet to every other robot in the team. Last year we used the second way, but this year we used broadcasting mainly because of the lesser bandwidth needed.

Therefore the data to send is also streamed to a memory buffer by the use of *NetSender*s and *NetReceiver*s as it is done for debug-communication. But this time it is sent via an UDP-based protocol by first sending the size of the buffer and then the data itself. Due to the limitations of the UDP-protocol a maximum of 1400 bytes is sent.

Like already mentioned we switched form using unicast-packets to broadcast-packets this year. Therefore the format of the packages was changed so that it contains the information for all other robots of the team.

Since every robot needs the IP-address of each other robot in his team, the “Dog-Discovery-Protocol” (DDP) was developed last year. Every 2 seconds each robot sends an UDP-broadcast packet to every other robot on the network. This packet contains a team-identifier and the actual team color (red or blue) of the robot. Every time a teammate receives a DDP packet, it checks for the right team-identifier and team color and adds this robot in the list of valid teammates. Only the packages from the ip-addresses on this list are accepted for the team-communication. If for some reason the robot did not receive any packet from the teammate for 10 seconds, it removes the robot from the list.

This communication scheme has proved to be very robust in testing during development and for real robot-soccer games on several events. Each robot is able to quickly find teammates and to communicate with them. When the robot runs out of battery or crashes because of a programing bug, the other robots still communicate among each other. After rebooting the robot reintegrates himself into the team communication within seconds.

On big RoboCup events like the German-Open or the world championship there usually are many wireless LAN networks in a very small area which interfere with each other. In this environment the packet-loss is very high and some of the data the robot sends is lost. Therefore data is sent every 100 ms so that the data received from other robots is as current as possible.

Special handling of the GameController messages. This year the “GameController” was not longer provided by Sony. It was completely rewritten by William Uther form UNSW and uses UDP-broadcasts to send the current score, penalties, time, etc. to the robots. For an easy integration into the framework a ”UDPBroadcastHandler” was created. This protocol-handler just calls the *netReceiver* which uses the corresponding streaming-operator to copy the data from the memory buffer into the *GameControlData* data-structure.

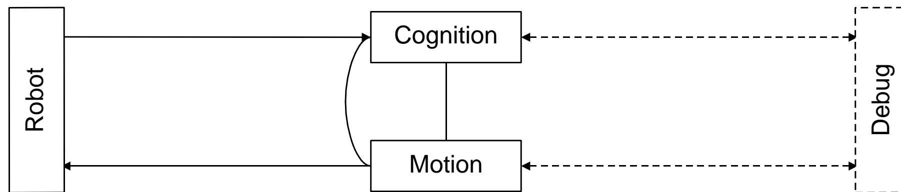


Figure 2.2: The process layout of Humboldt 2002, since RoboCup 2002 used by the whole GermanTeam.

2.2.3.3 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 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. But the source files have to be grouped based on the selected process layout for compilation and linking tasks, because in one layout, e. g., several files may 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 ongoing modifications, expansions, testing, and fine-tuning. Therefore, one major goal of implementing a *make engine* was to execute only those 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 the competitions in 2003, 2004 and 2005, a simple speed optimized pre-processor called *Depend* (*GT2005/Src/Depend*) (cf. Sect. 5.6) 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 (*$PDIR*, located in *GT2005/Build*) to avoid conflicts between different builds as well as the compulsion for a complete rebuild after changing the process layout, build variant, or compiler. Each such *$PDIR* contains the object files in the same subdirectory structure as the source code as well as a subdirectory called *bin* containing the resulting binaries. 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* (cf. Sect. 5.6) is used to completely generate all dependencies for the chosen build target in *GT2005/Build/*/*/depends.incl* each time compilation or linking takes place. Even with several hundreds of source files, this takes only a few seconds.

After that, all object files required for a certain binary can easily be determined by *Depend*. 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/R7000 or to simplify debugging e. g. by adding 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(6) project files easily from these dependencies since 2003. This simplifies maintaining the consistency between the source code tree and the project files. All scripts necessary for that can be found in *GT2005/Make/Dsp_generation/*. Since 2004 project files for Visual Studio 2003.NET can be generated too. Since 2005, only Visual Studio 2003.NET is supported.

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 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 GT2005

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 H.2.4.

This chapter describes most of the modules that were implemented. For some solutions only the chosen approach for the competition in Osaka is figured out in detail.

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).

For the calculation of the tilt and roll of the robot's body, there are two possibilities. They can be calculated by the measurements of the acceleration sensors or by the actual angles of the leg joints. By comparing long term averages and short term averages of the tilt and roll angle, 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.

For the ERS-7 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 the lower head tilt angle

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 y-axis by the upper tilt angle

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

8. translation along the positive z-axis by the distance along the z-axis between the center of upper tilt 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 each 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 images are processed using the resolution of 208×160 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 a relative high resolution, while the rest of the image can be scanned using a wider spaced grid.

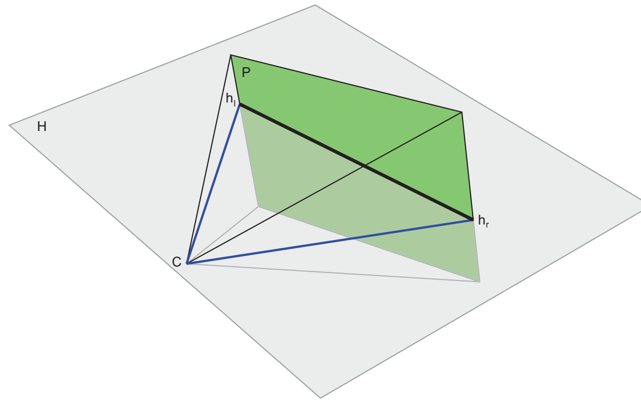


Figure 3.1: Construction of the horizon

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 as well as the current speed and direction of the motion of the camera.

3.2.1 Using a Horizon-Aligned Grid

Calculation of the Horizon. For each image, 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. 3.1). 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 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

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

with only z_l and z_r unknown. Let

$$i = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \quad (3.2)$$

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

$$R \cdot i = h_l \quad (3.3)$$

with the rotation matrix

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

leads to

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

In the same way follows

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

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.

Each grid line is scanned pixel by pixel from top to bottom and from left to right respectively. 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 for a ball, a sequence of some pink pixels is an indication for a beacon, an (interrupted) sequence of sky-blue or yellow pixels is an indication for a goal, a sequence white-green or green-white is an indication of 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 there were some pixels of the wrong class in between.

To speed up the object detection and to decrease the number of false positives, essentially two different grids are used. The main grid covers the area around and below the horizon. It is used to search for all objects which are situated on the field, i. e. the ball, obstacles, other robots, field lines, and the goals (cf. Fig. 3.2a). A set of grid lines parallel to and in most parts over the horizon is used to detect the pink elements of the beacons (cf. Fig. 3.2b).

Through these separated grids, the context knowledge about places of objects is implemented.

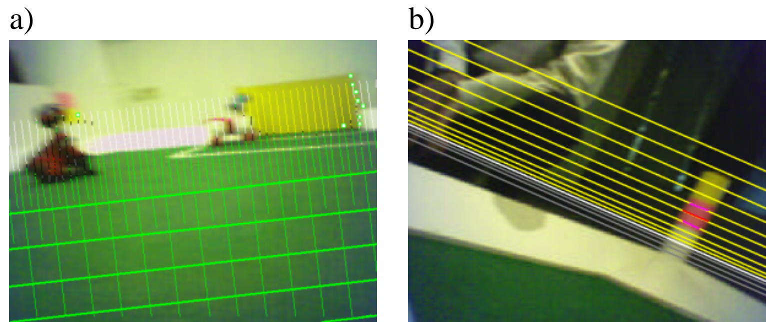


Figure 3.2: Different scanlines and grids. a) The main grid which is used to detect objects on the field. b) The grid lines for beacon detection.

3.2.2 Color Table Generalization

One of the key problems in the RoboCup domain is to reach a high degree of robustness of the vision system to lighting variations, both as a long term goal to mimic the adaptive capabilities of organic systems, as well as a short term need to be able to deal with unforeseen situations which can arise at the competitions, such as additional shadows on the field as a result of the participation of a packed audience. While several remarkable attempts have been made in order to achieve an image processor that doesn't require manual calibration (see for example [30], [55]), at the moment traditional systems are more efficient for competitions such as the RoboCup. Our goal here was to improve a manually created color table, to extend its validity to lighting situations which weren't present in the samples used during the calibration process, or to resolve ambiguities along the boundaries among close color regions, while leaving a high degree of control over the process in the hands of the user. To achieve this, we have developed a color table generalization technique which uses an exponential influence model. Further details on our solution can be found in [45].

3.2.3 Camera Calibration

Since the camera is the main exteroceptive of the ERS7 robot, a proper camera analysis and calibration is necessary to achieve percepts as accurate as possible. While the resolution of the ERS7's camera is $\approx 31\%$ higher compared to the model equipped on the Aibo ERS210 robot, our tests revealed some specific issues which weren't present in the old model:

- lower light sensitivity, which can be compensated by using the highest gain setting, at the expense of amplifying the noise as well;
- vignetting effect (radiometric distortion), which makes the image appear dyed in blue at the corners (cf. Fig. 3.3).

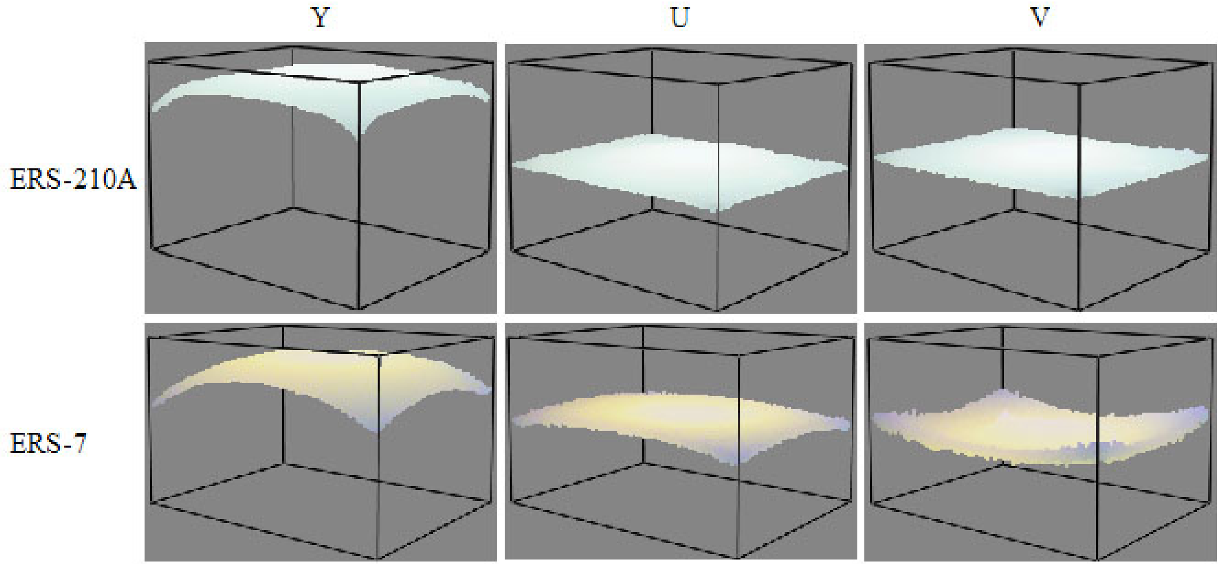


Figure 3.3: Comparison of the image of a white wall, captured by the ERS210 and ERS7 cameras. Y represents the brightness, U and V are the color bands of the image, which is provided by the camera in the YUV color space. The ERS7 images have a chromatic distortion at the corners; they also appear darker even if in this example, the camera gain was set to *high*, while in case of the ERS210A the setting was *low*

Geometric camera model. Instead of the classical and simple "pinhole" model, to describe the camera geometry we decided to use a more complete model which would also include the geometrical distortions of the images due to lens effects, called the *DLT model* (see [4].) This model takes into account the lack of orthogonality between the image axes s_θ, the difference in their scale (s_x, s_y), and the shift of the projection of the real optical center (principal point) from the center of the image (u_0, v_0):

- world coordinates, homogeneous:

$$p = \begin{bmatrix} x_w & y_w & z_w & 1 \end{bmatrix} \tag{3.7}$$

- image coordinates:

$$q = \begin{bmatrix} f \cdot x_i & f \cdot y_i & f \end{bmatrix} \tag{3.8}$$

- intrinsic parameters:

$$K = \begin{bmatrix} \sigma_x & \sigma_\theta & u_0 \\ 0 & \sigma_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.9}$$

- extrinsic parameters:

$$M = \begin{bmatrix} \ddots & \vdots & \iddots & \vdots \\ \cdots & R & \cdots & T \\ \iddots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.10}$$

- then the result is:

$$q = K \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot M \cdot p \tag{3.11}$$

In addition to this, we have also decided to evaluate augmented models which include radial and tangential non-linear distortions. Let ρ_1, ρ_2, ρ_3 be the coefficients of a 6th order radial distortion model, τ_1 and τ_2 the coefficients of a tangential distortion model, $\overline{x} = x - u_0$, $\overline{y} = y - v_0$, $r^2 = \overline{x}^2 + \overline{y}^2$, (x', y') the distortion corrected coordinates, according to [24] and [43]:

$$\begin{aligned} x' &= x + \rho_1 \bar{x} r^2 + \rho_2 \bar{x} r^4 + \rho_3 \bar{x} r^6 + \tau_1 (2\bar{x}^2 + r^2) + 2\tau_2 \bar{x}\bar{y} \\ y' &= y + \rho_1 \bar{y} r^2 + \rho_2 \bar{y} r^4 + \rho_3 \bar{y} r^6 + \tau_2 (2\bar{y}^2 + r^2) + 2\tau_1 \bar{x}\bar{y} \end{aligned} \tag{3.12}$$

In order to estimate the parameters of the aforementioned models for our cameras, we used a Matlab toolbox from Jean-Yves Bouguet (see [6]). The results showed that the coefficients (s_x, s_y, s_θ), are not needed, as the difference in the axis scales is below the measurement error, and so is the axis skew coefficient; the shift between the principal point (u_0, v_0) and the center of the image is moderate and dependent from robot to robot, so we have used an average computed from images taken from 5 different robots. As far as the non-linear distortion is concerned, the

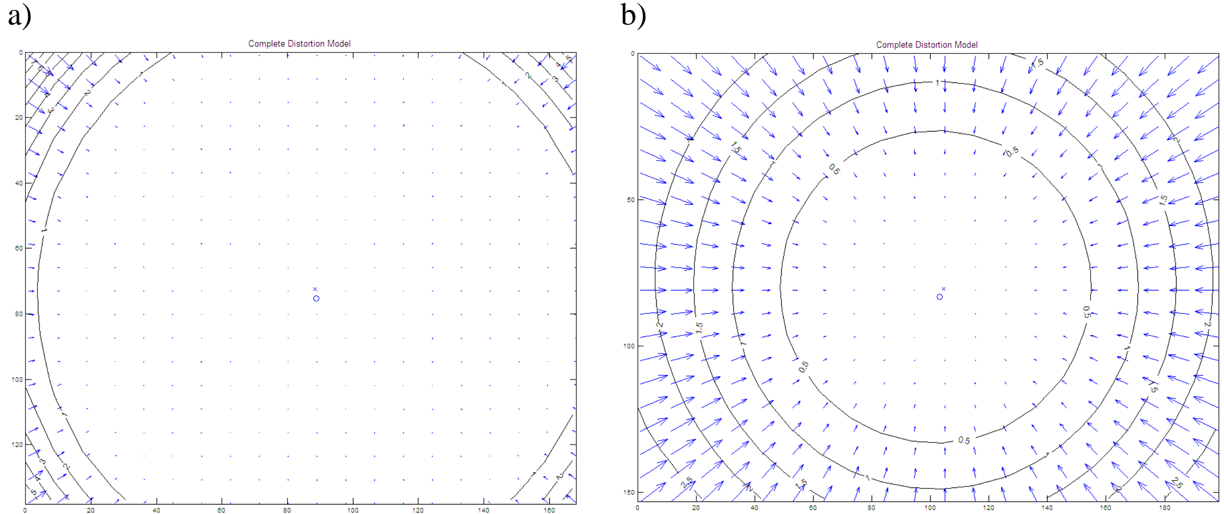


Figure 3.4: Complete camera distortion model. The blue arrows show the direction of the pixel displacement, while the magnitude is illustrated by the black iso-lines. a) ERS210: while most of the image exhibits negligible distortion, in the corners the error arrives up to 7 pixels of displacement b) ERS7: the distortion is distributed more uniformly accross the image, but the highest magnitude is significantly lower than on the old model (< 3 pixel).

results calculated with Bouguet's toolbox and illustrated in Figure 3.4 showed that on the ERS7 this kind of error has a moderate entity, and since in our preliminary tests, look-up table based correction had an impact of ≈ 3ms on the running time of the image processor, we decided not to correct it.

Radiometric camera model. As the object recognition is still mostly based on color classification, the blue cast on the corners of the images captured by the ERS7's camera is a serious hinderance in these areas. Our approach to correct such vignetting effect has been described in [45].

3.2.4 Detecting Field Lines and the Center Circle

Field lines and the center circle are part of every soccer field, be it robot soccer or human soccer. Therefore they will always be available, whereas artificial landmarks are likely to be abandoned in the near future.

As a first step towards 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 detection of points on field lines is still based on the change of the segmented color from green to white or the other way round.

The differentiation between a field line and possible borders or other white objects in the background is a bit more complicated. In most cases, these other objects have a bigger size in the image than a field line. But a far distant border for example might be smaller than a very close field line. Therefore the pixel, where the segmented color changes back from white to green after a green-to-white change before, 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 field line in the image. For the classification, these sizes are compared to the distance between the green-to-white and the white-to-green change in the image to determine if the point belongs to a field line or something else. The projection of the pixels on the field plane is also used to determine their relative position to the robot.

For every point classified as a field line, the gradient of the y-channel is computed (cf. Fig. 3.8c). This gradient is based on the values of the y-channel of the edge point and three neighbouring pixels, using a Roberts operator ([53]):

$$\begin{array}{l} s = I\left[\begin{array}{cc} x+1 & y+1 \end{array}\right] - I\left[\begin{array}{cc} x & y \end{array}\right] \\ t = I\left[\begin{array}{cc} x+1 & y \end{array}\right] - I\left[\begin{array}{cc} x & y+1 \end{array}\right] \\ |\nabla I| = \sqrt{s^2 + t^2} \\ \angle \nabla I = \arctan\left(\frac{s}{t}\right) - \frac{\pi}{4} \end{array} \tag{3.13}$$

where $|\nabla I|$ is the magnitude and $\angle \nabla I$ is the direction of the edge, that will be used by the self locator and for further processing.

In the next step, these lines are forwarded into a line finding algorithm. Clusters consisting of points describing the same edge of a field line are extracted and used to define line fragments. These line fragments are fused into field lines to calculate line crossings (cf. Fig. 3.5). All this is done using filtering heuristics to reduce the number of false positives. To extract even more information and to lessen the sensor ambiguity these crossings are characterized by specifying whether there is a field line on each side of the crossing or marked as lying outside the image.

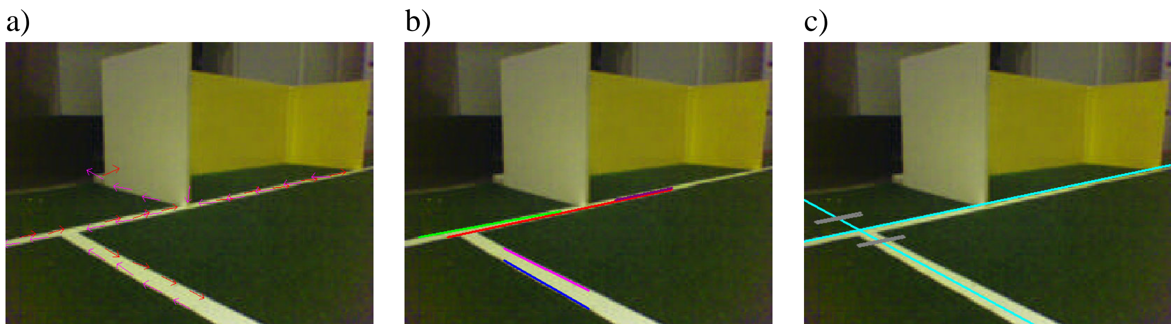


Figure 3.5: Finding lines and their crossings: a) Points on edges with direction hypotheses from gradients. b) Line fragments. c) Lines and their crossing.

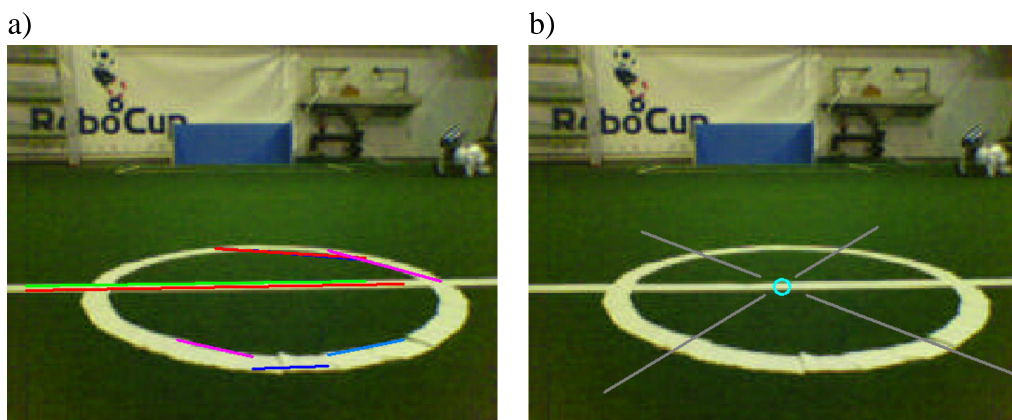


Figure 3.6: Finding the center circle: a) Line fragments describing tangents to the center circle. b) Verification of the center circle.

This information, together with the orientation of the line crossing, is finally forwarded to the self locator, which can use them as additional landmarks.

As a part of this algorithm the line fragments are used to detect the center circle. A curve in the image will result in several short line fragments describing tangents of this curve, like shown in Fig. 3.6a. Projecting these on the field and pairwise considering them as possible tangents of a circle of known radius therefore results in a center circle candidate. If enough tangents of a possible center circle are found, the candidates center is verified by the middle line crossing it and by additional scans (cf. Fig. 3.6b). Projecting this center to this middle line even increases the accuracy of the result. This way the center circle can be detected from a distance up to 1.5 m with high precision, and considering the orientation provided by the center line, it leaves only two symmetrical robot locations on the field (cf. Fig. 3.7).

3.2.5 Detecting the Ball

While scanning along the grid, runs of orange pixels are detected. Such orange runs are clustered based on their relative positions and proportion constraints, and each cluster is evaluated as a

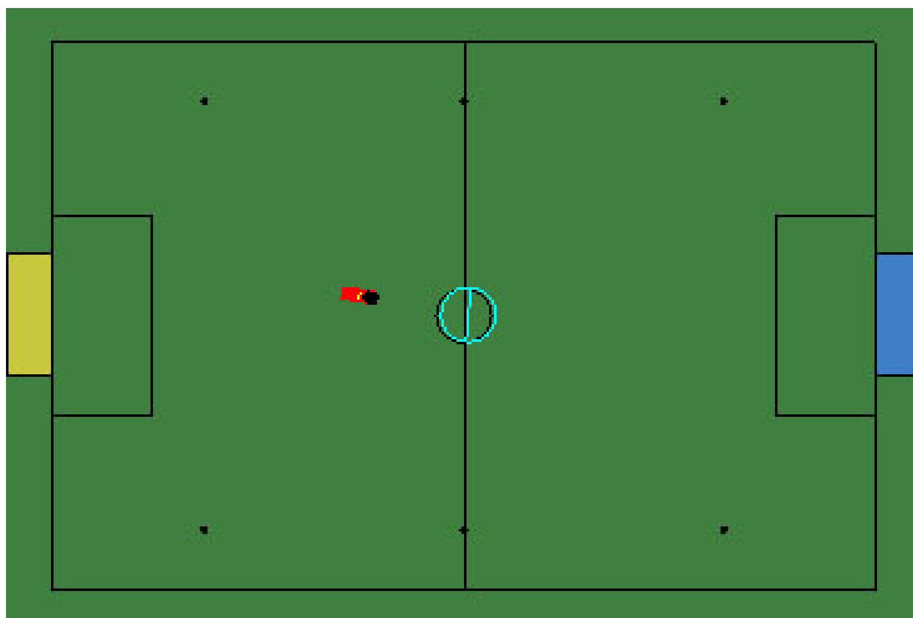


Figure 3.7: The found center circle with its orientation.

potential ball hypothesis. Starting at the center of each candidate ball cluster, the *ball specialist* scans the image in horizontal, vertical and both diagonal directions for the edge of the ball (cf. Fig. 3.8a). Each of these eight scanlines stops, if one of the following conditions is fulfilled:

- The last eight pixels belong to one of the color classes green, yellow, skyblue, red, blue.
- The color of the last eight pixels is too far away from ideal orange in colorspace. Thus the scanlines do not end at shadows or reflections whose colors are not covered by the colortable. These colors are not in the orange color class, because they can occur on other objects on the field.
- The scanline hits the image border. In this case two new scanlines parallel to the image border are started at this point.

The ending positions of all these scanlines are stored in a list of candidate edge points of the ball. If most of the scanned pixels belong to the orange color class and most of the scanlines do not end at yellow pixels (to prevent false ball recognitions in the yellow goal), it is tried to calculate center and radius of the ball from this list. For this calculations the Levenberg Marquardt method (cf. [12]) is used. At first only edge-points with high contrast which are bordering green pixels are taken into account. If there are not at least three of such points or the resulting circle does not cover all candidate edge points, other points of the list are added. In the first step of this fallback procedure all points with high contrast that are not at the border of the image are added. If this still does not lead to a satisfying result, all points of the list that are not on the image border are used. The last attempt is to use all points of the list. Depending on heuristics

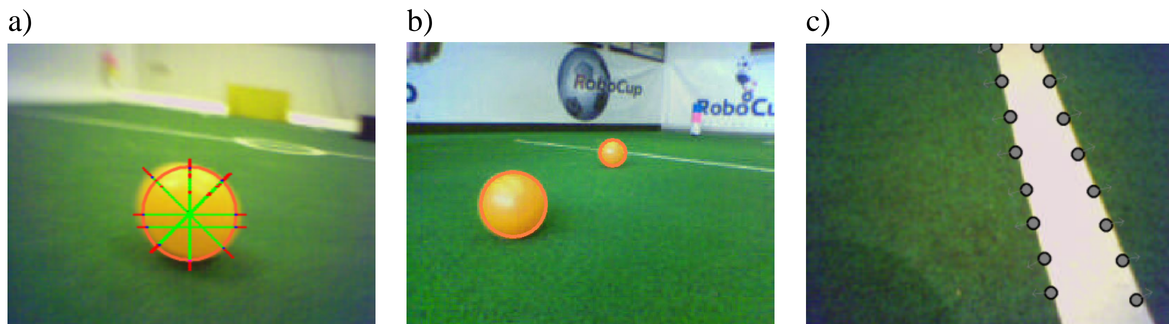


Figure 3.8: Percepts: a) The ball percept along with the scanlines of the ball specialist. b) Multiple ball percepts. c) Points on the edges of a line, also showing the computed gradients of the edges.

like roundness, percentage of orange and the expected size at that position in the image, the Ball-Specialist computes a confidence score for each ball hypothesis to be used by the particle filter based ball modelling (see Section 3.4).

3.2.6 Detecting Beacons

For detecting and analysing the beacons, we use a *beacon detector*. This module generates its own scanlines parallel to the horizon which are only looking for pink pixels. Their offsets increases with the distance to the horizon. Pink runs found this way are clustered depending on their horizontal and vertical proximity and merged into a line segment which should then be in the middle of the pink part of the flag (cf. Fig. 3.9a). From this line, 3 or 4 additional scanlines perpendicular to it are generated, looking for color of the other half of the flag and the top and bottom edges, using a modified SUSAN Edge Detector[1] (see [56]). Depending on the number of edge points found and whether some sanity critiria are met (ratio between the horizontal and vertical length of each part of a beacon, consistency between the vertical scanlines), a flag likelyhood is computed. If this value exceeds a confidence threshold, the current object is considered a beacon. In that case, the center of the pink part is calculated, and it is passed, along with the flag type (PinkYellow, YellowPink, PinkSkyBlue, SkyBluePink) to the *flag specialist*, which will perform more investigations.

From the centre of the pink area the image is scanned parallel and vertical to the horizon, looking for the borders of the flag. This leads to a first approximation of the flag size. Two more horizontal scans each are performed the top and bottom half to refine this estimate. To determine the height of the flag, three additional vertical scans are executed (cf. Fig. 3.9b). The size of the flag is computed from the leftmost, rightmost, topmost, and lowest points found by these scans (cf. Fig. 3.9c).

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

[1] This simplified version does not perform the non-maxima suppression and binary thinning.

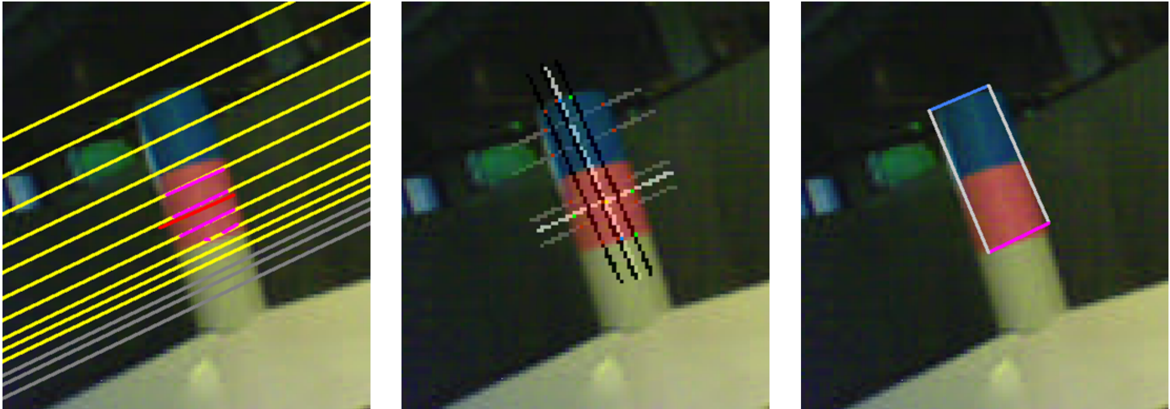


Figure 3.9: Three steps in beacon detection: (a) Scanlines searching for pink runs; the red segment shows the result of clustering; (b) The specialist detects the edges of the beacon; (c) The generated percept.

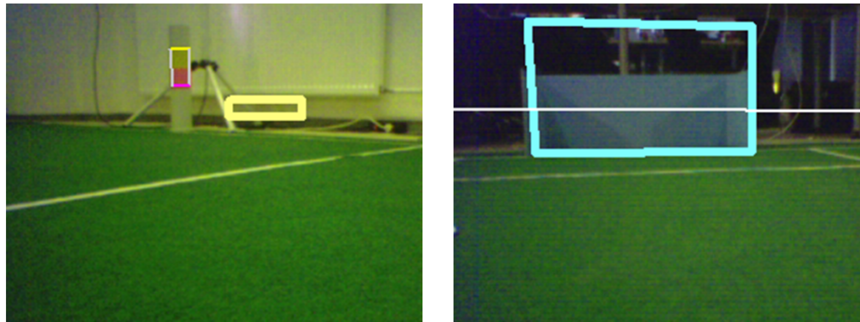


Figure 3.10: Problems with the *GT2004* goal recognition: (a) false percept, (b) size error

3.2.7 Detecting Goals

Except for the ball, the goals are the most important features to be detected by vision. They are very important landmarks for self-localization because goal percepts are unambiguous and carry information about the direction and distance relative to the current robot pose. Furthermore, compared to the beacons, goals are visible more often, especially in crucial situations where a dog is about to score. In fact, seeing the opponent's goal is in certain situations a trigger for a hard shot.

With the introduction of the new field and the removal of the field borders, we have encountered problems with the old goal recognizer (see figure 3.10). To overcome the problem of false goal sizes and ghost percepts, we pursue an entirely new approach. The key idea is to use more knowledge about the goal: the left and right edges are straight and parallel, the goal-coloured area is delimited by strong edges to neighbouring areas, and finally the goal has a minimum height in the image - absolute and with respect to the width in the image. Furthermore we know that the goal-coloured area is least likely to be obstructed by the goalie or shadows at its side edges (*goalposts*) and its top edge (*crossbar*).

To extract the goal's properties, a new analysis procedure has been implemented. It consists of a series of line scans that examine the regions of interest, namely the goalposts and the crossbar. The line scans are done to either explore the extend of a goal-coloured area (in the line's direction) or to investigate an edge. To do both in a reliable, noise-tolerant way, we use an extended colour classification on each pixel on the scan line and feed the result into an edge detection state machine.

Extended colour classification. Most modules of the *GermanTeam*'s vision are based on a look-up table for colour classification. For goal recognition, this yields the distinction between certainly goal-coloured pixels (*confirmed pixels*), non goal-coloured pixels (*outer pixels*) and unclassified pixels. However, since the look-up table is optimized to not misclassify any pixels[2], the latter case is quite common (see figure 3.11c). For those pixels we hence use a special classification based on the Mahalanobis distance of the pixel's colour c to an adaptive reference colour c_{ref}. It computes as

$$d = (c - c_{ref})^\top \, C^{-1} \, (c - c_{ref}),$$

where C is the typical covariance of the goal colour within an image. This was determined by manually selecting *all* pixels within the goal in a typical image and computing the colour covariance. For more accuracy these steps were repeated for each image from a set of reference images and C was set to the average of the resulting covariance matrices.[3] The reference colour c_{ref} is the running average over the last goal-coloured pixels. So the distance d reflects the degree of deviation to the local goal color shade taking normal variation into account.

For the state machine, the distance value of previously unclassified pixels is thresholded to distinguish between three cases: pixels with small colour deviation still likely to be a part of the goal (*inner pixels*), pixels with medium deviation as they typically appear on the edge (*intermediate pixels*) and pixels with large deviation (also regarded as *outer pixels*). Altogether this yields one of four classes for each pixel.

Edge detection. All line scanning starts from inside the goal-coloured area. The line is followed until it either crosses an edge or the colour has sloped away from the goal colour. Both can be detected by a combination of two simple counters:

- *Edge counter*: increased on outer pixels, reset on confirmed pixels.
- *Deviation counter*: increased on outer and intermediate pixels, slightly increased on inner pixels and largely decreased on confirmed pixels (clipped to non-negative values).

[2]This is strongly recommended because the goal recognition as described here works best with a rather sparse colour table. In detail, the requirements are as follows: avoid false classified pixels in the goal (however, blue pixels in the sky-blue goal are tolerated); limit green pixels to the carpet (especially blackish pixel which are common in the background must not be classified as green); avoid sky-blue and yellow in the background close to the goals.

[3]Note that the covariance C only represents the goal colour variation within one image, where the colour table also accounts for the variation across images. The manual selection of goal-coloured pixels does include pixels that are not classified as yellow, or sky-blue, respetively. And of course there is a separate matrix C for each goal colour.

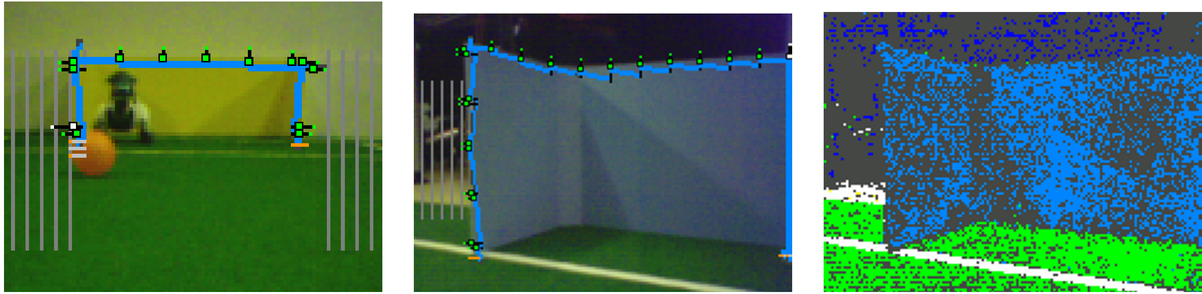


Figure 3.11: (a)-(b) Scanning strategy to analyse the goal (green dots show strong edge points, white dots show weak edges); (c) Colour table classification of image b, containing many unclassified pixels (gray).

In case of a strong edge there are many outer pixels very soon after the edge so that the edge counter reaches its threshold before the deviation counter does. The point reported as the edge is the last confirmed or inner pixel. Is there, however, a continuous colour gradient the deviation counter will reach its threshold first. The latter case should not not occur at the goal's edges, hence this is an indication that the goal-coloured area is not actually a goal. Inside the goal there might be pixels with deviating colour due to noise or shadows. Nevertheless, this would not be detected as an edge because subsequent pixels would then be goal-coloured again and reset the counters before they reach their thresholds.

Altogether this method is reliable, noise-tolerant and very efficient because it integrates the information from multiple pixels without the need for costly filter operations. Also it can determine the edge quality without explicitly computing a gradient.

Scanning strategy. The first step to recognizing goals is to discover any goal-coloured areas. To do this we look for accumulations of yellow or sky-blue pixels on the upper half of the common horizon-aligned grid (see Section 3.2). Once this has been detected, the analysis procedure described below is triggered immediately. To avoid that area to be discovered again, following grid lines intersecting with it are ignored. The same is done with grid-lines close to landmarks, to avoid their yellow or sky-blue parts to be considered as goal candidates.

From the left-to-right scanning order of vertical grid lines, we know that a goal is discovered near its left goalpost. From that point we follow the edge downwards and upwards by alternating scans towards and parallel to the edge (see figure 3.11b). Each iteration consists of two scans on horizontal lines towards the edge and two scans next to each other on vertical lines. The scans are repeated for higher reliability, choosing the inner edge point for horizontal scans (expecting an edge) and the scan that got further from the vertical ones (not expecting an edge). The scan lines along the goalpost are slightly tilted inwards to avoid crossing the edge. This might be necessary if they are not exactly perpendicular to the horizon due to motion skew (see also section 3.2.10). Furthermore their length is limited to regularly get in touch with the edge again.

In the rare case that a scan towards the edge does not find an edge within ten pixels, goalpost analysis is canceled. This may happen if only a fraction of the goal is successfully analysed and the same goal-coloured area is discovered again. In this case the starting point is not actually near the goalpost. This ensures that the left goalpost is not analysed more than once.

Figure 3.12: The goal recognition realizes that the left goalpost is occluded.

Next, starting from the top endpoint of the goalpost, we follow the crossbar. Depending on the view onto the goal, it may not be parallel to the horizon. Furthermore, with the crossbar actually being the top edge of the goal's side and back walls, it may appear curved to the left. Therefore the scans along the edge are regularly interrupted by scans towards the edge, if necessary adjusting the direction of the former. This ensures that the scan lines follow the edge as closely as possible, evading the area possibly obstructed by the goalie (see figure 3.11a).

The second goalpost is analysed similar to the first one, except that the scan upwards is not necessary. Finally, we check if there are green pixels below the goalposts' bottom endpoints, because this should not be true for other goal-coloured objects that may appear in the background or audience.

There are many cases where parts of a goalpost or the crossbar are not within the image. Given that, the scan along the edge will instead follow the image border, as long as the angle to the desired direction is less than 60 degrees.

Interpretation. Before interpretating the scanning results, it may be necessary to merge detected goal-coloured areas, e.g. if the crossbar is obstructed by a robot. However this is only done if several critia are met: both fragments need to have similar height and volume, the vertial offset must be limited, and most importantly the horizontal gap must not be more than a third of the new width.

After that, we determine which side edges of the goal hypothesises could be goalposts by checking the properties analysed before: it must have a minimum height of seven pixels (minimal height in which they appear from across the field), a minimum height relative to the other goalpost and the goal width, and the number of colour gradients (weak edges) on scan lines across the goalpost edge must be very low. Checking the straightness of the goalposts has not actually been implemented because the results were satisfactory as it is.

Based on this, we decide whether one of the hypotheses is a goal. Basically, this is considered to be true if at least one of its sides is a goalpost with green pixels below it. Altogether this reliably filters false percepts, and allows to only use both goalposts and the goal width for localization if they have both been seen (see figure 3.12).

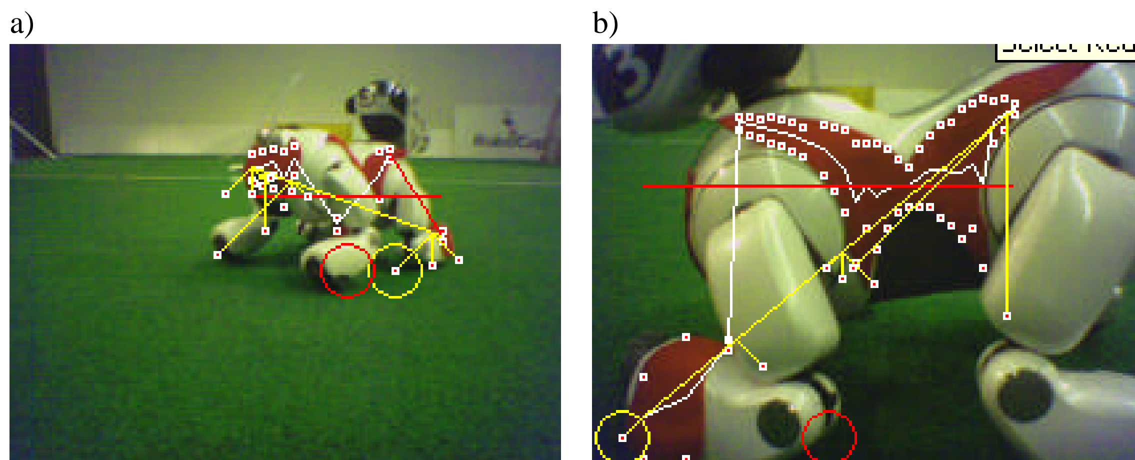


Figure 3.13: Recognition of other robots. a) A robot at 50 cm distance. The yellow circle shows the lowest foot point. The red circle shows the estimated robot position. b) A robot at 20 cm distance. The red line indicates the average height of the seen line percepts.

3.2.8 Detecting Robots

The overall target is to find the foot point of the robot to determine the relative distance to the robot. If the foot point is found, the distance is calculated by the distance-to-point method for the foot point. This is an accurate estimation of the real robot position.

To find the opponent players the standard scan lines were used. If there is a line percept with three pixel of the opponent player color, the player detection specialist is triggered. It searches beginning from the left line percept towards the right in the averaged height of all line percepts. It looks for a signature like white-red-white or red-white-red in case red is the opponents' color.

At each red-white transition further examination is started consisting of a scan line towards the ground, and one each to the left and right in the angle of 45 degrees, in order to find the lowest point of the robot. These scan lines accept the color white for the legs, the color of the jersey and colors nearby like pink and sky-blue. If a scan line gets stuck, two new scan lines were started in the remaining directions. After all scan lines have stopped, the lowest point is assumed to be the foot point of the robot. For the x coordinate the average between the left and right line percept is assumed. This works fine for robots in the distance between 20 cm and 150 cm.

During RoboCup 2005, the GermanTeam did not use the detection of other robots. Except the penalty shoot-out in the final. The penalty shooter used the player recognition to choose whether to grab the ball or shoot the ball depending whether a goalie was detected ahead.

3.2.9 Detecting Obstacles

While scanning the image 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

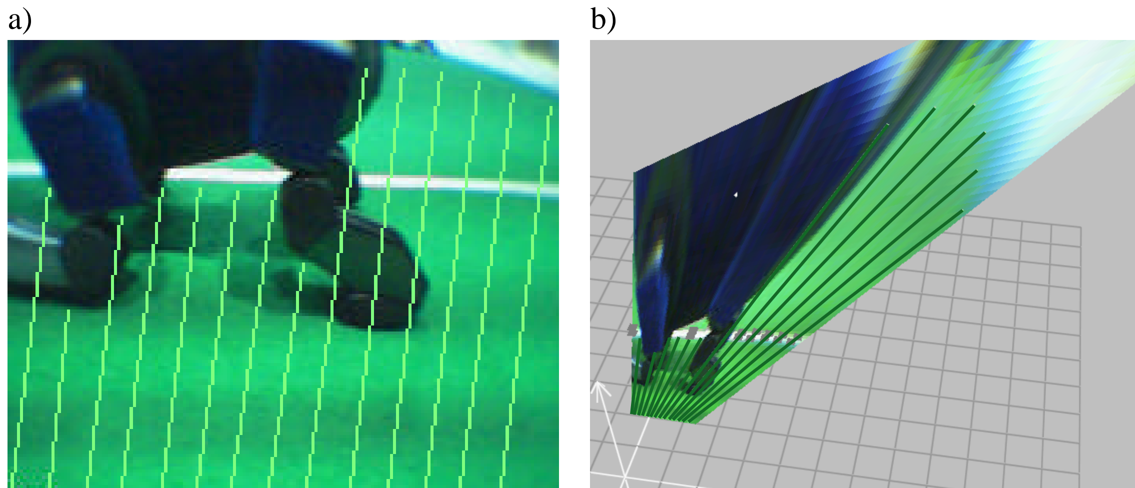


Figure 3.14: Obstacles Percept. a) An image with an obstacle. Green lines: projection of the obstacles percept to the image. b) The projection of the image to the ground. Green lines: obstacles percept.

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 (cf. Fig. 3.14a) 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).

The lines which indicate the free space usually start at the image bottom and end where the green of the ground ends or where the image ends (cf. Fig. 3.14b). 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.

3.2.10 Motion Compensation

The camera images are read sequentially from a “rolling shutter” CMOS chip. This has an impact on the images if the camera is moved while an image is taken, because each line in the image is captured at a different time instant: there is approximately a 30ms delay between the top scanline and the bottom. The resulting distortion from the camera motion can be compensated as we have shown in [45].

3.3 Self-Localization

The *GT2005 Self-Locator* implements a Markov-localization method employing the so-called Monte-Carlo approach [21]. 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. 3.19a). Each particle can be seen as the hypothesis of the robot being located at this posture. Therefore, the particles consist of a robot pose, i. e. a vector representing the robot's x/y-coordinates in millimeters and its rotation θ in radians:

$$pose = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} \tag{3.14}$$

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 particles 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.

3.3.1 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:

$$\Delta_{error} = \begin{pmatrix} 0.1d \times \text{random}(-1\ldots1) \\ 0.02d \times \text{random}(-1\ldots1) \\ (0.002d + 0.2\alpha) \times \text{random}(-1\ldots1) \end{pmatrix} \tag{3.15}$$

In equation (3.15), d is the length of the odometry offset, i. e. the distance the robot walked, α is the angle the robot turned.

In order to model the fact that the robot is not always able to move as expected, i. e. due to collisions with other robots, a number of particles is marked as not receiving odometry updates. These particles represent the hypothesis that the robot is currently blocked and unable to move. The number of particles without odometry updates can either be fixed or collision detection might be applied in order to reduce the number of particles that receive odometry updates when a collision was detected.

For each particle that is marked as receiving odometry updates, the new pose is determined as

$$pose_{new} = pose_{old} + \Delta_{odometry} + \Delta_{error} \tag{3.16}$$

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

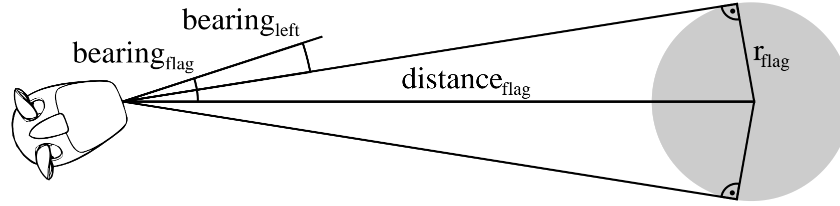


Figure 3.15: Calculating the angle to an edge of a beacon.

3.3.2 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 beacons and the goals, this localization approach only uses the directions to the vertical edges of the landmarks. However, although the points on edges determined by the image processor are represented in a metric fashion, they are also converted back to angular bearings. The advantage of using landmark 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.

The utilized percepts are bearings on the edges of beacons and goals delivered by image processing (cf. Sect. 3.2.6 and 3.2.7), bearings on line crossing positions and the center circle (cf. Sect. 3.2.4) as well as points on edges between the field and the field lines and the 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 edges of beacons, edges of goals, and edge points:

3.3.2.1 Beacons

The calculation of the bearing on the center of a beacon 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 beacon $distance_{flag}$, and the radius of the beacon r_{flag} are required (cf. Fig. 3.15):

$$bearing_{left/right} = bearing_{flag} \pm \arcsin(r_{flag}/distance_{flag}) \tag{3.17}$$

3.3.2.2 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.

3.3.2.3 Edge Points

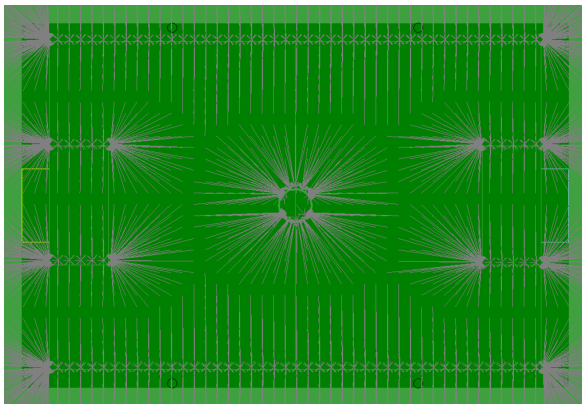
The localization also uses the points on edges determined by the image-processing system (cf. Sect. 3.2). Each pixel has an edge type (field line, yellow goal, blue goal, or field border), and by projecting it on the field, a relative offset from the body center of the robot is determined. Note that 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.

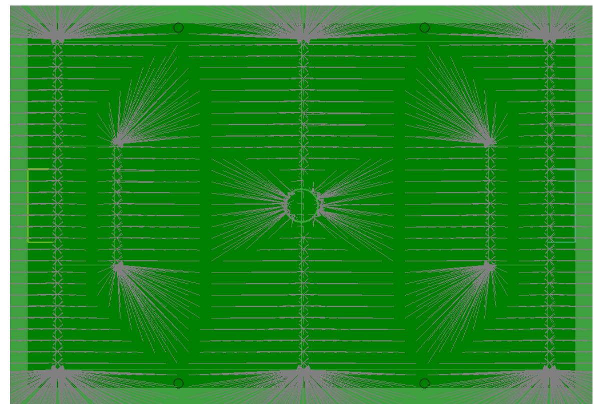
Since at the competition site in Osaka the area directly outside of the playing field was almost white, the outer field border was easily perceived by image processing (similar to the white field wall in previous years). Of course in the rules the area outside of the playing field is not defined, therefore it might be necessary to locally reconfigure the field border detection and its use for self-localization. The field lines are seen less often than the field border. The field border 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. However, it can be seen from nearly any location on the field.

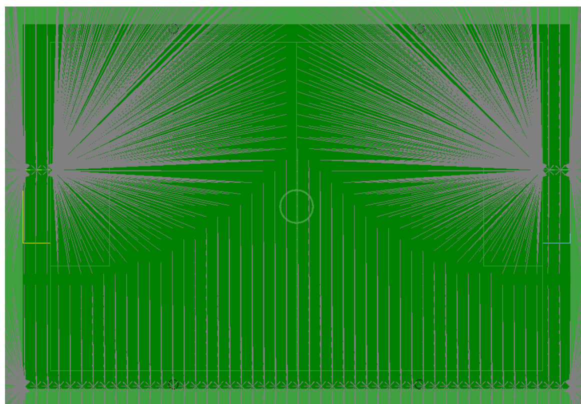
If the probability distribution for the pose of the robot had been modeled by a large set of particles, the fact that different edges provide different information and that they are seen in different frequency would not be a problem. However, to reach real-time performance on an Aibo robot, only a small set of particles can be employed to approximate the probability distribution. In such a small set, the particles 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 goals and beacons can determine the correct orientation on the field. Many particles will be located at the actual location of the robot, but several others are placed at the mirror symmetric position, because only the recognition of goals and beacons can discriminate between the two possibilities. For a longer period of time, no goal or beacon might be detected, whereas field lines are seen regularly. Under these conditions, it is possible that the particles on the wrong side of the field better match the measurements of 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 to mirror symmetric pose without ever seeing a goal or beacon. This is not the desired behavior, and it would be quite risky in actual soccer games. A similar situation problem could occur if the goalie continuously sees the front line of the penalty area, but only rarely its side lines. In such a case, the position of the goalie could drift sideways along because the front line of the penalty area does not provide any positional information across the field.

To avoid this problem, separate probabilities for goals, beacons, line crossings, horizontal field lines, vertical field lines and field border points are maintained for each particle.

Closest Model Points. For each particle of the probability distribution, we can calculate the absolute position of seen field lines on the field. For each of these points in field coordinates, it can be determined, where the closest point on a real field line of the corresponding type is located. Then, the horizontal and vertical angles from the camera to this model point are determined. These two angles of the model point are compared to the two angles of the measured point.


(a) Field lines along the field


(b) Field lines across the field


(c) One side of the playing field border

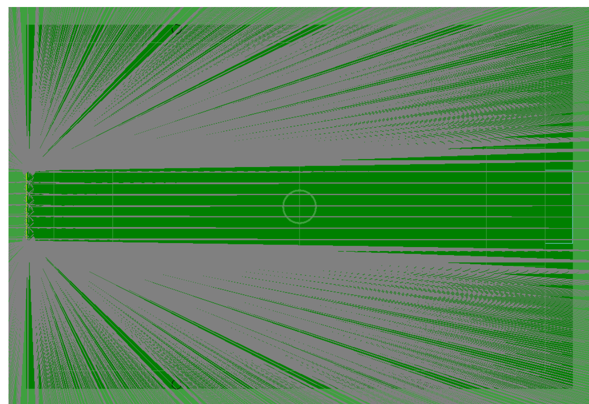

(d) Back side of a goal

Figure 3.16: Mapping of positions to closest edges

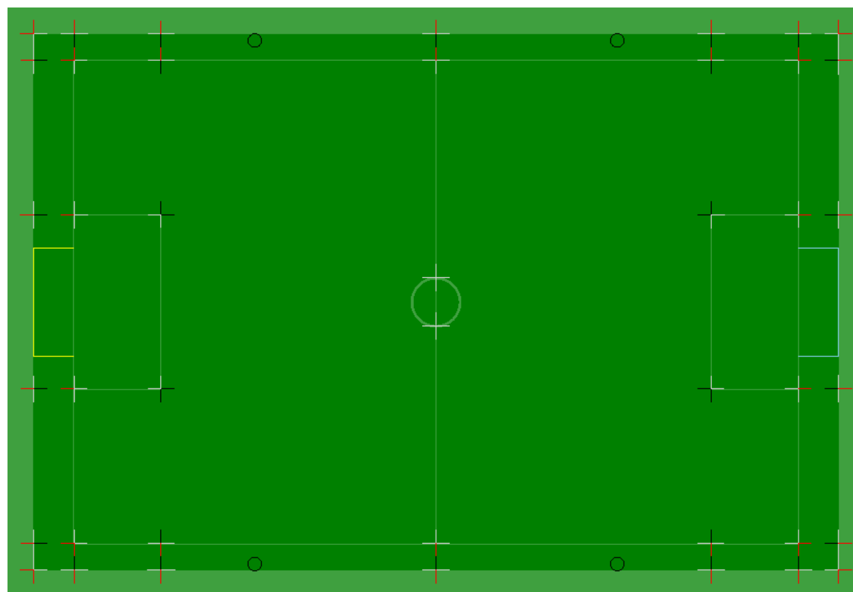


Figure 3.17: Possible locations and classifications of field line crossings.

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 100 particles. 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 3.16 visualizes the distances of measured points to the closest model point for the different edge types.

3.3.2.4 Line Crossings and the Center Circle

As the number of edge points that can be evaluated per frame is limited due to runtime restrictions, more information can be gathered by detecting special constellations of lines such as field line crossings and the center circle. Depending on the situation, image processing can detect the form and orientation of a line crossing or just its existence. This is taken into account by classifying line crossings. For each of the four sides of a crossing it is specified whether a field line continues in that direction or not, or whether it could not be determined (e. g. when the crossing is out of the image, too close to the image border, or if the crossing is occluded).

Similar to the evaluation of edge points each particle and observation is compared to the closest modeled crossing. In order to determine all possible locations on the field for a perceived crossing, its orientation and classification is matched against a list of perceivable crossing locations. These possible locations are visualized in Figure 3.17. Because of the way in which image

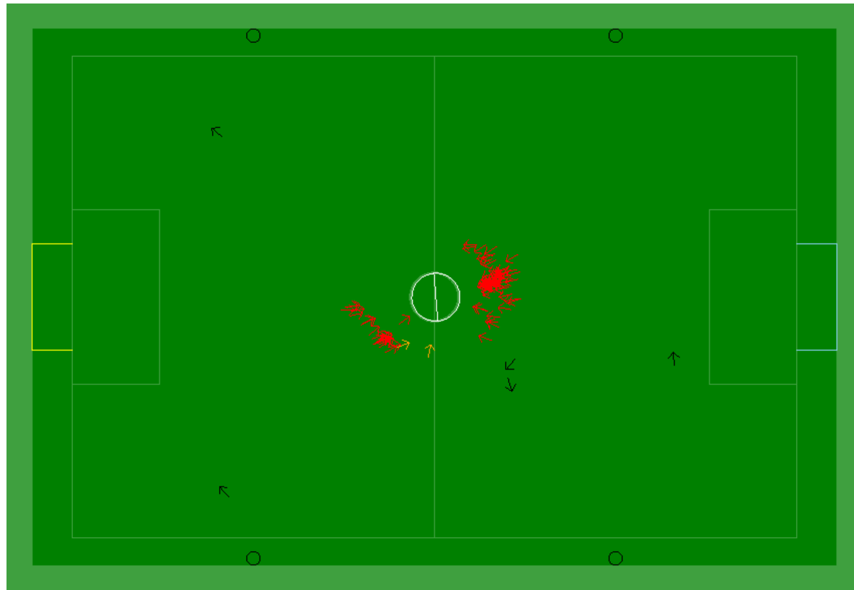


Figure 3.18: Particles gathering at the two possible postures derived from the center-cirlcle

processing detects field line crossings, points where a field line would cross the extension of another field line can also be perceived as field crossings. Therefore these points are also included in the list of line crossings (e. g. the crossings of the front lines of the penalty areas and the side field border lines). Since the area outside of the playing field is undefined, it is unknown how a crossing near the field border is classified on the outer side. Therefore in these cases any classification is matched (marked by red lines in Fig. 3.17).

As mentioned above, at the competition in Osaka the border towards the area outside of the playing field was easily perceivable. Therefore crossing points at the field border were also taken into consideration. Since this must not be the case on any playing field, it might be necessary to reconfigure the list of possible line crossing locations on different playing fields.

The increase in information that can be achieved by using line crossing especially improves selflocalization in the goal area and in the corners of the field.

As the center circle provides the same information as a classified crossing (its orientation is given by the center line) the update is done accordingly. When properly detected, the center circle reduces the number of possible poses to two (figure 3.18). Despite being a unique and precise percept, the center circle detection sufferes from a low detection range and several false-positives. Especially repeadetly detected false-positives can have a disastrous effect on the self-localization because there is only one modeled position; a fact that has to be regarded when assigning a weight to this percept.

3.3.2.5 Probabilities for percepts and the overall probability

The observation model only takes the bearings on the features into account that are actually seen, i. e., it is ignored whether the robot has not seen a certain feature 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:

$$s(\omega_{seen}, \omega_{exp}, \sigma) = \mathrm{e}^{-\sigma(\omega_{seen}-\omega_{exp})^2} \tag{3.18}$$

If multiple angles can be evaluated for a percept, e.g. horizontal, vertical angles and an orientation, the overall similarity of a particle for a certain edge type is calculated as:

$$q_{percept} = s(\omega_{(seen,1)}, \ldots, \omega_{(seen,n)}, \omega_{(exp,1)} \ldots \omega_{(exp,n)}, \sigma_1 \ldots \sigma_n) = \prod_{i=1}^{n} s(\omega_{(seen,i)}, \omega_{(exp,i)}, \sigma_i) \tag{3.19}$$

Where σ_i is a special weight for a certain percept and angle. For the similarity of the vertical angles the probability the robot's speed v (in mm/s) can be taken into account, because the faster the robot walks, the more its head shakes, and the less precise the measured angles are.

The parameters σ_i for a certain percept and angle is composed by measured variances v for each angle involved and a weight w for the whole percept:

$$\sigma_i = \frac{w_{percept}}{v_{(percept,angle)}} \tag{3.20}$$

As runtime is strongly limited on the Sony AIBO robot and calculating probabilities for a set of e.g. 100 particles is a costly operation, only a limited number of percepts can be considered each frame. While the number of beacons and goals that can be detected in one frame is already small because of the total number of those features on the field, the number of edge points and crossings has to be limited, thus increasing the possible portion of misreadings. To achieve stability against such 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 particle for each percept 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 particles, and therefore the probabilities will fall rapidly, and resampling will take place.

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

$$q'_{new} = \begin{cases} q'_{old} + \Delta_{up} & \text{if } q > q'_{old} + \Delta_{up} \\ q'_{old} - \Delta_{down} & \text{if } q < q'_{old} - \Delta_{down} \\ q & \text{otherwise.} \end{cases} \tag{3.21}$$

The overall probability p' of a certain particle is the product of the probabilities for the different kinds of percepts the particle can be updated with:

$$p' = \prod_{i=1}^{n} q'_i \tag{3.22}$$

3.3.3 Resampling

In the resampling step, the particles are moved according to their probabilities. Resampling is done in three steps:

3.3.3.1 Importance Resampling

First, the particles are copied from the old distribution to a new distribution. Their frequency in the new distribution depends on the probability p_i' of each particle, so more probable particles are copied more often than less probable ones, and improbable particles are removed. The number of particles that are removed is larger than the number particles duplicated. Therefore, at each step the particle set is reduced by a certain percentage, making room for new particles which are generated as described below.

3.3.3.2 Drawing from Observations

In a second step, new particles are added by so-called candidate postures. This approach follows the *sensor resetting* idea of Lenser and Veloso [36], and it can be seen as the small-scale version of the Mixture MCL by Thrun *et al.* [57]: 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.

Three 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 beacons seen. Estimated distances to these landmarks and odometry are used to update the bearings on these memorized beacons 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 beacon, and a bearing on a goal post or a beacon to determine the current posture. For each combination, one or two possible postures can be calculated.

3. As a single observation of an edge point 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.

The new particles are initialized with probability values 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.

If the number of particles that have to be added is larger than the number of postures in the template buffer, a template posture can be used multiple times, while adding random noise to the template postures. If no templates were calculated, random particles are employed.

3.3.3.3 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 particle is, the less it is moved. This can be seen as a probabilistic random search for the best position, because the particles 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 particles are moved according to the following equation:

$$pose_{new} = pose_{old} + \begin{pmatrix} 100(1-p') \times \text{random}(-1\ldots 1) \\ 100(1-p') \times \text{random}(-1\ldots 1) \\ 0.5(1-p') \times \text{random}(-1\ldots 1) \end{pmatrix} \quad (3.23)$$

3.3.4 Estimating the Pose of the Robot

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

3.3.4.1 Finding the Largest Cluster

To calculate the largest cluster, all particles 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 particles.

3.3.4.2 Calculating the Average

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

angle θ_{robot} is calculated as:

$$\theta_{robot} = \text{atan2}(\sum_i p'_i \sin\theta_i, \sum_i p'_i \cos\theta_i) \tag{3.24}$$

3.3.4.3 Certainty

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

$$c = \frac{m}{n} \cdot \frac{1}{m} \sum_i p'_i = \frac{1}{n} \sum_i p'_i \tag{3.25}$$

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.

As for a multimodal distribution a single value cannot represent well the whole positional probability density, this certainty is not only calculated for the overall winning sub-cube, but for the best n sub-cubes (actually 3). This more detailed representation, called the *RobotPoseCollection*, improves the reliability for modules deriving positions on the playing-field from the robots self-localization (such as the TeamBallLocator cf. Sect. 3.4).

3.3.5 Results

Figure 3.19 depicts some examples for the performance of the approach using 100 particles. The experiments shown were conducted with the simulator (cf. Sect. 5.2). They were performed 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, Padova, Lisbon, and Osaka, the method also proved to perform very well on real robots. Since 2002 this localization method works well enough in order to automatically position the robots for kickoff. For instance, the robots of the GermanTeam are just started somewhere on the field, and then—while still many people are working on the field—they autonomously walked to their initial positions. Even in 2004, the majority of the teams was not able to do this. As it was shown at the competitions in Osaka, the self-localization works very well on fields without an outer barrier. In 2003, it was also demonstrated that the GermanTeam can play soccer without the beacons.

3.4 Ball Modeling

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 position and speed.

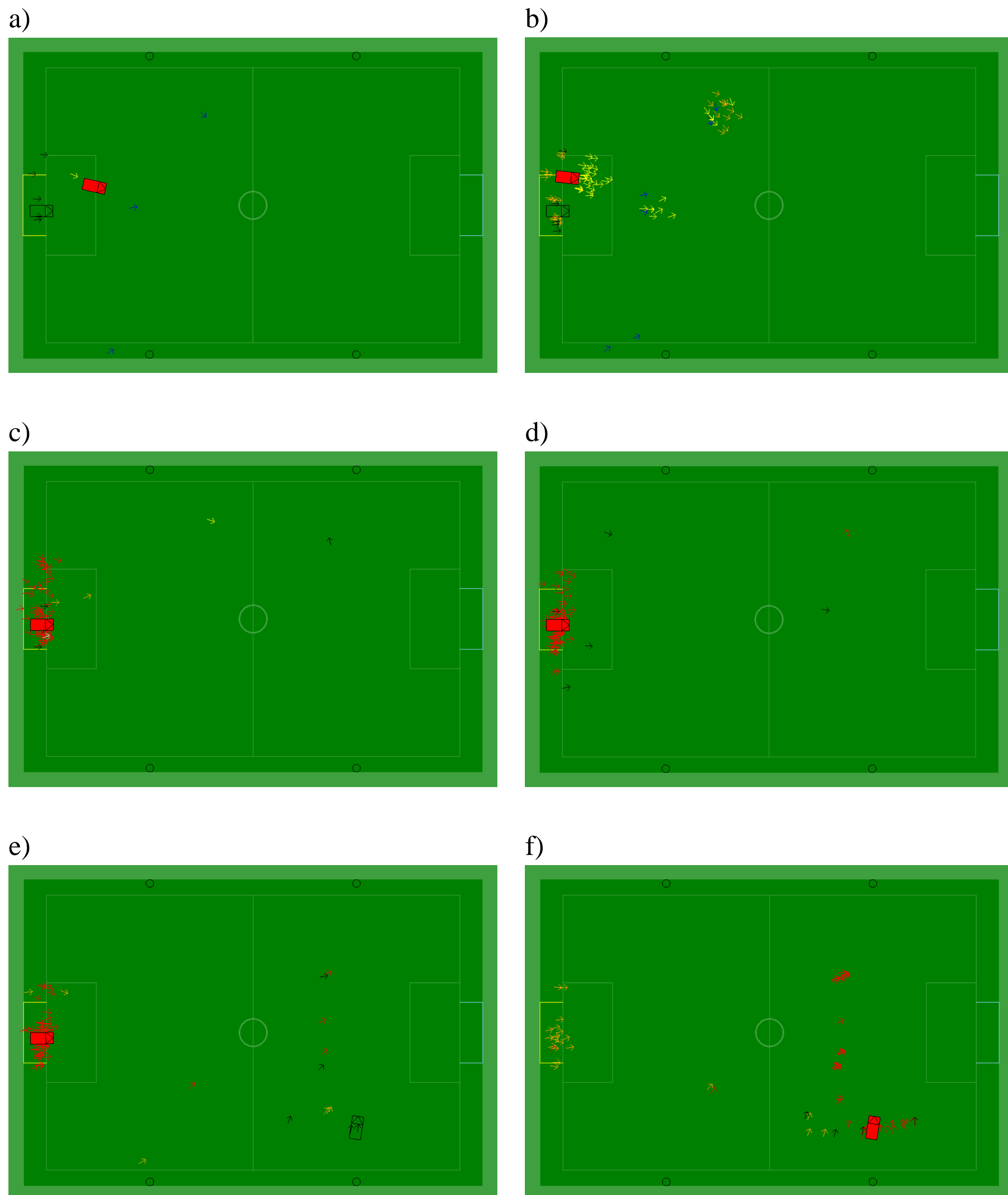


Figure 3.19: Distribution of the particles during the Monte-Carlo localization while turning the head. The outlined robot body marks the real position of the robot, the red body marks the estimated location. a) After the first image processed. b) After five images processed (150 ms). c) After 15 images (750 ms). d) After 75 images (2250 ms). e) Robot manually moved to another position. f) 15 images (750 ms) later.

3.4.1 Filtering of Ball Percepts using a Particle Filter

For the filtering of ball percepts a new approach based on particles was realized. The implementation is adapted to the particular needs of a ball locator and lays its main focus on a good reactivity of the system.

The advantage of a particle filter over, e.g., a Kalman filter is the possibility to deal with ambiguous situations by tracking different candidate ball positions and consider more than a single ball percept in a frame. Therefore each particle represents a possible ball position and velocity. The implemented approach consists of four steps:

1. Time Update, where the particle state is updated due to the robot's odometry and the current ball movement.

2. Measurement Update, which updates the particle positions based on the current ball percepts.

3. Calculation of ball position and velocity out of the particles.

4. Sensor Resetting

Time Update. In the Time Update the particles are first moved according to the robot's odometry as shown in equation (3.26) where $\vec{t}$ is the translation and α the rotation of the robot. This is necessary as the position and velocity are calculated in a robot-centric reference system.

$$\vec{s}_i = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \cdot \vec{s}_i - \vec{t} \qquad \vec{v}_i = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \cdot \vec{v}_i \tag{3.26}$$

Afterwards the particles are moved depending on the current calculated particle velocity and the time Δt between the last and the current frame assuming a constant speed model for the ball movement:

$$\vec{s}_i = \vec{s}_i + \vec{v}_i \cdot \Delta t \tag{3.27}$$

As the ball movement is not constant due to friction on the carpet, the velocity of the particle is decreased depending on the coefficient of friction μ, which was determined experimentally using a ceiling camera global vision system:

$$\vec{v}_i = \vec{v}_i - \mu \cdot 9810\frac{mm}{s^2} \cdot \Delta t \tag{3.28}$$

Finally the particle validity is decreased due to a system noise factor and the age of the last ball percept since the Time Update only predicts a new state of the particles.

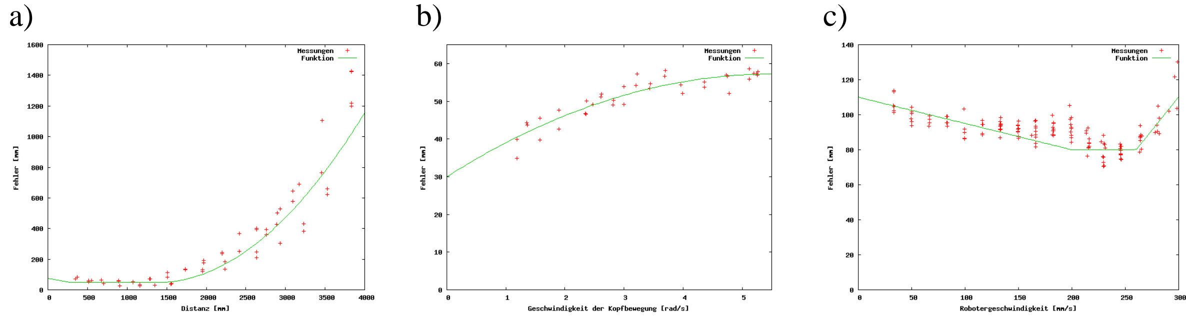


Figure 3.20: Ball measurement error functions learned from experimental data. a) Distance Error. b) Panning Velocity Error. c) Robot Speed Error.

MeasurementUpdate. The multiple ball percepts of the ImageProcessor are processed sequentially, starting with the percept with the highest validity. Percepts which are outside of the field are ignored. Additionally, feedback from the behavior is used when the ball is grabbed by the robot. In this case the ball position is modelled near the robot's own position.

During the update it is distinguished between particles near the considered percept and those farther away. Near particles are pushed closer to the percept. Their validity is increased depending on the current position validity and the accuracy of the percept. The influence of the percepts on the particle state depends on their expected accuracy which is calculated out of a percept validity provided by the ImageProcessor and the accuracy due to distance, panning velocity and robot speed. To determine the accuracies a test behavior was implemented, which measures the error values of the percept depending on these different factors. As a reference system a ceiling camera was used. Figure 3.20 shows the results of our experiments.

To avoid a big influence of wrong percepts, particles farther away are not moved, but their validity is updated depending on the measurement noise, which was determined experimentally, and the accuracy of the percept.

Finally for each particle a velocity is calculated out of the new and the previous particle position. The farther the particles are away from the robot the more they tend to get a high velocity because of the measurement noise. Therefore a large change in the velocity is prevented by restricting the increase of the absolute speed.

Calculation of ball position and velocity. For calculating the estimated ball position and velocity out of the particles we adapt the approach of the SelfLocator. The idea is to divide the field into a 10×10 grid and calculate the validity sum of the contained particles for each cell. Then the 2×2 with the biggest sum is determined. Figure 3.21 visualizes this algorithm.

Afterwards we calculate a weighted average for position and velocity of the particles in this grid by:

$$\overrightarrow{ball}_{pos} = \frac{\sum_{i=1}^{k} \vec{s}_i \cdot \mathrm{P}(s_i)}{\sum_{i=1}^{k} \mathrm{P}(s_i)} \qquad \mathrm{P}(ball_{pos}) = \frac{\sum_{i=1}^{\ell} \frac{\mathrm{P}(s_i)}{\mathrm{dist}\left(\overrightarrow{ball}_{pos},\, \vec{s}_i\right)}}{\sum_{i=1}^{\ell} \frac{1}{\mathrm{dist}\left(\overrightarrow{ball}_{pos},\, \vec{s}_i\right)}} \qquad \overrightarrow{ball}_{vel} = \frac{\sum_{i=1}^{k} \vec{v}_i \cdot \mathrm{P}(s_i)}{\sum_{i=1}^{k} \mathrm{P}(s_i)} \tag{3.29}$$

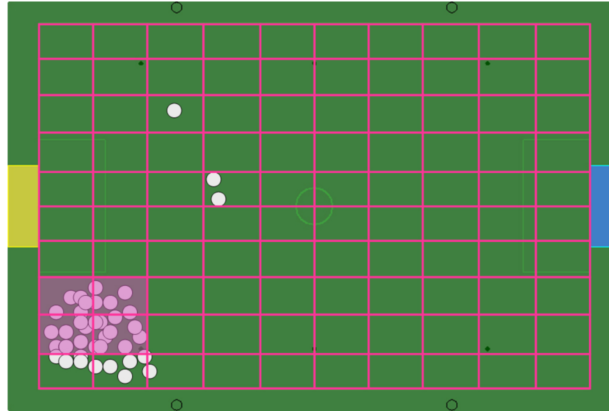


Figure 3.21: Grid for calculating the overall ball position, the pink area represents the chosen cluster.

where $\vec{s}_1, \ldots, \vec{s}_k$ are the particles inside the grid, $\vec{v}_1, \ldots, \vec{v}_k$ their velocities and ℓ is the number of particles.

Sensor Resetting. Particles with a position validity less than 0.05 are removed and replaced near the last percept. Beside that we assure that at least 5% of the particles are near the last percept. If this is not true, the particles with the lowest validities are removed and reinserted near this percept until this condition is fulfilled. If the last percept is too old the particles are alternatively spread around the last estimated ball position.

For the reinsertion of particles we use a gaussian distribution. The standard deviation depends on the time since the last ball percept. The older the percept or the lower the validity, the higher the standard deviation is. The initial validity of the replaced particles is set to 0.20.

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 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 with sufficient accuracy (ball position in coordinates relative to the robot) where the ball is, communicating the ball 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 can see 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 have to be distinguished:

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 into another direction. 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.

3.4.3 Team Ball Locator

The idea of the team ball locator is to use the particles of the whole team for calculating a “global” ball position. As it is too demanding to send all particles from the ball locator, so called **“representative”** particles are calculated and sent to the teammates. This solution makes the team ball locator also independent from the number of particles used in the ball locator.

3.4.3.1 Calculation of representative particles

Particles which will be communicated first have to be transformed into field coordinates. As it might happen that the localization status is uncertain and two mirror-symmetrical robot poses have similar validity, the SelfLocator calculates a so called “robot pose collection” which contains more than one possible position. For each robot pose and each particle of the ball locator a particle in field coordinates is derived.

Afterwards representative particles are calculated for this new set of particles with a recursive algorithm which divides the field into cells. Figure 3.22 shows the final step of the algorithm and

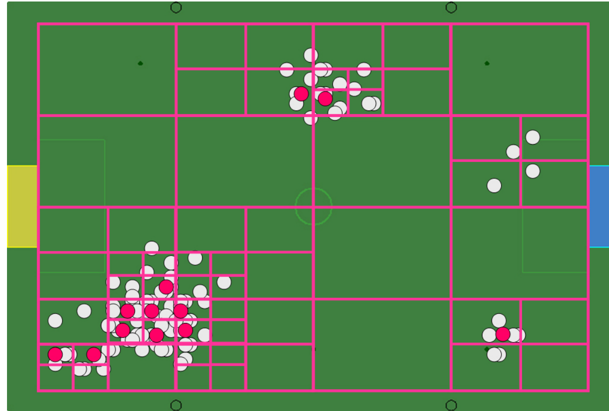


Figure 3.22: Final step of the algorithm which calculates the representative particles. The white circles represent the original particles, the red circles the resulting representative particles.

the calculated representative particles. It is distinguished between “Basic Cells” (which contain particles) and “Composite Cells” (containing 4 basic cells):

1. The whole field is initialized as a single Composite Cell

2. Calculate for each basic cell i with $i = 1, \ldots, 4$ the sum x_i of the particle validities in this cell. Let s be the sum of all particle validities in a composite cell, $s = \sum_{i=1}^{4} x_i$.

   - **if** $x_i < 0,05 \cdot s$: remove basic cell by deleting all particles contained in this cell.
   - **if** $0,05 \cdot s \leq x_i \leq 0,1 \cdot s$: do nothing.
   - **if** $x_i > 0,1 \cdot s$: transform the Basic Cell into a Composite Cell by subdividing it into 4 Basic Cells

3. Apply recursively step 2 until for all basic cells $x_i \leq 0.1 \cdot s$ is fulfilled or a recursion depth of 4 is reached.

4. Create a queue containing all representative particles by calculating a representative particle for each remaining cell as the weighted average of all particles in the cell.

The representative particles are communicated to the teammates. Each robot decreases the validities of the received particles depending on the network delay. Afterwards a communicated ball position is calculated out of the own representative particles and the communicated ones using the same algorithm as in the ball locator.

3.4.3.2 Reaction to loss of information

During a game it might happen that there is no data received from a robot for some frames. In this case the old information is not discarded, but the validities are decreased and the particles are spread with a gaussian distribution. The standard deviation of such gaussian depends on the number of frames where no particles were received from the robot.

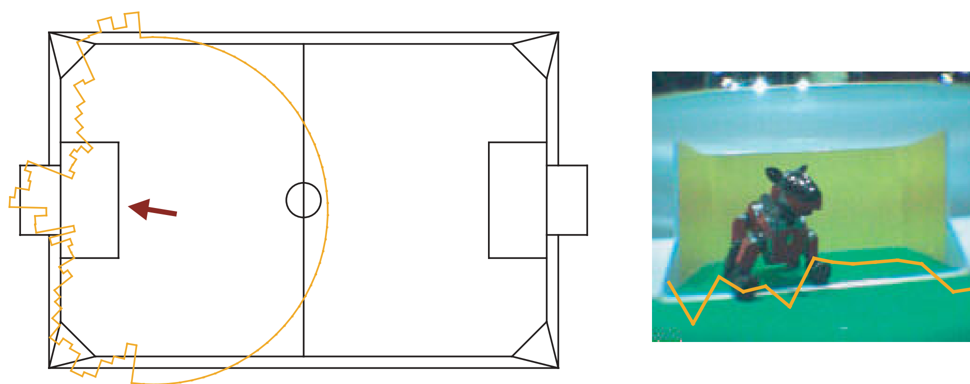


Figure 3.23: The obstacle model as seen from above and projected into the camera image. The robot is in front of the opponent goal.

If we receive at least one particle of a teammate all old particles from this robot are discarded and replaced by the new information.

3.5 Obstacle Model

Obstacle detection and the obstacle model used are described in detail in [27].

In the obstacle 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. 3.23). 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 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.

3.5.0.3 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 a detected obstacle or the upper boundary of the image (if no 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. 3.24).

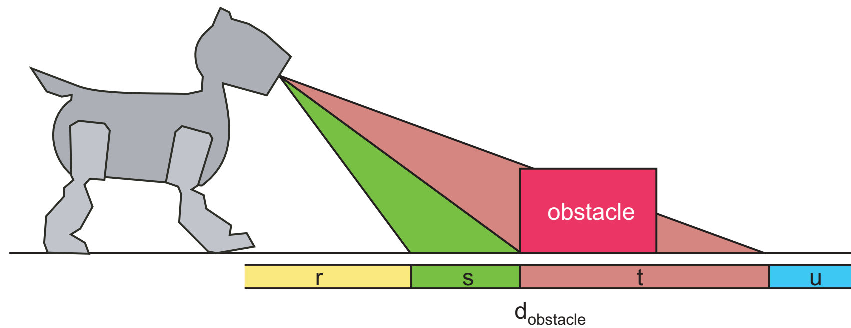


Figure 3.24: The above diagram depicts the robot looking at an obstacle. The robot detects some free space in front of it (s) and some space that is obscured by the obstacle (t). The obstacle model is updated according to the diagram (in this case the distance in the sector is set to $d_{obstacle}$ unless the distance value stored lies in r).

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.

3.5.0.4 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 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 Lisbon and Osaka 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.

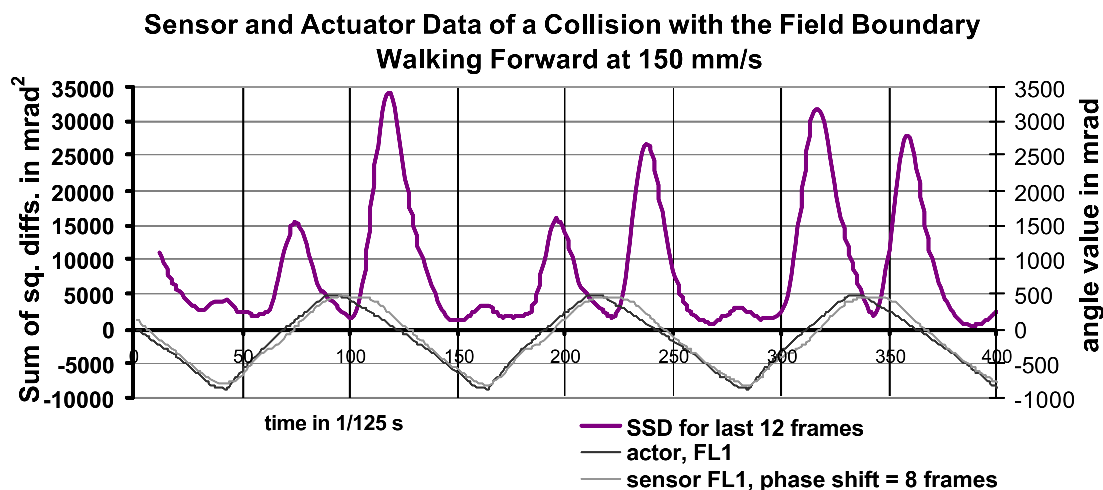


Figure 3.25: The graph shows the actuator and sensor curves and the sum of the squared differences (SSD). Peaks in the SSD curve correspond to collisions.

3.6 Collision Detector

Collision detection is described in detail in [26]. 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.

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. 3.25).

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. Although abrupt changes in motor commands made it harder to detect collisions during game situations than in the laboratory, this approach enabled us to detect and react to collision.

Another task was finding an "appropriate" behavior once a collision is detected. During game struggle for the ball we preferred not to react to collisions, because they are unavoidable in such situations. But therefore the robot reacted in situations, when he was far away from the ball and detected a collision during his movement towards the ball. In those cases he would perform a sideways movement to avoid the obstacle he just has collided with.

A sample behavior was also developed for the agent in which the robot would turn away from obstacles when a collision was detected. 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.

Determining Robot Positions by a Grid. 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. This is done for all percepts of the current robot and the percepts of the teammates. The validity of the communicated percepts is reduced by 10 percent, because small errors in the self localization generate ghost dogs.

The soccer field is discretized as a grid of a size of 15 cm x 15 cm. The positions of the percepts are converted into grid points and the value of the grid at this position is raised. In the next step all peaks were searched. The distance between the peaks is taken into consideration by selecting only peaks which have a certain distance to each other. Additionally, if more than one peak is in line with the own robot position only the nearest peak is used. In the third step the values of the neighbors of a peak grid point are reduced and added to the peak. In the last step all grid points are aged, so that the position of a robot which is not seen any longer is removed from the model within four seconds.

The process described above is only done for the opponent players to save time.

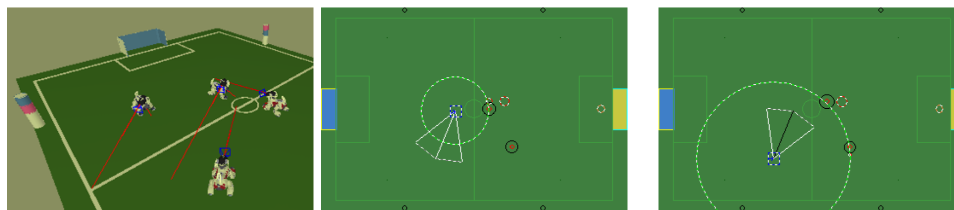


Figure 3.26: Player Model View in the Simualtor. a) Shows the current game situation b) Shows the Player Model of the first AIBO. The green circle the robot indicates the free space around the robot. The red circle inidicates the estimated opponent position. c) Shows the Player Model of the second AIBO.

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* [39, 40] since 2002 and improved it largely in 2003 and 2004. In 2005 the new language *YABSL* was introduced (see D.7). The description of behaviors using *YABSL* needs less source code than in *XABSL* and is better readable. With both languages a hierarchy of options can be described, following the architecture that was introduced together with *XABSL*. Appendix D gives an introduction into this system and a complete documentation can be found at the *XABSL* web site [38].

This section describes in detail the implemented strategies and behaviors of the GermanTeam 2005. Those who are not familiar with the *XABSL* language should probably read appendix D first.

The behaviors which the *GermanTeam* developed in *XABSL* for the RoboCup championships 2005 in Osaka are distributed among about 80 options. Figure 3.27 shows the option graph of the soccer related behaviors.

In general, the lower behaviors in the option hierarchy such as ball handling or navigation, have to react instantly on changes in the environment and are therefore very short-term and reactive. The more high-level behaviors such as waiting for a pass, positioning, or role changes

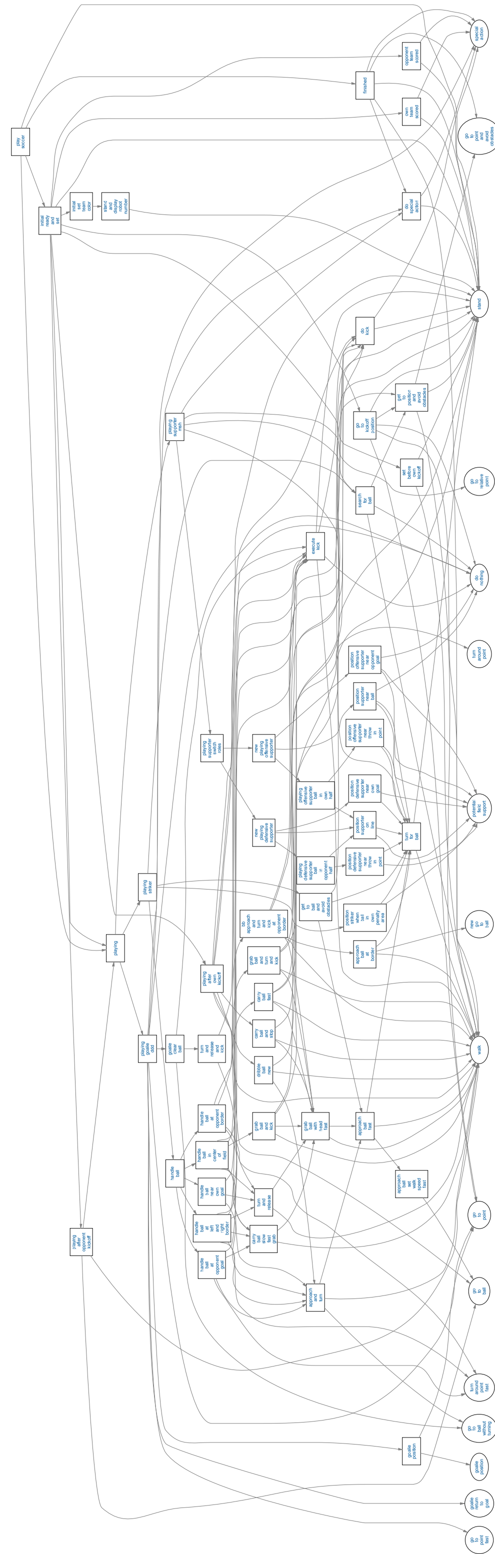


Figure 3.27: The option graph of the soccer-related behaviors of the *GermanTeam*.

try to prevent frequent state changes to avoid oscillations and make more deliberative and long-term decisions. This section describes from bottom to top how the *GermanTeam*'s robots play soccer starting with basic capabilities and finishing with the high-level team strategies.

An extensive automatically generated HTML documentation of these behaviors can be found at *http://www.ki.informatik.hu-berlin.de/XABSL/examples/gt2005/*. It is recommended to use this site as an additional source to this Section.

3.8.1 Ball Handling

The *GermanTeam* implemented a lot of sophisticated well tuned ball handling behaviors. This section shows step by step how the whole behavior is composed from simple options.

3.8.1.1 Approaching

All behaviors for ball approaching and dribbling are based on one single basic behavior: *"go-to-ball"* is responsible for walking to the ball. For the use in different contexts, it provides a variety of parameters. First, the body of the robot is always directed to the ball, restricted by the parameter *"max-turn-speed"*. The maximum walk speed is given by the parameter *"max-speed"*, making higher options responsible for slowing down near the ball. The *"max-speed.y"* parameter restricts the sideward component, allowing for sprinting with the "dash" walk type. For dribbling and the "turn kick" (cf. 3.8.1.2), *"y-offset"* specifies a y offset with that the robot shall arrive at the ball. If the robot is very close to the ball and if the ball is much to the left or right, the translation component is almost completely inhibited, making the robot only turn in order to avoid pushing the ball away with the front legs.

The ball handling behaviors do not reference the *"go-to-ball"* basic behavior directly but use the option *"approach-ball"* (cf. fig. 3.28). This option makes a distinction whether the robot is far away from the ball or close. In the first case, in state *"search-auto"*, the head-control mode *"search-auto"* is set. This lets the head of the robot look at the ball and – frequently, always after a certain time of consecutively perceived balls – shortly look around for landmarks and obstacles to improve self-localization. These head scans are disadvantageous near the ball. That's why if the robot gets closer to the ball than specified in the option parameter *"look-at-ball-distance"*, in state *"search-for-ball"* the head control mode is set to *"search-for-ball"*. This lets the head look at the ball only. To avoid frequent changes between these two states, there is a distance hysteresis of 5 cm between them. In both states, the option *"approach-ball-set-walk-speed"*, which controls the walk speed (see below), is referenced.

If the ball was not seen for 1.3 seconds in the *"search-auto"* state or not for 400 ms in the *"search-for-ball"* state, in state *"ball-not-seen"* the option *"turn-for-ball"* (see below) tries to redetect the ball. If the ball is seen again, the state *"ball-just-found"* remains active for 2 seconds, setting the head control mode *"search-for-ball"* in order to avoid further ball losses due to scanning around with *"search-auto"*.

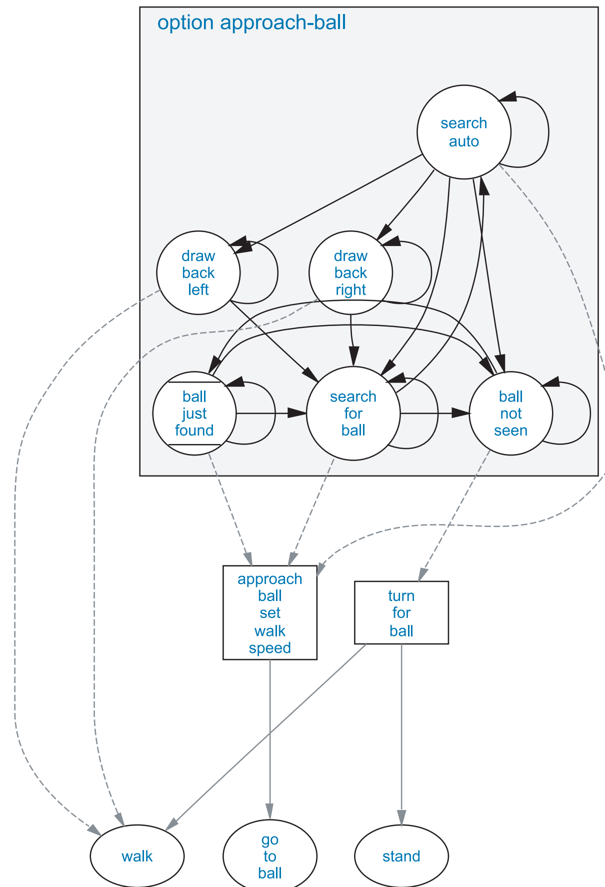


Figure 3.28: Option *"approach-ball"* controls the head movements while approaching the ball and handles collisions and ball losses. The complete documentation of the option can be found at *http://www.ki.informatik.hu-berlin.de/XABSL/examples/gt2004/option.approach-ball.html*.

Option *"approach-ball-set-walk-speed"* (cf. fig. 3.29a) controls the speed of ball approaching. It is only used by option *"approach-ball"*. In state *"fast"*, the basic behavior *"go-to-ball"* is executed with a fixed speed of 350 cm per second. If the robot gets closer to the ball than specified in parameter *"slow-down-distance"* (minus 2,5 cm hysteresis), in state *"slow"* the speed given in *"slow-speed"* is passed to *"go-to-ball"*. From the *"fast"* state, if the ball is farer away than 1200 cm and if the angle to the ball is between plus and minus 7 degrees, state *"dash"* becomes active. There *"go-to-ball"* is executed with walk type *"dash"*, a faster but not omnidirectional walking gait. (Note: In 2005 the same walking gaits were used for walk-type dash and for walk-type normal.

The ball approaching stops immediately after the ball was not seen for a certain time (see above). In this case, *"approach-ball"* references the option *"turn-for-ball"* (cf. fig. 3.29b) to redetect the ball. In the initial state *"ball-not-seen"*, the basic behavior *"stand"* is executed. Note that *"stand"* does not stop walking immediately but continuously slows down in order to

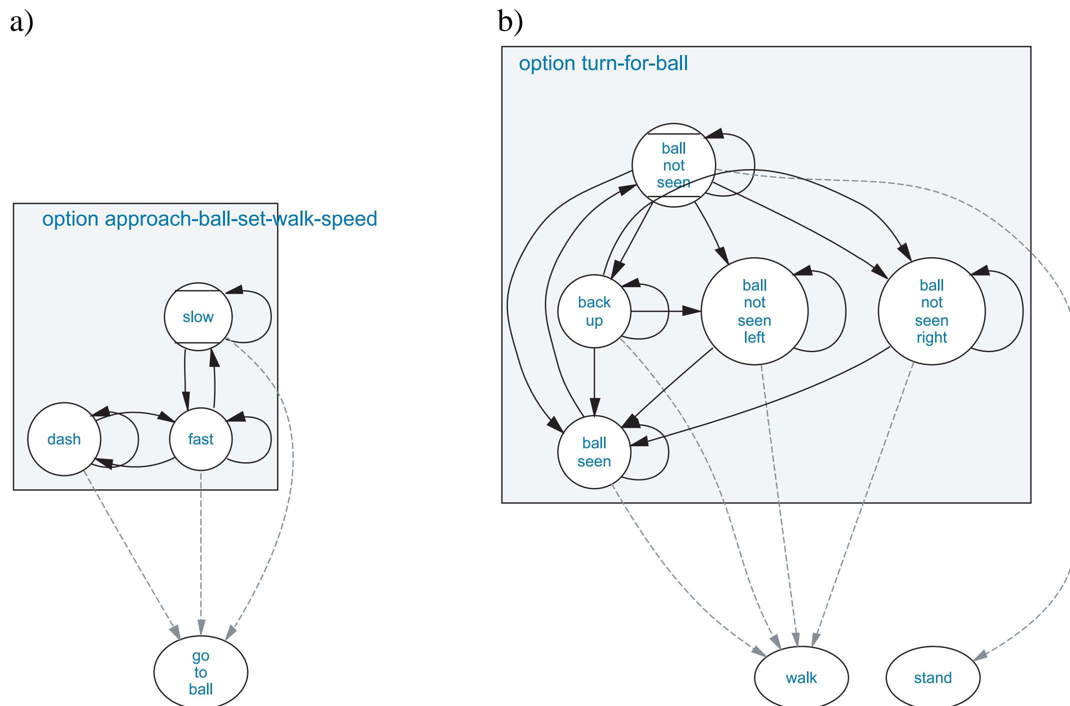


Figure 3.29: a) Option *"approach-ball-set-walk-speed"* controls the speed of ball approaching. b) Option *"turn-for-ball"* tries to redetect a previously lost ball.

avoid bumpy movements if the ball is redetected fast. As *"turn-for-ball"* can be activated from different contexts and situations, the time how long state *"ball-not-seen"* remains active depends on how long the ball was not seen and where it was seen last. The state remains active for at least 800 ms which are needed for *"stand"* to almost slow down. As long as the ball was seen 1.7 seconds before, *"ball-not-seen"* keeps active to give the head control a chance to make a complete scan around. If the ball was seen in the last 5 seconds and in the near, it is very likely that the ball is at the side of the robot. Therefore, in state *"back-up"* the robot walks backward for 1.5 seconds to redetect the ball. If that fails (or from *"ball-not-seen"* if all other conditions fail), the state *"ball-not-seen-left"* or *"ball-not-seen-right"* gets active, depending on whether the ball was previously seen left or right. The robot turns around using the *"walk"* basic behavior. The head control mode is set to *"look-left"* or *"look-right"*, letting the robot look into the direction of turning. Although the *"turn-for-ball"* option is not activated anymore from *"approach-ball"* when the ball is redetected, state *"ball-seen"* becomes active when the ball is seen again, turning the robot to the ball.

3.8.1.2 Dribbling

The option *"approach-and-turn"* (cf. fig. 3.30) dribbles the ball into the direction that is passed through the parameter *"angle"*. Already this behavior is able to get the ball reliably into the opponent goal. It is composed from mainly *"approach-ball"* and a few other short walk sequences

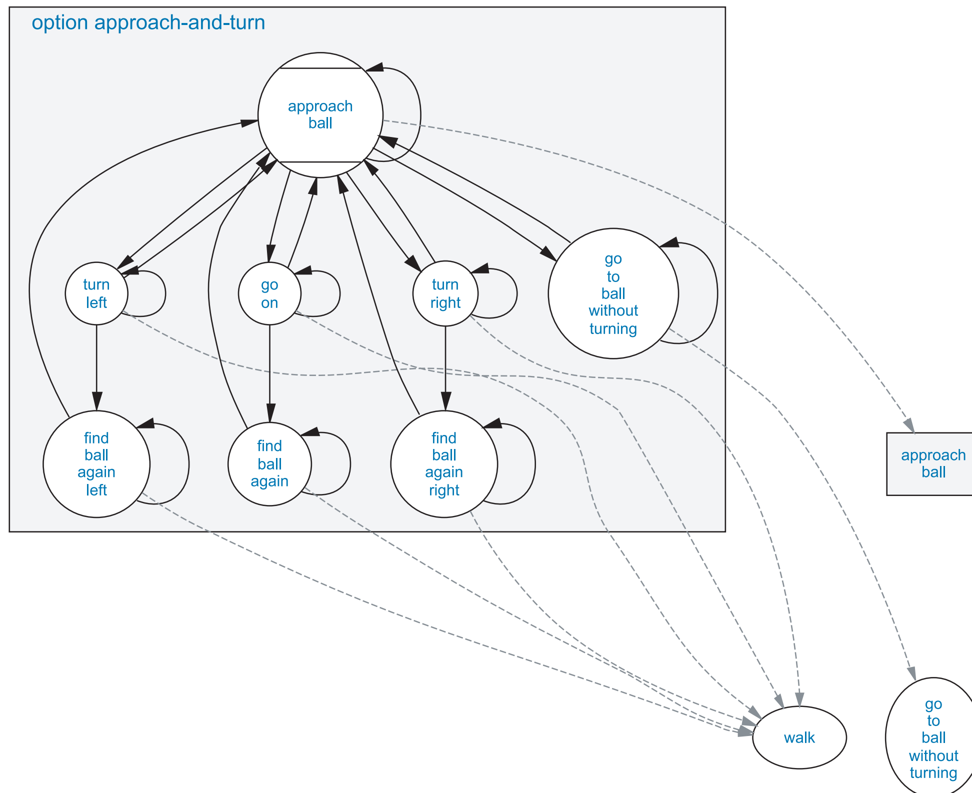


Figure 3.30: Option *"approach-and-turn"* dribbles the ball into given direction by pushing the ball with the chest or pulling it around with the front legs.

that push the ball into the desired direction. It does not use any kicks which makes it a fast and robust behavior.

State *"approach-ball"* activates option *"approach-ball"* with proper parameters for fast ball approaching. When it happens that the ball is very close and that the robot is already directed into the direction where the ball shall be dribbled to, state *"go-to-ball-without-turning"* is activated and basic behavior *"go-to-ball-without-turning"* is selected. Different from *"go-to-ball"*, this basic behavior uses only x and y translation. The advantage is that it is a bit faster near the ball than the normal ball approaching. As soon as the conditions above are not met anymore (with a hysteresis), the option returns to the state *"approach-ball"*.

If the ball is seen well and directly in front of the robot, one of the three dribbling moves starts: state *"turn-right"* becomes activated if the destination is more to the right than -30 degrees, *"turn-left"* gets active if the destination is more to the left than 30 degrees, and otherwise state *"go-on"* is chosen. In *"go-on"*, the robot just runs blindly straight ahead for 250 ms, pushing the ball forward with the front legs or chest. If after the time the ball is seen again or still seen, it is returned to state *"approach-ball"*. Otherwise, in state *"find-ball-again"*, the robot does not stop – as it would happen when in the *"approach-ball"* option the ball is not seen anymore – but still walks forward with a slow speed for maximum 500 ms, assuming that the ball is still in front of the robot and not at the side or behind.

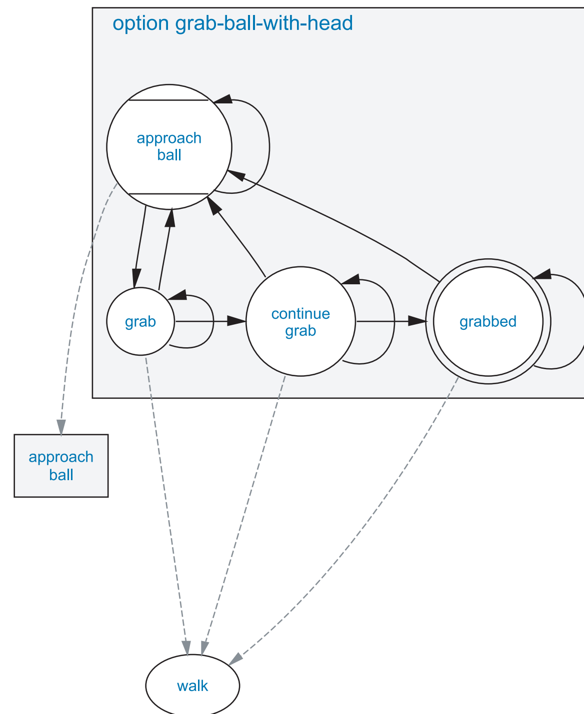


Figure 3.31: Option *"grab-ball-with-head"* grabs the ball with the head.

In the states *"turn-left"* and *"turn-right"*, the robot simultaneously walks forward and turns at the same time for 500 ms, pushing the ball reliably and strong into a direction of approximately 60 degrees. A different walk type, *"turn-kick"* is used in order to have the front legs more stretched to the front for safer guiding the ball with the outer leg. If the ball is not seen after the 500 ms, in the states *"find-ball-again-left"* and *"find-ball-again-right"* the robot does not stop but also walks straight ahead at a slow speed. Additionally, the head control mode *"search-for-ball-left"* or *"search-for-ball-right"* is set, which gives the head control a hint in which direction the ball was pushed and where to search first.

3.8.1.3 Grabbing

The behavior for ball grabbing is encapsulated in a separate option, *"grab-ball-with-head"* (cf. fig. 3.31). In the initial state *"approach-ball"*, option *"approach-ball"* is selected with a quite low speed near the ball. If the ball is in the correct position for grabbing, in state *"grab"* the robots walks forward at a low speed "onto the ball". The head control mode is set to *"catch-ball"* and the actual job of grabbing is done by the head control. The infrared distance sensor in the chest is used to measure the exact distance to the ball. If the ball is not at the chest yet, the head is lifted in order to push the ball not away with the head. Otherwise, the head is bended over the ball. The state *"grab"* is active without feedback for 1 second. After that, in state *"continue-grab"* it is checked with the infrared distance sensor whether the grab was successful (it has to be checked both in the head control and in the behavior control as

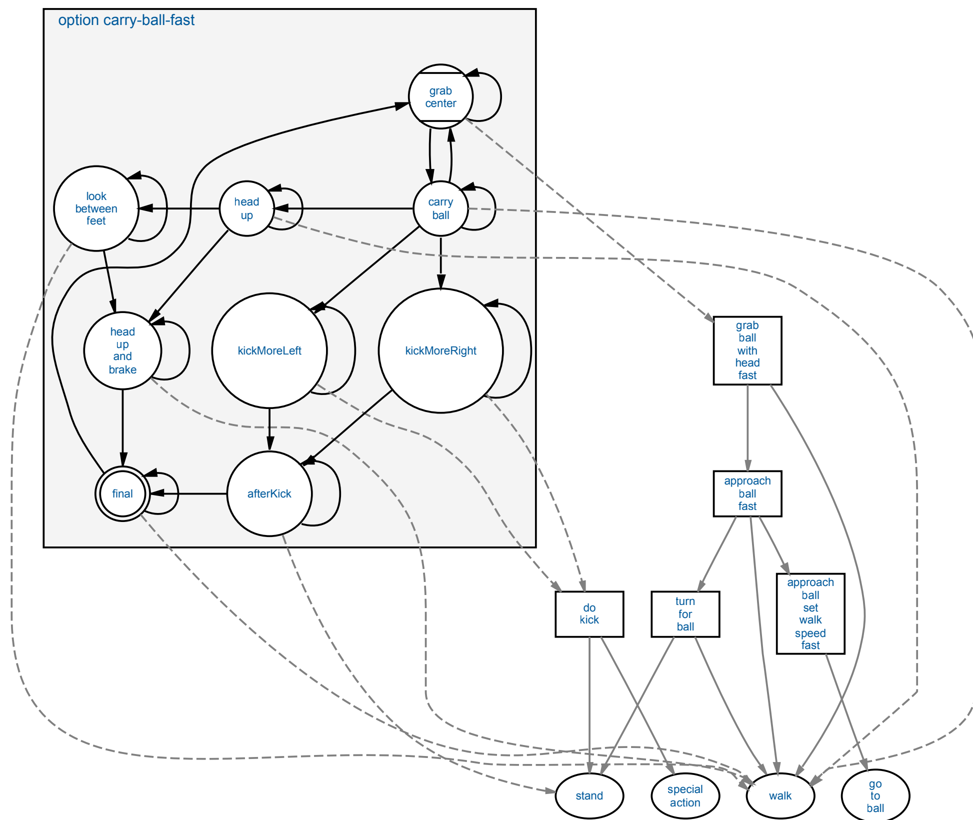


Figure 3.32: Option *"carry-ball-fast"* grabs and walks with the ball while touching and guiding it with the head or the mouth. Optionally a kick can be performed at the end oft the option.

a transmission of this information from the *Motion* to the *Cognition* process would last too long). If not, it starts from the beginning in the *"grab"* state. If the ball was grabbed, the option stays in the state *"grabbed"*, which is a target state to signal higher options that the whole behavior was successful. For different contexts there exist specialized versions of this option: *"grab-ball-with-head-fast"*, *"grab-ball-with-head-high"*, ...

3.8.1.4 Carry Ball

"carry-ball" means to dribble the ball while touching and guiding it with the head or the mouth. In this way the robot carries the ball instead of dribbling it. This behavior consists of three main phases:

1. The robot grabs the ball using the option *"grab-ball-with-head"* as described in 3.8.1.3
2. Then the robot carries the ball – without lifting its head! – using rotation and translation to a desired angle given by the parameter. If this angle is reached, the robot walks straight on. This second phase takes at most 3 seconds according to the rules (*"ball-holding"*).

3. Finally there are two different ways to release the ball, determined by the binary parameter *"stop-ball"*. If *"stop-ball"* is set to 0, the robot lifts its head and then continues walking forward with a reduced speed. As an result, the ball simply rolls away from the robot (and maybe rolls in the goal). When using the other possibility with *"stop-ball"* set to 1, the robot will first stop moving before lifting the head. In this way the ball will not roll away and the whole sequence can either be restarted easily or an other option can be selected by the super-behavior.

There are two slightly different variations of *"carry-ball"* implemented: *"carry-ball-slow"* and *"carry-ball-fast"*.
In the first one – *"carry-ball-slow"* – the robot grabs the ball, then the robot rotates with the grabbed ball without doing an translation to the desired angle (e.g. to the center of the goal). When the angle is reached, the rotation is stopped and the robot walks straight on using a x-translation only. This is the main difference between the two variations of *"carry-ball"*: In *"carry-ball-slow"* the robot first rotates and then walks straight on without turning anymore. The risk of loosing the ball is reduced because the turning can be done in a special walktype called *"turn-with-ball"*. After turning the walktype will be switched to normal.
In contrast to this, *"carry-ball-fast"* does rotation and translation at the same time after grabbing the ball. So the robot walks a curve to the desired angle. On the one hand, this saves time and the robot always keeps moving forward so its harder to attack it. On the other hand, the risk to loose the ball while walking and turning is increased because of only using the walktype normal. Optionally the robot can perform a kick after carrying the ball using the parameter *"kick-ball"*. A value of -1 kicks the ball foward-right, 1 means kicking forward-left and 0 means do not kick. An other problem of *"carry-ball-fast"* is that the rotation can only be done with at most 50 degrees per second. So if the robot is near to the goal boarder it may happen that it carries the ball out of the field. That is the reasons why we mostly used *"carry-ball-slow"*.

The main advantage is that the robot keeps hard control during the carrying of the ball, even in a dogfight situation. In addition to this, other robots (including the opponent's goalie) often can not see the ball and maybe get confused about that. On the other hand, the robot does not get any useful perceptions from its vision while carrying the ball because it is always looking to the ground. The robot can only use its odometry data to check if the desired angle is reached. So for example if the robot is blocked or bumped by another player, the robot may continue running in the wrong direction.

3.8.1.5 Pushing Backwards

The dribbling- and carry-behaviors are only reasonable when the robot is behind the ball (seen from the direction where the ball shall be played to). For all other cases, option *"turn-and-release"* grabs the ball with the head (the ball is shut between the front legs and the head), turns with the grabbed ball, and then releases the ball again.

In the initial state *"grab"* of option *"turn-and-release"* (cf. fig. 3.33), the option *"grab-ball-with-head"* is executed until it reaches its target state. After that, in the states *"turn-left"* and *"turn-right"*, the robot turns with the ball to the desired direction given by the option parameter

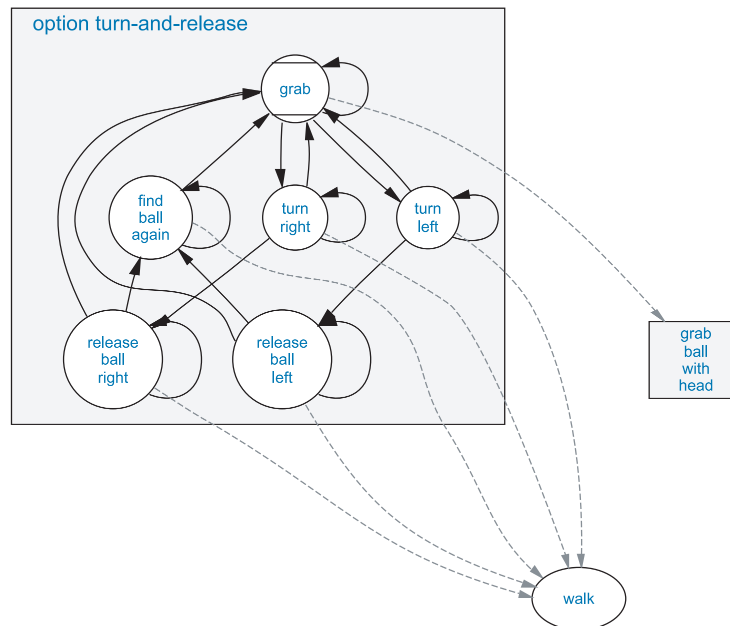


Figure 3.33: Option *"turn-and-release"* grabs the ball and pulls it around. The actual job of lifting the head in the right moment is done in the head control.

"angle". The head control mode is set to *"catch-ball"* in order to keep the ball grabbed. The special walk type *"turn-with-ball"* is set. With that, the robot uses almost only the hind legs for turning, the front legs enclosing the ball. After the difference to the target angle gets less than 110 degrees, in the states *"release-ball-left"* and *"release-ball-right"* the ball is released again. The head control mode is set to *"release-caught-ball-when-turning-right"* and *"release-caught-ball-when-turning-left"*. While the robot just continues to turn with walk type *"turn-with-ball"*, again the actual job is done by the head control. To give the ball a strong push with the outer leg, the head is lifted only if the current position in the walk cycle is between 0.77 and 0.85 when turning right or between 0.27 and 0.35 when turning left. If state and option *"approach-ball"* would be activated directly after that, the ball would be assumed to be lost as it indeed was not seen during turning. Therefore, in state *"find-ball-again"*, the robot has a chance to redetect the ball while slowly walking forward for maximum 500 ms.

3.8.1.6 Kicking

Kicking fast and precisely is crucial when playing robot soccer. Thus, the *GermanTeam* developed about 50 different kicks, suitable for almost all situations that can happen during a match. This large amount of specialized kicks requires a proper evaluation method to select which kick should be used in a certain situation.

Thereto, the *GermanTeam* followed three main goals for kicking: First, the robots should be able to play without any kicks. This goal was achieved first by implementing options like the above discussed *"approach-and-turn"* and *"turn-and-release"*. Second, there should be no

actions that try to establish a special situation in that a kick can be applied (like strafing or exact positioning at the ball). Instead, the robots should play the ball as if they were not using any kicks and kicks should be performed only if there was by chance an appropriate situation. And third, the kick selection itself should be more flexible, easy to extend, and, above all, not in *XABSL*, as it is indeed possible but very hard and time consuming to model and fine-tune the prerequisites of a kick in *XABSL*.

The goals two and three were achieved by introducing *kick selection tables*. A kick is retrieved from such a table by putting in the desired kick direction and the current x and y position of the ball. The look-up table stores for 12 discrete sectors (30 degrees each) of desired kick directions the start positions of appropriate kicks in a 1 cm wide grid, as shown in figure 3.34a) and c). To gain the table, a semi-autonomous teach-in mechanism was developed. Thereto, a robot stands on the playing field and kicks the ball several times with the same kick. Meanwhile, the starting position and the final position of the ball are measured relative to the robot. The results of such kick experiments can be seen in figure 3.34b) and d). For editing the kick selection table based on that data, a kick editor was developed.

As different situations on the field require different kicks, there are multiple kick selection tables. There are ones for the goalie, a field player playing in the center of the field, near the own goal, at the right border, at the left border, at the left opponent border, at the right opponent border, near the own goal, and near the opponent goal (cf. sect. 3.8.1.7).

To make the data stored in the kick selection table accessible to the *XABSL* behaviors, *XABSL* constants for the table and kick ids are generated automatically from the C++ implemented kick selection table. The decimal input function *"retrieve-kick"* is used to retrieve a kick, taking the desired kick direction and the id of the table to be used as parameters. If the returned kick is different from *"action.nothing"*, an appropriate kick was found for the current situation and the kick can be executed by using the option *"execute-kick"*.

3.8.1.7 Zones for Ball Handling

Some zones of the field require different behaviors than when playing in the center of the field. For instance, if the ball is at one of the sidelines, the robot has to turn behind the ball to avoid that the ball rolls out of the field. Near the own goal, the direction where to play the ball is not that important as to clear the ball just somewhere. And, at the opponent goal, there is much more precision needed than in the rest of the field.

Option *"handle-ball'* (cf. fig. 3.35) selects between these behaviors depending on where the ball is on the field. The initial state *"ball-in-center-of-field"* covers most of the area of the field, executing option *"handle-ball-in-center-of-field"*. At the borders and near the goals there are separate states, executing the corresponding ball handling options. To avoid oscillations between these states, a broad distance hysteresis was added there.

The options for the different zones combine approaching the ball, dribbling the ball, the *grab and carry* behaviors and select the kicks. How these behaviors are combined and which *kick selection table* is selected depends on the respective zone of the field.

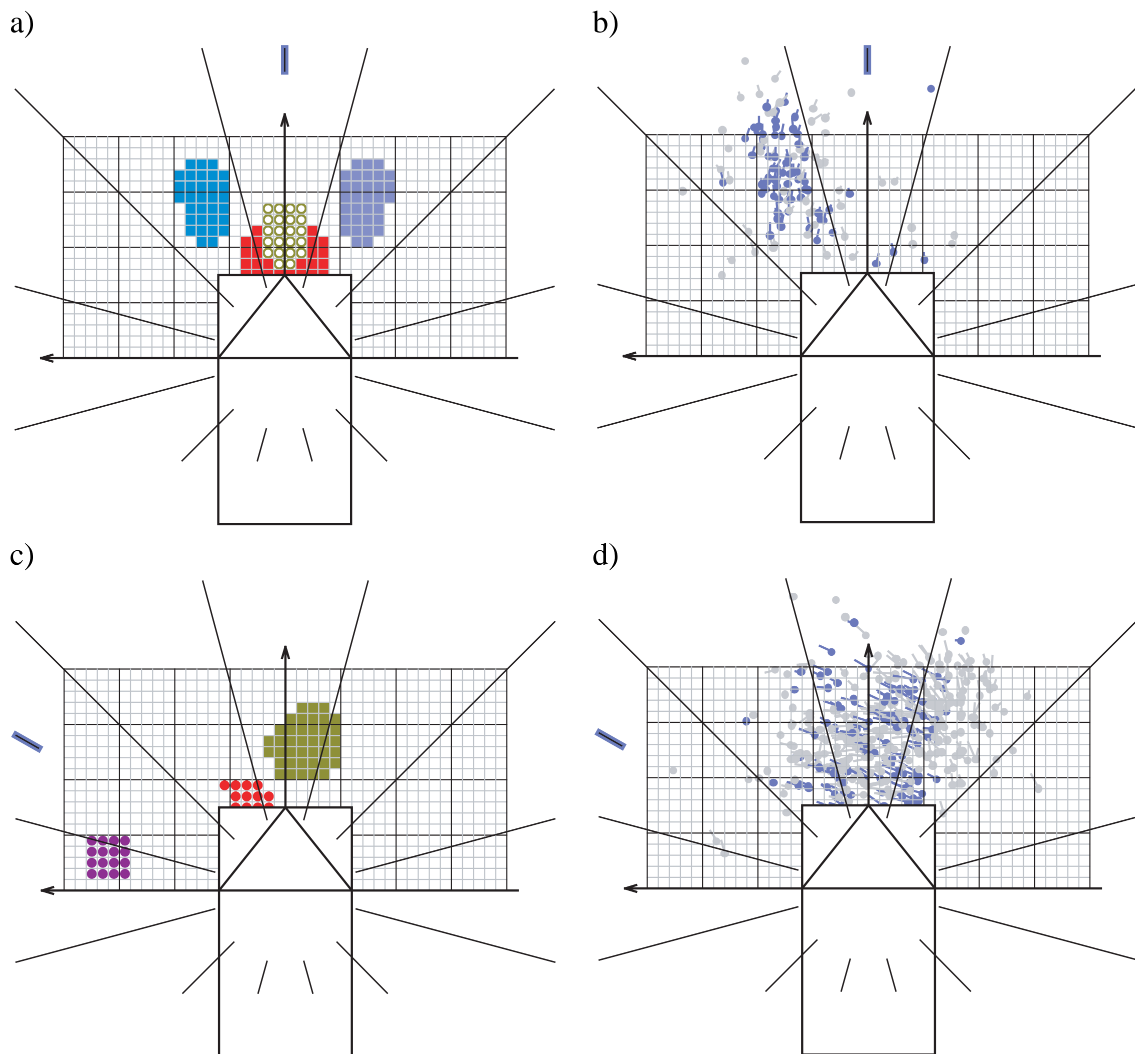


Figure 3.34: a) The kick selection table for the goalie when the desired kick direction is "forward" (in the sector between -15 and 15 degrees). If the current position of the ball is in the outer blue areas, the *"left-paw"* or *"right-paw"* kick is selected, in front of the robot (red area), kick *"chest-strong"*, and in a narrow range more distant in front of the robot (brown area) *"forward-kick-hard"*. b) Data recorded from kick experiments for the *"left-paw"* kick. The dots mark the position where the ball was perceived before the kick started. The lines out of the dots indicate in which direction and how far the ball was kicked in the experiment. All kick experiments in that the ball was kicked into the "forward" sector are highlighted blue. The area for *"put-left"* in a) was defined by taking these highlighted entries into account. c) The goalie kick selection table for the sector between 45 and 75 degrees. For the ball to the very left (purple area), *"put-left"* is selected, close to the robot (red area) *"hook-left"*, and in the brown area *"head-left"*. d) Kick experiments for the *"head-left"* kick, with those entries highlighted where the ball was kicked into the direction between 45 and 75 degrees.

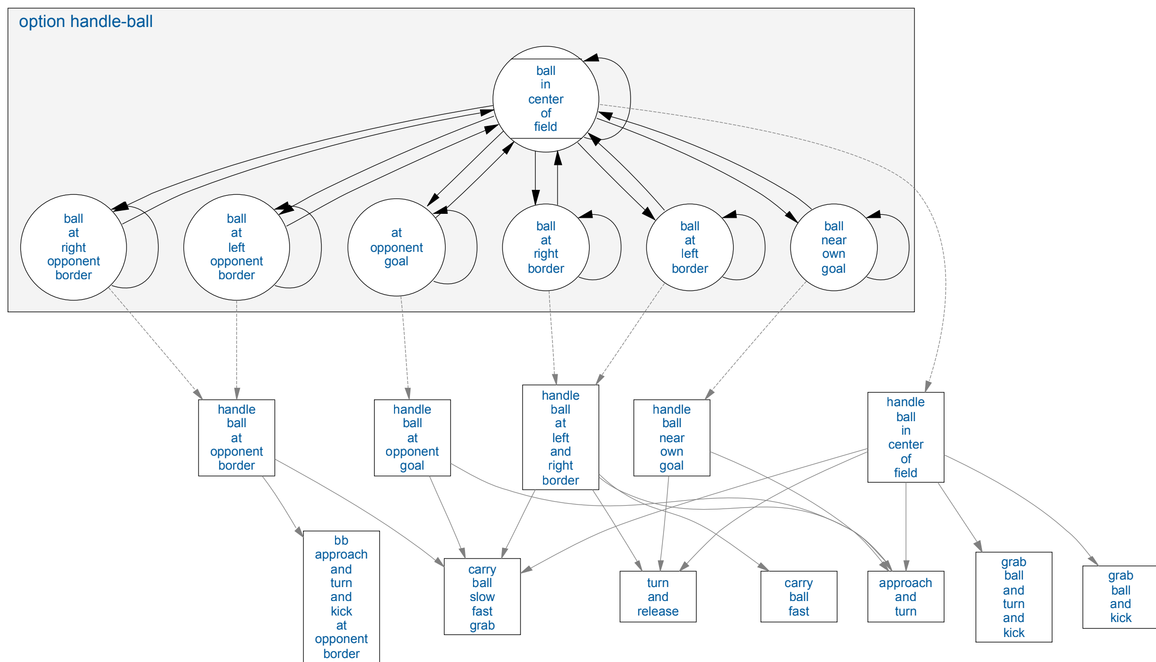


Figure 3.35: Option *"handle-ball"* selects between different behaviors for different zones on the field.

3.8.1.8 Transitions Between Ball Handling Behaviors

In the more high-level options, it is important to take into account when to do transitions between different behaviors. In general, all ball handling behaviors should be such that it is no problem to switch between them (which does not allow for strafing behaviors or behaviors for exact positioning for a certain kick). But there are some phases in behaviors such as ball grabbing, dribbling, or kicking, in that the behaviors should not be interrupted.

In *XABSL*, options have no chance to determine whether an option deep below in the option graph is in such a critical state. Therefore, the information whether the ball is handled at the moment is transmitted through an external variable, which can be queried through the Boolean input symbol *"ball.is-handled-at-the-moment"*. In the dribbling and kicking options, all states that execute a behavior that should not be interrupted set this variable by setting the enumerated output symbol *"ball.handling"* to *"handling-the-ball"*. In options higher in the option hierarchy, there are only transitions between states if *"ball.is-handled-at-the-moment"* is false.

Another principle for gaining smooth ball handling performance is that the higher the behavior in the option hierarchy, the less frequent transitions between states should be. A once selected behavior should always be continued unless there is a strong reason to change it.

Furthermore, if it happens that the ball is not seen anymore, the previously executed behavior should be continued until the ball is redetected (all ball handling options are based on *"approach-ball"*, which autonomously tries to redetect the ball using the *"turn-for-ball"* option). Therefore, there are only transitions in the higher options when the ball is just seen.

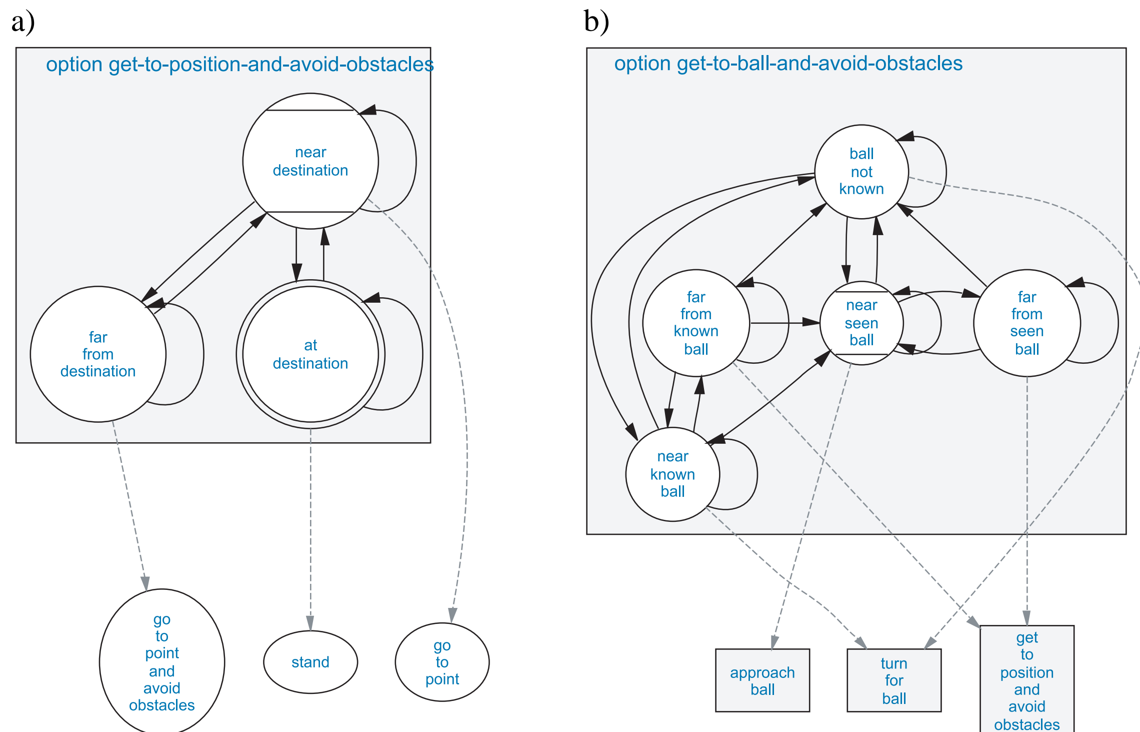


Figure 3.36: a) Option *"get-to-position-and-avoid-obstacles"* walks to a position avoiding obstacles on the way there. b) On top of that, *"get-to-ball-and-avoid-obstacles"* walks to the ball.

3.8.2 Navigation and Obstacle Avoidance

Navigation includes fast walking to a position with and without obstacle avoidance as well as positioning of the supporters (the players that do not handle the ball but try to reach a good position for support, pass interception, or defense).

3.8.2.1 Walking to a Position

There are two basic behaviors for walking to a position. First, *"go-to-point"* has the parameters *"x"* and *"y"* for the destination point, *"destination-angle"* for the orientation of the robot at the end, and *"max-speed"* for the maximum walk speed. As the rotation which is needed to reach the target angle is distributed over the whole distance to the target, it may happen that the robot walks backward.

Second, basic behavior *"go-to-point-and-avoid-obstacles"* uses the vision based obstacle model [27] to avoid obstacles on the way to the destination. Therefore, the robot has to walk forward to be able to detect the obstacles. The parameter *"avoidance-level"* defines how strict collisions shall be avoided. As it walks forward to its destination, it has no parameter for a target angle.

The option *"get-to-position-and-avoid-obstacles"* (cf. fig. 3.36a) combines these two basic behaviors. Far away from the destination, in state *"far-from-destination"*, *"go-to-point-and-*

avoid-obstacles" is used. As this basic behavior has problems near the target and as a target angle has to be reached, in state *"near-destination" "go-to-point"* is used. The distance from which on no obstacles shall be avoided can be set with the parameter *"no-obstacle-avoidance-distance"*. At the destination in state *"at-destination"*, the robot stops by using the basic behavior *"stand"*.

3.8.2.2 Walking to a Far Away Ball

The ball handling behaviors do not perform any obstacle avoidance and are therefore only executed near the ball. For longer distances, option *"get-to-ball-and-avoid-obstacles"* (cf. fig. 3.36b) is used.

For the ball position, there is a distinction between "seen" and "known". A "seen" ball position is a position that was modeled from perceptions made by the own camera of the robot. A "known" ball position is derived from a ball that was either seen or, after a time of 5 seconds in that no ball was seen, from a ball position that was transmitted over the Wireless LAN by team mates (the "communicated" ball position). As the "seen" ball position is measured and modeled relative to the robot, it is independent from localization errors. Instead the "communicated" ball position contains both the localization errors of the sending and the receiving robot and is therefore much more imprecise. That's why the "known" ball position can only be used to walk approximately into the direction of the ball but not for exact positioning near the ball or even ball handling.

If the ball is seen and far away, in state *"far-from-seen-ball"* the option *"get-to-position-and-avoid-obstacles"* is executed with a high speed parameter. If the ball is not seen but known and far, the same option is used in *"far-from-known-ball"* at a medium speed. Near the seen ball, in state *"near-seen-ball"*, option *"approach-ball"* is chosen. If the ball is not seen but known in the near, option *"turn-for-ball"* searches the ball, as from the short distance the robot would see the ball if the communicated ball position was correct.

3.8.2.3 Positioning

The *GermanTeam* employed artificial potential fields for the positioning of the supporters on the field [34]. The basic behavior *"potential-field-support"* has the parameters *"x"* and *"y"* for the destination point as well as *"max-speed"* for the maximum speed. Inside, a potential field of superposed force fields with repelling forces from obstacles, the own penalty area, and the ball, tries to navigate the robot to the requested point without collision and without obstruction of the ball handling robot. At the same time the body of the robot is always oriented towards the ball.

Amongst others, the option *"position-supporter-near-ball"* (cf. fig. 3.37) makes use of that basic behavior. It tries to support a ball handling robot by staying near the ball to be available if the other robot loses the ball for some reason. Additionally, opponent robots are pushed away or obstructed in approaching the ball.

The parameters *"x"* specifies the desired relative x offset in field coordinates and *"y"* the distance in the y direction to the ball. The actual side (in y direction) is chosen in the initial state *"choose-side"*. If the ball is at the left border ($y > 80$ cm), the robot positions right to the ball,

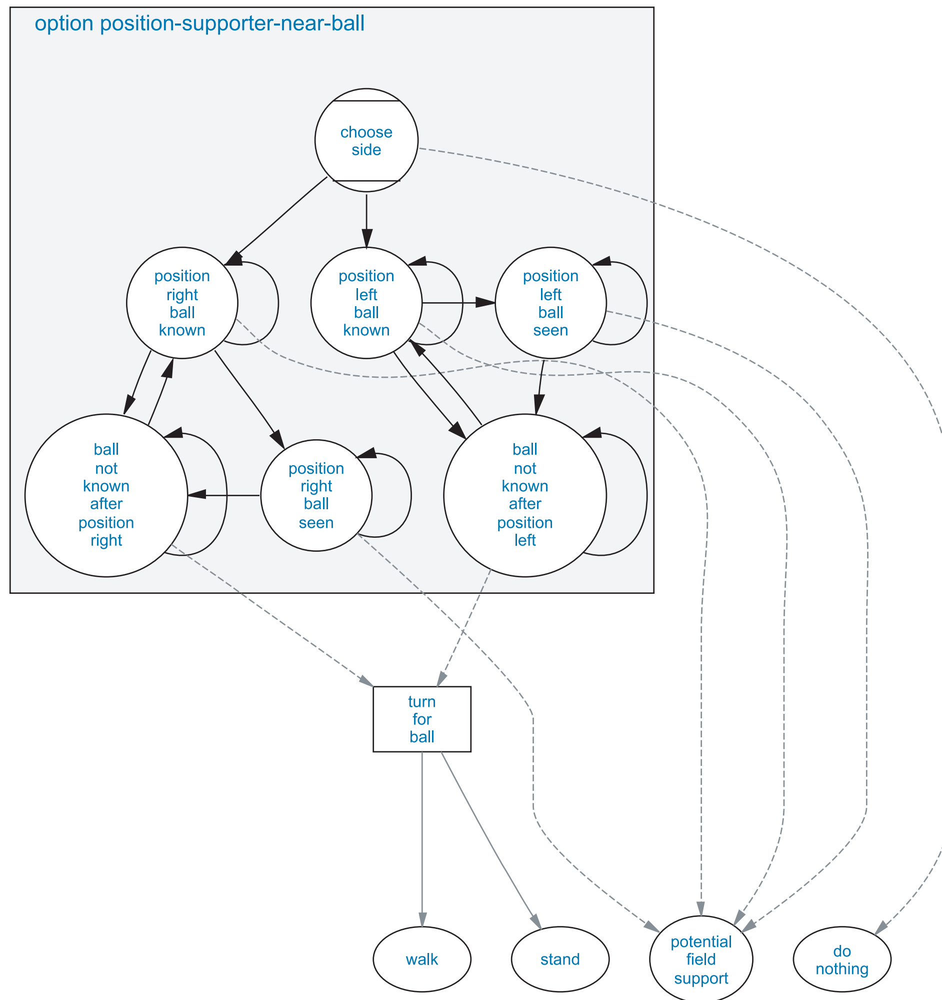


Figure 3.37: Option *"position-supporter-near-ball"* positions the robot near the ball. The speed is controlled depending on the reliability of the ball position.

vice versa at the right border. In the center of the field the robot chooses the side on that it is already.

As there is very often a crowd of robots around the ball, especially in games against weaker teams, the ball is often not seen, leading to an imprecise ball model. Therefore, the supporting robots try to keep calm and move cautious in order to stay well localized. For that, in the states *"position-left-ball-seen"* and *"position-right-ball-seen"* the maximum speed of movement is set to 350 mm/s second minus 20 mm/s for every second that the ball was not seen. If the ball is not seen but known (see above in sect. 3.8.2.2), in the states *"position-left-ball-known"* and *"position-right-ball-known"*, the robots walk only half that fast as the communicated ball position is very erroneous near the ball. For the case that the ball is neither seen or known, there are two states for option *"turn-for-ball"* in order to continue on the previous side if the ball is found again.

3.8.3 Player Roles

The four robots on the field have different roles. The player with the number one is always the goal keeper, the other three players change their roles dynamically. There is always only one robot at the same time that approaches the ball, the "striker". The "offensive supporter" positions in front of the ball or in the opponent half and the "defensive supporter" backs up from behind the ball and stays in the own half of the field.

3.8.3.1 Striker

The complete soccer playing behavior of the striker is implemented in option *"playing-striker"* (cf. fig. 3.38). In state *"get-to-ball"*, the option *"get-to-ball-and-avoid-obstacles"* (cf. sect. 3.8.2.2) is executed to approach the ball while avoiding obstacles on the way there. If the ball gets closer than 90 cm, in state *"handle-ball"* option *"handle-ball"* (cf. sect. 3.8.1.7) approaches and handles the ball without avoiding obstacles, as this would be disadvantageous. The back transition from *"handle-ball"* to *"get-to-ball"* is if the ball is farer away than 120 cm.

When the ball is inside the own penalty area (where the field players are not allowed to be in), in state *"ball-in-own-penalty-area"* the option *"position-striker-when-ball-is-inside-own-penalty-area"* positions the striker at the side of the penalty area, waiting for the goalie to clear the ball out of it.

Sometimes it happens that none of the four players of the own team is able to detect the ball for 12 seconds (for instance when two robots of the opponent team obstruct each other with the ball between them). Then, in state *"ball-not-known-for-long"* option *"search-for-ball"* walks along a fixed path between the left and right border of the field in order to redetect the ball. As also the supporters do that in other areas of the field, the whole field is covered and the ball is found again soon.

3.8.3.2 Supporters

The main task of the supporters is to position themselves well for pass interception, defense, and support of the striker. They cover the whole field in order to be able to be first at the ball, if the ball is kicked out of a crowd. At last, they try to stay away from the ball in order to not obstruct the striker.

The "offensive-supporter" stays most of the time in front of the ball and is implemented in option *"new-playing-offensive-supporter"* (cf. fig. 3.39a). If the ball is in the own half, in state *"ball-in-own-half"* the robot positions short behind the center line at the y position of the ball using option *position-offensive-supporter-ball-in-own-half*, waiting for a pass or a ball which is placed at a throw-in point. If the ball is inside the opponent half, in state *"position-supporter-near-ball"* the offensive supporter assists the striker by staying near the ball using option *"position-near-ball"* (cf. sect. 3.8.2.3). If the robot is still far away from the ball, in state *"get-to-far-ball"* the robot first walks there using option *"get-to-ball-and-avoid-obstacles"* (cf. sect. 3.8.2.2). If the striker plays the ball at the opponent border (which is detected through the position that the striker transmits over the WLAN), in state *"position-near-opponent-goal"*, the

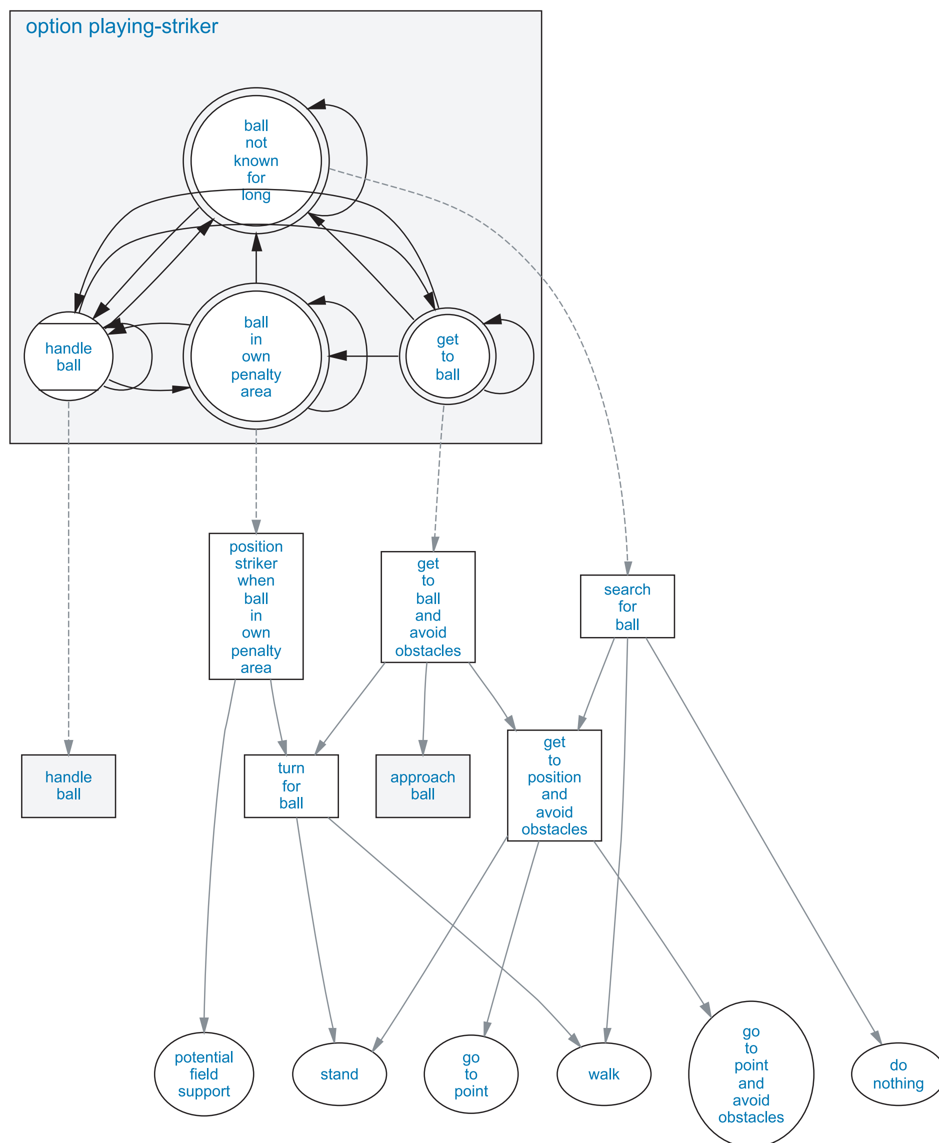


Figure 3.38: Option *"playing-striker"* implements a complete striker.

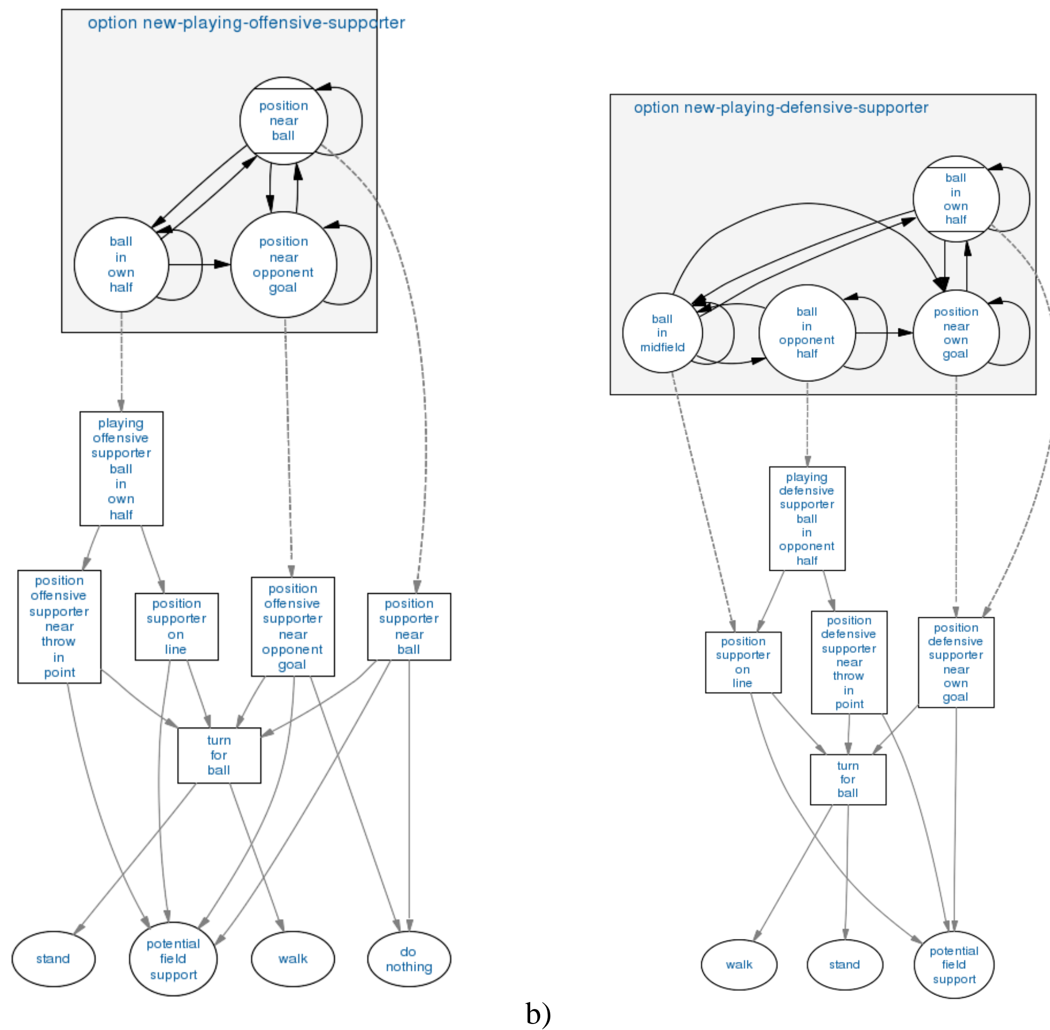


Figure 3.39: The options a) *"new-playing-offensive-supporter"* and b) *"new-playing-defensive-supporter"* decide where to position the robot.

supporter positions at the opposite corner of the penalty area using option *"position-offensive-supporter-near-opponent-goal"*, waiting for a pass or a failed kick of the striker.

Similar to that, the defensive supporter, implemented in option *"new-playing-defensive-supporter"* (cf. fig. 3.39b), mostly stays behind the ball. When the ball is in the opponent half (state *"ball-in-opponent-half"*), the robot positions itself in its own half using the option *"playing-supporter-ball-in-opponent-half"*. That options chooses whether the robot waits at one of the two throw-in points at the center line or keeps at the center of its own half following the movements of the ball in y-axis direction. The latter behavior is also executed, if the robot is in the state *"ball-in-midfield"*. If the ball is inside the own penalty area, the robot positions at the side of the penalty area opposite to the striker (state *"position-near-own-goal"* and option *"position-supporter-near-own-goal"*). Otherwise, it positions behind the striker to be there, if

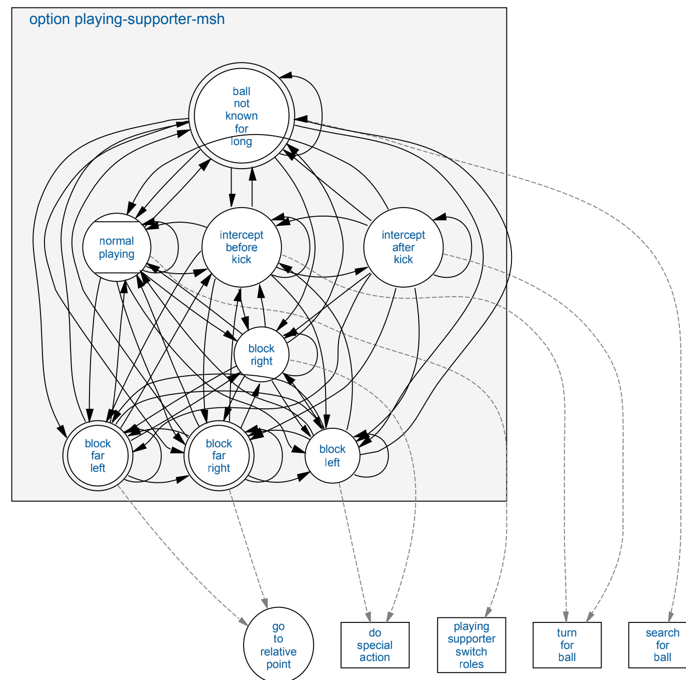


Figure 3.40: Option *"playing-supporter-msh"* intercepts kicks from the own team and blocks kicks of the other team.

the striker gets into difficulties.

Option *"playing-supporter-switch-roles"* selects between these two supporter options, depending on the role determined by the role negotiation process. It is executed from the initial state *"normal-playing"* of option *"playing-supporter-msh"* (cf. fig 3.40) for the positioning of the supporters. In state *"ball-not-known-for-long"* option *"search-for-ball"* is executed if the ball is not known for more than 12 seconds. If a ball is going to roll fast closely along the robot towards the own goal, it is stopped in states *"block-left"* and *"block-right"* by jumping to the side, or if the ball is rolling towards the goal but too far to be blocked it is attemped to catch the ball by walking sideways in states *"block-far-left"* and *"block-far-right"*. The actual analysis whether a ball can be blocked successfully is done in the ball locator module, storing the information in the ball model and providing it to the *XABSL* behaviors by the Boolean input symbols *"ball-rolls-by-left"* and *"ball-rolls-by-right"*.

In order to not loose the ball out of view when the striker kicks the ball somewhere, the striker notifies the other players on each kick through the Wireless LAN. Therefore, in all states of the ball handling options that prepare or perform a kick, the enumerated output symbol *"team-message"* is set to *"performing-a-kick"*. When the boolean input symbol *"another-teammate-is-performing-a-kick"* becomes true, in state *"intercept-before-kick"* the supporters stop position-

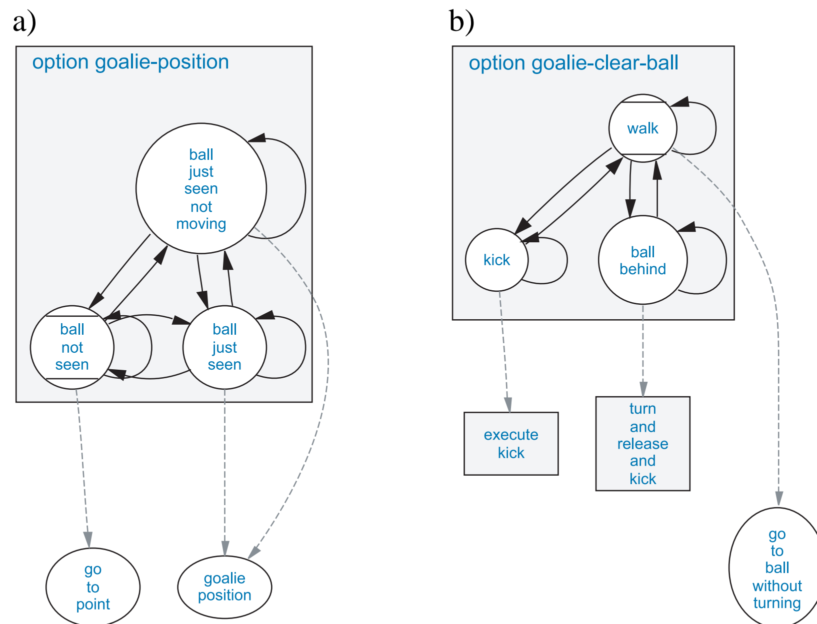


Figure 3.41: a) Option *“goalie-position”* positions the robot inside the goal. b) Option *“goalie-clear-ball”* tries to get the ball out of the penalty area.

ing but look only at the ball using head control mode *“search-for-ball”* and turn themselves for the ball, using option *“turn-for-ball”* (cf. sect. 3.8.1.1). After the striker finished its kick, *“another-teammate-is-performing-a-kick”* is not true anymore. In state *“intercept-after-kick”*, the robot still turns for the ball until the ball does not roll anymore (low ball speed), the ball passed the robot forward (x position of ball greater than of the robot), the ball is not seen anymore, or after a timeout of 3 seconds.

3.8.3.3 Goalie

The *GermanTeam* had one of the best defenses in the RoboCup 2005 tournament. This was achieved with an almost not moving goalie, standing at the right position for most of the time. As even small errors in the localization can make the robot believe that it is beside and not inside the own goal, the goalie behavior is much more dependent on good localization than the behaviors of the field players. The goalie behaviors have to support the localization with appropriate head movements and calm actions – it is very often a good strategy to let the goalie not move at all.

Option *“goalie-position”* (cf. fig 3.41a) makes use of the basic behavior *“goalie-position”*, which lets the robot position between the ball and the center of the own goal. To deal with errors in the localization, inside that basic behavior the robot’s position is corrected using the odometry and the ball position – the robot uses the ball as a landmark and the odometry is trusted more than the position provided by the self localization. If the robot does not move (the basic behavior requests a *“stand”* motion), in the state *“ball-just-seen-not-moving”* the head control mode is set to *“search-for-ball”* (the head looks only at the ball), allowing for a better detection of fast balls. Otherwise, in state *“ball-just-seen”* the self localization is supported by setting the head

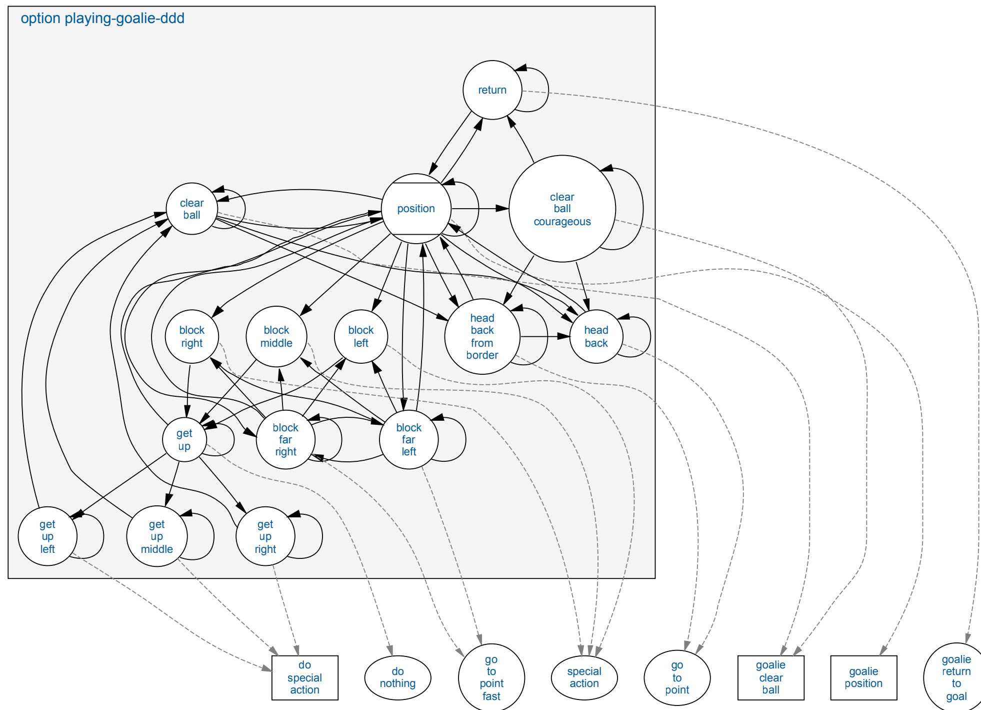


Figure 3.42: Option *“playing-goalie-ddd”* implements the goal keeper behavior.

control mode to *“search-auto”* (which scans also for landmarks). If the ball is not seen for 3.5 seconds, the robot walks to the center of the goal using basic behavior *“go-to-point”*.

Option *“goalie-clear-ball”* (cf. fig 3.41b) is responsible for the ball handling of the goalie. As there is mostly the striker and defensive supporter in the near, the task of the goalie is not to kick the ball very far (which requires strong and therefore dangerous kicks) but just to move it out of the penalty area. As there are often opponent robots that obstruct the goalie, no exact approaching of the ball is tried. Instead, in state *“walk”* the basic behavior *“go-to-ball-without-turning”* is used. Thereby the robot does not turn at all, which is faster than the normal *“go-to-ball”* basic behavior. If by chance the ball is in a good starting position for a kick (depending on a special kick selection table for the goalie, cf. sect. 3.8.1.6), in state *“kick”* a kick is performed. If it happens that the ball is behind the goalie (x position of the ball greater than the x position of the robot), in state *“ball-behind”* option *“turn-and-release-and-kick”* (cf. sect. 3.8.1.6) is used to clear the ball.

The complete goal keeper behavior is implemented in option *“playing-goalie-ddd”* (cf. fig. 3.42). In the initial state *“position”*, option *“goalie-position”* is selected. If the robot is far out the own penalty area for some reason, it returns to the goal in state *“return”* using basic behavior *“goalie-posion-return”*, which is faster than *“goalie-position”*. Similar to the supporters (cf. sect. 3.8.3.2), the goalie blocks fast balls by jumping left, right, or ahead (states *“block-middle”*,

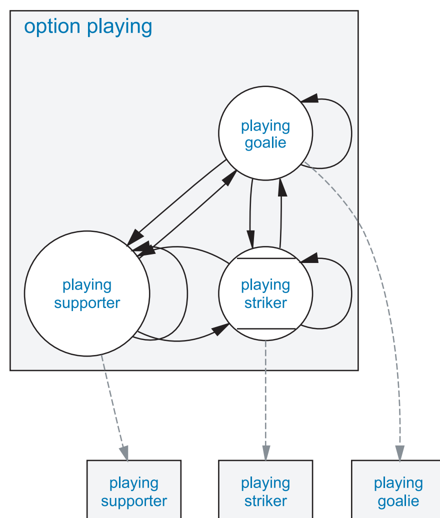


Figure 3.43: Option *“playing”* selects between the different roles.

“block-left”, *“block-right”*, *“block-far-left”*, and *“block-far-right”*). According to the position of the ball after a block, the goalie is able to perform a specific get up action, in order to obtain immediate control of the ball (states *“get-up-left”*, *“get-up-right”*, and *“get-up-middle”*).

Only when the ball is far inside the own penalty area (more than 20 cm over the line), in state *“clear-ball”* the option *“goalie-clear-ball”* is activated. There is already a transition back to state *“position”* when the ball is still inside the penalty area, 10 cm to the line. That’s why it happens very often that the ball is far inside the penalty area and the goalie does not move, standing between the ball and the center of the goal. But this is a very good strategy, as opponent strikers only have a chance to get the ball across a well positioned goalie when the goalie makes an error and opens a gap. Additionally, it can provoke that opponent strikers are taken out due to the “goalie-pushing” rule or that continuous pushes from the strikers let the ball roll out of the penalty area by chance. Only when there are no obstacles (opponent players) in the near, in state *“clear-ball-courageous”* the goalie also clears a ball that is in the outer parts of the penalty area and it returns to *“position”* when the ball is 7 cm out of the penalty area.

If the goalie does not see a previously seen ball anymore for more than two seconds, it is very likely that the ball is at the side of the robot, where it can not be redetected by scanning around with the head. Therefore, in state *“head-back”* the robot walks backwards to the rear wall of the own goal for maximum four seconds, hoping to redetect a ball that is at the side of the robot. If the robot is at the field border besides the goal and does not see the ball anymore, in state *“head-back-from-border”* it first walks to the center of the goal line in order to not collide with one of the goal posts.

3.8.3.4 Dynamic Role Assignments

Option *“playing”* (cf. fig. 3.43) assigns the different roles to the four robots. As only a specially marked robot is allowed to be inside the own penalty area, player one is always the goalie. But the

field players negotiate, which of them is the striker or a supporter. Therefore, all players transmit trough WLAN the time, how long they will approximately need to reach the ball. This time is computed such:

```
estimatedTimeToReachBall = distanceToBall / 0.2
  + 400.0 * fabs(angleBetweenBallAndOpponentGoal)
  + 2.0 * timeSinceBallWasSeenLast;
```

For every 10 cm to the ball it is assumed that the robot needs 500 ms to get there. The angle between the ball and the opponent goal is multiplied with 400 ms and added, preferring robots that are already behind the ball (no time is added) over robots that would have to grab the ball with the head or that would have to turn behind it (maximum 400 ms $\times\pi/2$ is added). In the last term, two seconds are added for every second that the ball was not seen, preferring robots that see the ball well.

For the role negotiations, the robot with the least estimated time to reach the ball is chosen to be the striker. To stabilize the decision, the player that is already the striker gets a time bonus of 500 ms. From the other robots, the robot with the higher x position (plus a bonus of 30 cm for the current offensive supporter) becomes the offensive supporter.

As an exception, if a supporter positions in front of the opponent goal (option *"position-offensive-supporter-near-opponent-goal"*, cf. sect. 3.8.3.2), it becomes immediately a striker if the ball is between the robot and the opponent goal.

If the WLAN does not work, a fallback with semi-fixed mappings from robot numbers to roles is applied: Player number two becomes striker if the ball is not far in the opponent half (x position of the ball less than 50 cm) and if the ball was seen in the last five seconds. Otherwise, it is a defensive supporter, staying in the own half. Players three and four become strikers when the ball is not far in the own half (x position of ball greater than -50 cm), otherwise they are offensive supporters. This can lead to situations (when the ball is in the center of the field) in that all three field players are strikers, which does not look very good.

The computed role is provided to the *XABSL* behaviors through the enumerated input symbol *"role"*. However, in option *"playing"* this role is not directly mapped onto the states for the different roles. For example, if the striker performs a kick or has the ball grabbed (the Boolean symbol *"ball.is-handled-at-the-moment"* is true), option *"playing"* remains in state *"playing-striker"*. Additionally, if the supporters intercept a pass (option *"playing-supporter"* is not in one of its target states), there is also no transition to other states. This is helpful if a ball is kicked in the direction of a supporter. It becomes only a striker when the ball passed the robot or if the ball does not roll anymore, preventing the robot from running into the wrong direction and possibly pushing the ball back.

3.8.4 Game Control

The *GermanTeam* supports the RoboCup Game Manager to minimize human interaction during the games. This program is operated by a co-referee and sends via WLAN the state of the

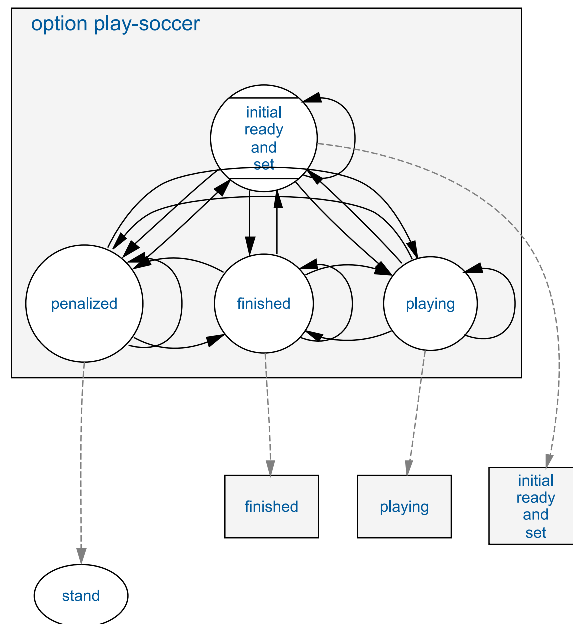


Figure 3.44: Option *"play-soccer"* is the root option of the option graph.

game (*initial*, *ready*, *set*, *playing*, *penalized*, or *finished*), the current score, the team color, and which team has kick-off to both teams. If the WLAN does not work for some reason, there is a sophisticated standardized interface to set these states manually through the buttons of the robot.

If all the game states would be implemented in one option, the number of transitions between states would be unmanageable high, as there are both transitions for messages from the game controller and button press events. Additionally, the *GermanTeam* added also transitions that are needed when the game controller is wrongly operated. That's why the implementation of the game control is distributed over three options: *"play-soccer"*, *"initial-ready-and-set"*, and *"initial-set-team-color"*.

The option *"play-soccer"* (cf. fig. 3.44) is the root option of the option graph. It has a state for the *"penalized"* game state where the robot does not move, a state for the *"finished"* game state where the option *"finished"* is executed (cf. sect. 3.8.5), and a state for the *"playing"* game state where the option *"playing"* (cf. sect. 3.8.3.4) is executed. All other game states are managed by the state *"initial-ready-and-set"*, executing the option with the same name. As the option *"initial-ready-and-set"* also executes a kick-off behavior when the *"playing"* message was received, *"play-soccer"* switches only from *"initial-ready-and-set"* to *"playing"* when *"initial-ready-and-set"* is in its target state, indicating that the kick-off behavior is finished.

The option *"initial-ready-and-set"* implements the game states *"initial"*, *"ready"*, and *"set"*, as well as the post-kick-off behavior. As the kick-off positions and the post-kick-off behaviors are different for own and opponent kick-off, there are always two option states for each game state.

In the beginning, if there was a goal, in the state *"own-team-scored"* or *"opponent-team-scored"* the corresponding option performs a short happy or sad cheering move (cf. sect. 3.8.5). When these options reach their target states, the option switches to the states for the *"ready"* game state.

In the *"ready"* states, the option *"go-to-kickoff-position"* lets the robots autonomously walk to their kickoff positions. These positions are read from input symbols to make them easy to configure. For own kick-off, one robot (robot four) is allowed to go to the center circle. If it gets close to that, in *"go-to-kickoff-position"* the state *position-exactly* becomes active, trying to position the robot very precisely and such that after the kick-off the robot can kick the ball straight ahead through the biggest gap between the opponents. Additionally, robot three positions at the center line close to the border. To avoid that opponent teams adapt to the *GermanTeam*'s kick-off strategies, there are different variants, which are selected randomly.

With the *"set"* message from the game manager or by touching the head button, the states for the *"set"* game state are reached. Before own kick-off, the option *"set-before-own-kickoff"* is executed. This option lets robot three, which positioned at the centerline, perform a different standing pose, allowing him a faster start after the kick-off.

After the *"playing"* message from the game controller or after a pressed head button, the states *"playing-after-own-kickoff"* and *"playing-after-opponent-kickoff"* become active, executing the corresponding options. In option *"playing-after-opponent-kickoff"* the target state is immediately reached for the goalie, robots three, and four. This lets the option *"initial-ready-and-set"* reach its target state *"playing"*, which again allows for a transition to *"playing"* in option *"play-soccer"*. But robot two keeps standing for 4.5 seconds to avoid that three field players run for the ball, possibly causing problems with the role negotiations. In the option *"playing-after-own-kickoff"*, the goalie and player two immediately start playing using the same target state mechanism. Player four performs a strong kick straight ahead. Player three runs blind with the "dash" walk type for 2 seconds along the border into the opponent half. If player four hits the gap between the opponent robots, player three can approach the ball at the opponent border before the opponent team does. However, this strategy worked well only against weaker teams.

If the robots are operated by hand, the option *"initial-ready-and-set"* is in the states for the *"initial"* game state at the beginning. Both execute the option *"initial-set-teamcolor"*, which allows for manual setting of team color through the back buttons.

3.8.5 Cheering and Artistry

The Sony Four Legged League is highly interesting to watch because the robots behave very life-like and the game is highly engaging. To make the games more enjoyable for the crowds, cheering (and crying) behaviors were implemented in addition to the wide range of kicks. After each goal, in option *"own-team-scored"* (cf. fig. 3.45a) one of four happy looking cheering motions is executed. After a few seconds, the option reaches its target state and the robots walk back to their kick-off positions. Accordingly, option *"opponent-team-scored"* selects between four annoyed and sad looking motions.

a) b)

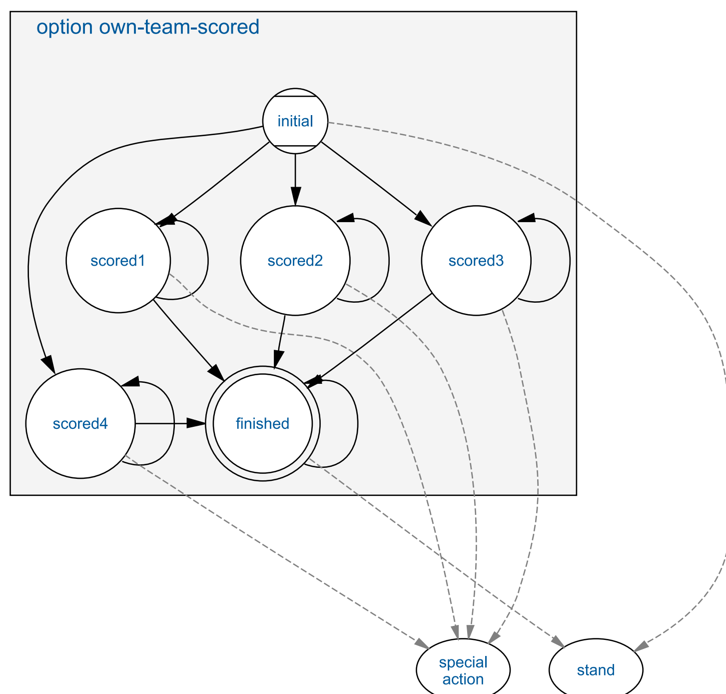


Figure 3.45: a) Option *"own-team-scored"* only chooses one out of four *"scored"* states, executes them for a few seconds, and then terminates. b) At the end of option *"finished"*, all robots walk to the center of the field for being also on the winner photo.

After the game, when the own team lost, in option *"finished"* the robots just let their heads hang down and behave sad. But when the own team won, the choreography is a bit more complex. All robots slowly walk to the center of the field. During this, every seven seconds, they stop walking and perform synchronously some cheering motions. After a while, all robots arrive in the center of the field and continuously perform headstands, which gives a good foreground for the winner photo (cf. fig. 3.45b).

Besides the cheering motions for the soccer games, many other demos and artistry choreographies were developed with *XABSL*.

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

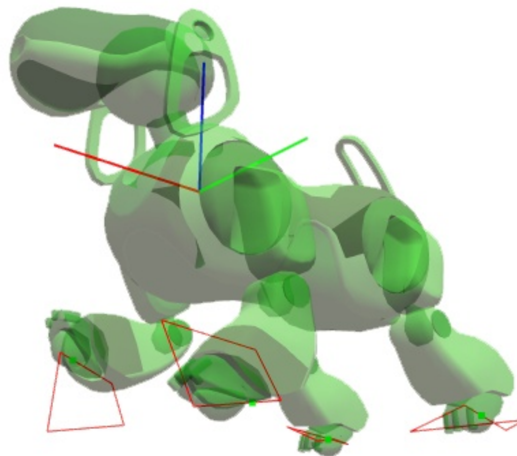


Figure 3.46: Walking by moving feet in 3 dimensional polygons

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.

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.

As a result of the actions requested from behavior control, the motion module produces a buffer containing current 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 can 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 based on the idea of the pWalk from UNSW [25]. A main feature of our implementation is that the engine and the parameters used are separated. The engine offers a huge set of parameters. This allows creating completely different walks with the same engine by having different parameter values. A class containing several sets of parameter values for different walk requests is given to the constructor of the engine. The engine itself interpolates between the provided parameter sets to generate a parameter set according to the walk request. Further it is possible to transmit new parameters via the wireless network from RobotControl to test the resulting walk at runtime.

3.9.1.1 Approach

The general idea is to calculate the position of the feet relative to the body while they move on curves defined by the parameters of the walking engine(cf. Fig. 3.46). The necessary joint angles to reach the foot position are calculated by inverse kinematics.

The control engine for the Sony Aibo in all the approaches described in the literature is based on static inverse kinematics, not taking dynamic effects into account. This because a complete dynamic model for this robot is not available, and it is also not computationally feasible for the on-board resources, which have to be shared with other processes such as cognition and image processing. As a result, there are significative differences between the controlled locus and the real locus that a foot describes during a walk; the loci of the fastest walks were differing the most [23]. These differences may result out of the idleness of the motors, kinematics errors etc. Consequently we created in 2005 a more flexible walking engine which controls the feet on a path described by a three dimensional polygon with less restrictions regarding the locus than our former approach.

For the direction of walking the four legs are more or less treated as wheels. Seen from above, the polygons are rotated to the desired walking direction (cf. Fig. 3.47). Every combination of walking forward, walking sideways, and turning results in turning around an Instantaneous Center of Rotation (ICR). This determines the step direction for each leg.

The walking speed is defined by the size of the polygons. The time for one step is constant but when walking faster the step length is bigger and equivalent to the speed a wheel would have at the same position.

3.9.1.2 Parameters defining a gait

As mentioned before, the actual walking style the engine generates is mainly defined by the set of parameters applied. Due to the mentioned observations on the real loci it appeared evident that giving more flexibility in the shape of the loci could give better results in terms of faster walks, particulary we introduced three dimensional polygons instead of the common two dimensional shapes.

Another goal was to keep the overall number of parameters low, because hand tuning them is not feasible, and the time to optimize the parameters with machine learning approaches greatly

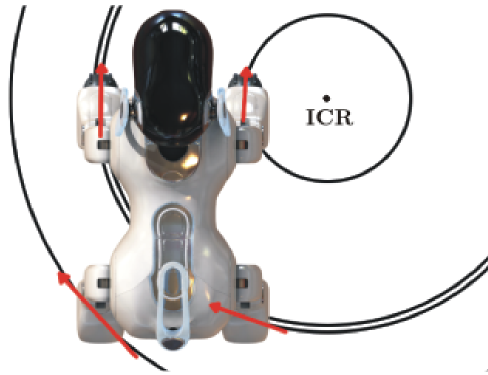


Figure 3.47: Principle of treating legs as wheels. Walking and turning combined results in a rotation around an Instantaneous Center of Rotation (ICR). Instead of really using wheels the robot makes steps (red) with the same direction and speed a wheel would have.

profits from a smaller search space, especially considering the long times required to evaluate the fitness.

So, we decided to use 3 dimensional polygons with n vertices $P_1, \ldots, P_n$. Additionally n timing parameters are needed to specify the amount of time needed for the foot to travel from vertex P_i to P_{i+1}, for $1 \leq i < n$, and from P_n to P_1. Further we restricted the walking gait to a trot gait, so two diagonally opposite legs are at the same time in the air, while the other remain on the ground. This results in a total number of parameters for a single leg of $3n + n$. Considering the symmetry between the left and right side of the robot, for a 4 legged robot we have a total of $8n$ parameters.

3.9.1.3 Combining several optimized parameter sets

Unfortunately a parameter set optimized for a certain walking direction, for example fast forward, is not necessarily good for walking sideways or backwards or rotating. For example, the fastest walk used by the GermanTeam in 2004 was only useful for straight forward walking with sidewards and rotational components almost null. Consequently, this parameter set was used only for "sprints" to the ball, while in the more common game situations the robot was switching to a slower but more neutral with respect to direction parameter set.

The problem with "hard" switching of parameter sets while walking is that the abrupt transition can cause stumbling and tripping of the robot, so we decided to implement a technique to smoothly interpolate among parameter sets. Thus, we can use optimized parameters for the main walking directions without incurring in the negative effects of stumbling or unwanted directional changes. To be able to interpolate between the parameter sets, we constrained them to have all the same number of vertices.

The used polygons can be classified into four groups. One group for walking forward, one group for walking sidewards and one group for the rotation. Figure 3.48 shows the matrix of 12 polygon sets which are used to generate by interpolation an optimal set for each walk request. The most important *group1* for forward/backward walking includes 5 polygon sets: good omni-

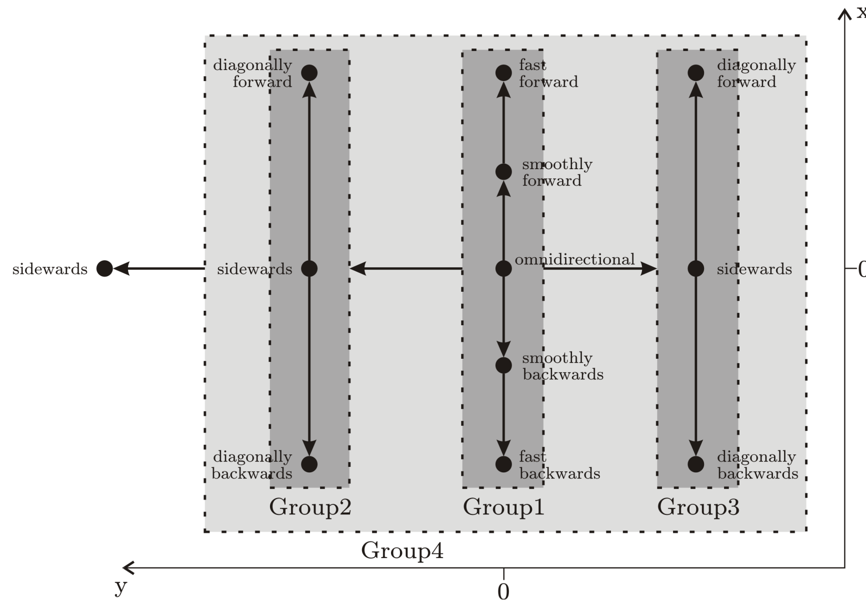


Figure 3.48: The interpolation scheme for a walk request

directional movement, moving smoothly forward/backward and moving fast forward/backward. The *group2* and *group3* contain the optimized polygons for walking sidewards and sidewards diagonally. *Group4* consits of only one polygon set for the rotational component.

To calculate a polygon set for a certain walk request, at first a polygon set for *group1* (see Figure 3.48) is calculated by linear interpolation between two sets of this same group according to the x-component of the request. Similarly, a second polygon set is generated for *group2* or *group3*, depending on the y-component of the request. Out of these two resulting intermediate polygon sets, a final polygon set for *group4* can be calculated. This suffices for motion requests having only a translational component. In presence of an additional rotational component, this polygon set has finally to be interpolated with the special set for rotations.

3.9.1.4 Odometry correction

In theory the robot should walk exactly as fast as its walking model calculates it, but obviously there is a difference to reality. To minimize this difference the GermanTeam used only few global correction factors until 2003. Such a correction factor is the quotient of maximum calculated speed and maximum resulting real speed. That gives a simple estimation for the distance the robot really walks when it moves its legs. The disadvantage is, that increasing the maximum speed also increases the average difference between calculated and real speed, because the relationship between those two is non-linear.

The introduction of three dimensional polygons as loci for the feet brings up a new problem for the odometry. While in the 2 dimensional, rectangular case, the ground phase of the feet and therefore the theoretical odometry was easy to measure, this approach fails for the case of 3 dimensional polygons. The ground phase of the feet can not be identified, this is especially true

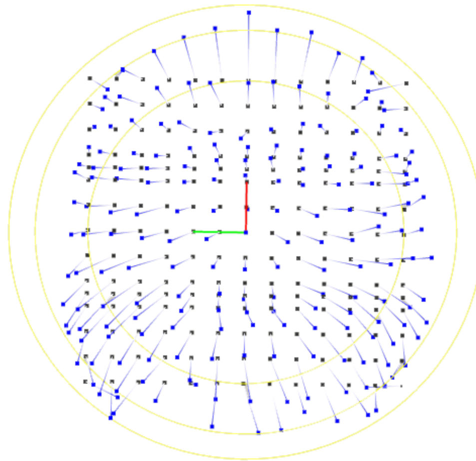


Figure 3.49: Visualization of the layer with translation movement only of the odometry correction table

for more than four vertices per polygon. Due to this fact, in a first approach we calculated the longest distance in x and y direction of the polygon and cut it at the maximum and minimum points into two halves, consisting of according segments of the polygon. The trajectory of these two halves of the polygon which is nearer to the ground was used as the ground phase and therefore used for calculating the odometry. The resulting odometry of this approach was poorer in accuracy than the ones of the walks which are based on the two dimensional polygons and the faster speed of the walk did not countervail the disadvantage of the less precise odometry information. Thus, we created 3 dimensional odometry correction tables which map a controlled speed to the real measured speed. The tables are 3 dimensional because of the ability of the robot to combine walking in x and y direction as well as rotation.

A layer of the odometry correction table is shown in Figure 3.49. The black dots represent the nodes of the grid, the blue dots show the according speed. The apparent disadvantage that the robots' real walking speed may differ from its controlled speed and only the odometry feedback is corrected using the look up tables is negligible because our closed loop low level behavior control continuously readjusts the walk requests to the current situation.

3.9.1.5 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:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = Rot_y(-q_1) Rot_x(q_2) Trans \begin{pmatrix} 0 \\ 0 \\ -l_1 \end{pmatrix} Rot_y(-q_3) Trans \begin{pmatrix} 0 \\ 0 \\ -l_2 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{3.30}$$

$Rot_{x/y}(\alpha)$ 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:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(q_1) & 0 & -\sin(q_1) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(q_1) & 0 & \cos(q_1) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(q_2) & -\sin(q_2) & 0 \\ 0 & \sin(q_2) & \cos(q_2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -l_1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\begin{pmatrix} \cos(q_3) & 0 & -\sin(q_3) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(q_3) & 0 & \cos(q_3) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -l_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{3.31}$$

Matrix multiplication results in

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(q_1)\sin(q_3)l_2 + \sin(q_1)\cos(q_2)\cos(q_3)l_2 + \sin(q_1)\cos(q_2)l_1 \\ \sin(q_2)l_1 + \sin(q_2)\cos(q_3)l_2 \\ \sin(q_1)\sin(q_3)l_2 - \cos(q_1)\cos(q_2)\cos(q_3)l_2 - \cos(q_1)\cos(q_2)l_1 \\ 1 \end{pmatrix}. \tag{3.32}$$

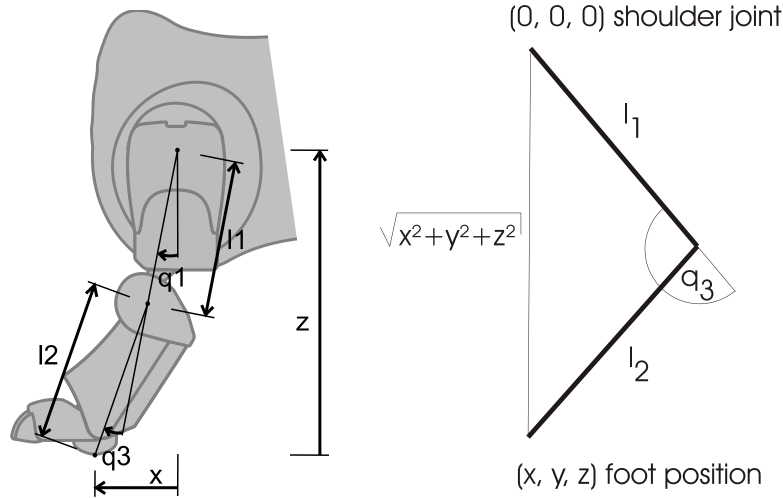


Figure 3.50: leg side view, calculation of knee joint q_3 via law of cosine

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.

According to the law of cosine (cf. Fig. 3.50)

$$\cos\alpha = \frac{l_1^2 + l_2^2 - x^2 - y^2 - z^2}{2l_1 l_2} \tag{3.33}$$

with upper limb length l_1 and lower limb length l_2.

With

$$|q_3| = |180° - \alpha| = \arccos\frac{x^2 + y^2 + z^2 - l_1^2 - l_2^2}{2l_1 l_2} \tag{3.34}$$

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 (3.32) (geometrically apparent cf.

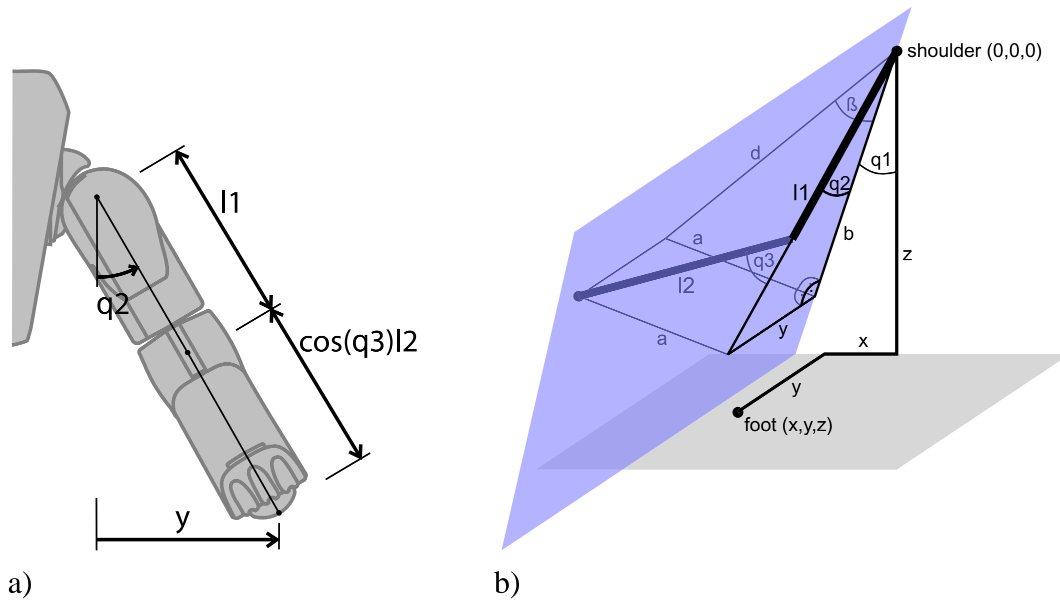


Figure 3.51: a) leg front view, calculation of shoulder joint q_2 b) leg model, calculation of shoulder joint q_1 with several helping variables

Fig. 3.51a)

$$\begin{aligned} y &= \sin(q_2)l_1 + \sin(q_2)\cos(q_3)l_2 \\ &= \sin(q_2)\left[l_1 + \cos(q_3)l_2\right]. \end{aligned} \tag{3.35}$$

Consequently

$$q_2 = \arcsin\left(\frac{y}{l_2\cos(q_3) + l_1}\right). \tag{3.36}$$

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

Calculation of shoulder joint q_1. Finally the joint angle q_1 can be calculated. According to equation (3.32) (cf. Fig. 3.51b)

$$\begin{aligned} x &= \cos(q_1)\sin(q_3)l_2 + \sin(q_1)\cos(q_2)\cos(q_3)l_2 + \sin(q_1)\cos(q_2)l_1 \\ &= \cos(q_1)\sin(q_3)l_2 + \sin(q_1)\left[\cos(q_2)\cos(q_3)l_2 + \cos(q_2)l_1\right]. \end{aligned} \tag{3.37}$$

When defining

$$a := \sin(q_3)l_2, \tag{3.38}$$

$$b := -\cos(q_2)\cos(q_3)l_2 - \cos(q_2)l_1 \tag{3.39}$$

and

$$\beta := \arctan\left(\frac{a}{b}\right), \tag{3.40}$$

$$d := \sqrt{a^2 + b^2} \quad \left(= \frac{b}{\sin(\beta)}\right), \tag{3.41}$$

so that

$$a = d\cos(\beta), \qquad b = d\sin(\beta), \tag{3.42}$$

equation (3.37) simplifies to

$$\begin{aligned} x &= \cos(q_1)a - \sin(q_1)b \\ &= d\left[\cos(q_1)\cos(\beta) - \sin(q_1)\sin(\beta)\right] \end{aligned} \tag{3.43}$$

which can be transformed to

$$x = d\cos(q_1 + \beta). \tag{3.44}$$

Hence

$$|q_1 + \beta| = \arccos\left(\frac{x}{d}\right). \tag{3.45}$$

The sign of $q_1 + \beta$ can be obtained by checking the z-component of equation (3.32). As in equations (3.37)-(3.44) this results in:

$$\begin{aligned} z &= \sin(q_1)\sin(q_3)l_2 - \cos(q_1)\cos(q_2)\cos(q_3)l_2 - \cos(q_1)\cos(q_2)l_1 \\ &= \sin(q_1)\sin(q_3)l_2 - \cos(q_1)\left[\cos(q_2)\cos(q_3)l_2 + \cos(q_2)l_1\right] \\ &= \sin(q_1)a + \cos(q_1)b \\ &= d\left[\sin(q_1)\cos(\beta) + \cos(q_1)sin(\beta)\right] \\ &= d\sin(q_1 + \beta). \end{aligned} \tag{3.46}$$

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.

Anatomy Corrections. The above calculations assume that a vertically stretched leg results in an arc of 0 degrees at all joints. Whereas this might be true for a wire netting model, it is incorrect for an Aibo. Adding constant offsets as in Fig. 3.52 compensates for this difference.

3.9.1.6 Gait Evolution

The task to find a fast and effective parameter set describing the walk becomes more and more difficult with an increasing number of parameters. Finding the fastest possible walk using a walking engine with n parameters means to find the representation of the fastest walk in an n-dimensional search space. For a large number n this is not feasible by trying different parameter combinations by hand. Until 2004 the GermanTeam used two different approaches to optimize the gait parameters. They mostly differed in the way, the speed of the resulting walk and therefore the fitness of an individual has been determined. One approach used the self localization,

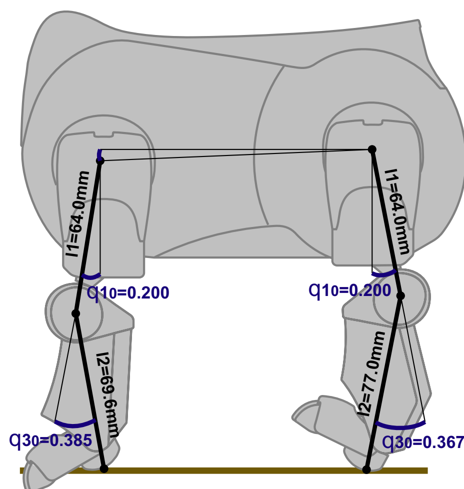


Figure 3.52: Sending an angle of 0 degrees to all joint motors results in vertical legs but not in angles of 0 degrees between the joints.

the other one used the touch sensors of the hind legs to measure the covered distance of a feet on the ground [51]. While the approach based on self localization needs a lot of time to determine the speed and the resulting measurements are quite noisy, the other approach based on the proprioceptive touch sensors restricts the hind legs of the robots to move in a way, that the sensors have ground contact while walking.

Due to these disadvantages of both approaches and the fact that the number of parameters which define our walk has significantly increased, we used an external ceiling camera to measure the speed of the robot both accurately and fast. We were able to identify the fitness within 3 seconds for every individual. We applied different learning techniques to this problem and found out that the so called (μ, λ) evolution strategy with self–adapting mutation strength leaded very quickly to a (local) optimum in this high dimensional search space [5]. Finding the fastest forward walk needed only 45 minutes of training. The table in Figure 3.53 shows the resulting maximum walking speeds.

forward	backwards	sideways	diagonal forward	diagonal backwards	rotation
45.1 cm/s	40.5 cm/s	34.4 cm/s	42.1 cm/s	43.5 cm/s	200 °/s

Figure 3.53: Maximum walking speeds of the GermanTeam

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 makes it possible to play the ball from different positions relative to the robot to various directions. The behavior is responsible for choosing a useful 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 are being executed. Transitions make it possible to define conditions, i. e. 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 a C data structure with its own compiler described in section 5.5. The generated data is loaded from 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. This ensures that 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 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. Additionally the table may contain entries giving a constant speed for a special action. The table also contains an indication of the walk cycle the special action is suitable to be executed in, when switching from walking to the special action. This can smooth the overall motion.

Teaching and Inverse Kinematics. Special Actions are recorded by “teaching”. The robot’s joints are brought in the desired position(s) by hand and are then recorded. A full motion is constructed by playing back a squence of such snap shots. The sequence can later be edited and optimized.
To create more life-like motions that utilize the robot’s entire body, inverse kinematics is used in much the same manner that it is used for generating the walking motions: The four paw positions on the ground are specified and the relative position and orientation of the robot’s trunk is also be specified. The joint angles are calculated from this data using inverse kinematics. Using this approach, robot motions using all of its legs synchronously can easily be created, allowing for much quicker and stronger motions than the ones usally realized by teaching. (The “head kick” is an example of a kick that was designed using inverse kinematics. It builds up additional momentum by rotating the robot’s body, adding force to the actual head kick.)

3.9.3 Head Motion Control

The module *HeadControl* calculates gaze directions for the robot. It receives *HeadControlModes* 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* which forwards them directly to the motors (cf. Sect. 3.9). *HeadControl* receives sensor data and the internal world model and it is part of the *Motion* Process.

The Aibo ERS7 head has three degrees of freedom: neck tilt, head pan, and head tilt. This is in contrast to the Aibo ERS210 which has the ability to roll its head (instead of the second tilt). This required some rework of the existing inverse kinematics.

The *HeadControl* for the ERS-7 was mostly rewritten to get a more intelligent *HeadControl*. Our approach to improve the quality of the *HeadControl*, was to develop a behavior, which gets more information about the own position on the field during watching the ball. This means, the gaze needs to be adjusted in a way, that the ball and any useful landmarks are in sight. Additionally, the gaze direction should only change if it is necessary and spend most of the time on watching the ball to prevent losing the ball and disturbing the speed measuring of the ball locator. Another concern was to guide the gaze towards the positions of landmarks to improve the quality of the selflocator results.

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 and gaze directions were developed.

Additionally, XABSL was used to model the different modes and states of attention.

It was also found out in experiments that the ERS7 has considerable more trouble tracking the ball compared to the ERS210 due to the new robot design (the quality of the camera images is not as good and the servo motor response is more sluggish).

The actual robot behavior can communicate with the Head Control by means of setting a *HeadControlMode*. This allows the behavior module to request certain attention modes. The most important ones are “search-for-ball” and “search-auto”. The former forces the camera to track the ball only (e.g. when the robot is about to kick the ball) whereas the latter allows the robot to look at landmarks too (e.g. when it is far away from the ball or needs to re-localize).

3.9.3.1 Geometric Considerations

To direct the robot’s gaze in the direction of a given target, head joint angles are calculated using inverse kinematics. This is outlined in the following paragraph. Furthermore, geometric considerations regarding the landmarks and optimizing gaze direction are discussed in the following paragraphs. The following is implemented as helper methods.

Look At Point. This method is used extensively in the head control to determine the gaze direction. From the coordinates $\vec{r} = (x, y, z)$ in the robot coordinate system, head joint angles are calculated. For the ERS210, a unique analytical solution could be derived. In case of the ERS7, two of the joints (“neck tilt” θ_1 and “head tilt” θ_3) are not independent. This means that

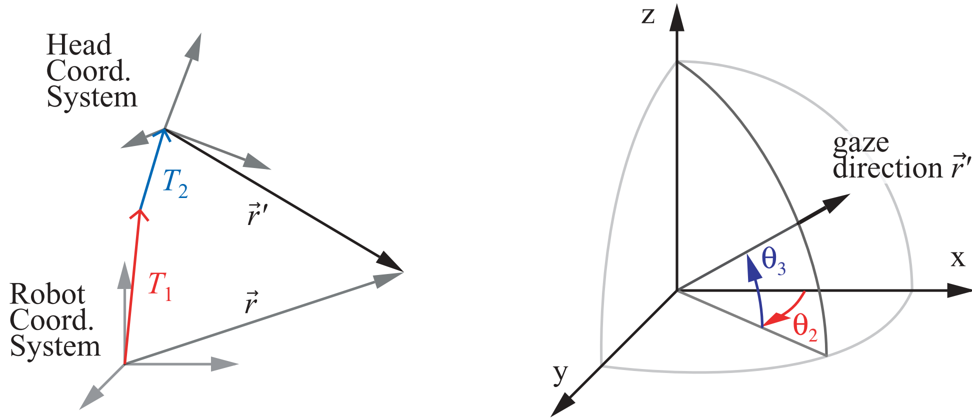


Figure 3.54: *Left:* the target point is transformed from the robot coordinate system to the shoulder coordinate system (T_1) to the head coordinate system (T_2). *Right:* transformation from cartesian into spherical polar coordinates yields the desired joint angles.

more than one unique solution exist: the robot can use both of the two joints to raise its head (and any combination of the two). The neck tilt joint is located at the robot's shoulders (at the base of its neck) whereas the two other joints are located at the end of the neck (we assume them to be at exactly the same point).

When deriving a solution, the assumption is made, that it is desireable for the robot to have the neck joint as far up as possible. This means that the robot's head is as far away from the ground as possible thus minimizing the distance error when looking at something on the ground. Setting the neck tilt to zero (straight up), the problem is reduced to two degrees of freedom which can be solved analytically.

To derive a solution, the target point is transformed from the coordinate system relative to the robot into the "shoulder coordinate system" (this is where the neck tilt joint is attached). Since the neck joint is set to a constant, the transformation into the "head coordinate system" can be performed. In this coordinate system, the remaining calculations are basically a transformation from cartesian coordinates into spherical polar coordinates (see fig. 3.54):

$$\theta_1 = \theta_c, \qquad \theta_2 = \arctan\frac{y'}{x'}, \qquad \theta_3 = \arctan\frac{\sqrt{x'^2 + y'^2}}{z'} \tag{3.47}$$

where $\vec{r}_{\text{head}} = (x', y', z')$ is the vector pointing in the desired gaze direction in the head coordinate system, and θ_c denotes the constant angle of the neck tilt.

There are cases, however, where the solution exceedes the physical limits of the robot's head tilt joint, $\theta_{\text{total tilt}} > \theta_{3,\text{ max}}$ (e.g. if the target is very close). In this case, the robot can look at the target if it uses the neck tilt. To achieve this, the head tilt is set to the maximum and neck tilt is set to the the missing tilt: $\theta_3 = \theta_{3,\text{ max}}$ and $\theta_1 = \theta_{\text{total tilt}} - \theta_{3,\text{ max}}$. This is, of course, only an approximation and more accurate solutions can be found. The presented solution turned out to have a big advantage which is not immediately obvious when looking at the calculations: when the robot tries to look at something that is not quite within reach, it will look over its shoulder

whereas other solutions tended to look between the robot's feet. Looking over the soulder has two advantages: the robot sees more of its surroundings (it is more likely to see the ball and landmarks) and the joints are closer to their "default" values (this means that very small head movement has to be performed once a target actually comes into view).

Calculate Closest Landmark. To be well localized, the robot needs to frequently look at landmarks. To minimize the time the robot needs to look away from the ball, the closest landmark with regard to the current gaze direction (or any other desired direction) is calculated. This is done by calculating the relative angle to all landmarks and then determining the minimal absolute angle with respect to the reference angle.

The actual head motion that we want to achieve is a sweeping motion that tries to look at as many landmarks as possible along the way (until it reaches a time limit or the physical joint limit). These scans are performed alternately left and right. Therefore, the algorithm was adjusted to also calculate the closest landmark in a given direction w.r.t. the current gaze direction (this is done by taking into account the sign of the relative angle to the landmark).

In experiments, it was observed that small errors in the localization caused the calculation of the closest landmark to oscillate. To make the scanning motion robust against such oscillations, the current result of the calculation is compared to the last result and the current scan direction, making sure that the direction of the scan is maintained.

Look At Ball And Closest Landmark. When looking only at the ball, landmarks sometimes come into view (e.g. the goal). The gaze direction of the robot can, however, be altered in such a way that the robot still has full view of the ball but is also able to see landmarks that are close (and would otherwise be outside its field of view). Fig. 3.55 illustrates how the gaze direction can be altered when landmarks are near (this is particularly important for landmarks that can only be used if they are fully in the field of view of the robot).

To determine if two objects can be looked at simultaneously, their relative angle to the camera is calculated. If the difference between the two is smaller than the opening angle of the camera, the two objects' centers can be looked at simultaneously. To make sure that the objects fully fit into the image, their angular width needs to be considered too. This is done for the two dimensions independently (pan and tilt).

The gaze direction is changed from being centered on the ball to a direction where the robot can just see the landmark. This is important for ball tracking to still function reliably. Also, some buffer is added at the image edges (lessening the effective opening angle) so that the ball is always fully within the field of view and cannot be lost easily and to compensate for the sluggishness of the robot's head joints.

If the ball is close to the robot, the gaze direction remains fixed on the ball. On the one hand, this is because the ball takes up most of the camera image and effective adjustment is no longer possible. On the other hand, the assumption is made that if the ball is close, the robot is about to kick it (or interact with it in some other way) so it is of greatest importance to know exactly where the ball is.

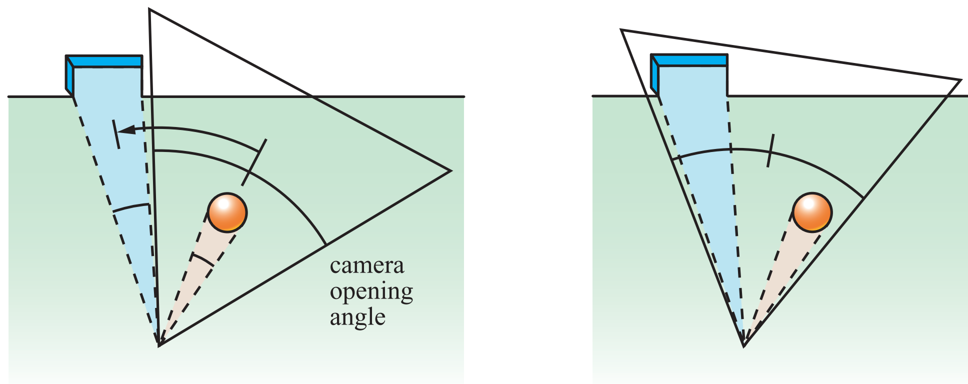


Figure 3.55: Top down view of the field of view of the robot. In the left diagram, the robot's field of view is centered on the ball. The right diagram shows the optimized gaze direction which enables the robot to perceive the goal too.

3.9.3.2 Head Path Planner

Often the head describes simple movements to collect information, like scanning for beacons and landmarks. To provide an easy tool, the *HeadPathPlanner* was developed. The planner is initialized with an array of joint settings and timings. It calculates a path along the given angles. By repeatedly calling a function for every clock step the angles for the head joints are returned, until the last position of the path is reached.
The planner takes the starting head position and the speed limits of the head joints for a optimal head movement into account. The speed limits can be retrieved by using the *HeadControlMode* calibrateHeadSpeeds. This function has been initiated by the observation that the joint speeds differ from robot to robot.

3.9.3.3 Landmark State

Because there is no information stored in the selflocator about the seen beacons, the class *LandmarkStates* stores the time of sight for every single beacon. This is used by the *HeadControl* to look, if possible, on different beacons, which improves the quality of the selflocator. In the *HeadControl* behavior the time between the two last different beacons is needed to decide, whether to look at ball or beacons in search-auto mode. Furthermore, if no beacon was seen recently, the *HeadControl* starts a search for beacons to avoid a total dislocation.

3.9.3.4 State Machine

The robot faces various situations during the game. Those situations require the robot to direct its attention towards different targets (as the ball when chasing the ball or landmarks when trying to (re-)localize). This can be modeled in a state machine that controls the robot's gaze direction. Roughly, the following situations can be distinguished: look at the ball, look at landmarks, search for the ball when the ball has been lost. A tradeoff has to be made between looking at the ball

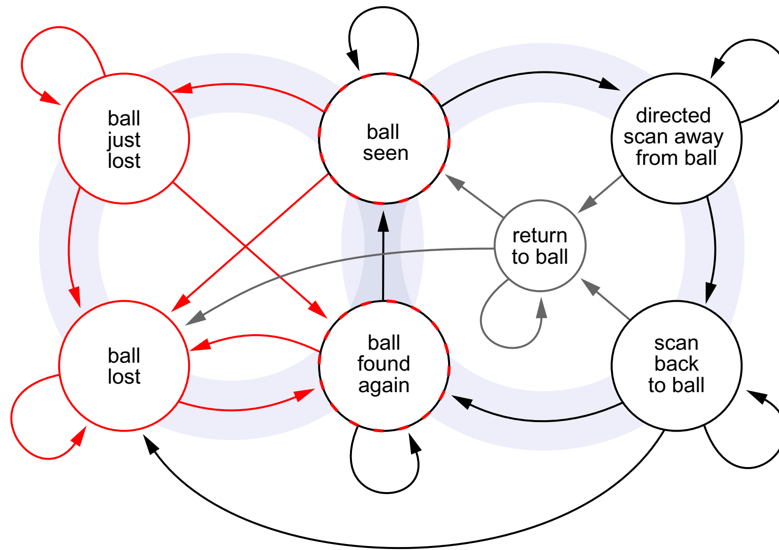


Figure 3.56: State machine used for tracking the ball. On the right side of the diagram, states are shown related to "intentionally not looking at the ball but at landmarks instead". To the left (highlighted in red) the states are shown that the robot passes through/visits when it has lost sight of the ball.

as much and as long as possible and looking at or scanning for landmarks. Both are equally important. Looking only at the ball for prolonged periods of time causes localization to deteriorate for two reasons: first of all, the opening angle of the robot's camera is limited and too few landmarks come into view when the robot is only looking at the ball. On the other hand, odometry data is of relatively poor quality requiring the robot to frequently perform vision updates for its localization.

The state machine is shown in figure 3.56. There are three types of states that can be grouped together: states that are active when the ball is seen, those that are involved in intentionally looking away from the ball, and those that are active when the ball is lost. The individual states are described here and some of the transitions are pointed out (for a more detailed description, please use the generated XABSL documentation):

Ball Seen. The robot sees the ball and is looking at it.

Ball Just Lost. The robot has recently seen the ball but is not currently seeing it. The gaze direction is the direction where the ball was recently seen. This is a step between Ball Seen and Ball Lost to not starting a ball search, if only the ball is partially covered or the image processor could not find the ball during some frames.

Ball Lost. The robot has not seen the ball for quite some time and is performing left and right scanning movements with its head. Once the ball has been found, there is a transition to the state "Ball Found Again".

Directed Scan Away From Ball. When the ball has been seen for a certain period of time, the robot starts a scanning motion to the right or the left side off the ball. This motion directs the camera towards landmarks. The state will be exited either if the scanning motion has been performed for a certain time period, if the head has reached its joint limits, or if there is no landmark in the direction of the scan (e.g. if the next landmark in the scanning direction is behind the robot).

Scan Back To Ball. This state returns the robot's gaze to the ball. The state is necessary because the robot does not see the ball but "knows" where the ball is, in other words the "ball is not lost".

Ball Found Again. This state is reached when the ball is seen again after it has not been seen. This state stabilizes the robot's gaze on the ball because if the ball was not seen before, it will most likely re-appear at the edge of the field of view and percepts may be of poor quality. Additionally, this state gives the ball locator some time to calculate the ball speed before the *HeadControl* begins to look at ball and landmarks.

Return To Ball. This state is used to quickly return the robot's gaze towards the ball. If the *HeadControlMode* was set to search-auto and the robot is performing a scan for landmarks, setting the *HeadControlMode* to search-for-ball causes a transition to this state. It does essentially the same as "Scan Back To Ball" only quicker.

Other Modes. This state catches every head movement which is not implemented in an own Basic Behavior.

3.9.3.5 Basic Behaviors

The Basic Behaviors are used to describe atomic behaviors. For a more detailed description, please use the generated XABSL documentation

look-at-ball. This will only look at the seen ball position.

look-around-at-seen-ball. This Basic Behavior is used by Ball Just Lost to begin a ball search. The resulting head movement descibes a rectangle around at the last seen ball position.

directed-scan-away-from-ball. This calculates the smallest head pan angle to the next beacon. On the results of this calculation, the side to look at is choosen. While this Basic Behavior is active a scan for more beacons is done, until the head pan joint reaches its stop.

look-at-ball-and-closest-landmark. If the *HeadControlMode* is in search-auto, the headcontrol attempts to adjust the gaze simultaneously towards the ball and one landmark. The landmark which will be chosen depends on the distance and the angle between landmark and ball. For instance, far landmarks on the ground can probably not be detected by the image processor or a big angle increases the probability that the ball rolls out of sight when moving fast.

directed-scan-for-landmarks. This Basic Behavior calculates the closest landmark and moves the head towards it, depending on the last scanned side. If a landmark is reached, the closest one will be calculated and aimed. This repeats until the head pan angle limit is reached. The gaze stops for some time at every landmark to avoid blurred images of the landmark.

other-head-movements. This Basic Behavior is used for simple head movements to avoid a Basic Behavior implementation of every single, simple movement like Look Left, Look To Stars, Look Between Feets, etc. Most of the moves, which are implemented here, use the *HeadPathPlanner* or set head joints directly.

Chapter 4

Challenges

The RoboCup competition consists besides the robot soccer tournament also of so called *challenges*. The goal of these challenges is to give teams a platform to present their research in addition to the robot soccer and to look towards possible rule changes for the soccer matches like e.g. reducing the amount or position of the beacons around the field. The ideas and algorithms used by the GermanTeam to solve the technical challenges are presented in this chapter.

In the so called OpenChallenge we presented our debug mechanisms and our related debug Tool RobotControl which is described in chapter 5.1 and Appendix G.

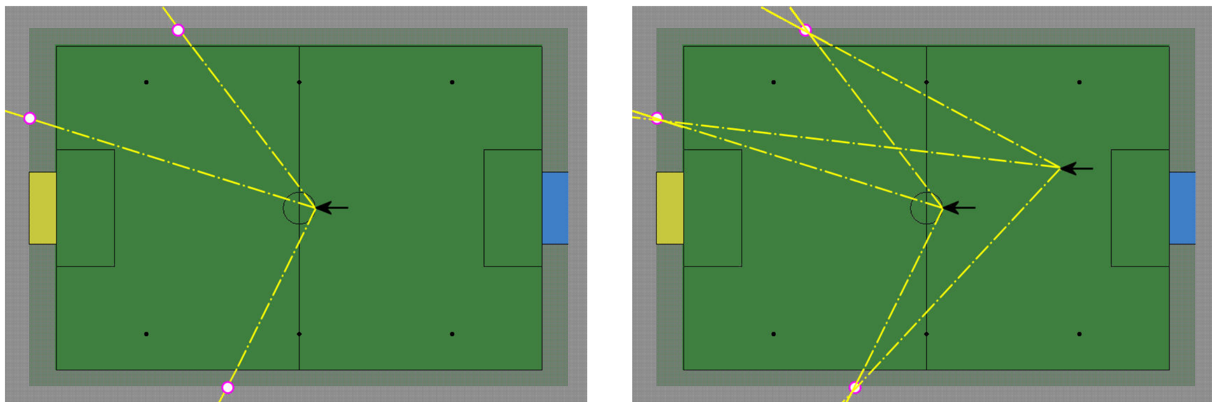
4.1 almost SLAM Challenge

In the Sony 4 Legged League the playing field is equipped with various unique landmarks which can be distinguished by their color. In order to support the leagues' development towards a more soccer-like environment this challenge demands the use of less strictly designed landmarks with initially mostly unknown positions. SLAM is the abbreviation for "**S**imultaneous **L**ocalization **A**nd **M**apping", an aim that can hardly be reached with the hardware limitations of the Aibos. Therefore in the almost SLAM Challenge the problem of SLAM is heavily reduced.

4.1.1 The Challenge Rules

The challenge is split into two stages. Additional landmarks are placed at random positions on the field border. In the first step the robot should explore the field within one minute. In the second step the normal goals and beacons are either covered or removed. Now, the robot should move to 5 positions on the field using the additional information gathered in the first step. The score is calculated from the number of points reached, the distances between the robot and the target positions and the total time needed to complete stage two. The detailed challenge rules can be found on the Sony 4-legged Robot League homepage[1].

[1] http://www.tzi.de/4legged/pub/Website/Downloads/Challenges2005.pdf



(a) For each new flag recognized the values on a line to the percept are being increased

(b) As the corridor where the flags are placed is known, moving the robot is not required to determine the positions

Figure 4.1: Mapping of new flags

4.1.2 Problem Analysis

The Problem basically consists of three parts. The recognition, mapping and usage of formerly unknown landmarks. In order to obtain the best possible result, the limitations for the design and placement of the additional landmarks should be considered. As the particular form of the landmarks is most probably hard to be identified, they should not be considered to be unique, making the center circle the only unique landmark on the playing field. Generally the additional landmarks can not be assumed to be detected often which puts a higher importance on the field lines. As our standard SelfLocator highly relies on the percepts generated from field lines, the use of additional landmarks can be reduced to resolving the symmetry of the field. Apart from that we can assume more accurate odometry information than in normal playing situations, as hardly any collisions are to expected.

4.1.3 Our Approach to the almost SLAM Challenge

Relying mostly on the field lines, we designed the following approach.

In the first step the robot scans both halves of the playing field for pink landmarks (those are guaranteed) from two symmetrical positions directly in front of the center circle (figure 3.18). In a map a value is incremented along a line from the robots position to the detected landmark each time a percept is generated (figure 4.1(a)). As the landmarks are placed on a narrow border around the field, their position can directly be approximated without moving the robot (figure 4.1(b)). This way two maps with new landmarks are generated, one for each half of the field.

In the second step the robots' position is unknown but can be determined to be on one of two possible positions as soon as the center circle is visible. If the center circle is not visible, it can be found by moving the robot where the (green) field ends in the longest distance. This distance can be precisely determined whenever the field border can be detected, which is usually the case when the robot is in a close to medium distance to the border. Once the center circle is found,

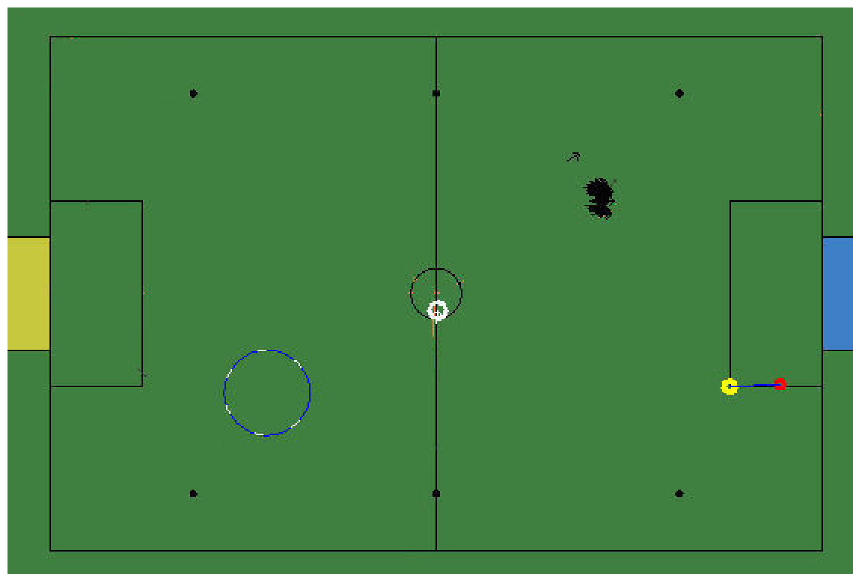


Figure 4.2: The falsified position is tracked to improve stability and accuracy

the robot positions itself according to the center circle, taking one of the two positions used in step one to map the landmarks. Similar to step one, the half of the field the robot is currently looking at, is scanned for landmarks. The detected landmarks are mapped onto the landmark positions in the two maps created before. For each map, the differences between currently seen and expected bearings are accumulated. Based on this accumulated difference the correct position can be determined and the target positions can be reached using the field lines only. To improve both accuracy and stability of the pose of the robot, the falsified position is tracked, too. Pose hypotheses occurring in an area around this position are mirrored near to the real pose of the robot (figure 4.2). Using this approach we were able to win the almost SLAM Challenge at the world championship 2005 in Osaka.

4.2 Variable Lighting Challenge

Color tables are currently the most precise method for color segmentation used in the Sony-4-legged-League. The method of color table generalization described in Sect. 3.2.2 not only improves color tables for better segmentation under constant lighting situations, but also improves the stability for small lighting changes. For greater changes though a more adaptive approach is needed.

4.2.1 The Challenge Rules

The Variable Lighting Challenge is based on the regular penalty shootout with a blue robot trying to shoot as many goals as possible into the yellow goal during three minutes. Also positioned on the field, there are two non moving red robots, one robot on the goal line and one somewhere

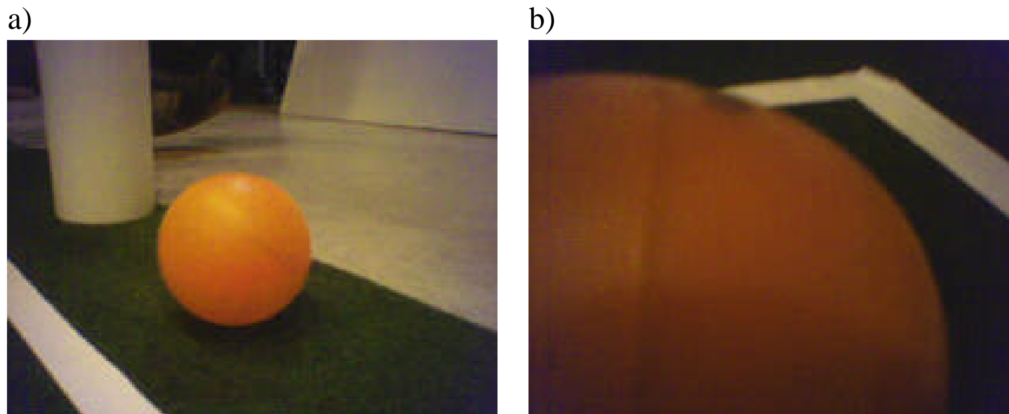


Figure 4.3: Two different situations occuring in a normal game situation even without changing the overall lighting of the field: a) The ball on a bright spot on the field. b) The ball lying in the robots own shadow.

outside the penalty area. During the time of the challenge the lighting on the field will change according to a schedule previously fixed by the referee but unknown to the competing teams. The winner is the team which scored the most goals.

4.2.2 Problem Analysis

The main task is to get a correct image segmentation all times, although the lighting conditions and therefore the color values, given by the camera, change. The challenge is, that the color classes move within the whole color space in a non-linear way, when the lighting conditions are changed.

Especially distinguishing the colors yellow, orange and red i. e. goal, ball and red jerseys of the robots, respectively, is a problem. Because of the poor camera quality, even small changes in illumination, e. g. shadows, can make the AIBO identify a goal as a ball, which is fatal for the game play and sometimes also for the fragile neck joint. The same is true for orange found in red jerseys.

4.2.3 Our Approach to the Variable Lighting Challenge

Our approach consists of a classification of the lighting conditions preceding the actual color classification. Based on features extracted during the scanning process of the past images, the best color table is selected for segmentation of the image.

A simple feature is the average intensity for each color channel over the last 5 images, which can be computed nearly without additional computational time during a normal image analysis. This way the AIBO can distinguish fast and reliably between different illumination levels and therefore choose the appropriate color table (cf. Fig. 4.3 and 4.4). Because of the usage of generalized color tables, the segmentation was very good even in illumination levels lying between the reference levels.

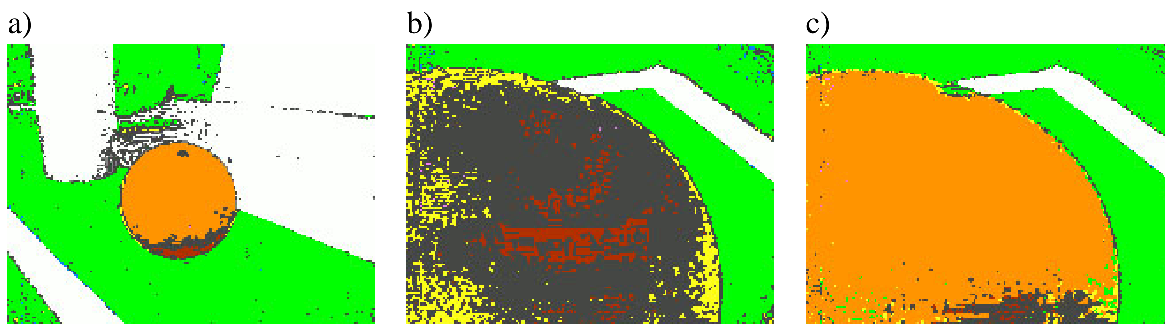


Figure 4.4: The segmentation of the images of Fig. 4.3: a) Segmented with a color table fitting these lighting conditions. b) Segmented with the same color table as in a). c) The automatically chosen one for darker lighting conditions.

This dynamic selection of color tables gave good results, both under lab conditions and during the GermanOpen in Paderborn. The vision part performed as good as with a constant lighting level and both ball detection and self localization worked well. In the Challenge held at the RoboCup in Osaka we were not able to perform equally well due to a configuration error in the camera settings.

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.

RobotControl presented in Section 5.1 is a remote debugging tool that can connect to real and simulated robots. It provides a vast number of dialogs.

The simulator (cf. Sect. 5.2) is able to simulate up to eight robots. The complete source code that was developed for the robot is also compiled and linked into this application. That allows algorithms to be tested and debugged very easily. While providing less visualization tools than robot control, the simulator supports the connection to several robots at once.

A ceiling camera system for external detection of the robots and the ball on the field was introduced as a development tool (cf. Sect. 5.3).

MakeStick is a tool for writing memory sticks. It can copy compiled code to the sticks and allows for configuring existing sticks. Code is copied using the script *copyfiles.bash* that provides several options to influence what is copied.

The *Universal Resource Compiler* (URC, cf. Sect. 5.5) is used by the GermanTeam for compiling motion description files containing, e. g., the kicks as well as for generating YABSL files defining symbols that also exist in the C++ code to be used by the behavior control.

Depend is the heart of the build system of the GermanTeam. It automatically generates all dependencies required to compile the code and is also able to create the project files required by Microsoft's Visual Studio.

The *Emon Log Parser* (cf. Sect. 5.7) was used to get as much information as possible out of the log files produced by the Open-R SDK Emergency Monitor.

5.1 RobotControl

There has been a major change in RobotControl this year. The complete line of development for RobotControl from the past years has been abandoned and RobotControl was rewritten from scratch. To ease the conversion to the new RobotControl, the old RobotControl is included in the source code release. The new tool is called RobotControl2 in the source code release of

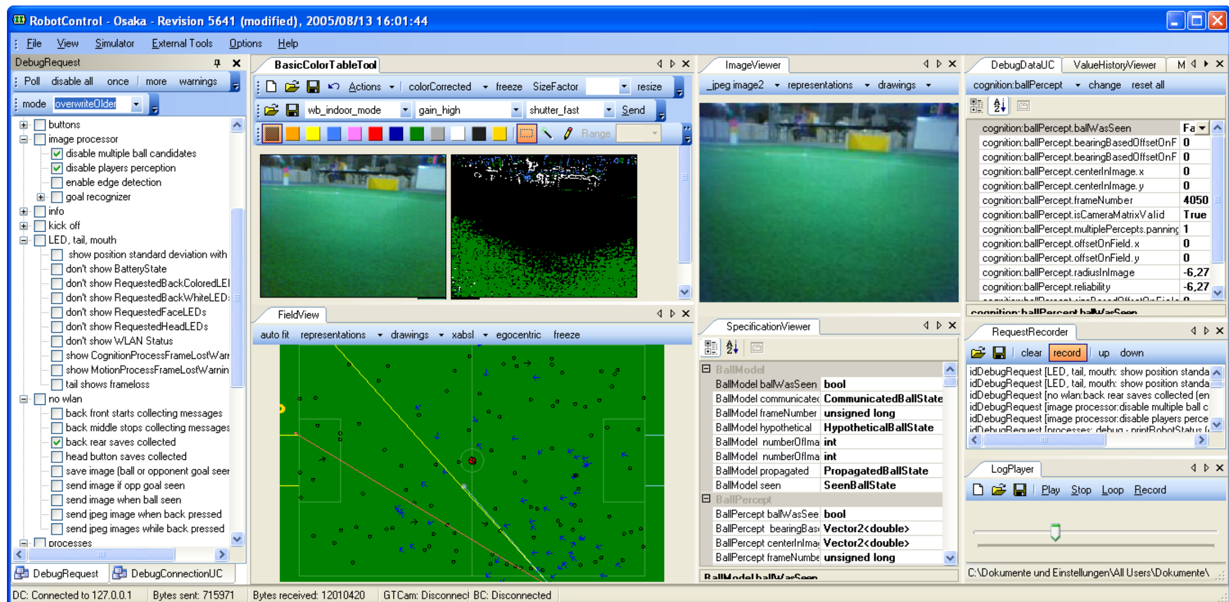


Figure 5.1: The RobotControl application

the GermanTeam. This document however describes only the new RobotControl, which will be referred to as RobotControl hereafter.

In contrast to the simulator SimRobot (see next section), RobotControl (cf. Fig. I.1) is intended to be a general support tool that helps to increase the speed and comfort of the software development process.

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.

Generally speaking, RobotControl serves as a front-end and GUI to the debugging techniques. It provides standard visualization and modification dialogs, which are connected to the corresponding debugging techniques on the robot. Thus enabling an out of the box use of the debugging techniques. This solves common tasks in debugging robot control software

- Switches, switching on and off of parts of the code for debugging purposes
- Visualization of data, especially of internal representations of the robot
- Modification of data, especially algorithm dependent parameters, as well as the use of test data
- Control, switching off different algorithms used to play robot soccer

Almost all of RobotControl's functionality was programmed into toolbars, user controls and so called *Managers*. 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 the Simulator: In the Simulator most of the interaction with the program is done using a text console whereas in RobotControl many graphical user interfaces exist. On the one hand, as many tasks require a graphical user interface, the Simulator provides only a small portion of the functionality of RobotControl. On the other hand RobotControl does not include a simulator or simulated processes. This is also a major difference to earlier RobotControl releases.

Appendix I describes the structure and mechanisms of RobotControl in detail.

5.2 Simulator

SimRobot is a kinematic robotics simulator that was developed at the Universität Bremen [46]. It is written in C++ and is distributed as public domain [49]. 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 of the 32 bit versions for Microsoft Windows is continued. The version used in 2004 and 2005 is an intermediate release that is only available as part of the GermanTeam code releases. The next official release, already presented at the last RoboCup Symposium [35], will include a physical simulation as well as a new generic description language for robot simulations, and will be released soon. It is currently also ported to Linux, but that version will probably appear later.

SimRobot consists of three parts: the *simulation kernel*, the *graphical user interface*, and a *controller* that has to be 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 E 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.

The following sections will give a brief overview of SimRobot, and how it is used to simulate a team of robots.

5.2.1 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, sensors, and actuators. Objects can contain further objects, e. g. the base joint of a robot arm contains the objects that make up the arm.

The kernel is platform independent. It is connected to a user interface and a controller via a well-defined interface. This enables an easy porting to other platforms.

The current release has a completely revised kernel which contains several important features compared to the previously used simulator:

OpenGL graphics: All functions for drawing objects are part of the simulation kernel. Through using OpenGL (cf. Fig. 5.2) instead of an own library for 3-D graphics, SimRobot now benefits from modern graphics hardware and offers a fast display of the simulated environment.

Hardware accelerated offscreen rendering is a feature which several manufactures implement on their graphics hardware. SimRobot is able to detect and use this acceleration for the generation of camera images. On computers which do not support this feature, a quite slow, software-based implementation is used.

An XML-based modeling language is now used to describe the simulation scenes. This offers the possibility of using a vast variety of existing tools for editing and validating scenes.

The SimRobot kernel is able to simulate the following classes of objects:

Bodies. Currently, bodies can be modeled as spheres, cylinders or as a collection of polygons. To each object, a surface class is assigned, which determines the color of the object. 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.

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 three types of actuators: rotational joints, translational joints, and objects moving in space in six degrees of freedom. SimRobot is currently only a kinematic simulator; thus it cannot directly simulate walking machines. Therefore, the motion of the simulated AIBOs is generated by a trick: the GT2005 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 under the assumption that the body roll is always zero. The computation is performed by the same function that is also calculating the position of camera in the code that is actually running on the robots.

Sensors. The current version of SimRobot provides, in contrast to previous releases, only two different kinds of sensors:

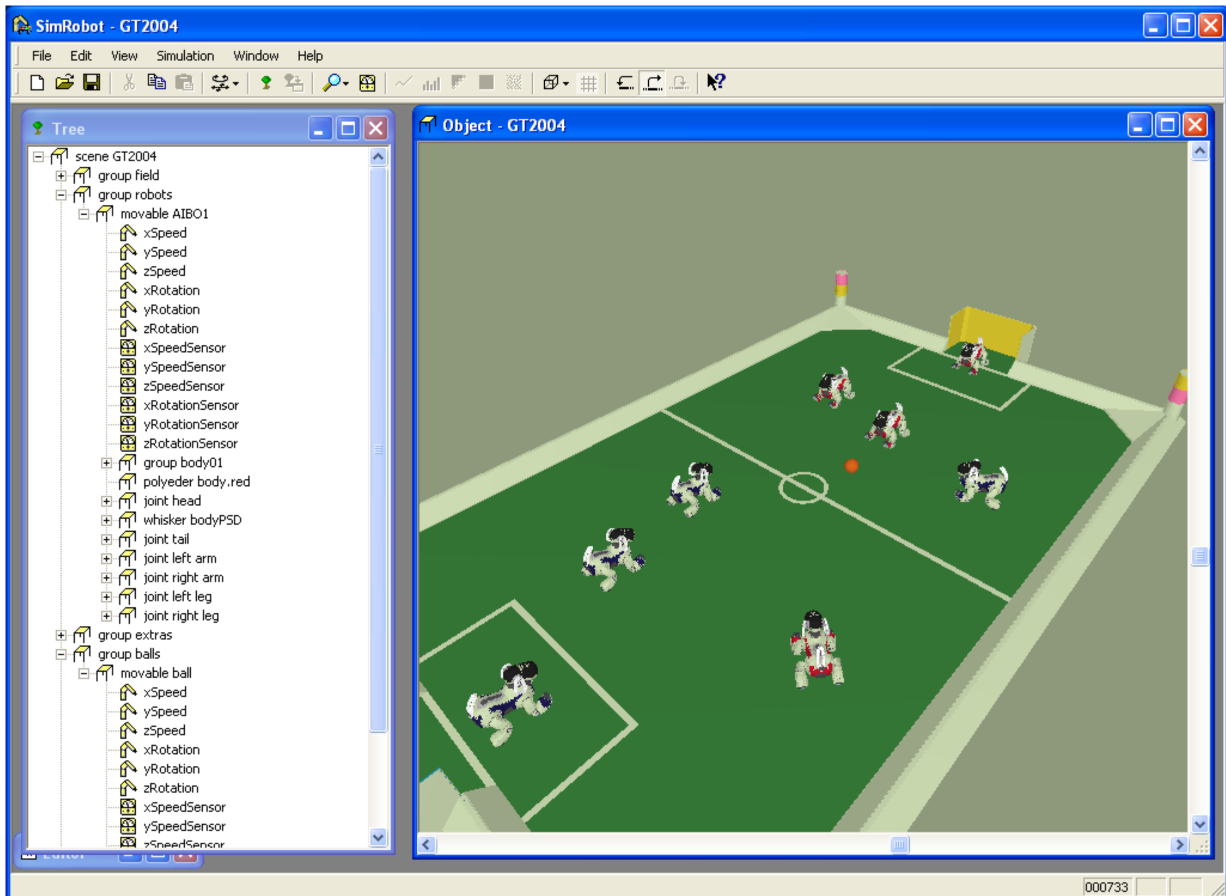


Figure 5.2: SimRobot simulating the GermanTeam 2004.

- A *camera* which computes a two-dimensional array of RBG pixels which have a color depth of 24 bits.
- A *whisker* which imitates the behavior of an infrared sensor. On the one hand, it is used to simulate the PSD sensors in AIBO's head and body. 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.

5.2.2 User Interface

The user interface of SimRobot includes an editor for writing the required scene definition files. If such a file has been written and has been compiled error-free, the scene can be displayed as a tree of objects (cf. Fig. 5.2, left 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 (cf. Fig. 5.3, upper right window). Objects may be displayed as wire-frames, simple flat shaded polygons or smooth shaded polygons whose surface brightness is dependent on their current angle to a global light source.

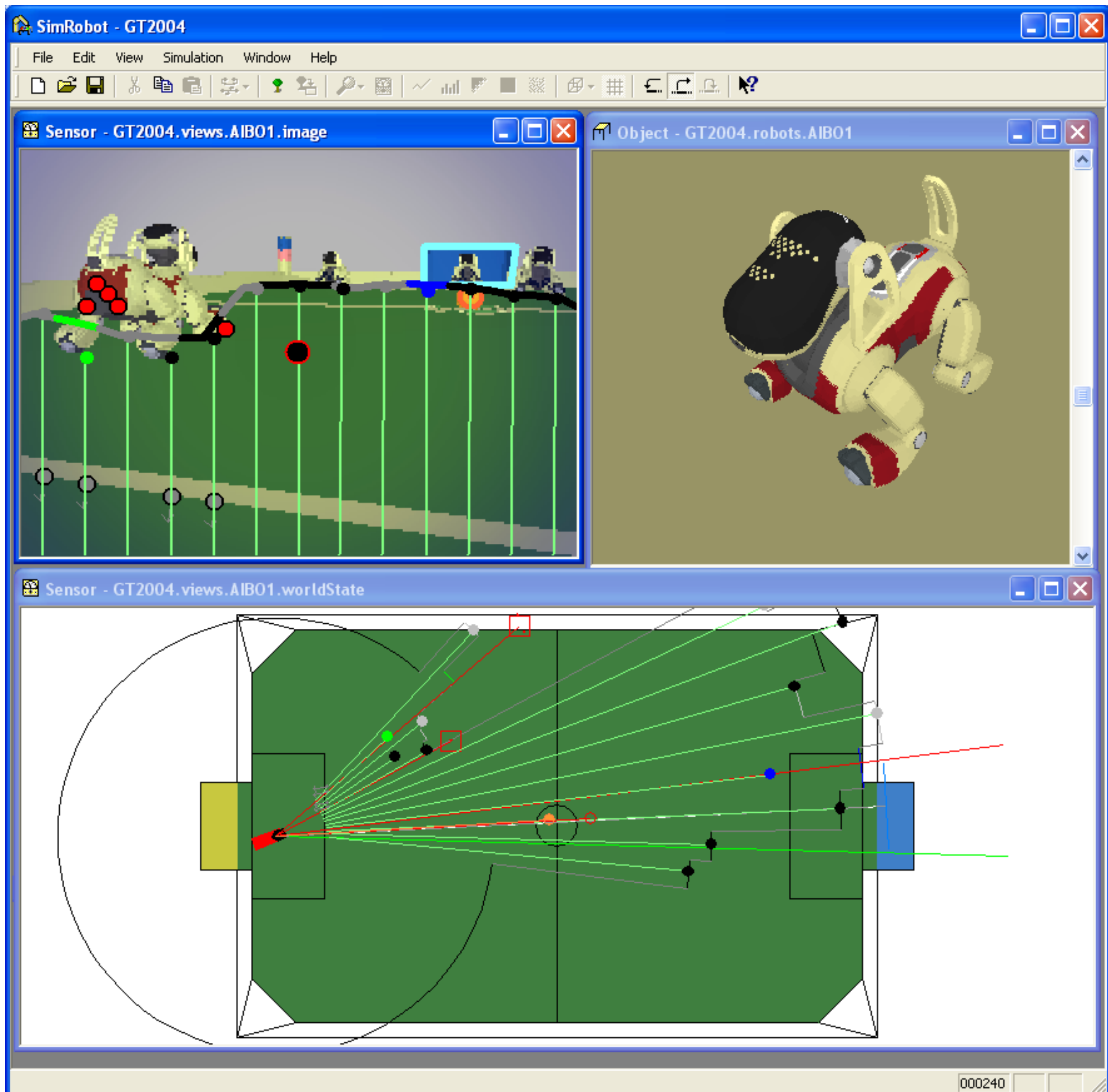


Figure 5.3: SimRobot displaying an image from a simulated camera, a single robot, and a user-defined view.

While data from the camera sensor can only be displayed as a color image, data from distance sensors can be depicted as line graphs, column graphs, and monochrome images. 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 always 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. The code of the GermanTeam 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 C.

5.2.3 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 lower window in figure 5.3 is a view that contains the soccer field overlaid by a visualization of the robot's percepts and the currently estimated world state.

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.

5.3 Ceiling Camera

The rules of the Robocup Sony-4-legged-league clearly state that remote imaging and processing is not allowed during the games. The robots have to use their onboard computers and the pictures from the built-in camera to compute their world models. In contrast to this, the small-size league uses cameras mounted at the ceiling to track their robots which are radio controlled by an external computer. While a ceiling mounted camera is therefore not allowed during games of the Sony-4-legged-league, it can prove quite useful during the development. A camera system which oversees the whole field can provide the engineers with a global view of the situation. The robots and the ball can be tracked and their position, heading and movement can be compared to the data computed by the autonomous robots.

During the year 2004, the Darmstadt Dribbling Dackels and the Microsoft Hellhounds, two of the four sub-teams the GermanTeam consists of, independently developed software which builds a global world model by interpreting the image from ceiling mounted cameras above the field. In the following the approach of the Technische Universität Darmstadt shall be outlined. A detailed description can be found in the diploma thesis of Ronnie Brunn and Michael Kunz which is available online but published in German language only [11].

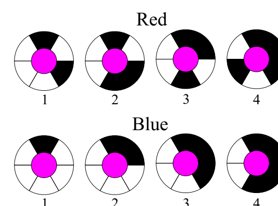
A high resolution fire wire camera was mounted at the ceiling of the laboratory and equipped with a wide angle lens to enable the camera to view the whole field. The image from the ceiling mounted camera is disturbed by the perspective and the optical distortions of the camera lens. These distortions are compensated by the software and the objects on the field are recognized.

The ball is detected in a similar way as by the algorithms used by the robots themselves. However, some adjustments for the special case of a viewpoint originating at the ceiling above the field were applied. The detection of the robots proved more complicated and required the robots to be equipped with markers on their backs (cf. Fig. 5.4(a)). These markers were designed to minimize the distraction and obstruction of the autonomous robots. Therefore, only the colors pink, black and white were used and the marker was kept quite small. The resulting size is a compromise between the minimization of the obstruction and maximization of accuracy and reliability of the recognition by the ceiling-camera system. The design of the markers can be seen in Figure 5.4. A pink disk in the center of the circular marker is used for initial detection within the image. The black and white sectors circling the pink provide a binary code which is unique even if rotated. Therefore, this code allows both the identification of the robot, i.e. team color and player number, and the computation of the rotation and by this the heading of the robot (cf. Fig. 5.4(b)).

The global world model containing position, heading and speed of all recognized objects on the field is distributed to all workstations used to debug the robots software during development. The debug application "RobotContol" communicates with the ceiling-camera system over network and allows the user to compare the global world model to the local models transmitted by the robots nearly in real time. For sophisticated analysis using even better synchronized data, a system for compensating the offset of the time between robot and ceiling-camera system has been developed. It uses the lighting of a lightbulb as an common event in both data streams from robot and ceiling-camera to compute the offset and as such the synchronization of recorded logfiles.

Additionally, the data broadcasted by the ceiling-camera system can be received by the robots themselves enabling them for example to play with a quasi flawless localization or recognition of teammates and opponent players. This enables the developers to program the robots behavior even at an early time where the other modules are not computing the necessary data yet. Another

(a) A robot equipped with a marker.



(b) Different markers allow identification and recognition of position and heading.

Figure 5.4: Markers for recognition of robots by the ceiling mounted camera.

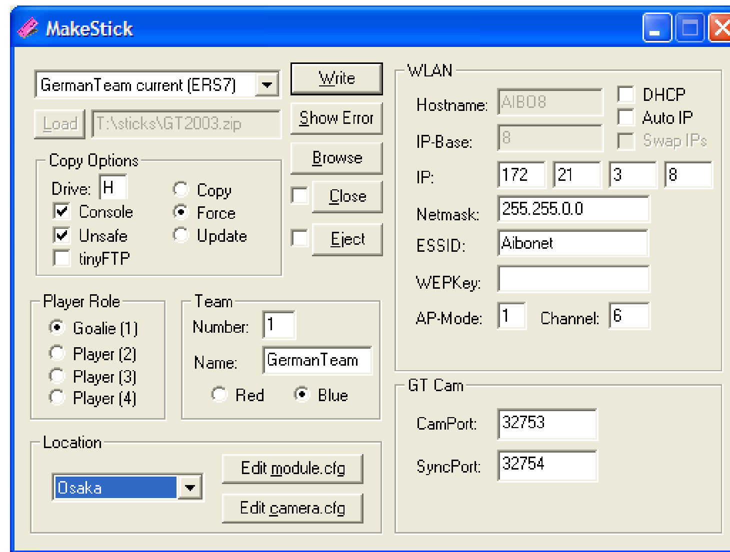


Figure 5.5: The MakeStick tool

use case would be the automatic evaluation of gaits and as such the possibility of autonomous learning of new gait parameters. This could be accomplished by using the robots self-localization as well, but this would not nearly be as accurate as the data provided by the ceiling mounted camera, especially while the robot is moving.

The main advantage of the whole effort happens to be the chance of impartially comparing various approaches and different versions of modules during actual games. For example the autonomous localization of the robot or its model of the ball can be compared to the global view provided by the ceiling mounted camera. If a developer makes changes to the module, he or she can test wether the results have really improved or not. Errors and glitches in the robot's software can be detected and the accuracy of different approaches could be compared in situations which are more realistic than manually set-up test cases.

5.4 MakeStick

MakeStick allows the user to copy data to a memory stick and/or configure the stick afterwards. *Configuring* means to set a robot's role, its team color, its team identifier, its location, and the parameters required for wireless networking. The code copied to the stick can just have been compiled, or it was compiled long ago and stored in a zip archive afterwards. Although the code can run on both the ERS-7 and the ERS-210, the correct version of the operating system for the target system has to be copied. And last but not least, for practice games, not only the current code of the own team can be copied and configured, but also the code of other teams (currently, only the code of CMPack'02, CMPack'03, and CMPack'04, and hopefully CMDash'05 is supported).

5.4.1 Installation

MakeStick is a small application. If it is compiled using the configuration *MakeStick Release* with Visual Studio 2003 .NET, the resulting executable in the directory *Bin* is only 72 kB in size. This allows it to be installed as an autoplay handler for the memory stick drive. So MakeStick is always started when a memory stick is inserted. To accomplish this, Microsoft's TweakUI can be used. Using this tool, MakeStick can be established as a handler for *music files* in *My Computer/Autoplay/Handlers*, because the code of the GermanTeam contains wav-files. After inserting such a memory stick, Windows will ask for the handler to use (if it does not, reset the default handler in the properties dialog of the memory stick drive), and MakeStick can be selected as default handler.

5.4.2 Usage

Figure 5.5 shows the dialog components of MakeStick. The combobox at the top of the dialog allows selecting the kind of operation performed:

- The *GermanTeam current* options copy the GermanTeam 2005 code that was compiled last together with the operating system either for the ERS-7 or the ERS-210 using the script *copyfiles.bash*. The stick is configured afterwards with the information specified in the dialog.
- The *configure* options configure memory sticks that already contain code. Here, it can be selected between *GermanTeam 2003*, *GermanTeam 2004* (which is compatible to German-Team 2005), and *CMPack*. Depending on the selection, the other dialog elements will be adapted.
- The *from ZIP* options unpack an image file to the memory stick and configure it afterwards. Again, *GermanTeam 2003*, *GermanTeam 2004/2005*, and *CMPack* code is supported.

5.4.2.1 Actions

Write executes the command that was selected. In most cases, a shell script will be started in a separate window.

Show Error starts the *emon log parser* (cf. Sect. 5.7) to analyze the information that was written to the memory stick the last time the robot software crashed.

Browse opens an Explorer window that shows the contents of the memory stick. This button is useful if MakeStick is used as default handler for the memory stick drive.

Close closes MakeStick. The checkbox left to the button decides whether MakeStick is closed after *Write*, *Show Error*, or *Browse* were successfully executed.

Eject ejects the memory stick from the selected drive. This is required for some machines. The checkbox left to the button decides whether MakeStick is automatically ejects the memory stick after *Write* was successfully executed.

Load opens a file selector dialog to select the compressed stick image file to be unpacked to the memory stick.

5.4.2.2 Copy Options

This section allows specifying how and where the data will be copied:

Drive selects the memory stick drive. This selection is very important because this drive may be formatted without any warning.

Console. The operating system copied supports the WLAN console.

Unsafe. This option deactivates the Open-R *EmgcyMon* module. This module is responsible for shutting down the robot if the battery load is too high. This option has to be selected with care, because it deactivates a safety system of the operating system.

tinyFTP. This option activates copying the tinyFTP server to the memory stick.

Copy/Force/Update. If *Copy* is selected, all files are copied to the memory stick. If *Force* is selected, the stick will be quick-formatted before. *Update* will only copy newer files. It will not change the operating system. So, whenever a memory stick with an unknown configuration is used, *Force* is the best selection. Afterwards, *Update* ensures the fastest copy times.

5.4.2.3 Player Role

Each robot has a role in its team. This role is fixed at least for the goalie. But also for field players, it is important that they have at least one distinguishable feature. So depending on the code that is configured, the player role is either interpreted as a real role or just as a player number. All robots in a team must have different roles.

5.4.2.4 Team

Number. This field configures the number that is used to identify the team for the GameController.

Name. The robot's of the GermanTeam use UDP broadcasts to find each other in the network. To be able to distinguish between teammates and opponents, a team name is transmitted. This name can be configured here.

Red/Blue specifies the color of the team.

5.4.2.5 Location

Since the GermanTeam consists of four sub-teams using a single software repository, some configuration information is location-dependent (color tables, module selection, etc.). All these different settings are stored in a sub-directory below *Config/Location*. The dropdown list allows selecting one of them. Please note that only the selected location information is copied to the memory stick, so all other configurations will be missing. Therefore, selecting a location only makes sense when actually copying the current GermanTeam code to a memory stick, and not when unpacking or only configuring it.

Edit modules.cfg displays a dialog that allows changing the solutions selected for the current location on the local hard drive. It will not configure the memory stick. Therefore, it has to be used before the current code is copied to the memory stick using the button *Write*.

Edit camera.cfg displays a dialog that allows changing the camera settings for the current location on the local hard drive. It will not configure the memory stick. Therefore, it has to be used before the current code is copied to the memory stick using the button *Write*. However, since the code of the GermanTeam still relies on color tables, changing the camera settings without changing the color table makes no sense. So this option may be of more use in future.

5.4.2.6 WLAN

In this section, the wireless connection of the robot is configured. To ease the configuration of whole teams in practice games, the hostname and the IP-address of a robot is based on its role and its team color.

DHCP. Activates DHCP.

Auto IP. If activated, IP addresses are calculated from the *IP-Base*, the *team color*, and the *player role*. Otherwise, the IP address has to be specified completely.

Swap IPs. Normally, in *auto IP* mode the IP-addresses of a blue team always start above the IP-addresses of the red team. Here, this sequence can be inverted.

IP-Base specifies the base of the last byte of the IP-address. The resulting address of a robot is computed as $base + 4 \times team + role$.

IP allows specifying the first three bytes of the IP-address.

Netmask. The subnet mask required.

ESSID. The ESSID of the wireless network.

WEPKey. The key required for a WEP-encrypted network. If none is specified, no encryption will be used.

AP-Mode. If an access point is used, specify 1 here. If the ad-hoc mode is used together with a computer running Windows XP, specify 2. Otherwise, use 0.

Channel specifies the network channel used by the access point or the ad-hoc connection.

5.4.2.7 GT Cam

This section allows configuring the ports used to listen to the data sent by the ceiling camera modules *GT Cam* that provides ground truth.

CamPort. The port data is received on.

SyncPort. The port used for synchronization.

5.5 Universal Resource Compiler

The *Universal Resource Compiler* (formerly known as *Motion Net Code Generator*) parses a set of motion specifications described in a special language and compiles it into a file that is read be the special actions module (cf. Sect. 3.9.2). 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 data structure is loaded from the *SpecialActions* module (cf. Sect. 3.9.2).

The *Universal Resource Compiler* also generates a YABSL representation of all motions allowing new motions to be used in the behavior (cf. Sect. 3.8) without having to specify them in more than one place, as well as several other symbol definitions for YABSL.

For interactively testing motions and their transitions, the active motion request can be modified using RobotControl's *DebugDataUC*, selecting *motion:motionRequest*. Furthermore, a the dialog *Motion Designer* is provided to transmit new motion descriptions to the robot and execute them without the need to recompile anything.

5.5.1 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 motion file starts with a line that defines the name of the special action. By convention, it is identical to the name of the motion file, e.g.:

```
motion_id = swing
```

After that, any sequence of the following lines is possible:

1. Empty lines are simply ignored.

2. Comments start with a " or // by convention. They are also ignored.

3. A label is defined by the keyword *label* followed by an identifier, e. g. *label loop*. Please note that the special label *start* must be defined before the first set of joint angles or transition.

4. A transition is defined by the keyword *transition*, followed by the condition, the name of the target motion file, and a label in that target motion file. For instance

   ```
   transition swing swing loop
   ```

   means "when the execution of this motion reaches this line, and the special action currently selected is *swing*, then jump to the special action *swing* at the label *loop*".

   There is a special condition *allMotions* which means "whatever motion is currently selected". So

   ```
   transition allMotions swing loop
   ```

   is an unconditional jump to motion file *swing* at label *loop*.

   There is also a special motion file *extern.mof* that represents the possibility of switching to another kind of motion, e.g. walking. *extern* is also the entry point from other kinds of motion, e.g. when it is switched from *walk* to *specialAction*, the motion net is entered in motion file *extern* at label *start*. Therefore *extern.mof* contains transitions to all other motion files, and all other motion files contain a transition back to *extern*, typically as their last line:

   ```
   transition allMotions extern start
   ```

5. A line starting with *pid* followed by the index of the joint the PID values should be changed of and the three PID parameters themselves.

6. A set of joint angles consists of 20 entries, the first 18 of which define actual joint angles, the 19th entry defines whether to interpolate or not (0 or 1), and the final entry defines how many 8 ms frames the execution of this motion should take. Each angle is either a number, defining the angle in milliradians, or the character ˜ (tilde), meaning "do not overwrite the

current angle", so special actions keep the joint angle set by the head control module. If interpolation is activated, intermediate joint angles are calculated and sent to the robot to slowly reach the target angles. Without interpolation, the target angles are sent immediately and kept for the entire duration of this motion.

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.

5.6 Depend

The directory structure of the GermanTeam is not based on the layout of the processes that are to be build as binaries. Source files may belong to different sets of binaries according to the chosen process layout.

Depend solves the problems that occur when multiple teams try to resolve the dependencies and compile as well as link different binaries for different purposes. (See Sect. 2.2.4 for a detailed description of the problem). *Depend* enormously speeds up the process of resolving dependencies while leaving great flexibility to developers.

Depend is used to completely generate all dependencies for the chosen build target in *GT2005/Build/*/*/depends.incl* each time. Even with several hundreds of source files, this takes only a few seconds.

Depend was developed for the competitions in 2003,2004 and 2005. It is a simple speed optimized pre-processor written in *C*.

Implementation. First of all, *Depend* reads all **.cpp* and **.h* in all subdirectories of *GT2005* into memory and sorts their paths alphabetically. That speeds up further processing dramatically, because most files have to be touched several times and would normally be searched somewhere in the include path and loaded from hard disk each time. Given the main source file of the process, such as *Motion.cpp*, *Depend* calculates all object files needed to link the process binary as well as all header files needed to compile each of the object files.

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.

To be able to do that calculation faster than a normal preprocessor, *Depend* uses a couple of assumptions. None-system *#include*s are not allowed inside *#if* (but are allowed inside *#if[n]def*), includes must be case-sensitive and have to use slashes instead of backslashes. The implementation of a function declared in a header file is assumed to be found in the header file itself or in a *cpp* file with the same name in the same directory. Any offence against these rules results in an appropriate error message.

Depend generates dependency files accepted by *make*. It probably only works and compiles under Windows.

5.7 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 *GT2005/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> <layout> <configuration>
```

So, the path to the emon.log of the crash has to be provided as well as the names of the process layout, e. g. *CMD*, and the configuration of GT2005 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.

A simpler way to use the *emonLogParser* is to call the script *showError.bash* followed by the drive letter of the memory stick drive. The script will use the */OPEN-R/APP/CONF/CONFIG.CFG* on the memory stick to automatically determine the process layout and the configuration.

Chapter 6

Conclusions and Outlook

The GermanTeam now exists for more than four years. Over the years, the results achieved in the RoboCup competition got better and better, culminating in winning the world championship in 2004 and 2005. The general architecture developed for RoboCup 2002 has proven to be sustainable, still satisfying our needs, as well as the needs of at least six other teams worldwide. Only a few changes were applied over the years. For RoboCup 2003, switching from the Greenhills-based environment to the gcc-based environment required only minor changes in the platform dependent part. For RoboCup 2004, the robot dependent parts of the code were virtualized, so that the compiled binaries run on both the ERS-210 and the ERS-7[1]. For RoboCup 2005, the C++-based debugging tool RobotControl was restructured and completely rewritten in C#. All other changes are described in Section 1.5, and they are not repeated here.

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.

In RoboCup 2005, four other teams (the Dutch AIBO Team, the Hamburg Dog Bots, SPQR+Sicilia, and Wright Eagle) were also using the GermanTeam code base. Wright Eagle even reached the quarter final. Also from a scientific point of view, the GermanTeam provides a good foundation for doing research. Between 2003 and 2005, it was the team with most publications at the RoboCup Symposium, e.g., 10% of the talks at the RoboCup Symposium 2005 were given by a member of the team [18, 28, 35], and besides there were two poster presentations [45, 52].

[1] However, the actual code does not behave properly on an ERS-210, because some differences such as computing power and the orientation of the third head joint cannot be compensated for.

first Round Robin	
GermanTeam – ARAIBO	4:0
GermanTeam – Mi PAL Team Griffith	5:1
second Round Robin	
GermanTeam – WrightEagle	1:1
GermanTeam – Dutch AIBO Team	5:0
GermanTeam – Hamburg Dog Bots	3:0
Quarter Final	
GermanTeam – JollyPochie	4:0
Semi Final	
GermanTeam – CMDash	5:0
Final	
GermanTeam – NUBots	4:3

Table 6.1: The results of the GermanTeam in Osaka

6.1 The Competitions in Osaka

The GermanTeam scored very well at the past RoboCup world championship. During both round-robin, we became winner of our groups (4:0, 5:1, 1:1, 5:0, 3:0). During quarterfinals, we defeated JollyPochie by 4:0. We won against CMDash by scoring 5:0. Finally, we got world champion as we bet NUBots in the finals in a penalty shoot-out by 4:3, after the game was a draw in the regular game time at a score of 2:2.

As every year, teams were invited to take part at the *Technical Challenge*. Each team has to demonstrate, that it is able to solve a number of scientific problems with the robots. In 2005, we placed third at the technical challenge, mainly due to outstandingly winning the "Almost SLAM Challenge" (see Chapter 4 for details).

6.2 Future Work

The GermanTeam owns a powerful code basis for the next year's work. For the RoboCup Dutch Open in April 2006, 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

In the future we will continue to integrate 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" [14] 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. The creation of a common world model will help to fusion knowledge of different agents and to create a common base for cooperation and action taking, making team play and cooperation more effective.

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.

The next step is to analyze, how the gained knowledge from the Sony Four Legged League can be used to create a new robot architecture for humanoid robots. Thus we will examine how portability of a software architecture can be achieved for different hardware platforms. The portability of developed algorithms thereby becomes as important as the design of the algorithm itself.

6.2.2 Technische Universität Darmstadt

In 2005 a global vision system based on a ceiling mounted camera was developed. This system will be applied in order to systematically evaluate and improve different localization methods.

Furthermore, using accurate reference data from global view not only the self-localization but also a number of other algorithms can be evaluated and even automatically improved through learning and optimization methods. The plans of the Darmstadt Dribbling Dackels until the competitions in 2006 are to improve the ball modeling based on current algorithms implementing Kalman Filters and Particle Filters, the recognition and modeling of the other players, the ball passing and cooperation of players, the ball controlling and kicking capabilities as well as the walking capabilities of the robots. We plan to integrate these developments in a new world model incorporating informations, such as position, orientation and velocity of all objects, which are shared with the other groups in the GermanTeam for different research purposes (team behavior planning, opponent's plan recognition, scheduling of resources and others).

6.2.3 Universität Bremen

To achieve the scientific goals described in ch. 1.2.3, the members from the Universität Bremen will continue their work on an integrated, fully probabilistic world model which contains adequate estimations of the poses of all robots on the field. The approach from Kwok and Fox [33] appears to be very suitable for this project, since it is able to incorporate the states and intentions of the different actors in the environment. This allows a close interconnection with plan recognition approaches. Additionally, the perceptions of different robots, which are exchanged by all team members among each other, have to be integrated to achieve a world model of an adequate completeness.

To exploit the information from the world model, the robot's behaviors have to be changed to be able to adapt themselves to the positions and targets of the opponent robots. To achieve this, the potential fields approach [34], which has been used for several positioning behaviors (cf. 3.8.2.3), has to be extended by a representation of the aims of opponent robots.

Another research topic is still the elimination of manual color calibration. Although several approaches have already been presented [17, 31, 30], none of them was used in real soccer games so far. A rather robust recognition of the beacons has already been implemented. It is based on similarity to prototypical colors and the presence of certain spatial relations between neighboring surfaces.

During the last year, SimRobot has been extended by rigid body dynamics [35] to allow a more detailed and realistic simulation of collisions, ball movement, and walking as well as kicking motions of the robots. This extended simulator has been used in parallel to SimRobotXP (cf. 5.2) so far. After correcting some instabilities und inaccuracies, we are going to replace the old simulator completely.

6.2.4 Universität Dortmund

Since our recent considerable growth in team size, we feel that we have reached a point where it is ineluctable for us to leave the GermanTeam and take part in RoboCup 2006 as a new entry.

We wish to thank the other universities and members of the GermanTeam for the fruitful cooperation achieved through all these years, both from a scientific as well as human point of view.

Together we have proved that spatial locality is not a stringent requirement for a successful cooperation in a highly competitive environment.

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 2005 for travel support. All members of the GermanTeam thank the Deutsche Forschungsgemeinschaft (DFG) for funding parts of their respective projects. The team from Darmstadt thanks Vereinigung von Freunden der Technischen Universität zu Darmstadt e.V. [1], and Fachbereich Informatik of TU Darmstadt [2] for financial support. Allied Vision Technologies [3] supplied Darmstadt with the ceiling camera. The team members from Dortmund thank Microsoft MSDNAA, Lachmann & Rink GmbH, Garz & Fricke and Mabotic for their effective cooperation and sponsoring and the DAAD (German Academic Exchange Service) and the Fachbereich Informatik of Dortmund University for travel support. The team members from Bremen thank Evolution Robotics and the International Office of the Universität Bremen for travel support.

The members of the GermanTeam 2005 also want to thank the members of the GermanTeam 2002, 2003, and 2004 for creating the foundation for the continuing success of the GermanTeam and for writing the previous years' team reports [13, 47, 48] that were 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 1.
- 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.

[1] http://www.tu-darmstadt.de/freunde
[2] http://www.informatik.tu-darmstadt.de
[3] http://www.alliedvisiontec.com

- 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).
- RobotControl 1 includes parts of SimRobot developed by Thomas Röfer, Uwe Siems, Christoph Herwig, and Jan Kuhlmann (http://www.informatik.uni-bremen.de/simrobot).
- This product includes SimRobot XP developed by Tim Laue and Thomas Röfer.

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 (or at least within) this software.

A.1 Required Software

- Microsoft Windows 2000/XP
- Microsoft Visual C++ 6.0 SP6 or Visual C++ 2003 .Net (can be installed anywhere)
- Cygwin 1.5 including CygIPC 2 (can be installed anywhere).
- gtk+ 2.2.1 for Cygwin (unpack it to Cygwin-Path */*)
- Open-R SDK 1.1.5-r2 and MIPS developer tools 3.3.3 (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;*.

On the homepage of the GermanTeam (http://www.robocup.de/germanteam) an archive is provided that contains the correct versions of Cygwin, gtk+, and Open-R together with some scripts that automate the installation process. They even update the search path of Visual Studio 6, but the search path of Visual Studio 2003 .NET still has to be updated manually (or VisualStudio has to be installed afterwards). It is strongly advised to use the provided package to get a suitable build environment.

A.2 Source Code

The source code has to be unpacked anywhere in a directory *XXX\GT2005*. 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 compilation.

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

Doc contains the documentation generated by *Doxygen* from the source files.

Make contains makefiles, batch files, and Visual C++ project files. The workspaces *GT2005.dsw* (Visual Studio 6) and *GT2005.sln* (Visual Studio 2003 .NET) are located here, i. e. the files that have 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* (i. e. *Open-R*), *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 with the main purpose 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 everything 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. F), and message queues. However, the code of the Windows tools such as RobotControl cannot be found here.

A.2.2 Tools Code

Src/Depend contains the code of the tool (cf. Sect. 5.6) to create the dependencies for all robot builds as well as for RobotControl and SimGT2004.

Make/DSP_generation contains a few scripts to create Visual Studio (6 as well as 2003.Net) project files from these dependencies generated by *Depend* (cf. Sect. 2.2.4.4).

URC contains the code of the tool (cf. Sect. 5.5) to create new motions, i. e. special actions (cf. Sect. 3.9.2), Xabsl symbols, etc.

RobotControl2 contains the code of the tool with the same name (cf. Sect. 5.1).

RobotControl contains the code for the legacy version of RobotControl.

SimRob95 contains parts of the old version of the kinematic robotics simulator SimRobot that is used in RobotControl 1 (cf. Sect. 5.1).

SimRobXP contains the new, OpenGL-based version of SimRobot.

A.3 The Developer Studio Workspace GT2005.dsw/.sln

The Developer Studio Workspace GT2005.dsw/.sln 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 *_GT2005*, *_RobotControl*, and *_Simulator*. Please note that legacy code such as the old version of *SimRobot* does not support *Doxygen* and generates no proper help files.

_GT2005 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 layout (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*. If no drive is associated to your HOSTNAME in the script, it is tried to find the memory stick drive by searching the main directory of all drives for a sub-directory *open-r* or a file *memstick.ind*. If the search fails, *E:* is used as default. Try *copyfiles.bash --help* to see all options. Instead of calling *copyfiles.bash* manually, you can use the more comfortable tool MakeStick (cf. 5.4)

_**RobotControl 2** can be built in different configurations. The executable will be copied to *GT2005\Bin*. The *(*Debug) configuration will include debug-information to the binaries, *(*Release) will not.

_**Simulator** currently only supports a single configuration. The executable, together with the help file, will be copied to *GT2005\Bin*.

The other projects generate libraries (*GUI*, *GuiLib*, *SimRobotCore*, *SimRobotForRobot-Control*), source code (*SpecialActions* by executing the universal resource compiler *URC* cf. Sect. 5.5)), tools (*Depend* cf. Sect. 5.6), or preprocessed behavior (*Behavior* cf. Sect. 3.8) 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 *Behavior*. 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 let the robots play and how to use the debug tool on the PC. In addition, the main configuration files used by the robots are described.

B.1 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.1.1 location.cfg

This text file contains the name of a subdirectory under *Config\Location*. 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. In addition, in the subdirectory there are two further subdirectories *ers7* and *ers210* that allow to specify different configurations for each type of robot. If *location.cfg* 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 *GT2005\Config* on the PC. In the code release, the file contains the name "Osaka", so the configuration files in that subdirectory are used.

B.1.2 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.1.3 camera.cfg

This file describes the camera settings. It must contain the settings that were active when the color table was created. The file consists of three textual keys:

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

B.1.4 player.cfg

With this text file, it is possible to set the team color and the number of a robot. The goalie must always have player number 1. The team identifier is used for the *Dog Discovery Protocol* to identify all robots of the same team, so that can establish connections between each other.

```
// teamColor red | blue
teamColor blue
// playerNumber 1 | 2 | 3 | 4
playerNumber 2
// teamIdentifier string (up to 15 characters)
teamIdentifier GT2005
```

B.1.5 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[1]. 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 *DebugData* user control 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 *GT2005SelfLocator* depend on these correction values.

B.1.6 wlanconf.txt

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

B.1.7 coeff.c{*u,v,y*}

These files contain the coefficients calibrated off-line for the color correction. They can be generated from a log file with a tool which is not included in the GermanTeam source code distribution, however this shouldn't be needed as they work fine for ERS7 robots under a very wide range of lighting situation, provided that the camera white balance mode is set to *Indoor*; the correction effect is valid but weaker in *Outdoor* mode, and currently untested in *Fluorescent* mode. Do not use these coefficients on ERS210(A) robots, as the result would be an artificially induced chromatic distortion; to disable the color correction, simply make sure that these files are not present in the currently selected configuration folder.

[1]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.

Appendix C

Simulator Usage

C.1 Introduction

The simulator is based on SimRobot [49], 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, the simulator shares the user interface with SimRobot. This user interface is documented in the online help file that comes with SimRobot. Please note that because the version of SimRobot used by the GermanTeam is only an intermediate release, the help file contains a completely outdated manual on the scene description language. The description language used is now based on XML.

The adaptation of the simulator to the GermanTeam code is also in an intermediate state, as the GermanTeam code itself. The simulator can be compiled in the two configurations *_Simulator CMD Debug* and *_Simulator CMD Debug Server*. The latter partially supports the new debugging techniques and is meant to be used with RobotControl. The former only supports the old debugging methods and is described in this document. In future, the old debugging techniques will cease to exist and the simulator will only and fully support the new ones[1].

The simulator is the second Windows tool of the GermanTeam besides RobotControl. While RobotControl focuses on *interaction*, the simulator has its strength in *automation*. In addition, RobotControl focuses on the connection to a single robot, while the simulator can connect to several real or simulated robots at once, and it can replay several log files in parallel. The main input channel of the simulator 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[2]. Therefore, the simulator 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. Each process layout has its own set of views, and message handling is not dependent on Windows idle time. The approach

[1] In fact, while writing this document the transition has already taken place, but the code release reflects the code used by the GermanTeam in July, 2005, and newer developments are not included.

[2] Although RobotControl also supports scripting, its abilities in that area are not finished yet.

requires a lot less synchronization, which also makes the simulator faster than RobotControl. On the other hand, there are a some things that cannot be done with the simulator and require the use of RobotControl. 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.

C.2 Getting Started

The simulator can either be started directly from the Windows Explorer (from *GT2005\Bin*), from Microsoft Developer Studio, or by starting a scene description file[3]. 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 *scene GT2005*. The view can be adjusted by using the context menu of the window.

After starting a simulation, a script file may automatically be executed, setting up the robots as desired. The name of the script file is the same as the name of the scene description file but with the extension *.con*. 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 *scene*, the objects in the group *robots*, and all *information views*.

C.3 Scene View

The *Scene View* (cf. Fig. C.1) appears if the *scene* is opened from the *Tree View*. The robots are modeled in great detail, e.g. the state of its LEDs. Please note that the face LEDs of the ERS-7 are not simulated, but all other LEDs are. The view can be rotated around two axes, and it supports several mouse operations:

- Left-clicking an object allows dragging it to another position. Active and inactive robots and the ball can be moved in that way.
- Right-clicking allows rotating objects around their body centers, in the case of the Aibo this is the middle between its forelegs.
- Select an *active* robot (see below) by double-clicking it. Robot console commands are sent to the selected robot only (see also the "robot" command).
- Buttons on the robot can be "touched" by left-clicking them. However, in the scene view, they are typically too small to be useful, but they can easily be pressed when a single robot is displayed.

[3]This will only work if the simulator was started at least once before.

Figure C.1: Scene views showing the whole field and two robots of the red team

C.4 Information Views

In the simulator, *information views* are used to display debug drawings. These are generated by the robot control program, and they are sent to the simulator via *message queues*. The views are defined in the source code. They are instantiated separately for each robot. There are six kinds of views related to information received from robots: *image views*, *color space views*, *field views*, *Xabsl views*, *sensor data views*, and *timing views*. Field and image views display debug drawings received from the robot, whereas the other views visualize certain color channels, the current color table, specific information about the current state of the robot's behavior, its sensor readings, and the timing of the modules it executes. The available image and field views are defined in *GT2005\Src\Platform\Win32\SimRobot\RobotConsole.cpp*. They can be selected from the *Tree View* (cf. Fig. C.2 left).

C.4.1 Image Views

An image view (cf. left of Fig. C.3) 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*. Each drawing is underlain by an image, e.g. the camera image, the color classified image, both either with or without color correction, or any *image drawing*. This underlay is specified by the name of the view.

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

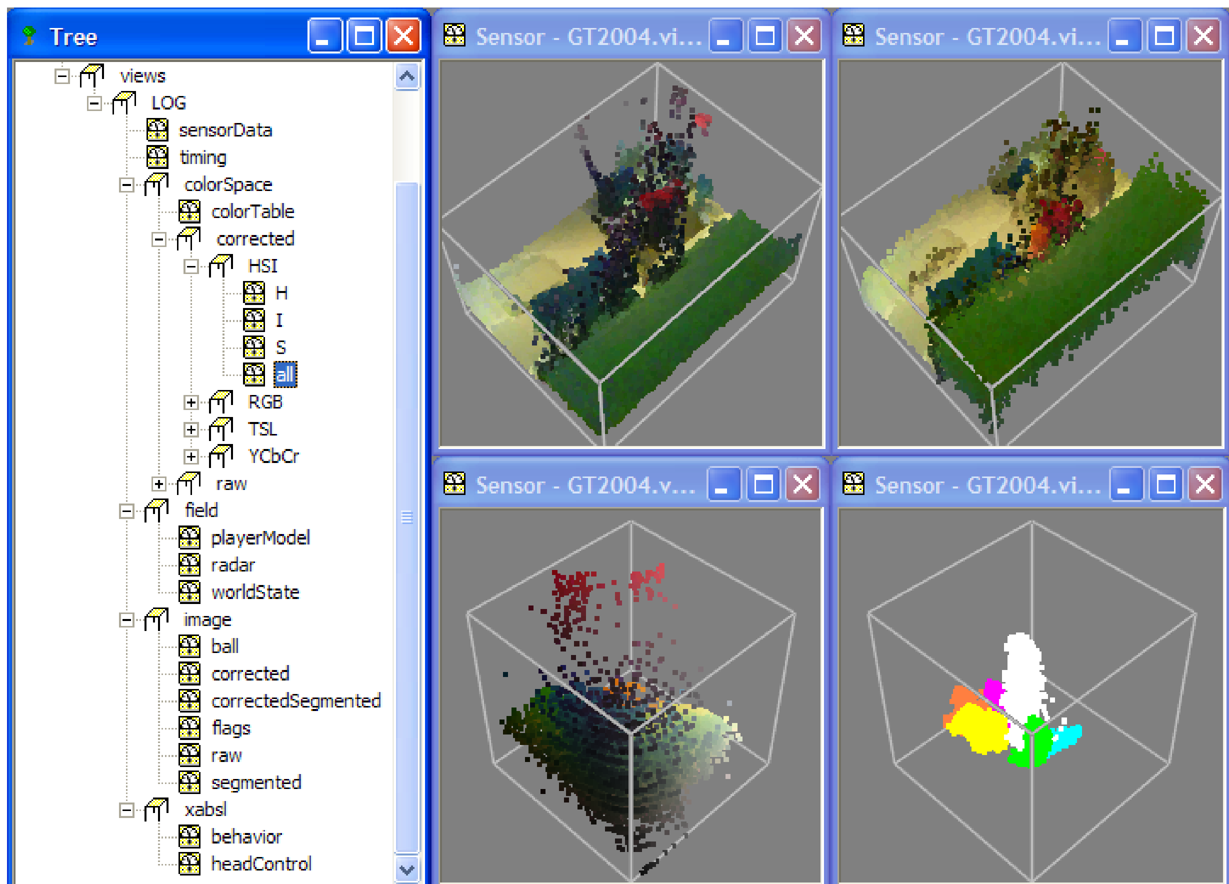


Figure C.2: Tree view, color channel views, image color space view, and color table view

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::imageProcessor_horizon,
  Drawings::imageProcessor_scanLines,
  Drawings::perceptCollection,
  Drawings::selfLocator
END_VIEW(image)
```

To display all this information, console commands (cf. Sect. C.6) 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
```

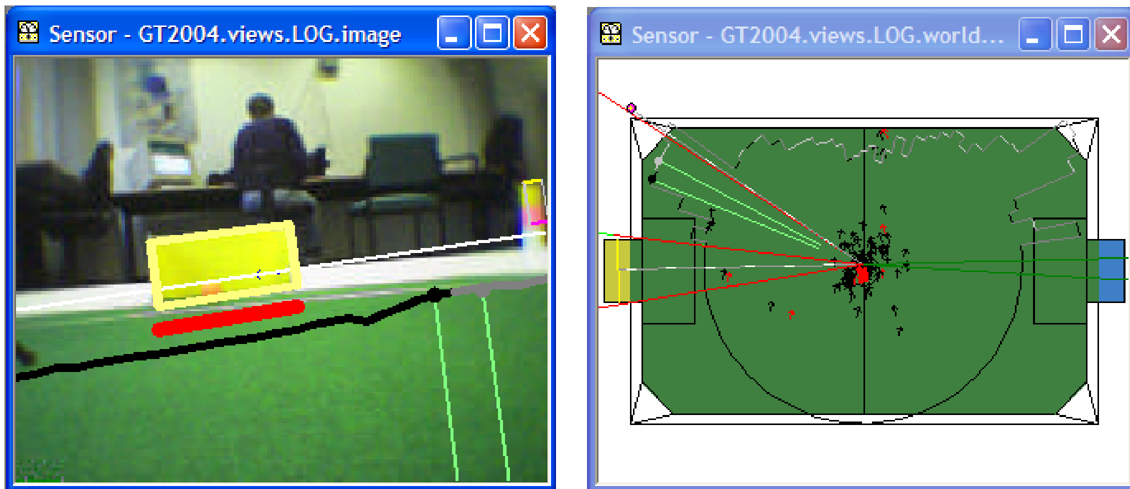


Figure C.3: Image view and field view with several debug drawings

```
dk send_selfLocator_drawing every 100 ms
```

If color table editing is activated, image views will react to the following mouse commands (cf. commands *ct on* and *ct off* in Sect. C.6.3):

Left mouse button. The color of the pixel selected is added to the selected color class. Depending on the current configuration, the neighboring pixels may also be taken into account and a larger cube is changed in the color table (cf. commands *ct imageRadius* and *ct colorSpaceRadius* in Sect. C.6.3).

Left mouse button + Shift. The color class of the pixel selected is chosen as the current color class. It is a shortcut for *ct <color>* (cf. Sect. C.6.3).

Left mouse button + Ctrl. Undoes the previous action. Currently, up to ten steps can be undone. All commands that modify the color table can be undone, including, e. g., *ct clear* and *ct load* (cf. Sect. C.6.3).

Left mouse button + Shift + Ctrl. The color class of the pixel selected is removed from the color table for all image pixels of the same class. Thereby, it is possible to remove, e. g., all orange from a certain image.

C.4.2 Color Space Views

Color Space Views visualize image information in 3-D (cf. Fig. C.2 right). They can be rotated by clicking into them with the left mouse button and dragging the mouse afterwards. There are three kinds of color space views:

Color Table. This view displays the current color table in YCbCr space. Each entry that is assigned to a certain color class is displayed in a prototypical color. The view is useful while editing color tables (cf. Fig. C.2 down right).

Image Color Space. This view displays the distribution of all pixels of an image in a certain color space (*HIS*, *RGB*, *TSL*, or *YCbCr*). It can be displayed by selecting the entry *all* for a certain color space in the *Tree View* (cf. Fig. C.2 down middle).

Image Color Channel. This view displays an image while using a certain color channel as height information (cf. Fig. C.2 up right).

C.4.3 Field Views

A field view (cf. right of Fig. C.3) 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. C.6) 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.

C.4.4 Xabsl Views

Two Xabsl views are part of each set of views. One can display information on the general robot behavior, the other on the behavior of the head. The information displayed is configured by the console commands *xis* and *xos* (cf. Sect. C.6.3). In addition, the debug keys *sendXabslDebugMessagesForBehaviorControl* or *sendXabslDebugMessagesForHeadControl* must have been set (cf. Sect. C.6.3).

```
# set behavior control solution to GermanTeam 2005
sr BehaviorControl GT2005-soccer
```

Figure C.4: XABSL views and sensor data view in the simulator

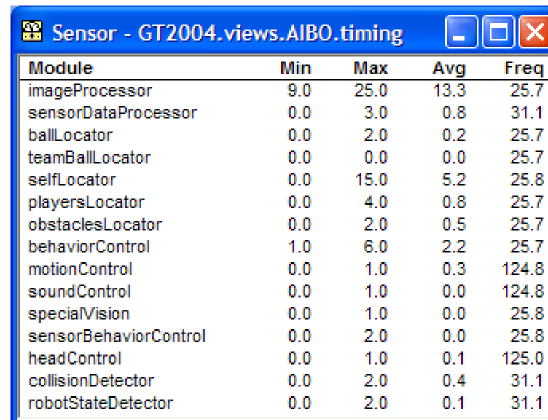
```
# request Xabsl debug messages
dk sendXabslDebugMessagesForBehaviorControl on

# 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 search-for-ball
```

C.4.5 Sensor Data View

The sensor data view displays all the sensor data taken by the robot, e.g. all joint angles, the distance measurements made by the PSDs, etc. To display this information, the debug key *send-SensorData* must have been executed (cf. right view in Fig. C.4).



Sensor - GT2004.views.AIBO.timing

Module	Min	Max	Avg	Freq
imageProcessor	9.0	25.0	13.3	25.7
sensorDataProcessor	0.0	3.0	0.8	31.1
ballLocator	0.0	2.0	0.2	25.7
teamBallLocator	0.0	0.0	0.0	25.7
selfLocator	0.0	15.0	5.2	25.8
playersLocator	0.0	4.0	0.8	25.7
obstaclesLocator	0.0	2.0	0.5	25.7
behaviorControl	1.0	6.0	2.2	25.7
motionControl	0.0	1.0	0.3	124.8
soundControl	0.0	1.0	0.0	124.8
specialVision	0.0	1.0	0.0	25.8
sensorBehaviorControl	0.0	2.0	0.0	25.8
headControl	0.0	1.0	0.1	125.0
collisionDetector	0.0	2.0	0.4	31.1
robotStateDetector	0.0	2.0	0.1	31.1

Figure C.5: The timing view in the simulator

C.4.6 Timing View

The timing view displays statistics about the speed of certain modules (cf. Fig. C.5). It shows the minimum, maximum, and average runtime of an execution of modules in milliseconds. In addition, the frequency is displayed with which the module was executed. All statistics sum up the last 100 invocations of the module. The timing view only displays information on modules the debug key for sending profiling information of which is activated, i.e. to display information about the speed of the image processor, the debug key *sendImageProcessorTime* must have been activated.

C.5 Scene Description Files

The language of scene description files is based on XML, and it is currently not documented. The scene description files provided include several files to keep the main files small: *Surfaces.scn*, *Field.scn*, *ERS210*, and *ERS7.scn*. These files cannot be opened directly by the simulator, they can only be included by other files.

In the main files, such as *GT2005.scn*, three groups can be edited:

<group name=”robots”>. This group contains all *active* robots, i. e. robots for which processes will be created. So, all robots in this group will move on their own. However, each of them will require a lot of computation time.

<group name=”extras”>. Below the group *robots*, there is the group *extras*. It contains *passive* robots, i. e. robots that just stand around, but that are not controlled by a program. Passive robots can be activated by moving their definition to the group *robots*.

<group name=”balls”>. 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 *GT2005\Config\Scenes*. Please note that there are two types of scene description files: the ones required to simulate one or more robots (about 2 KB in size, but they include larger files), and the ones that are sufficient to connect to a physical robot or to replay a log file (about 1 KB in size).

C.6 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. The first kind can only be used in a script file that is executed when the simulation is started. The second kind are *global commands* that either change the state of the whole simulation, or are sent to all robots at all times. The last type is *robot commands* that affect currently *selected robots* only (see "robot" command to find out how to select robots).

C.6.1 Initialization Commands

sc <name> <a.b.c.d> (ers210 | ers7). Starts a wireless connection to a real robot. The first parameter defines the name that will be used for this robot. The second parameter specifies the ip address of the robot. The optional parameter specifies whether the robot that will be contacted is an ERS-210 or an ERS-7. This is necessary because some information used on the PC differs between the two models. The default is the ERS-7. The command will add a new robot to the list of available robots using *name*, and it adds a full set of views 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 the simulator is exited, the connection will be terminated.

sl <name> <file> (ers210 | ers7). 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, *GT2005\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\\hello* to load the log file *c:\logs\hello.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).

C.6.2 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 *GT2005\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 initial — ready [(blue — red) [<blueScore> <redScore>]] | set | playing | finished. Game control. The command is sent to all robots. The *ready*-command is interpreted according to the team color of each robot.

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 forwardKickHard* 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. On the ERS-7, *roll* represents the second tilt axis.

robot ? | all | <name> {<name>}. Selects a robot or a group of robots. The console commands described in the next section are only sent to *selected robots*. By default, only the robot that was created or connected last is selected. Using the *robot* command, this selection can be changed. Type *robot ?* to display a list of the names of available robots. A single simulated robot can also be selected by double-clicking it in the *Scene View*. To select all robots, 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, *the simulator* simulates real-time, but it is running slower. By default this option is switched off.

 # <text>. Comment. Useful in script files.

C.6.3 Robot Commands

bc <red%> <green%> <blue%>. Defines the background color for 3-D views. The color is specified in percentages of red, green, and blue intensities.

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.

cp (indoor | outdoor | fluorescent) (low | mid | high) (slow | mid | fast) [**save**]**.** Set camera parameters. The first parameter defines the white balance, the second the gain, and the third one the shutter speed. Changing the camera parameters only has an effect on real robots. If *save* is specified, the configuration will be written to the *camera.cfg* of the currently selected location.

ct off | on | undo | <color> | (load | save) <file> | (clear | shrink | grow) [**<color>**] **| send** [**<ms> | off**] **| imageRadius <number> | colorSpaceRadius <number>.** This command controls editing the current color table. The parameters have the following meaning:

- **on | off.** Activates or deactivates mouse handling in image views. If activated, clicking into an image view modifies the color table (cf. Sect. C.4.1). Otherwise, mouse clicks are ignored.
- **undo.** Undoes the previous change to the color table. Up to ten steps can be undone. All commands that modify the color table can be undone, including, e. g., *ct clear* and *ct load*.
- **<color>.** Selects the given color as current color class.
- **(load | save) <file>.** Loads or saves the color table. The default directory is the current location. The default extension is *.c64*.
- **(clear | shrink | grow)** [**<color>**]**.** Clears, grows, or shrinks a certain color class or all color classes.
- **send** [**<ms> | off**]**.** Either sends the current color table to the robot, or defines the interval in milliseconds after which the color table is sent to the robot automatically (if it has changed). *off* deactivates the automatic sending of the color table.
- **imageRadius <number>.** Defines the size of the region surrounding a pixel that is clicked on that will be entered into the color table. 0 results in a 1×1 region, 1 in a 3×3 region, etc. The default is 0.
- **colorSpaceRadius <number>.** Defines the size of the cube that is set to the current color class in the color table for each pixel inserted. 0 results in a $1 \times 1 \times 1$ cube, 1 in a $3 \times 3 \times 3$ cube, etc. The default is 2.

dk ? [<**pattern**>] | **(** <**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. The resulting list can be shortened by specifying a search pattern after the question mark. 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. Please note that there currently is a problem with debug keys that are not permanently switched on or off. Since *<number>*, *every <number>*, and *every <number> ms* are interpreted on a frame basis, they may behave different than expected if the code checking these debug keys is not executed in every frame. For instance, the image processor is not executed in every frame of process *Cognition*, because this process waits for new sensor data, and the image processor is only executed if also a new image has arrived. So if the image processor checks a certain debug key that is not always active, it may miss some of the frames in which the key is active. This is the reason why sending a single image (*dk sendImage 1*) does not work in all cases.

hcm ? [<**pattern**>] | <**mode**>**.** Sets the head control mode. Type *hcm ?* to get a list of all available head control modes. The resulting list can be shortened by specifying a search pattern after the question mark.

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** | **pause** | **forward** | **backward** | **repeat** | **goto** <**number**> | **clear** | **(keep** | **remove)** <**message**> {<**message**>} | **(load** | **save)** <**file**> | **cycle** | **once.** The command supports both recording and replaying log files. The latter is only possible if the current set of robot processes was created using the initialization command *sl*. The different parameters have the following meaning:

?. Prints statistics on the messages contained in the current log file.

start | **stop.** If replaying a log file, starts and stops the replay. Otherwise, the commands will start and stop the recording.

pause | **forward** | **backward** | **repeat** | **goto** <**number**>**.** The commands are only accepted while replaying a log file. *pause* stops the replay without rewinding to the beginning, *forward* and *backward* advance a single step in the respective direction, and repeat just resends the current message. *goto* allows jumping to a certain position in the log file.

clear | **(keep** | **remove)** <**message**>**.** *clear* removes all messages from the log file, while *keep* and *remove* only delete a selected subset based on the set of message ids specified.

(load | **save)** <**file**>**.** These commands load and save the log file stored in memory. If the filename contains no path, *GT2005\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.

cycle | **once.** The two commands decide whether the log file is only replayed once or continuously repeated.

mr ? [<**pattern**>] | <**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. The resulting list can be shortened by specifying a search pattern after the question mark. 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** | **log** <**file**>**.** Switches the output of text messages on or off, or redirects them to a text file. 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. In addition, text messages can be stored in a log file, even if their output is switched off. The file name has to be specified after *msg log*. If the file already exists, it will be replaced. If no path is given, *GT2005\Config\Logs* is used as default. Otherwise, the full path is used. *.txt* is the default extension of text log files. It will be automatically added if no extension is given.

pr continue | **ballHolding** | **illegalDefender** | **illegalDefense** | **obstruction** | **goaliePushing** | **playerPushing** | **pickup** | **leaving** | **damage.** Penalize robot. The command sends one of the nine 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.

replace. If the mode is set to *replace*, only the newest message of each type is preserved in the queue. 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 ? [**<pattern>**] | **<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. The resulting list can be shortened by specifying a search pattern after the question mark.

so (worldState | **percepts) (off** | **show** | **replace** [**<noiseLevel>**] **).** The command configures the use of so-called *oracled world states* and *oracled percepts*. This allows the modules calculating the world state or the percepts to be switched off without a failure of the robot. In addition, the image generation itself can be switched off to speed up the simulation and allow the use of several simulated robots in parallel. Sending oracled world states or percepts can either be deactivated (*off*), one of them can be activated just to be able to display oracled data (*show*), or it can replace the data produced on the robot (*replace*). In addition, a noise level between 0 and 1 can be specified. By default, no oracled data is sent. Note that this command only has an effect on simulated robots.

sr ? [**<pattern>**] | **<module> (?** [**<pattern>**] | **<solution>** | **off).** Sends a solution request. This command allows switching the solutions for a certain module. To deactivated a module, either type *sr <module> disabled* or *sr <module> off*. Type *sr ?* to get a list of all modules. To get the solutions for a certain module, type *sr <module> ?*. In both cases, the resulting lists can be shortened by specifying a search pattern after the question mark.

tr ? [**<pattern>**] | **<type>.** Sends a tail request. Type *tr ?* to see all available tail requests. The resulting list can be shortened by specifying a search pattern after the question mark.

xbb [**hc**] **(?** [**<pattern>**] | **unchanged** | **<behavior>** {**<num>**} **).** Selects a Xabsl basic behavior. The command suppresses the basic behavior currently selected by the Xabsl engine and replaces it with the behavior specified by this command. Type *xbb ?* to list all available Xabsl basic behaviors. The resulting list can be shortened by specifying a search pattern after the question mark. 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 Xabsl engine. The command *xbb* only works if a Xabsl behavior was loaded with the command *xlb* (see below). If the command is immediately followed by *hc*, it applies to head control, otherwise it applies to main behavior control.

xis [**hc**] **(?** [**<pattern>**] | **<inputSymbol> (on** | **off)).** Switches the visualization of a Xabsl input symbol in the *Xabsl View* on or off. Type *xis ?* to list all available Xabsl input symbols. The resulting list can be shortened by specifying a search pattern after the question mark. The command *xis* only works if a Xabsl behavior was loaded with the command *xlb* (see below). If the command is immediately followed by *hc*, it applies to head control, otherwise it applies to main behavior control.

xo [**hc**] **(?** [<**pattern**>] | **unchanged** | <**option**> {<**num**>} **).** Selects a Xabsl option. The command suppresses the option currently selected by the Xabsl 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 Xabsl options. The resulting list can be shortened by specifying a search pattern after the question mark. Use *xo unchanged* to switch back to the option currently selected by the Xabsl engine. The command *xo* only works if a Xabsl behavior was loaded with the command *xlb* (see above). If the command is immediately followed by *hc*, it applies to head control, otherwise it applies to main behavior control.

xos [**hc**] **(?** [<**pattern**>] | <**outputSymbol**> **(on** | **off** | **?** [<**pattern**>] | **unchanged** | <**value**> **)).** Show or set a Xabsl output symbol. The command can either switch the visualization of a Xabsl output symbol in the *Xabsl View* on or off, or it can suppress the state of an output symbol currently set by the Xabsl engine and replace it with the value specified by this command. Type *xos ?* to list all available Xabsl output symbols. To get the available states for a certain output symbol, type *xos <outputSymbol> ?*. In both cases, the resulting list can be shortened by specifying a search pattern after the question mark. Use *xos <outputSymbol> unchanged* to switch back to the state currently set by the Xabsl engine. The command *xos* only works if a Xabsl behavior was loaded with the command *xlb* (see above). If the command is immediately followed by *hc*, it applies to head control, otherwise it applies to main behavior control.

xsb [**hc**]**.** Sends the compiled version of the current Xabsl behavior to the robot. If the command is immediately followed by *hc*, it applies to head control, otherwise it applies to main behavior control.

C.7 Examples

This section presents some examples of script files to automate various tasks:

C.7.1 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 robot, 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 AIBO1 172.21.3.201 ers7
```

```
# 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 GT2005

# 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
```

C.7.2 Replaying a Log File

The example script shown was used to test the GT2005ImageProcessor. It instantiates a robot named *LOG1* that is fed by the data stored in the log file *GT2005\Config\Logs\logFile-with-images.log*.

```
# replay a log file
sl LOG1 logFile-with-images ers7

# 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 GT2005
sr SelfLocator GT2005
sr BallLocator GT2005
sr PlayersLocator GT2005
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
```

C.7.3 Remote Control

This script demonstrates joystick remote control of the robot.

```
# connect to a robot
sc 172.21.3.201 ers7

# 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 forwardKickHard
jbc 2 mr headLeft
jbc 3 mr headRight
jbc 4 mr demoSit
jbc 5 mr demoScratchHead
jbc 6 tr wagHorizontalFast
jbc 7 tr noTailWag
```

Appendix D

Extensible Agent Behavior Specification Language

The *Extensible Agent Behavior Specification Language (XABSL)* [39, 40] is an XML based language for behavior engineering. It simplifies the process of specifying complex behaviors and supports the design of both very reactive and long term oriented agent behaviors. It is not only a behavior modeling or description language – instead, behaviors written in *XABSL* can be transformed automatically into an intermediate code which is executed directly on a target platform using the *XabslEngine* class library. Together with the interpreter and a variety of tools for visualization and debugging, behavior developers get a complete system for behavior specification, documentation, testing, execution, and debugging. The whole *XABSL* system can be downloaded for free from the *XABSL* web site [38].

Section D.1 describes hierarchical finite state machines for action selection as the behavior control architecture behind *XABSL*. Section D.2 gives an overview of the *XABSL* language and section D.3 provides a brief introduction to the language elements and the syntax. Section D.4 deals with some technical issues related to the use of XML techniques and the tools that were developed in conjunction with *XABSL*. Section D.5 describes the runtime system *XabslEngine*. Finally, section D.6 relates the architecture and the language to other approaches.

D.1 Hierarchies of Finite State Machines

In *XABSL*, behavior modules (*options*) that contain state machines for decision making are ordered in a hierarchy, the *option graph*, with atomic *basic behaviors* at the leaves.

D.1.1 The Option Graph

An *XABSL* behavior specification consists of a set of behavior modules called *options* and a set of distinct simple actions (skills) called *basic behaviors*. Both options and basic behaviors can have parameters. The options are ordered in a hierarchy – complex behaviors are composed from

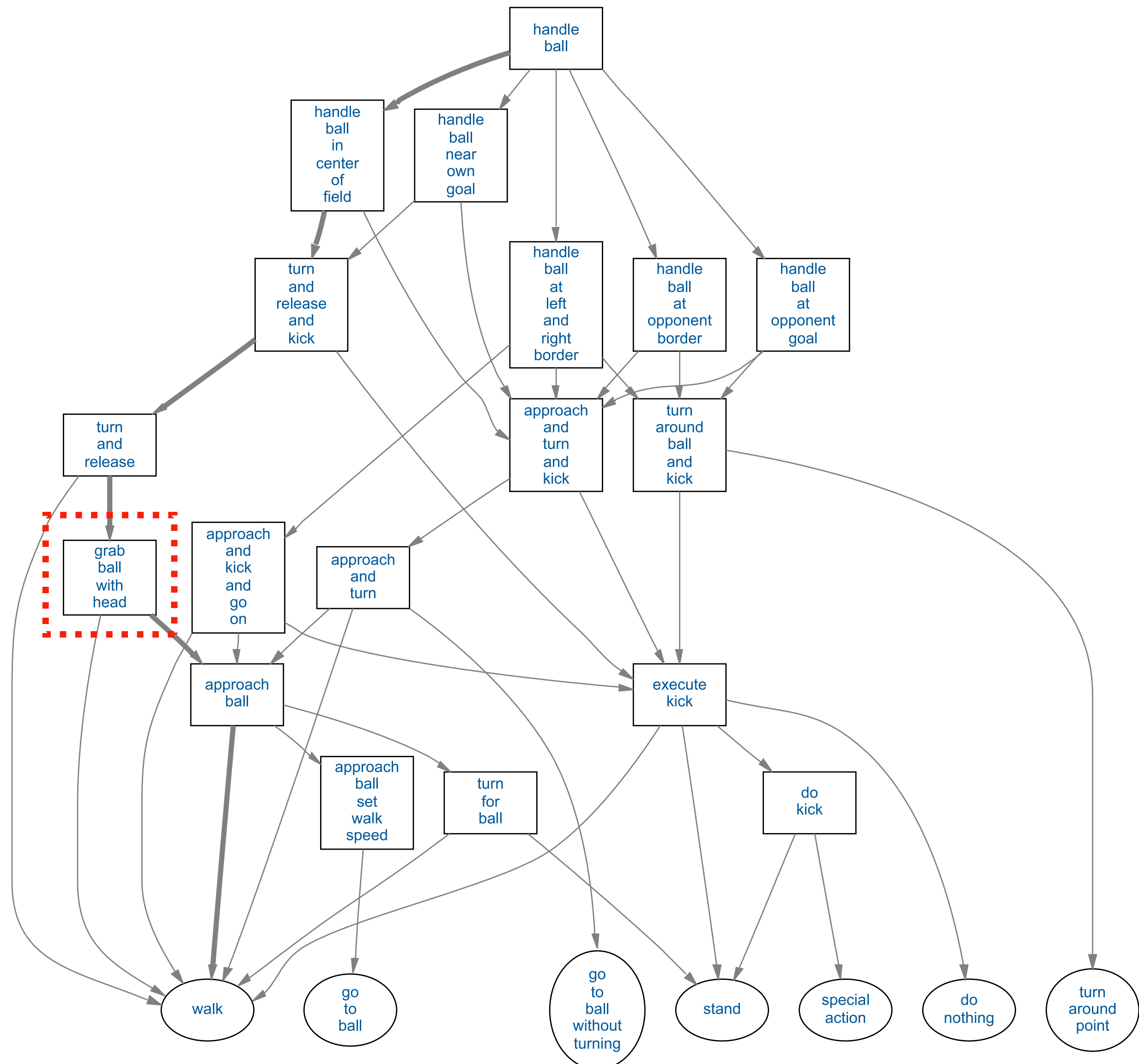


Figure D.1: An example for an option graph from the robot soccer domain (the ball handling part of the *GermanTeam*'s soccer behaviors for the world championships 2004 in Lisbon). Boxes denote options, ellipses denote basic behaviors. The edges show which other option or basic behavior can be activated from within an option. The thick edges mark one of the many possible option activation paths. The internal state machine of option *"grab-ball-with-head"* (marked with the dashed rectangle) is shown in figure D.2.

simpler ones. Each option uses a set of other subordinated options and/or basic behaviors to realize a certain behavior.

For example in figure D.1, the option *"grab-ball-with-head"* (a behavior for grabbing and holding the ball between the front legs and the head of an Aibo robot) is composed of the option *"approach-ball"* (a behavior for walking to the ball) and the basic behavior *"walk"* (a behavior for blind walk). Each basic behavior and option can be used from more than one other option. This allows reusing the same behaviors in different contexts. E.g. in figure D.1 a few other options than *"grab-ball-with-head"* use the option *"approach-ball"*. This helps behavior developers to modularize their behaviors. In the example, only one behavior for ball approaching was developed and fine-tuned and then used by very different other options.

The option hierarchy can be seen as a rooted directed acyclic graph, called the *option graph*. The basic behaviors are the leaves (terminal nodes) of this graph. The "topmost" option (at the root of the graph) is called the *root option*. Note that in *XABSL* it is possible to specify option graphs that contain loops (and are for this reason not acyclic). But the runtime system is able to detect such loops at startup and denies work if the graph is not acyclic.

In the architecture, action selection means to activate, parameterize, and execute one of the basic behaviors. Therefore, the root option (which is always active) activates and parameterizes one of its subsequent options, this subsequent option again activates and parameterizes one of its subsequent options or basic behaviors and so on until a basic behavior is activated, parameterized, and executed. As the option graph is directed and acyclic, always exactly one of the basic behaviors is reached and executed.

In *XABSL*, a subset (sub-graph) of the options and basic behaviors which is spanned by a specially marked option, the *root option*, is called an *agent*. (As the option graph does not need to be connected completely, it is not possible to determine a single root option of the graph – *agents* mark the root options of different trees.)

D.1.2 State Machines

Within options, the activation of subordinated behaviors is done by finite state machines. Figure D.2 shows an example of such a state machine. In each option, exactly one state is marked as the *initial state*. This state gets activated when the option becomes newly activated. An arbitrary number of states can be declared as *target states*. This allows indicating that a behavior is finished as higher options can query whether a subsequent option reached a target state. Each state is connected to exactly one subsequent option or subsequent basic behavior. Note that more than one state can be connected to the same subsequent option or basic behavior. Always exactly one state of an option is active. This state determines, which of the subordinated behaviors is activated and how its parameters are set.

Each state has a *decision tree*, which selects a transition to either another or the same state. Figure D.3 gives an example for such a decision tree. For the decisions, the following information can be used: Parameters passed by higher options, the world state, other sensory information, and messages from other agents. As timing is often important, it can also be taken into account how long the state and the option are already active. In addition, the success of a subsequent option can be tested by querying whether the subsequent option reached one of its target states.

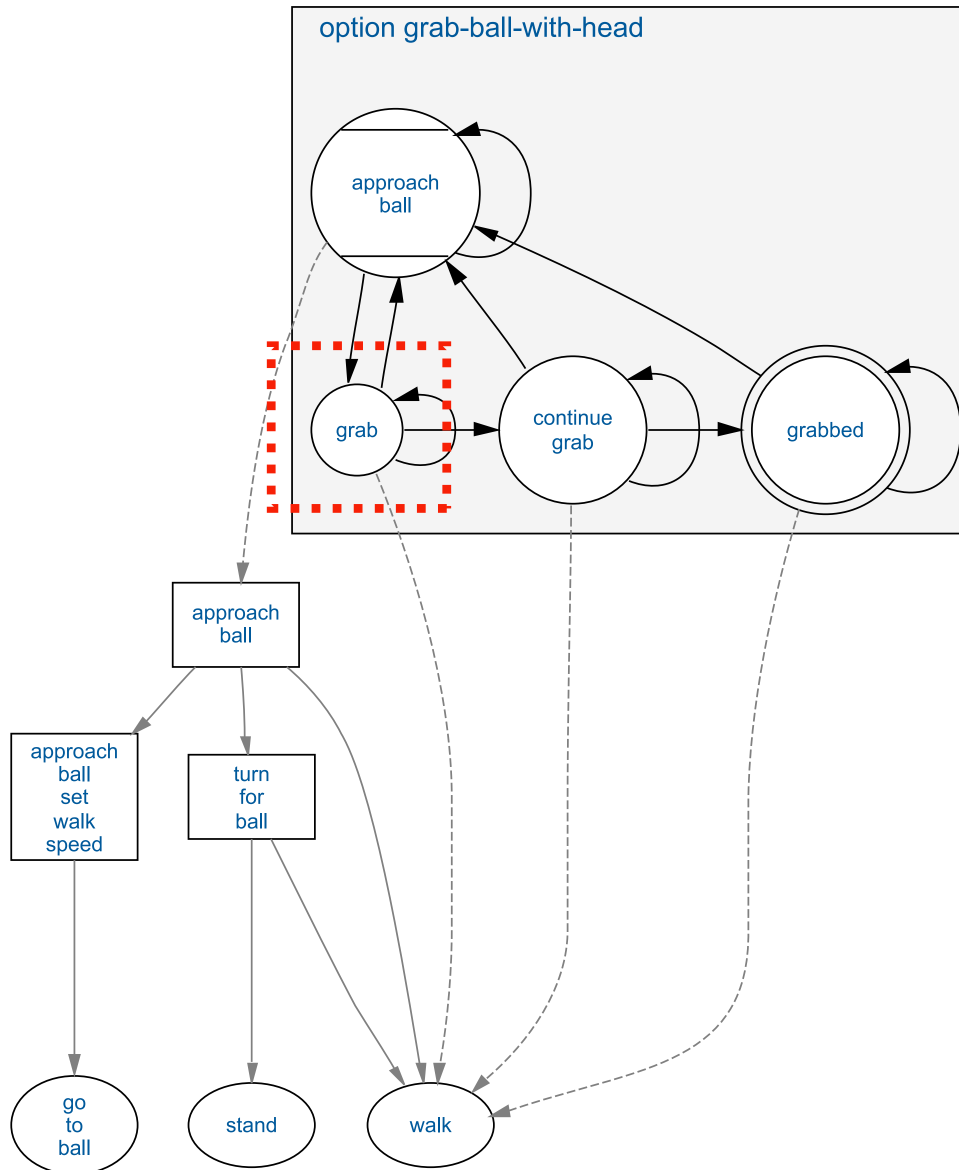


Figure D.2: An example for an option's internal state machine (the option *"grab-ball-with-head"* from the example in figure D.1). Circles denote states, the circle with the two horizontal lines denotes the initial state, the double circle denotes a target state. An edge between two states indicates 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. The decision tree of state *"grab"* (marked with the dashed rectangle) is shown in figure D.3.

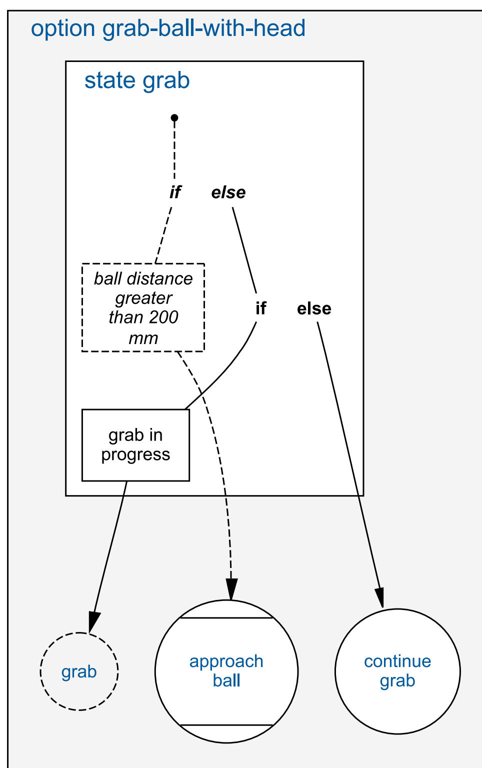


Figure D.3: An example for a decision tree of a state (state *"grab"* of option *"grab-ball-with-head"* in figure D.2). The leaves of the tree are transitions to other states. The dashed circle denotes a transition to the same state. The pseudo code of that decision tree is shown in figure D.4.

As each state has its own decision tree, the decisions are made not only dependent on the representation of environment's state but also on the decisions that were done in the past. When the active state is taken into account, hysteresis functions between states are possible. That means if there is a transition from state A to state B for a certain condition, this condition can be different than for the transition from B to A. Thus, behaviors can be preferred once they were selected to avoid oscillations.

In the robot soccer example from figure D.2, the option *"grab-ball-with-head"* is initially in the state *"approach-ball"*. As long as the state is active, the subsequent option *"approach-ball"* is activated with certain parameters, making the robot move towards the ball. As soon as the ball gets closer than a threshold, the decision tree of state *"approach-ball"* selects a transition to state *"grab"*. State *"grab"* becomes the active state and the subsequent basic behavior *"walk"* is executed with parameters such that the robot walks onto the ball. If it somehow happens that during that the ball gets farer away than another, the decision tree of state *"grab"* selects a transition back to state *"approach-ball"*. Otherwise, after a certain time a transition to state *"continue-grab"* is selected (cf. fig. D.4).

```
if ((ball.time-since-last-seen-consecutively < 200)    // ball distance greater than 200 mm
    && (ball.consecutively-seen-time > 100)
    && (ball.seen.distance > 200)
    && (ball.seen.distance < 800)                  )
{
  transition-to-state(approach-ball);
}
else
{
  if (time-of-state-execution < 1000)   // grab in progress
  {
    transition-to-state(grab);
  }
  else
  {
    transition-to-state(continue-grab);
  }
}
```

Figure D.4: The pseudo code of the decision tree of state *"grab"* (cf. fig. D.3).

D.1.3 Interaction with the Environment

To access the information about the world that is needed for decision making, symbolic representations are used. The world model of the agent system is divided into simple and non-structured information items, called the *input symbols*. In the ball grabbing example, amongst others the symbol *"ball.seen.distance"* is used to reference the distance to the seen ball in the world model.

The main actions of the agent system are controlled by the basic behaviors. It does not matter whether these actions are generated completely reactively using closed sensor-actuator loops or whether intermediate representations such as a world model are used in addition. In embodied agents, the basic behaviors usually control the agent's locomotion system. E.g. in the soccer behaviors of the *GermanTeam*, the basic behaviors were used to control all leg movements of the robots (walking and kicking).

Besides the execution of basic behaviors, the environment can be influenced by setting special requests, the *output symbols*. Each state within an option can set such output symbols to certain values to control perception processes or additional actuators. For instance, for the robots of the *GermanTeam*, an important actuator independent from the leg movements is the head. The output symbol *"head-control-mode"* is used to set a general mode how to move the head independent from the selected basic behavior. This mode is then used by other parts of the software to control the head movements. But also LED and sound output and messages to team mates are triggered with output symbols.

D.1.4 The Execution of the Option Hierarchy

An *XABSL* behavior implementation is always a part of a wider agent program. The surrounding software has to process the sensor readings, build up (if necessary) a world model, manage the

communication to other agents, control the actuators and so on. At some point in this *sense-think-act cycle*, the program passes the control to the *XABSL* system to execute the option graph. Before, all data needed for decision making have to be up-to-date. Afterwards, the actions generated by the basic behaviors and the additional requests set by the output symbols have to be (processed and) sent to the actuators of the agent system.

Each time the option graph is executed, a basic behavior becomes selected and executed. The *XABSL* system has to be executed as frequent as required for the reactivity of the action system. Usually, it is called as often as new data can be obtained from the agent's main sensor. For instance on the Aibo robots of the *GermanTeam*, the *XABSL* behaviors are always executed after a newly perceived image was processed.

The execution of the option graph starts from the root option (cf. sect. D.1.1) of the agent. The decision tree of the active state of the root option is executed to determine the next active state, which can of course be the same as before. For the subsequent option of the active state, again the decision tree of the active state is executed and so on until the subsequent behavior of a state is a basic behavior.

Each time a decision tree activates another or the same state, the newly activated state sets the parameters of the subsequent option or basic behavior and the state's output symbols. Note that output symbols that were set during this process can be overwritten by options lower in the option graph. If an option was not active during the last execution of the option graph, the state machine is reset (the initial state is activated).

The *option activation path* (cf. fig. D.1) follows the path from the root option to the currently activated basic behavior through all active options. As each option activates only one subsequent behavior at a time and as the graph is rooted, directed, and acyclic, such a path exists and contains no branches. The *time of option activation* is the time, how long an option was consecutively activated. This time is set to zero when an activated option was not active during the last execution of the option graph. Accordingly, the *state execution time* is the time how long the active state was consecutively activated.

The option activation path including the option activation time, active state, and state activation time for all of its options constitute the global state of an *XABSL* agent. The generated actions of the system depend on this state, the perceptions and the world model (and, if the basic behaviors have persistent states, on these states).

D.2 Behavior Specification in XML

Implementing such an architecture totally in C++ proved to be error prone and not very comfortable [10]. The source code became very large and it was quite hard to extend the behaviors. Therefore, the *Extensible Agent Behavior Specification Language* (*XABSL*) was developed to simplify the behavior engineering process.

The *XABSL* language and supporting tools are completely based on XML techniques. Figure D.5 shows an example of an *XABSL* XML notation. The reasons to use XML instead of defining a new grammar from scratch were the big variety and quality of existing editing, validation, and processing tools, the possibility of easy transformation from and to other languages as well as the

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE symbol-and-basic-behavior-files SYSTEM "../symbol-and-basic-behavior-files.dtd">
<option xmlns="http://www.ki.informatik.hu-berlin.de/XABSL2.2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://www.ki.informatik.hu-berlin.de/XABSL2.2
../../../../Tools/Xabsl2/xabsl-2.2/xabsl-2.2.option.xsd" name="grab-ball-with-head" initial-state="approach-ball">
  &ball-symbols;
  &head-and-tail-symbols;
  &motion-request-symbols;
  &special-action-symbols;
  &strategy-symbols;
  &robot-state-symbols;
  &common-basic-behaviors;
  &simple-basic-behaviors;
  &options;
    <common-decision-tree>
    <if>
      <condition description="ball distance greater than 200 mm">
        <and>
          <less-than>
            <decimal-input-symbol-ref ref="ball.time-since-last-seen-consecutively"/>
            <decimal-value value="200"/>
          </less-than>
          <greater-than>
            <decimal-input-symbol-ref ref="ball.consecutively-seen-time"/>
            <decimal-value value="100"/>
          </greater-than>
          <greater-than>
            <decimal-input-symbol-ref ref="ball.seen.distance"/>
            <decimal-value value="200"/>
          </greater-than>
          <less-than>
            <decimal-input-symbol-ref ref="ball.seen.distance"/>
            <decimal-value value="800"/>
          </less-than>
        </and>
      </condition>
      <transition-to-state ref="approach-ball"/>
    </if>
  </common-decision-tree>
  ...
  <state name="grab">
    <subsequent-basic-behavior ref="walk">
      <set-parameter ref="walk.type">
        <constant-ref ref="walk-type.normal"/>
      </set-parameter>
      <set-parameter ref="walk.speed-x">
        <decimal-value value="200"/>
      </set-parameter>
      <set-parameter ref="walk.speed-y">
        <decimal-value value="0"/>
      </set-parameter>
      <set-parameter ref="walk.rotation-speed">
        <multiply>
          <decimal-input-symbol-ref ref="ball.seen.angle"/>
          <decimal-value value="2"/>
        </multiply>
      </set-parameter>
    </subsequent-basic-behavior>
    <set-output-symbol ref="head-control-mode" value="head-control-mode.catch-ball"/>
    <set-output-symbol ref="ball.handling" value="handling-the-ball"/>
    <decision-tree>
      <if>
        <condition description="grab in progress">
          <less-than>
            <time-of-state-execution/>
            <decimal-value value="1000"/>
          </less-than>
        </condition>
        <transition-to-state ref="grab"/>
      </if>
      <else>
        <transition-to-state ref="continue-grab"/>
      </else>
    </decision-tree>
  </state>
  ...
</option>
```

Figure D.5: An example for an *XABSL* XML notation: a source code fragment for the state *"grab"* (cf. fig. D.3) of option *"grab-ball-with-head"* (cf. fig. D.2).

general flexibility of data represented in XML languages. The syntax and even all constraining relations between the language elements are specified in XML schema, so no other compile or validation tools than standard XSLT / XML processors are needed[1]. Many XML editors are able to check whether an *XABSL* document is valid at runtime. A high validation and compile speed results in short change-compile-test cycles.

Standard XSLT transformations are used to compile *XABSL* documents to an intermediate code for the runtime system and to generate extensive documentations. Note that the figures D.1, D.2, D.3, and D.4 were generated automatically from the XML source in figure D.5.

An aftereffect of this restriction to standard XML technologies and tools is that the language had to be adapted to existing tools to some extend. For example, some constructs had to be introduced only for the compatibility with the used XML editor. And, which is also not typical for a programming language, there is a relatively strict distribution of language elements onto different file types, which is required for efficient processing of the data (in previous versions of *XABSL*, the complete specification of the behaviors was in only one file, which made editing very slow).

Agent behavior specifications based on the architecture introduced in the previous section can be completely described in *XABSL*. There are language elements for options, their states, and their decision trees. Boolean logic ($||$, $\&\&$, !, $==$, $!=$, $<$, $<=$, $>$, and $>=$), simple arithmetic operators ($+$, $-$, $*$, $/$, and $\%$), and conditional decimal expressions (comparable to the ANSI C question mark operator, $a\ ?\ b\ :\ c$) can be used for the specification of decision trees and parameters of subsequent behaviors. 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 model) and to output functions (e.g. to set requests for other parts of the information processing). For each variable or function that one wants to use in conditions, a symbol has to be defined. This makes the *XABSL* framework independent from specific software environments and platforms. An example:

```
<decimal-input-symbol name="ball.x" measure="mm"
  description="The absolute x position on the field"/>
<decimal-input-symbol name="utility-for-dribbling" measure="0..1"
  description="Utility for dribbling"/>
<boolean-input-symbol name="goalie-should-jump-right"
  description="A ball rolls along to the right"/>
```

The first symbol *"ball.x"* simply refers to a variable in the world state of the agent system, *"utility-for-dribbling"* stands for a member function of an utility analyzer and *"goalie-should-jump-right"* represents a complex predicate function that determines whether a fast moving ball is headed to the right portion of the own goal. In options, these symbols then can be referenced.

[1]The only exception is the check for loops in the option graph. This can not be done by validating documents against XML Schema and is therefore checked by the runtime system at startup.

The developer may decide whether to express complex conditions in *XABSL* by combining different input symbols with Boolean and decimal operators or by implementing the condition as an analyzer function in C++ and referencing the function via a single input symbol.

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.

D.3 The XABSL Language

This section gives a brief introduction to the syntax and the semantics of the *XABSL* language. Thereby, the formal structure of the grammar is, as usual in the XML world, displayed with syntax diagrams (e.g. fig. D.6) instead of textual representations such as EBNF or others. A complete language reference can be found at the *XABSL* web site [38].

D.3.1 Symbols, Basic Behaviors, and Option Definitions

Symbols, basic behaviors, and option definitions are referenced from inside options. In order that it can be checked whether a referenced symbol (or option parameter etc.) exists, they all have to be declared in definition files (comparable to header files in C++).

First, there are definition files for symbols. There can be many of them for grouping symbols thematically. The element *"symbols"* is the root element of such a symbol file (cf. fig. D.6). *XABSL* has six different symbol types that can be declared in arbitrary order inside a symbols element: A *"boolean-input-symbol"* represents a symbol for a Boolean, and a *"decimal-input-symbol"* a symbol for a decimal variable or function (the *XabslEngine* uses the data type double for decimal values). Besides the attribute *"name"*, which is the id of the symbol and which is referenced from inside options, it has additional attributes that are needed for the generation of the HTML documentation. A *"decimal-input-function"* is a prototype for a parameterized decimal function. Each parameter of a function is defined in a separate *"parameter"* child element. The element *"enumerated-input-symbol"* represents a symbol for an enumerated variable or function. Each enumerated item is defined in a single *"enum-element"* child element. Output symbols are declared with *"enumerated-output-symbol"*, like the *"enumerated-input-symbol"* element with *"enum-element"* child elements. The element *"constant"* defines a decimal constant.

Basic behaviors are written in C++. Nevertheless, in basic behavior files, a prototype has to be declared for each of them. The element *"basic-behaviors"* (cf. fig. D.7) is the root element of such a file and has to have at least one child element of the type *"basic-behavior"*, which defines a prototype for a basic behavior. Optionally it has *"parameter"* child elements which declare a parameter that can be passed to the corresponding basic behavior written in C++.

Every option is 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 has an *"option-definitions"* (cf. fig. D.7) root element. Inside, *"option-definition"* elements define a prototype for an option. As the *"basic-behavior"* element, it can have *"parameter"* child elements that specify parameters of an option.

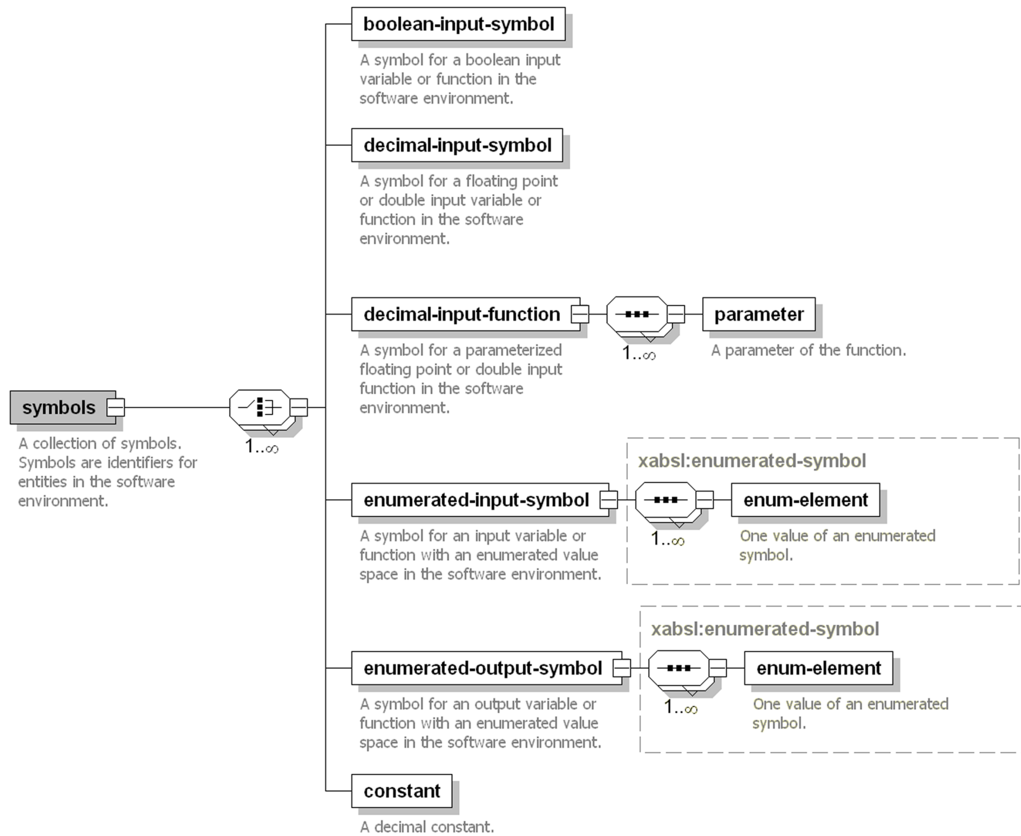


Figure D.6: The syntax of the element *"symbols"*.

D.3.2 Options and States

The root element of an option file is the *"option"* element (cf. fig. D.7). Inside that, the files for all referenced symbol definitions and basic behavior and option prototypes are included using a DTD include mechanism (cf. sect. D.4).

After the included *"symbols"*, *"basic-behaviors"*, and *"option-definitions"* child elements, a *"common-decision-tree"* child element can follow. This is a decision tree which is carried out before the decision tree of the active state. If no condition of the common decision tree proves to be true, the decision tree of the active state is carried out. This can be used to reduce the complexity of implementation when the conditions for a transition are same in each state. If the common decision tree uses expressions that are specific for a state (*"time-of-state-activation"* or *"subsequent-option-reached-target-state"*), these expressions refer to the state that is currently active. The child elements of a *"common-decision-tree"* are the same as in the normal decision tree of a state, which is explained later in this section.

Followed by the optional *"common-decision-tree"*, each option has to have at least one *"state"* child element, which represents a single state of an option's state machine (cf. fig D.8). Its first child element is either a *"subsequent-option"* or a *"subsequent-basic-behavior"*, determining which subsequent behavior is executed when this state is active. If the referenced option

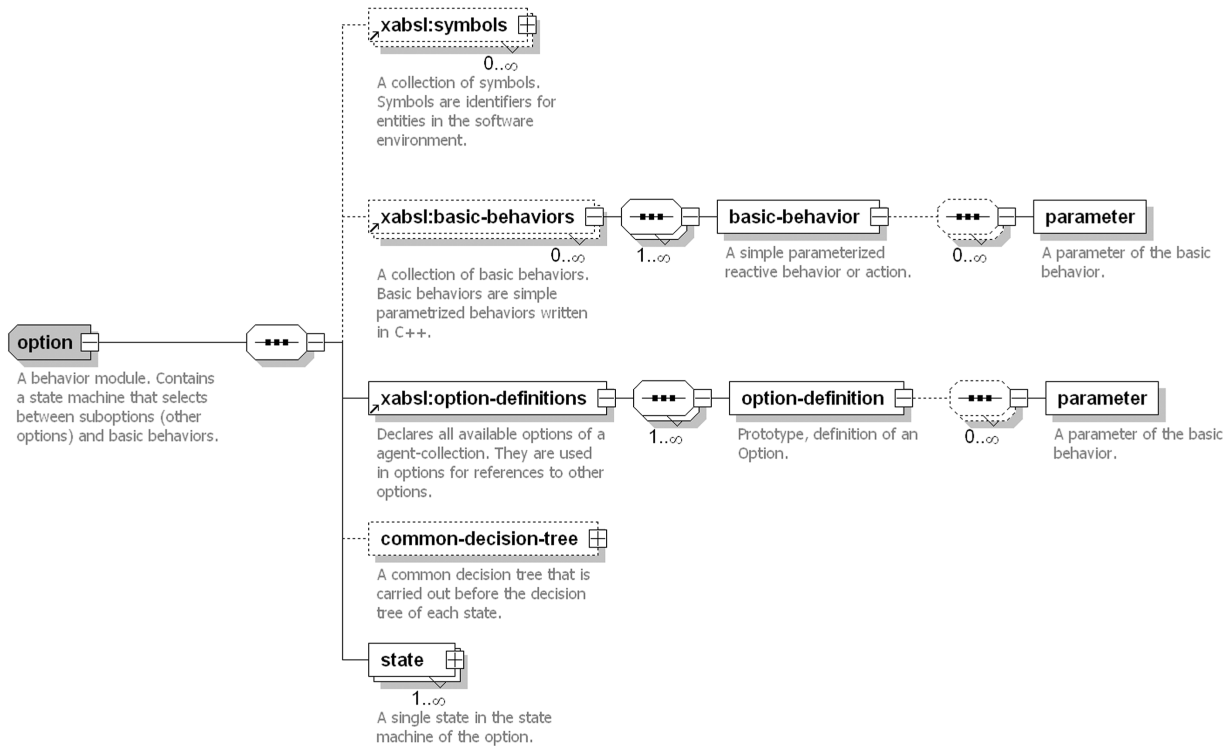


Figure D.7: The syntax of the element *"option"*.

or basic behavior has parameters, these can be set with *"set-parameter"* child elements. If a state does not set all parameters of a subsequent behavior, the execution engine sets the remaining parameters to zero. The child element of the *"set-parameter"* element is a decimal expression, which are described later in this section.

After the definition of the subsequent behavior, output symbols can be set by inserting *"set-output-symbol"* child elements. Note that the state machine is carried out first and only the state being active afterwards 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 only applied to the software environment when the option graph was executed completely.

Each state has a decision tree. The task of this decision tree is to determine a transition to a following state (which can be the same state). Consequently, the leaves of a decision tree are transitions to other states. The element *"decision-tree"* itself is of the type *"statement"* (cf. fig. D.9). A *"statement"* can either be an if, else-if, else block, or a transition to a state. The *"transition-to-state"* element represents a transition to another state.

An if, else-if, else block consists of an *"if"*, optional *"else-if"* 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, which allows for complex nested expressions. The *"condition"* element has a Boolean expression (cf. next section) as a child element.

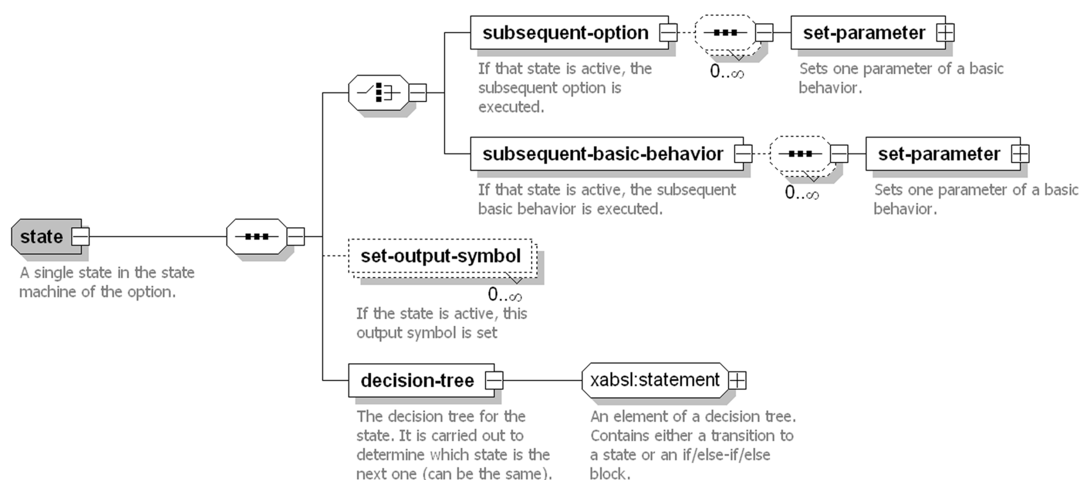


Figure D.8: The syntax of the element *"state"*.

D.3.3 Boolean and Decimal Expressions

A *"boolean-expression"* can be one of the elements shown in figure D.10. A *"boolean-input-symbol-ref"* references a Boolean input symbol. The element *"enumerated-input-symbol-comparison"* compares the value of an enumerated input symbol with a given enumerated value. The elements *"and"* and *"or"* represent the Boolean && and || operators and have at least two *"boolean-expression"* child elements. In contrast, *"not"* has only one *"boolean-expression"* child element and represents the Boolean ! operator.

The elements *"equal-to"*, *"not-equal-to"*, *"less-than"*, *"less-than-or-equal-to"*, *"greater-than"*, and *"greater-than-or-equal-to"* are the $==$, $!=$, $<$, $<=$, $>$ and $>=$ operators. They all have two *"decimal-expression"* child elements, which are described below.

The expression *"subsequent-option-reached-target-state"* is true when the subsequent behavior of the state is an option and when the active state of the subsequent option is marked as a target state. Otherwise this statement is false. It can be used to give a feed-back to higher options that a behavior is finished.

Elements from the *"decimal-expression"* group (cf. fig. D.11) can be used inside some Boolean expressions and for the parameterization of subsequent behaviors.

A *"decimal-input-symbol-ref"* references a decimal input symbol. A *"decimal-input-function-call"* represents a call to a decimal input function. For each parameter of the function, a *"with-parameter"* element must be inserted. If a parameter is not set, the executing engine sets the parameter to zero.

The element *"with-parameter"* has a child element from the *"decimal-expression"* group.

A *"constant-ref"* references a constant which was defined in a *"symbols"* collection, a *"decimal-value"* is a simple decimal value, e. g. *"3.14"*, and *"option-parameter-ref"* references a parameter of the option.

The elements *"plus"*, *"minus"*, *"multiply"*, *"divide"*, and *"mod"* stand for the arithmetic $+$, $-$, $*$, $/$ and $\%$ operators. They all have two child elements from the *"decimal-expression"* group.

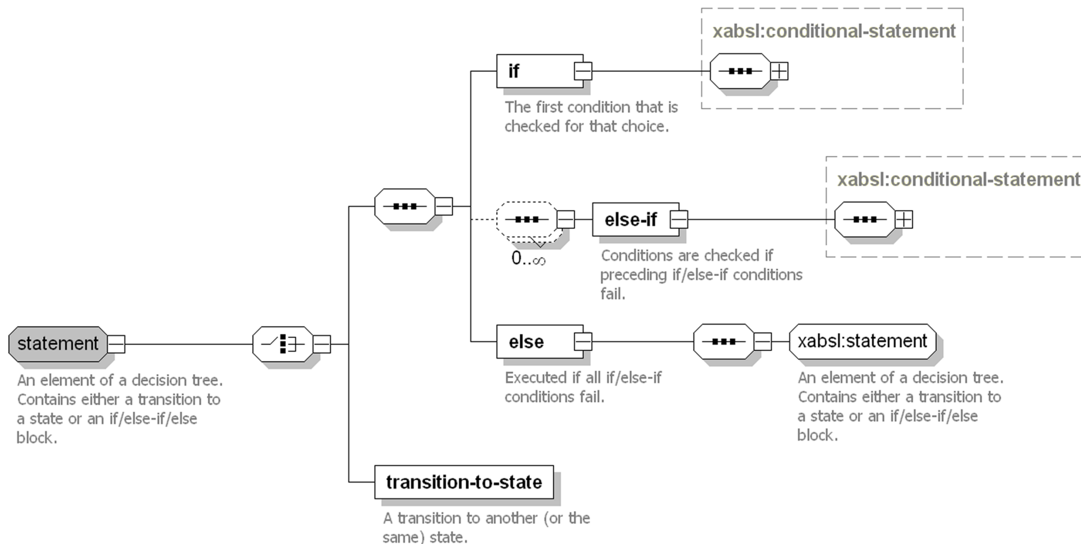


Figure D.9: The syntax of the group *"statement"*. Amongst others, the element *"decision-tree"* is of this type.

The element *"time-of-state-execution"* can be used to query how long the state has already been 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). Analogically, element *"time-of-option-execution"* represents the time the option has already been active. This time is reset if the option was not active during the last execution of the engine. It may also happen here that the option activation path above the current option changes without this time being reset.

The statement *"conditional-expression"* works such as an ANSI C question mark operator. A *"condition"* which has a *boolean-expression* child element is checked. If the condition is true, the decimal expression *"expression1"*, otherwise *"expression2"* is returned. It is mainly used to set parameters of subsequent behaviors (which have to be decimal) dependent on a condition.

D.3.4 Agents

The file "agents.xml" is the root document of an *XABSL* behavior specification. It includes all the options and defines agents. Figure D.12 shows the structure of the *"agent-collection"* element. It has *"title"*, *"platform"*, and *"software-environment"* elements that are only used for generating the HTML documentation.

With an *"agent"* element, an agent is declared by referencing a root option from the set of all options. After the definition of the agents and the included option prototypes, 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.

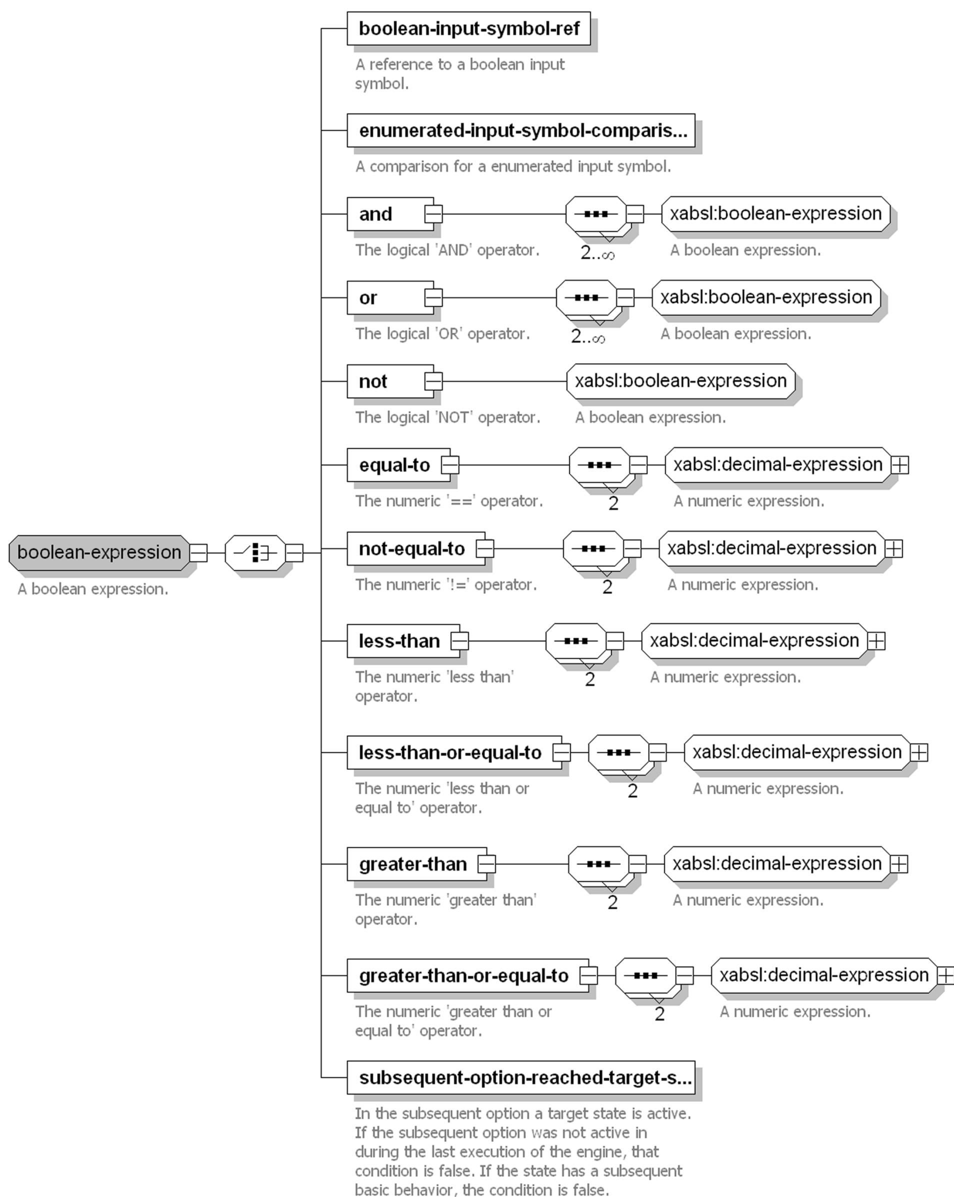


Figure D.10: The syntax of the group *“boolean-expression”*. Elements from this group are used inside conditions of decision trees.

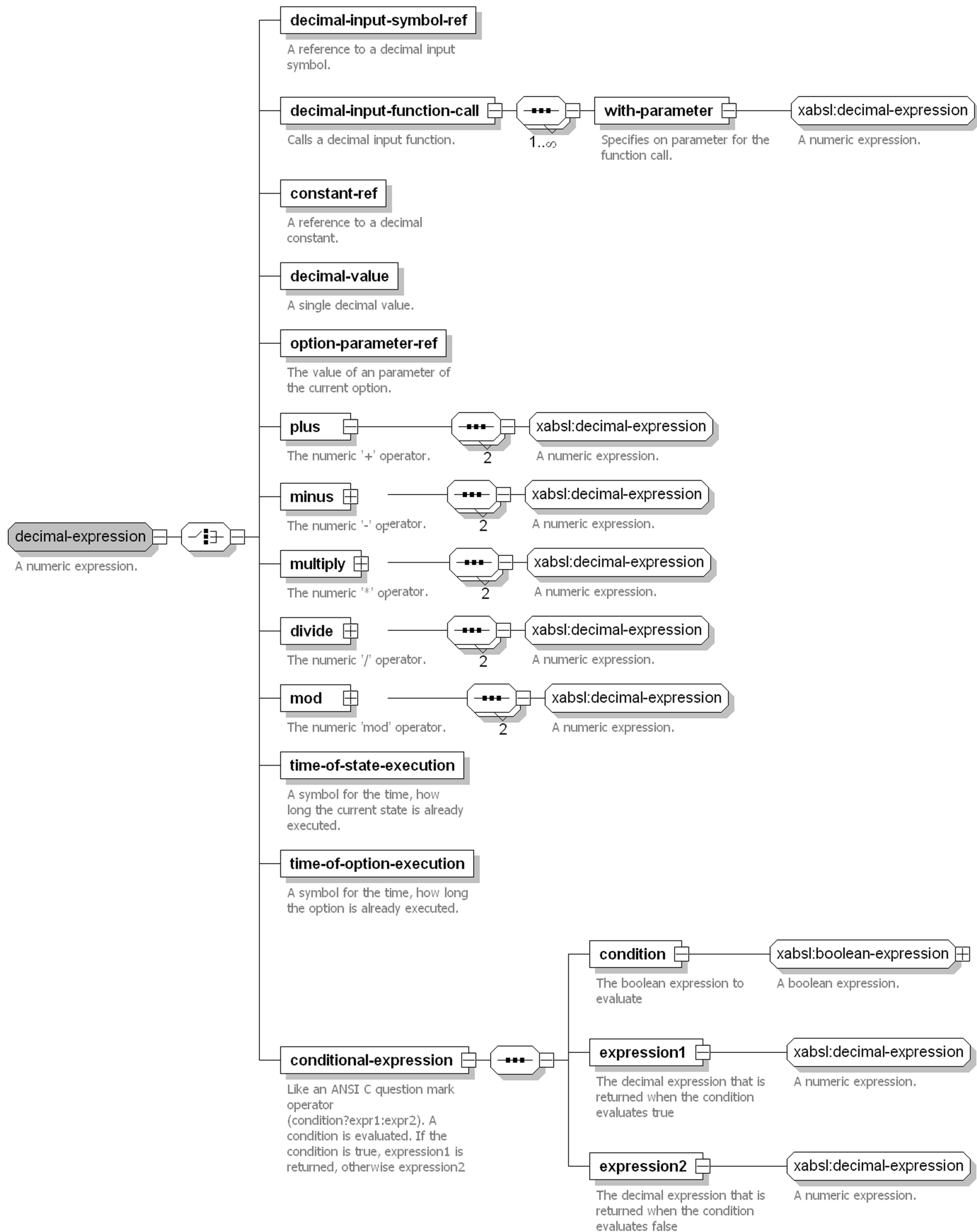


Figure D.11: The syntax of the group *"decimal-expression"*.

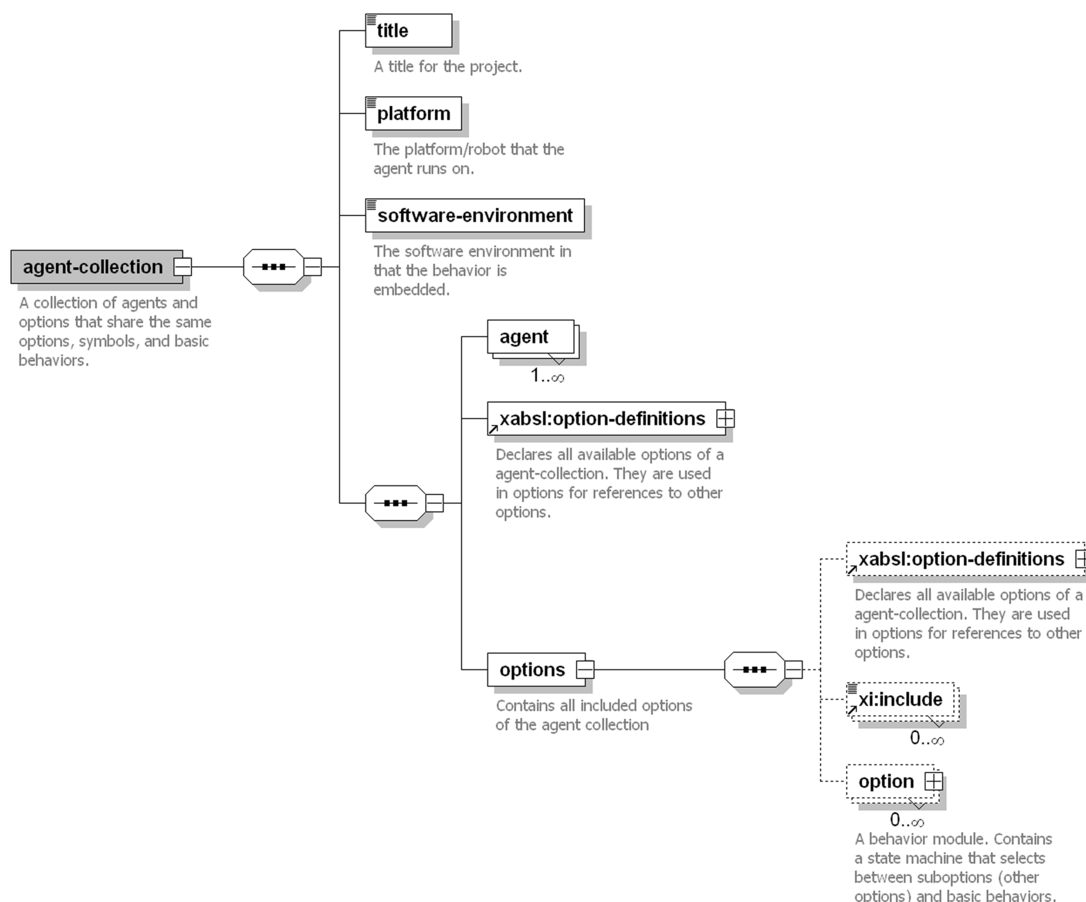


Figure D.12: The syntax of the element *"agent-collection"*.

D.4 Mechanisms and Tools

XABSL is an *XML 1.0* [7] dialect that is specified in *XML Schema* [20]. Schemas are used instead of DTDs as only they allow to specify complex identity constraints. For instance, for all decimal input symbols there is a *key* defined which guarantees that the names of the symbols are unique. If inside an option such a decimal input symbol is referenced, a *key reference* assures that the referenced symbol exists in the key.

An *XABSL* agent behavior specification is distributed over many files, which helps the behavior developers to keep an overview over larger agents and to work in parallel. The XML schemas for all the different file types can be found at the *XABSL* web site [38].

D.4.1 File Types and Inclusions

Figure D.13 shows the different file types that are part of an *XABSL* agent behavior specification. Symbol files contain the definitions of symbols, basic behavior files prototypes for basic behaviors and their parameters, and option files contain a single option. The file "options.xml" defines

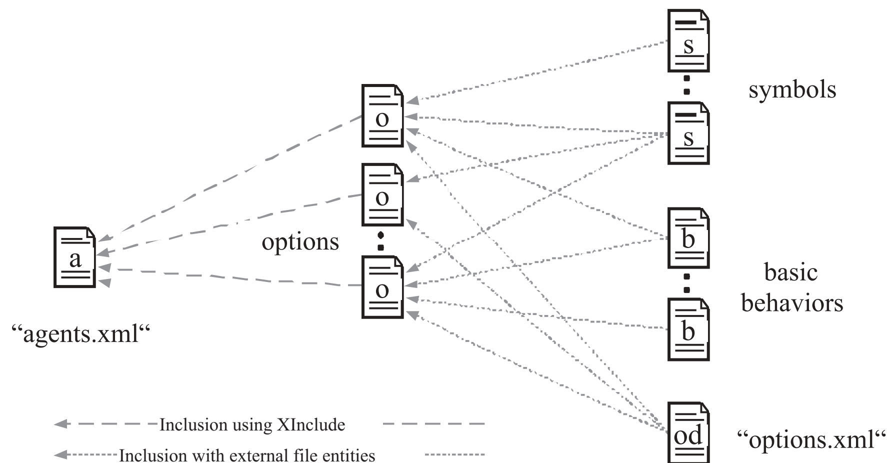


Figure D.13: Different file types of an *XABSL* specification and include mechanisms.

prototypes for each option and its parameters. The file "agents.xml" includes all the option files and defines the agents and their root options.

Two mechanisms for including one XML file into another are used. When using *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 inserted by for instance *&mySymbols;*. Most XML editors support this mechanism. It allows checking the validity of an option inside the XML editor. The disadvantage is that no cascading inclusions are possible.

With *XInclude* [42] a file is directly included into another one with a statement such as this: *<xinclude href="another-file.xml"/>*. An XInclude processor later resolves these includes for further processing. The disadvantage is that most XML editors do not resolve XInclude statements for validation.

D.4.2 Document Processing

Standard XSLT [15] transformations are used to generate three types of documents from *XABSL* source documents: an *intermediate code* which is executed by the *XabslEngine*, *debug symbols* containing the names of all named elements, and an extensive HTML-documentation containing SVG-charts for each agent, option, and state.

The run-time system *XabslEngine* uses an intermediate code instead of parsing the *XABSL* XML files directly, thus no XML parser is needed. (On many embedded computing platforms XML parsers are not available due to resource and portability constraints.)

The generated debug symbols contain the names of all options, basic behaviors, parameters, and symbols. They make it possible to implement platform and application dependent debugging tools for monitoring option and state activations as well as input and output symbols. For instance, the *Xabsl2 Behavior Tester Dialog* (cf. fig. D.16) was integrated into the *RobotControl* application, the general debug tool of the *GermanTeam*.

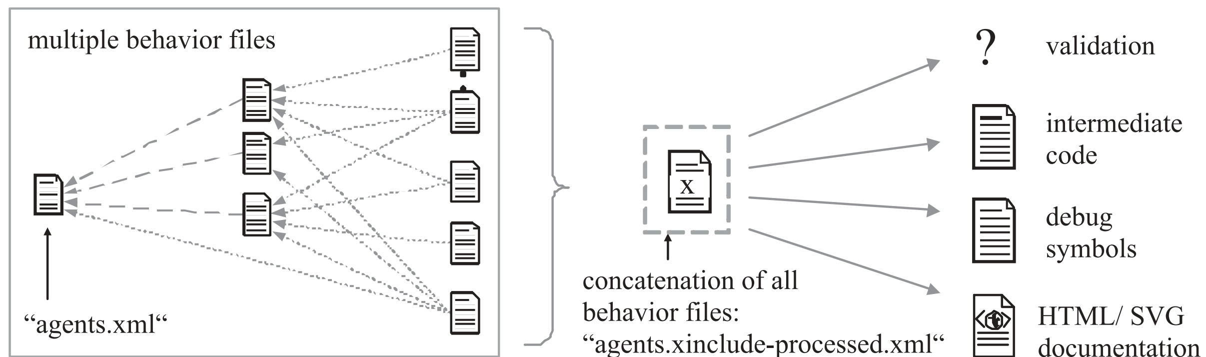


Figure D.14: Document generation in *XABSL*

The HTML documentation helps the developers to understand what their behaviors do. Almost all information specified in the XML files is clearly visualized; there are SVG charts for each option graph, state machine, and decision tree. As it would have been nearly impossible to generate these charts directly with native XSLT transformations (it is very difficult to place nodes and edges such that there is little overlapping), the “dot” tool of the AT&T Graphviz [22, 3] graph drawing suite was used. This program takes structural descriptions of the graphs as input and renders charts from it, ensuring a good layout and little overlappings between objects. As an XML wrapper for the input language of the “dot” tool, the *Dot Markup Language* (DotML) [37] was developed. Note that the figures D.1, D.2, and D.3 were generated automatically from *XABSL* documents with DotML and “dot”.

Figure D.14 shows how all the different documents are generated. Because an *XABSL* agent behavior specification is distributed over many XML files, firstly, 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 XSLT style sheet “generate-intermediate-code.xsl” is applied to “agents.xinclude-processed.xml” to generate the intermediate code. The debug symbols are created with “generate-debug-symbols.xsl”. Similar to the *XABSL* behaviors, the generated documentation is also distributed over many files. To increase the compile speed, only for the changed *XABSL* source files the documentation pages are rebuilt. Therefore, 13 different XSLT style sheets exist for the documentation generation.

For the correct call of all the different XSLT style sheets and DotML scripts, a complex Makefile was developed, which is described in detail on the *XABSL* web site [38].

D.5 The XabslEngine Class Library

The *Xabsl2Engine* is the *XABSL* runtime system. It is written in plain ANSI C++ [19] and does not use any extensions such as the STL [44]. It is platform and application independent and can easily be 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 to be registered at the engine. The engine provides extensive debugging interfaces for monitoring the activation of options and states, the values of the symbols, and the parameters of options and basic behaviors. Instead of executing the engine from the root option, single options and basic behaviors can be tested separately.

A complete API documentation of the class library is available at the *XABSL* web site [38].

D.5.1 Running the Xabsl2Engine on a Specific Target Platform

As the class library is application and platform independent, message and error handling functions as well as file access routines have to be implemented externally.

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. The engine uses that class to state errors and to raise error messages. The Boolean variable "*errorsOccurred*" can be used to determine whether errors occurred during the creation or execution of the engine.

Afterwards, a class that gives the engine read access to the intermediate code has to be derived from *Xabsl2InputSource*. The code does not inevitably have to be read from a file, but can also be read from a memory region or any other stream. The pure virtual functions *open()*, *close()*, *readValue()*, and *readString()* have to be implemented.

The intermediate code contains comments (//...) that have to be skipped by the read functions:

```
// 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 "*6 0 52.5 1 13*" has to be read from the file.

Finally, a static function that returns the system time in milliseconds has to be defined, e.g.: *static unsigned long getSystemTime()* .

D.5.2 Registering Symbols and Basic Behaviors

After creating an instance of the *Xabsl2Engine* by passing a reference to an error handler derivate and a pointer to the time function as parameters, all the symbols and basic behaviors can be registered at the engine. Note that this has to be done before the option graph is created.

As the behaviors written in *XABSL* use symbols to interact with the software environment of the agent system, for each of these symbols the corresponding variable or function has to be registered to the engine. The following example binds the variable *aDoubleVariable* to the symbol "*a-decimal-symbol*" which was defined in the *XABSL* agent behavior specification:

```
pMyEngine->registerDecimalInputSymbol("a-decimal-symbol",
                                      &aDoubleVariable);
```

If the value of the symbol is not represented by a variable but by a function, this function has to be registered at the engine. Moreover, 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 all other symbol types works in a similar way.

All basic behaviors are 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. Furthermore, the parameters of the basic behavior have to be declared as members of the class and have to be registered using *registerParameter(..)*. Afterwards, an instance has to be registered to the engine with the *registerBasicBehavior(..)* function for each basic behavior class.

D.5.3 Creating the Option Graph and Executing the Engine

After the registration of all symbols and basic behaviors, the intermediate code can be parsed using the *createOptionGraph(..)* function.

If the engine detects an error during the execution of the option graph, the error handler is invoked. This can happen if the intermediate code contains a symbol or a basic behavior that was not registered before. By using the *Xabsl2ErrorHandler* member variable *errorsOccured*, it can be checked whether the option graph was created successfully or not.

If no errors occurred during the creation, the engine can be executed with *execute()*. This function executes the option graph once at a time. Starting from the selected root option, the state machine of each option is carried out to determine the next active state. After that, the state machine for the subsequent option of this state is carried out again and again until the subsequent

Xabsl2Profiler

Load Log | Search | 1290

1290 5 (40ms)	turn-and-release grab	grab-ball-with-head approach-ball	approach-ball search-for-ball	approach-ball-set-w slow
1285 5 (40ms)	approach-and-turn approach-ball		approach-ball search-for-ball	approach-ball-set-w slow
1280 5 (40ms)	approach-and-turn approach-ball		approach-ball ball-just-found	approach-ball-set-w slow
1270 10 (80ms)	approach-and-turn approach-ball		approach-ball ball-just-found	approach-ball-set-w slow
1265 5 (40ms)	approach-and-turn approach-ball		approach-ball ball-just-found	approach-ball-set-w slow
1255 10 (80ms)	approach-and-turn approach-ball		approach-ball ball-just-found	approach-ball-set-w slow
1245 10 (80ms)	approach-and-turn go-on			
1240 5 (40ms)	approach-and-turn approach-ball		approach-ball search-for-ball	approach-ball-set-w slow
1230 10 (80ms)	approach-and-turn approach-ball		approach-ball search-for-ball	approach-ball-set-w fast

Figure D.15: The *Xabsl2 Profiler* allows to analyze changes in the behaviors over time. Each line reports a change in the state of the *XABSL* system. In the left column, a timestamp and the number of frames with no change of state is displayed. The other columns show the corresponding option and state activations on "levels" of the option graph (each option was automatically assigned to such a level for better visualization). A red cell indicates that another option was activated on a certain level, yellow stands for a state change, and green means that the parameters of a subsequent behavior changed.

behavior is a basic behavior, which is executed then, too. Finally, 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 in the beginning is the root option of the first agent. The agent can be switched using the function *setSelectedAgent(..)*.

D.5.4 Debugging Interfaces

The engine provides rich debugging interfaces that can be used to develop monitoring and debugging tools.

Instead of executing the option graph with *execute()*, single basic behaviors or options can be parameterized and executed separately. 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. For tracing the values of symbols, the engine provides access to the symbols stored. 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 applied to the software environment by using the function *setOutputSymbols()*.

Based on that interface, two debug tools were integrated into the *RobotControl* [50] application, the general debug tool of the *GermanTeam*. First, the *Xabsl Behavior Tester* (cf. fig. D.16) allows tracing the option activation path, the parameters and execution times of options, states,

Xabsl2 monitor and tester
playing-striker Send headcontrol phys. robot
(option selected - no parameters available)
ball.handling unchanged

Agent:	GT2004 - soccer		
Option Activation Path:			
playing-striker	320.9 s	handle-ball	142.7 s
handle-ball	142.7 s	ball-in-center-of-field	142.7 s
handle-ball-in-center-	142.7 s	turn-and-release-and	6.1 s
turn-and-release-and	6.1 s	turn-and-release	6.1 s
turn-and-release-and-kick.angle		170.00	
turn-and-release-and-kick.table-id		0.00	
turn-and-release	6.1 s	grab	6.1 s
turn-and-release.angle		170.00	
grab-ball-with-head	6.1 s	approach-ball	6.1 s
approach-ball	113.4 s	search-for-ball	0.0 s
approach-ball.look-at-ball-distance		500.00	
approach-ball.slow-down-distance		350.00	
approach-ball.slow-speed		170.00	
approach-ball.y-offset		0.00	
approach-ball-set-ws	1.8 s	fast	1.8 s
approach-ball-set-walk-speed.slow-dow		350.00	
approach-ball-set-walk-speed.slow-spe		170.00	
approach-ball-set-walk-speed.y-offset		0.00	
Active Basic Behavior:		go-to-ball	
go-to-ball.distance		0.00	
go-to-ball.max-speed		350.00	
go-to-ball.max-speed.y		350.00	
go-to-ball.max-turn-speed		0.00	
go-to-ball.target-angle-to-ball		0.00	
go-to-ball.walk-type		0.00	
go-to-ball.y-offset		0.00	
Generated Action:		walk : normal,347.9 -3.5 -0.0	
output symbols:			
head-control-mode		search-for-ball	
input symbols:			
ball.seen.distance		474.96	
ball.seen.angle		-1.07	
obstacles.robot-is-stuck		false	

Figure D.16: The *Xabsl2 Behavior Tester*, a part of the *RobotControl* application, makes use of the debugging interfaces of the *XabslEngine*.

and basic behaviors, as well as the values of input and output symbols. Into the other direction, single options or basic behaviors can be selected and parameterized manually for execution.

Second, the *Xabsl Profiler* (cf. fig. D.15) can be used to analyze behaviors over time. For that, log files containing the option activation path are recorded and visualized such that it can be seen for how long states and options were active. This helps to detect state oscillations or unused states.

D.6 Discussion

The *XABSL* system is a tool that can be used for decision making in autonomous agents. Because the language has no elements that are specific for a certain agent system and due to the independence of the run-time system *XabslEngine* from specific software platforms, *XABSL* can be applied in very different agent architectures and platforms.

That is why it depends on the chosen agent architecture and the implemented behaviors whether an *XABSL* agent behavior specification is reactive or deliberative. If the criterion for that distinction is that the environment is represented and modeled in persistent states, integrating past information, then it depends on whether either the agent system directly passes the sensor readings to the *XABSL* behaviors or a world model is built up and made available. But as the state based approach tends to continue once selected behaviors, there are persistent states of intention. If seen from that perspective, *XABSL* is clearly deliberatively.

In the taxonomy of Russel and Norvig [54], *XABSL* agents are *goal based agents*, although there are no explicit goals. But implicitly the implemented behaviors (options) have goals, which are decomposed into sub-goals (subsequent options). Previous goals and intentions (option and state activations) are kept.

The architecture is hierarchical, as complex behaviors are composed from simpler ones. But it is not layered, because although more long-term and deliberative behaviors reside in higher levels of the hierarchy and more low-level and reactive behavior on lower levels, there is no conceptual differentiation between different levels of the option graph.

XABSL does not contain a classical planning component in the meaning that plans are derived automatically from the current world model or future simulations, but it is possible to add such mechanisms to the agent system and to make the results available to the *XABSL* behaviors through input symbols.

The *XABSL* architecture is behavior-based [2] as high-level behaviors are constructed from a set of reactive basic behaviors. Thereby due to the use of finite state machines always only one basic behavior is selected at the same time. But nevertheless, it is possible to combine different behaviors *continuously* [1] inside the basic behaviors, for instance by using potential fields.

The system is used inside of existing agent architectures for decision making. *XABSL* can neither be used to model a complete agent system nor is it able to control the complete agent program (instead, it is frequently called from the agent program). This is in contrast to many other languages such as the *Behavior Language* [8], *COLBERT* [32], or *CDL/ MissionLab* [41], which model complete agents including sensory and motor control capabilities.

Additionally and also in contrast to these systems, *XABSL* does not translate the behavior specifications into the code of the native programming languages (such as C++) but directly interprets an intermediate code. Thus, it is not necessary to recompile the programs if the behaviors change, leading to a shorter change-compile-test cycle.

The language can be best compared with the *Configuration Description Language* (*CDL*), a part of the *MissionLab* system. As CDL, *XABSL* allows to completely specify agent behavior based on hierarchies of finite state machines. But *XABSL* has a higher expressiveness in conditions for state transitions so that CDL documents could be transformed into *XABSL* documents, but not vice versa.

D.7 YABSL

In the past the behavior of the GermanTeam was written in *XABSL*. As mentioned before *XABSL* has a lot of advantages over just using C++ or plain C for implementing state machines for the behavior. Despite of its advantages there are also some major disadvantages caused by the use of XML, like generating large and complex code. To get rid of them *YABSL* was introduced this year to make writing behaviors more easy.

The idea behind *YABSL* was not to replace the whole architecture provided by *XABSL* but to ease the writing of behavior code. To reach this goal *YABSL* provides a syntax similar to C++. This makes the code more “human readable” as it is much shorter and easier to survey. Despite of the “human readable” code the architecture used by the robots remained unchanged by this process because there should always be the possibility of using *XABSL* or *YABSL* as one desires. To provide this flexibility a new compiler has been implemented using Ruby which is able to transform *XABSL* code to *YABSL* and vice versa. This makes it possible to use the features provided by YABSL while still using the old tools for example for building the documentation.

The main advantage from *YABSL* over *XABSL* is the shortness of the source code. The similarity to C++ makes it easy to read and understand the code. As an example here is a snippet of the same code first in *XABSL* and then in *YABSL*.

approach-and-kick written in *XABSL*.

```
<condition description="kick possible">
  <and>
    <boolean-input-symbol-ref ref="ball.just-seen"/>
    <less-than>
      <decimal-input-function-call ref="abs">
        <with-parameter ref="abs.value">
          <decimal-input-symbol-ref ref="ball.seen.relative-speed.y"/>
        </with-parameter>
      </decimal-input-function-call>
      <decimal-value value="150"/>
    </less-than>
    <less-than>
      <decimal-input-symbol-ref ref="ball.seen.relative-speed.x"/>
      <decimal-value value="150"/>
    </less-than>
    <not-equal-to>
      <decimal-input-function-call ref="retrieve-kick">
        <with-parameter ref="retrieve-kick.angle">
          <option-parameter-ref ref="approach-and-kick.angle"/>
        </with-parameter>
        <with-parameter ref="retrieve-kick.table-id">
          <option-parameter-ref ref="approach-and-kick.table-id"/>
        </with-parameter>
      </decimal-input-function-call>
```

```
      <constant-ref ref="action.nothing"/>
    </not-equal-to>
  </and>
</condition>
```

approach-and-kick written in *YABSL*.

```
/** kick possible */
if (ball.just_seen && abs(value = ball.seen.relative_speed.y) < 150 &&
    ball.seen.relative_speed.x < 150 &&
    retrieve_kick(angle = @angle, table_id = @table_id) != action.nothing)
```

As one can see the code written in *YABSL* is much more concise and easier to understand than the *XABSL* part. Both parts do exactly the same. With the use of YABSL the size of the source has been reduced by about 50 %. Another advantage from using *YABSL* instead of *XABSL* is the incremental compile ability of the compiler which speeds up the compilation and linking process enormously.

As *XABSL* was mostly written in an XML editor which provides syntax highlighting and auto completion of the code, *YABSL* should also provide this features. This was achieved by developing a Visual Studio plugin for auto completion and syntax highlighting of *YABSL* code. Next to this features the *YABSL* files are now also part of the Behavior project which is generated each time the behavior is compiled. This makes it even easier to edit the behavior specific files directly in Visual Studio.

Appendix E

Processes, Senders, and Receivers

E.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 ability of Aperios to exchange data between processes via the wireless network by using the TCPGateway. 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.

E.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 *GT2005\Src\Processes*. Process layouts are named after the processes that exist in them, and in fact, since 2003 there was only one layout, namely *CMD* that consists of the processes *Cog-*

nit(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 *GT2005**Config**Processes*\processLayout*object.cfg*, following the format */MS/OPEN-R/APP/OBJS/*shortName.*bin*,
- insert a line in *GT2005**Config**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 *GT2005**Src**Processes*\processLayout with the full name,
- and, append the last line of all *depend_for_* files in *GT2005**Make**dsp_generation* by an entry following the pattern *shortName=filename* and run *makecmd.cmd* in the same directory. The command will recreate all project files to include the new process.

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.

E.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 F). 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()*.

E.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. Please note that all classes that are derived from *class Streamable* automatically implement these operators.

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;
}
```

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

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

E.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!

E.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 F

Streams

F.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. It is also possible to derive classes from *Streamable* and implement the mandatory function *serialize(In*, Out*)*. The concepts in iostreams where extended by mechanisms to gather information on the structure of data while serializing it.

There are reasons not to use the C++ iostreams library. The C++ iostreams library 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. The iostreams library is not fully implemented on all platforms, namely not on Aperios. Another reason not to use the C++ iostreams library is that the protocol of the streamed data is explicit only in the streaming operators, rendering the easy serialization and deserialization of data impossible in languages other than C++.

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.

F.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
- **OutTextRaw.** Formats data as raw text (same output as "cout")

Out. Abstract class

- **OutStream<PhysicalOutStream,StreamWriter>.** Abstract template class
  - **OutBinaryFile.** Writing into binary files
  - **OutTextFile.** Writing into text files
  - **OutTextRawFile.** Writing into raw text files
  - **OutBinaryMemory.** Writing binary into memory
  - **OutTextMemory.** Writing into memory as text
  - **OutTextRawMemory.** Writing into memory as raw text
  - **OutBinarySize.** Determine memory size for binary storage
  - **OutTextSize.** Determine memory size for text storage
  - **OutTextRawSize.** Determine memory size for raw text storage
  - **OutBinaryMessage.** Writing binary into a MessageQueue
  - **OutTextMessage.** Writing into a MessageQueue as text
  - **OutTextRawMessage.** Writing into a MessageQueue as raw 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

F.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. *GT2004\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.

For writing to text streams without separation of entries and space escaping, for example *OutTextRawFile* can be used instead of *OutTextFile*. It formats the data such as known from the ANSI C++ *cout* stream. The example above is formated as following:

```
13.14000Hello Dolly
42
```

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);
```

F.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. To make a class streamable

it is furthermore possible to derive the class from *Streamable* and implement the mandatory function *serialize(In*,Out*)*.

The protocol of streamed data lies in the streaming operators and serialize functions. This made the implementation of non C++ debugging tools impossible. Therefor the streaming mechanism was extended by techniques to gather the structure of streamed data.

F.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;
  }
}
```

F.4.2 Streamable

Another way of making a class Streamable is to derive the class from *Streamable* and implement the abstract function *serialize(In*,Out*)*. For data types derived from *Streamable* streaming operators are provided, meaning they may be used as any other data type with standard streaming operators implemented.

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

class Sample : public Streamable
{
  public:
    virtual void serialize(In* in, Out* out)
    {
        if( in != null)
        {
            *in >> a >> b >> c >> d;
        }
        else
        {
            *out << a << b << c << d;
        }
    }
    int a,b,c,d;
};
```

F.4.3 Streaming Protocols

Since the actual layout, also called protocol, of the serialized data is explicitly modeled in the streaming operators and serialize functions to be able to deserialize serialized data means to have knowledge about the structure of data. To allow non C++ applications to make use of serialized data, the process of streaming data was extended by acquiring additional structural information about the data streamed. This includes the data types of streamed data as well as names or id´s which follow from the structure of classes. The process of acquiring name and type of members of data types is automated.

The acquisition of the protocol of streamed data types is eased by a set of macros. There are macros for streaming operators and the serialize function respectively.

- `STREAM_REGISTER_BEGIN();` indicates the start of a streaming operation, as well as the start of a member and structure acquisition by the *Stream Handler*.
- `STREAM_BASE( s);` streams the base class.
- `STREAM( s);` streams the member s, retrieving its name in the process.

- STREAM_ENUM(s, numberOfEnumElements, getNameFunctionPtr); streams the member s of type enum, retrieving its name in the process, as well as the name and primitive of all possible values.
- STREAM_ARRAY(s); streams a static array s.
- STREAM_DYN_ARRAY(s, count); streams a dynamic array s with length count.
- STREAM_REGISTER_FINISH() indicates the end of the streaming operation for this data type.

These macros can be used in the streaming operators. Which can for instance look as follows:

```
virtual void serialize( In* in, Out* out)
{
    STREAM_REGISTER_BEGIN();

    STREAM(test);

    STREAM_DYNAMIC_ARRAY( testArray, 10);

    STREAM_REGISTER_FINISH();
}
```

which streams the member `test` and a dynamic array called `testArray`.

For the use in streaming operators the set of macros is duplicated and extended by macros which also have a parameter for the containing data type.

A process wide instance of the *StreamHandler* gathers the structural information of data types, including names and types of member, in a data structure. The data held by the *StreamHandler* is streamable.

F.4.4 Streaming using *read()* and *write()*

There also is another 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. The approach also breaks virtual functions, rendering its use with data types derived from Streamable dangerous, to say the least. 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.

F.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 G

Debugging Mechanisms

Debugging mechanisms are an integral part of the *GermanTeam*'s system architecture. This chapter describes the basic structures and components, mainly for transmitting messages.

G.1 Exchanging Messages Between Robots and PC

Besides the package-oriented inter-object communication with senders and receivers, *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. *Debug keys* are used to request certain messages.

G.1.1 Message Queues

The class `MessageQueue`[1] allows to store and transmit a sequence of type safe and time-stamped messages. On the Windows platform, it is implemented as a dynamic list[2]. In Open-R, where dynamic memory allocations are expensive, a static memory area is used, whose size must be defined in advance. These two different methods are implemented in the platform dependent class `MessageQueueBase`[3], which is used by `MessageQueue` to store messages.

With `myQueue.setSize(size)` the maximum size of a message queue (all messages + overhead) is defined. On the Open-R platform, a memory area of that size is allocated once. If it is full, further messages are discarded without notification. On all other platforms, `setSize()` is ignored.

As almost all data types have streaming operators, it is very easy to store them in message queues. Class `MessageQueue` provides different write streams for different formats: Mes-

[1]Definition and implementation in *Src/Tools/MessageQueue.h* and *.cpp* .

[2]Based on class `List`, definition and implementation in *Src/Tools/List.h* and *.cpp* .

[3]Included platform independent via *Src/Platform/MessageQueueBase.h*, definition and implementation for Aperios/Open-R in *Src/Platform/Aperios1.3.2/MessageQueueBase.h* and *.cpp* , for Windows and Linux in *Src/Platform/Win32Linux/MessageQueueBase.h* and *.cpp* .

sages that are stored through `out.bin` are formatted binary. The stream `out.text` formats data as text and `out.textRaw` as raw text.

After that all data of a message was written to the queue, it must be finished with `out.finishMessage(id)`. The ID[4] specifies the type of the message and is used to distribute the message later on. Additionally, `MessageQueue` automatically stores the system time when the message was written, the team color and robot number of the sending robot, and a flag indicating whether the message was sent from a physical or a simulated robot.

Some examples for writing into a `MessageQueue`:

```
Image myImage;
myMessageQueue.out.bin << myImage;
myMessageQueue.out.finishMessage(idImage);

int i = 3;
myMessageQueue.out.text << "a b" << i << " c"
MyMessageQueue.out.finishMessage(idText);

myMessageQueue.out.textRaw << "a b" << i << " c"
MyMessageQueue.out.finishMessage(idText);

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

In the first example an image is streamed in binary format to the message queue `myMessageQueue`. The type of the message is `idImage`. Then a simple text message (format `text`, ID `idText`) is written. The result is:

```
a\ b 3 \ c
```

The third example is the same as the one before except that the stream `out.textRaw` is used:

```
a b3 c
```

In the fourth example, four integer numbers are written binary with the ID `id4FunnyNumbers`.

The following functions transmit data between different message queues: `copyAllMessages(otherQueue)` copies all messages into another queue and

[4] All IDs are defined in *Src/Tools/MessageQueue/MessageIDs.h* .

`moveAllMessages(otherQueue)` moves all messages there. With `clear()` all messages are deleted.

Message queues are exchanged between processes such as all other packages by streaming them through the debug senders and receivers. These streaming mechanisms can also be used to stream a message queue into a file or to transmit debug data via the Wireless Network. Note that log files are simply streamed-into-file message queues.

To declare a new message type, an ID for the message must be added to the enum `messageID`[5]. Additionally, a string for the type is added to function `getMessageIDName(MessageID id)`[6]. Note that new message IDs have to be added at the end of enum `messageIDs` as otherwise old logfiles would not work anymore.

G.1.2 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. For each message, with `myQueue.handleAllMessages(handler)` a *message handler* is invoked.

All message handling classes are derived from class `MessageHandler`[7] and have overwritten the virtual function `handleMessage(InMessage& message)`:

```
class MyMessageHandler : public MessageHandler
{
  virtual bool handleMessage(InMessage& message)
  {
    switch (message.getMessageID())
    {
    case idImage:
      message.bin >> myImage;
      return true;
    case idXY:
      message >> otherQueue;
      return true;
    case idZ:
      return otherMessageHandler.handleMessage(message);
    default:
      return false;
    }
  }
};
```

[5]In *Src/Tools/MessageQueue/MessageIDs.h* .
[6]In same file.
[7]Definition in *Src/Tools/MessageQueue/InMessage.h* .

Messages are read from a MessageQueue via streams. Thereto, message.bin provides a binary formatted stream, message.text a text stream and message.config a text stream that skips comments. In the example above, all messages of the type idImage are streamed binary formatted into the local variable myImage. With message >> otherQueue a message is copied completely into another MessageQueue. With otherMessageHandler.handleMessage(message) another class derived from MessageHandler is called to handle the message. To read a message a second time, the read position of the stream is reset before with message.resetReadPotition().

Function handleMessage(..) has to return whether the message was handled. An interface to the message is given in parameter message[8]. The ID of the message can be queried with message.getMessageID() and message.getTimeStamp() returns the time when the message was put into a queue[9]. The team color of the sending robot is returned by message.getTeamColor() and the robot number by message.getPlayerNumber(). Finally, the function getMessageWasSentFromAPhysicalRobot() determines whether the message was sent from a physical or simulated robot.

G.1.3 Sending Messages

To simplify the access to outgoing message queues a macro is defined[10]:

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

For example

```
OUTPUT(idText, text, "Could not load file " << filename);
```

outputs an text message.

To safe processing time, the macros are ignored in the *Release* configuration.

G.2 Message Queues and Processes

Each process[11] has two message queues: debugOut for outgoing and debugIn for incoming messges. Debug messages are transmitted between processes such as normal packages.

[8]Instance of class InMessage, definition and implementation in *Src/Tools/MessageQueue/InMessage.h* and *.cpp* .

[9]Copying messages between queues does not change the time.

[10]In */Src/Tools/Debugging/Debugging.h* .

[11]Class Process, definition and implementation in *Src/Tools/Process.h* and *.cpp* .

Thereto, with the macro `DEBUGGING`[12] the receiver `theDebugReceiver` for and the sender `theDebugSender` are defined for `debugIn` and `debugOut`.

The static function `getDebugOut()`[13] is used by the debugging macros to access `debugOut` from an arbitrary position in the source code.

G.2.1 Message Handling

Before the execution of `Process::main()`, the `handleMessage(..) of each process is called for every message in debugIn`. For example in the `Cognition` process incoming messages are distributed such:

```
bool Cognition::handleMessage(InMessage& message)
{
  switch (message.getMessageID())
  {
  case idSensorData:
    message.bin >> theSensorDataBufferReceiver;
    processSensorData = true;
    return true;

  case idLinesSelfLocatorParameters:
    pSelfLocator->handleMessage(message);
    return true;

  ...
  default:
    return Process::handleMessage(message);
  }
}
```

Some messages are directly streamed into representations. For example `idSensorData` messages are written into the variable `theSensorDataBufferReceiver`. Other messages are handled by modules and their solutions respectively. For example messages with the ID `idLinesSelfLocatorParameters` are passed to the `handleMessage(..)` function of the `SelfLocatorSelector` (cf. sect. H.2.4), which calls the `handleMessage(..)` method of the currently selected solution.

All messages that were not handled in `Cognition::handleMessage(..)` are passed to `Process::handleMessage(..)`. This function processes common messages such as `idDebugKeyTable` and `idSolutionRequest`. For the remaining messages, all modules are queried automatically whether they want to handle the message. If not, an error message is displayed.

[12]Definition in *Src/Tools/Process.h* .

[13]Definition and implementation in *Src/Tools/Debugging/Debugging.h* and *.cpp* .

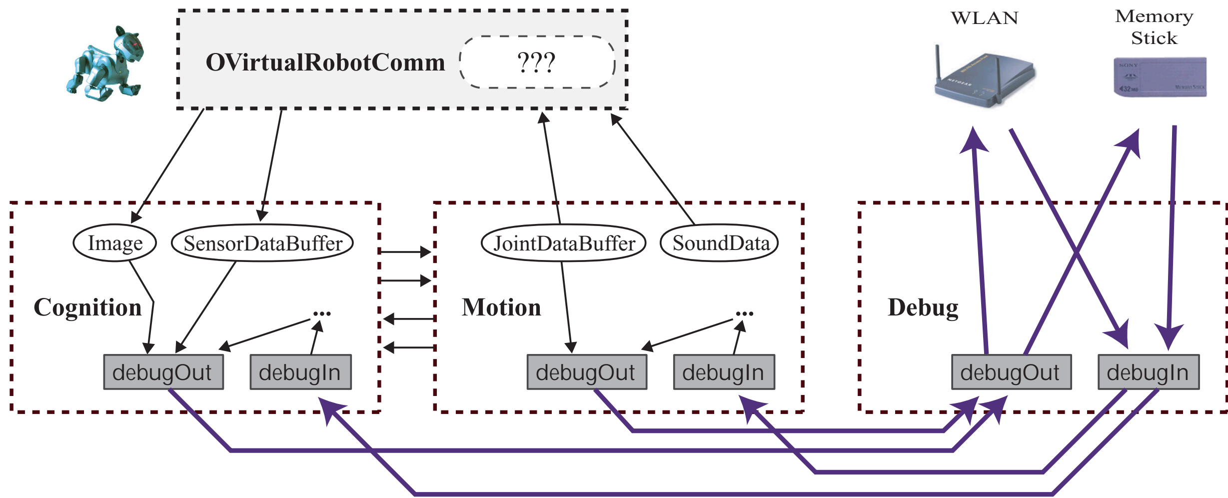


Figure G.1: Data flow between processes in the CMD process layout.

G.2.2 The Process Debug

The process `Debug`[14] manages the communication of the robot programs with the tools on the PC. For each of the other processes (in the CMD layout: `Cognition` and `Motion`) it has a sender and receiver for message exchange (cf. fig. G.1).

Messages that arrive via the WLAN from the PC are stored in `debugIn`. Additionally it is also possible to transmit messages to a physical robot with the MemoryStick. Thereto, one writes the content of the outgoing message queue of into the file *requests.dat* on the Memory-Stick. The `Debug` process reads this file at startup and puts all the containing messages into `debugIn`. With this mechanism, it is possible to transmit data and requests to physical robots without a working WLAN connection. Function `Debug::handleMessage(..)` distributes all messages in `debugIn` to the other processes.

All messages from `Cognition` and `Motion` are stored in `debugOut`. If a WLAN connection was established, they are sent from there to the PC. To avoid communication jams, it is possible to send a `QueueFillRequest`[15] to the `Debug` process. It specifies how to process messages in `debugOut` :

- `immediateReadWrite` (send everything): All messages in the queue `debugOut` are transmitted. Note that this can result in jams.
- `overwriteOlder` (real-time mode): For each message type, only the newest message is transmitted. This prevents the queue from getting bigger and bigger if the WLAN is not fast enough. It is the standard setting in *RobotControl*.
- `rejectAll`: The queue `debugOut` is cleared.

[14]Definition and implementation in *Src/Processes/CMD/Debug.h* and *.cpp* .
[15]Definition and implementation in *Src/Tools/Debugging/QueueFillRequest.h* und *.cpp* .

- `collectNSeconds`: Breaks the WLAN traffic for n seconds. This can be used to collect data without the slowing down impact of the WLAN. After the time, all data are sent at once.
- `toStickImmediately`: All messages are appended to the file *Logfile.log* on the MemoryStick instead of transmitting them via the WLAN. Together with requests in *requests.dat* (see above), this can be used to get data from a phyical robot without a working WLAN connection.
- `toStickNSeconds`: Same as above, but the messages are written only after n seconds. As writing to a memorystick slows down the system very much, this is useful when big amounts of data (for instance images) shall be collected and written.

G.3 Common Debug Mechanisms

Based on message queues, the *GermanTeam* developed a rich variety of high-level debugging mechanisms. Three common high-level debug tools are introduced in this section. Note that the debugging techniques have undergone a complete overhaul. Compared to previous years the most important change is the use of textual, meaning string representations as identifiers for debug switches. In previous years one had to usually extend enumerations to make use of debugging techniques, meaning boring and endless compile-test-cycles. This has been overcome this year, rendering the use of the techniques easier than ever. For legacy reasons the source code release also contains the 2004 versions of some of the debugging techniques introduced here. To use the new debugging techniques the code must be compiled with `NEWDEBUGGING` defined. See earlier TeamReports for references of the old techniques.

All of the mechanisms bring C++ macros with them, which provide a consistent and easy to use interface to the mechanisms. The macros are left blank in the release, also called non debugging version of the code which is compiled to play the competition games.

G.3.1 Debug Requests

Debug Requests are used to enable and disable parts of the source code. They can be seen as a runtime switch available only in debugging mode. This can be used to trigger certain debug messages to be sent, as well as to switch on certain parts of algorithms. They are textual keys which allow a certain statement to be executed on the enabling of the textual key. An architecture was implemented to provide a seemless integration into the source code.

G.3.1.1 Macros

A number of macros ease the use of the mechanism as well as hide the implementation details.

- `DEBUG_RESPONSE( id, expression);` executes the expression if the id is enabled.

- `DEBUG_RESPONSE_NOT( id, expression);` executes the expression if the id is not enabled. Also expression is executed in the release mode of the software. This enables for switching off on request.

These macros can be used anywhere in the source code, allowing for easy debugging. For example:

```
DEBUG_RESPONSE("test", test());
```

calls the `test()` function in case the Debug Request with the string id "test" is enabled. *Debug Requests* are commonly used to send message on request, as the following example shows.

```
DEBUG_RESPONSE("send motion data",
    OUTPUT( idText, text, motionData);
);
```

sends the motion data as textual message if the *Debug Request* with the id `"send motion data"` is activated.

G.3.1.2 Debug Request Architecture

Debug Requests consists of an id. This is the name of the debug switch.

Debug Requests fulfill 2 constraints. The first being that only those Debug Requests are available which are actually executed. This is due to the modularized software architecture of the German Team. The idea is that only those *Debug Request* id´s are available, that are used in the currently switched on *Solutions*. The second constraint is that using a *Debug Request* only results in the compilation of the part of the source code the id is used in. This is achieved by using string representations as id´s. All 2 constraints make it unnecessary to keep a higher level organization for *Debug Requests* like in enumerations or global lists. If the id is not used in any of the following macros anymore, it will not be communicated as an available *Debug Request*. In fact it just does not exist anymore. This makes it very easy for debugging switches in deprecated solutions to be extinguished. Because they disappear with the corresponding solution.

There are different states to *Debug Requests*. They can be on, off, or requested a certain amount of time. Messages can be used to switch on *Debug Requests*, acquire available *Debug Requests* as well as get the status of available *Debug Requests*, i.e. are they switched on. The infrastructure provided accounts for these use cases.

The *Debug Request Table* keeps track of switched on debug requests. It keeps a list of all *Debug Requests* switched on, even if the connection to the debug tool breaks or is deliberately closed, as well as if the part of code the *Debug Request* is used in, is not executed.

G.3.1.3 Polling

The *Debug Request* with the id `"poll"` has a special meaning. *Polling* is a mechanism developed for gathering available *Debug Requests*, meaning gathering the id´s of all *Debug Requests*,

which have been used or passed by in the last frames of the process operating. When an id has not made itself known, a message will be constructed of the type `idDebugResponse` , which can be used to gather the available id´s. The logic of the polling response is hidden in the macros.

G.3.2 Debug Images

Visualization is one of the big demands on debugging techniques. Since the robots operate in a special domain and have the camera as their main sensor, it is easy to come up with visualizations requests which involve pictures taken by the camera. *Debug Images* are a concept that allows for the easy manipulation of pictures taken by the camera. This is used for low level visualization of image processing debug data. Every image has an associated textual id, which allows for referencing it in the manipulation process, as well as the packaging and sending of the *Debug Image*. The id can be used in number of macros allowing for the manipulation of image data.

G.3.2.1 Macros.

- `N_DECLARE_DEBUG_IMAGE(id);` declares a debug image with the specified id.
- `N_INIT_DEBUG_IMAGE(id, image);` initializes a *Debug Image* with the given id via copy constructor from the given image.
- `N_SEND_DEBUG_IMAGE(id);` sends *Debug Image* with the given id as bitmap.
- `N_SEND_DEBUG_IMAGE_AS_JPEG(id);` sends a *Debug Image* with the given id as jpeg encoded image.
- `N_DEBUG_IMAGE_GET_PIXEL_Y(id, x, y);` returns the Y-channel value of the given pixel (x,y) of the given image.
- `N_DEBUG_IMAGE_SET_PIXEL_YUV(id, x, y, y, u, v);` sets the Y,U and V-channel of a given pixel (x,y) of the image with the given id.
- `N_DECLARE_DEBUG_GRAY_SCALE_IMAGE(id);` declares a special image type.
- `N_SET_COLORED_PIXEL_IN_GRAY_SCALE_IMAGE(id, x, y, color);` allows for the manipulation of the special image type grayscale
- N_GENERATE_DEBUG_IMAGE(id,expression) allows for the encapsulation of debug image macros within a *Debug Request*

These macros can be used anywhere in the source code, allowing for easy creation of *Debug Images*. For example:

```
NDECLARE_DEBUG_IMAGE("test");

NINIT_DEBUG_IMAGE( "test", image);
```

```
NDEBUG_IMAGE_SET_PIXEL_YUV("test", 0, 0, 0, 0, 0);

NSEND_DEBUG_IMAGE_AS_JPEG("test");
```

initializes a *Debug Image* from another image, sets the pixel (0,0) to black and sends it afterwards as JPEG encoded image.

G.3.3 Debug Drawings

This is a mechanism allowing the developer to draw to a virtual drawing paper. The mechanism provides a 2 dimensional drawing paper and a number of drawing primitives, as well as a mechanism for request, sending and drawing of these primitives to the screen of the debug application of the developer. *Debug Drawings* consist of a textual id, as well as a number of primitives. Furthermore every id has an associated type, which enables the debug application to draw the id to the right drawing paper. Id, type and description are all string based representations. So far 2 standard drawing papers or types are provided, called `"drawingOnImage"` and `"drawingOnField"` . This refers to the 2 standard applications of Debug Drawings, namely drawing something into an image and on a 2 dimensional representation of the field. However the mechanism itself is not restricted to these 2 types, but is rather fully extensible with respect to id and type. Note however that a certain id is bound to a certain type, meaning it is only of that type. The primitives, which include such geometric figures as line, rectangle, circle as well as a representation for brushes or pens and colors are packaged in a single structure.

G.3.3.1 Macros

A number of macros ease the use of the mechanism as well as hide the implementation details.

- `NDECLARE_DEBUGDRAWING( id, type, description);` declares a *Debug Drawing* with the specified id, type and description. The type and description are bound to the id.

- `NCIRCLE(id, x, y, radius, penWidth, penStyle, penColor);` draws a circle with the specified radius, pen width, pen style and pen color at the coordinates x and y to the virtual drawing paper.

- `NDOT(id, x, y, penColor, fillColor);` draws a dot with the pen color and filling color at the coordinates x and y to the virtual drawing paper.

- `NLARGE_DOT(id, x, y, penColor, fillColor);` draws a large dot with the pen color and filling color at the coordinates x and y to the virtual drawing paper.

- `NARROW(id, x1, y1, x2, y2, penWidth, penStyle, penColor);` draws an arrow with the pen color, width and style from the point (x1, y1) to the point (x2,y2) to the virtual drawing paper.

- `NLINE(id, x1, y1, x2, y2, penWidth, penStyle, penColor);` draws a line with the pen color, width and style from the point (x1, y1) to the point (x2,y2) to the virtual drawing paper.
- `NOCTANGLE(id, x1,y1,x2,y2,x3,y3,x4,y4,x5,y5,x6,y6,x7,y7,x8,y8, color, fill);` draws an octangle.
- `RECTANGLE(id, x1,y1,x2,y2, penWidth, penStyle, penColor);` draws a rectangle.
- `NCOMPLEX_DRAWING(id,expression);` can be used to encapsulate a number of drawings with one id.

These macros can be used anywhere in the source code, allowing for easy creation of *Debug Drawings*. For example:

```
NDECLARE_DEBUG_DRAWING("test", "drawingOnField", "draws a test debug
drawing");

NRECTANGLE("test", 0,0,100,100, 10, ps_solid, black);
```

initializes a drawing called `"test"` of type `"drawingOnField"` which draws a test drawing, consisting of a rectangle.

G.3.3.2 Drawing Manager

Since there is a message sent for every primitive and every message contains the drawing id, resulting in up to 100 messages and more, the id must be mapped to a somewhat less space or memory consuming data type then string. This mapping is done by the *Drawing Manager*, which also keeps track of the id´s drawing type. The string based id´s are mapped to integers. The *Drawing Manager* sends special messages on request revealing the mapping, enabling the corresponding debug application to ”decode” the drawing id´s. The *Drawing Manager* also handles the corresponding description and sends them together with the type, so that connected debug applications are able to use type and description for their designed purpose.

G.3.4 Debug Data

Debug Data is a concept which allows for the reading and modifying of data. Every streamable data type (cf. Sect. F) is able to be manipulated and read, if its structure is gathered while streamed (cf. Sect. F.4.3). This allows for truly generic manipulation of the data in for instance generic dialogs.

For all data types, with existing streaming operators a special mechanism was developed to allow for the request, viewing and changing of data of that type. The developed approach takes care of such tedious tasks as message handling and so on. The request mechanism is based on the Debug Request mechanism. While the sending and modification is based on the implementation

of a streaming operator of the type of the data to be changed. The specification acquired by the *Stream Handler* can be used by the debug application to gain insight into structure and member names of complex data types.

To ease the use a macro was defined:

- `MODIFY( id, object)` modifies object with the given id on debug data change request.

The *Debug Data Table* handles the incoming requests to change certain data.

G.3.5 Stopwatch

Stopwatches are a mechanism to allow for the measurement of time, specifically execution time of parts of the code. The measured times are identified by textual id´s, which also serve as identification for the request of the measured execution time.

- `NSTOP_TIME_ON_REQUEST(eventID, expression)` stops the execution time of the expression on request of the eventID.

Appendix H

Mechanisms for Modules and Solutions

The members of the *GermanTeam* often work on the same problems in parallel to follow their own research interests and to compete as separate teams in national RoboCup competitions. In order to keep different approaches integrated into a common source code repository, the *module architecture* was developed.

H.1 Division of Information Processing into Tasks

In this architecture, the complete information processing of the robot programs is divided into single "tasks" such as images processing or LED control. This distribution together with well defined interfaces was defined by the *GermanTeam* in a meeting in 2001 and did not change much since then (cf. fig. H.1).

Modules. Each single task is encapsulated in a *module*. It exchanges data with other modules as well as sensors and actuators only through external *representations*. This means that if a module is executed, it reads its input from external data structures and stores the results of computation in other external representations. Modules do not call each other (and even do not "know" each other)[1].

Representations. *Representations* are data structures that are exchanged between modules. They are defined in separate classes in directory *Src/Representations/*[2].

To be able to develop different solutions for modules, these representations should be as common and general as possible. In order that developers of different modules and solutions interpret the data in the same way, they should be well documented and have clear semantics. All representations have streaming operators so that they can be easily transmitted between processes and between robot and PC.

[1]The only exception is module *MotionControl*. This internally executes one of the modules *WalkingEngine*, *GetupEngine* and *SpecialActions*.

[2]This directory is exclusively for data structures that are exchanged between modules, all others should be placed in *Src/Tools/* .

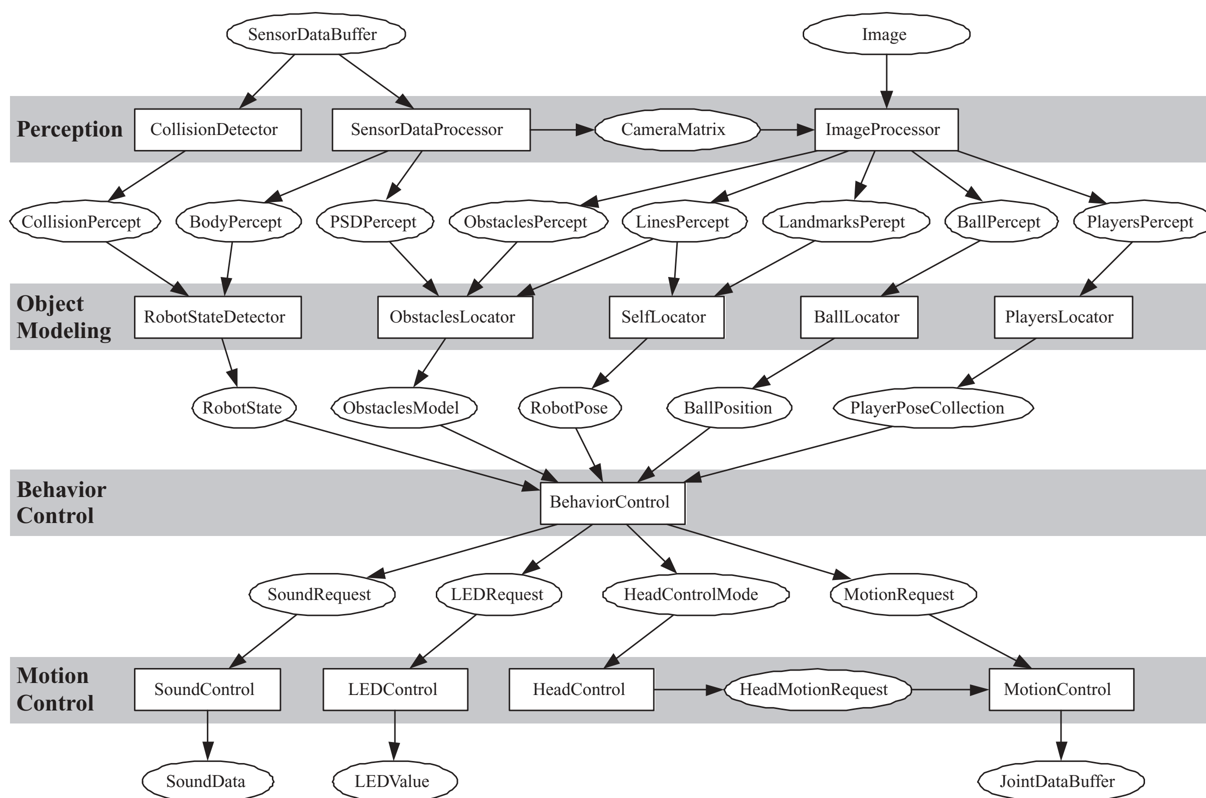


Figure H.1: Simplified graph of the *GermanTeam*'s modules (boxes) and representations (ellipses).

Special representations are those that are exchanged between the robot operating system and the control programs (images, sensor data, sound, and motor commands). They reimplement the Open-R data types and have platform dependent streaming operators in order to make the module architecture independent from any platform.

For example class *Image*[3] encapsulates the image data as they are received from the robot system. The only platform depending part of `Image` is the streaming operator that reads an image from a stream[4]. By that, the image processing module (as all other modules) runs both on physical robots and on the PC in a simulated robot.

Solutions. For each module it is defined, which representations it can access for reading and which representations it can write to. Thanks to these fixed interfaces, it is possible to develop different *solutions* for a module in parallel in the same source code basis. Solutions can be ex-

[3]Definition and implementation in *Src/Representations/Perception/Image.h* and *.cpp* .

[4]The reading streaming operator for the Open-R platform is implemented in *Src/Platform/Aperios1.3.2/Sensors.cpp*. It reads the data as they are sent from Open-R. In *Src/Platform/Win32/Sensors.cpp* the streaming operator for the Windows platform is defined. It reads the data as they were written by the `Image` writing streaming operator (implementation in *Src/Representations/Perception/Image.cpp*) .

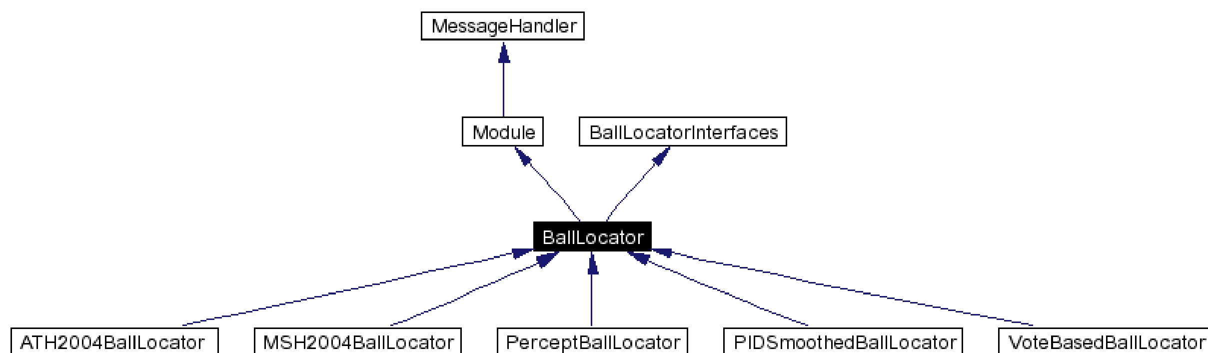


Figure H.2: The class hierarchy of module *BallLocator* (as in January 2004).

changed at runtime without affecting the overall system. Additionally, for some debugging scenarios it is important to switch off single modules completely.

With that, new approaches to a problem can be added to the existing code without changing recent developments. It is possible to compare solutions by switching them at runtime. And it makes it possible that the four members of the *GermanTeam* work all the time on the same code repository but can compete against each other at national RoboCup competitions.

H.2 Defining Modules and Solutions

Each module together with its solutions resides in a separate directory in *Src/Modules/*. For example the files for the module *BallLocator* reside in *Src/Modules/BallLocator/*.

The class hierarchy for a module and its solutions looks a bit complex (cf. fig. H.2), but this section will show that this structure is needful.

H.2.1 Class Module

Each module is derived from class `Module`[5]:

```
class Module : public MessageHandler
{
public:
  virtual void execute() = 0;

  virtual bool handleMessage(InMessage& message)
  { return false; }

  virtual ~Module() {};
};
```

[5]Definition in *Src/Tools/Module/Module.h* .

It allows to execute all modules in the same way through the parameterless `execute()` function (the interfaces of a module are passed to it as references to external representations at startup). As `Module` is derived from `MessageHandler` (cf. sect. G.1.2), it is possible to overwrite the `handleMessage(..)` function and to receive messages in a solution.

H.2.2 Interface Classes

If the interfaces of a module would be passed separately to the constructor of a module, changing the interfaces would be very time consuming as the constructors of all derived solutions would need to be changed too. That's why the interfaces of each module are defined in a separate class. For the *BallLocator* module this interface class is called `BallLocatorInterfaces`[6].

In January 2004 it looked such:

```
class BallLocatorInterfaces
{
public:
  BallLocatorInterfaces(
    const OdometryData& odometryData,
    const CameraMatrix& cameraMatrix,
    const BallPercept& ballPercept,
    const RobotPose& robotPose,
    SeenBallPosition& seenBallPosition,
    PropagatedBallPosition& propagatedBallPosition,
    unsigned long &time
    )
    :
  odometryData(odometryData),
    cameraMatrix(cameraMatrix),
    ballPercept(ballPercept),
    robotPose(robotPose),
    seenBallPosition(seenBallPosition),
    propagatedBallPosition(propagatedBallPosition),
    timeOfImageProcessing(time)
  {}

  const OdometryData& odometryData;
  const CameraMatrix& cameraMatrix;
  const BallPercept& ballPercept;
  const RobotPose& robotPose;

  SeenBallPosition& seenBallPosition;
```

[6]Definition in *Src/Modules/BallLocator/BallLocator.h* .

```
  PropagatedBallPosition& propagatedBallPosition;
  unsigned long &timeOfImageProcessing;
};
```

Thus the class contains references to all representations that are accessed by a module. Those representations which are the input are stored with `const` references, as the module is not allowed to change them. All references are initialized in the constructor.

H.2.3 Base Classes For Modules

Each solution of a module is derived from a common base class which is derived from `Module` and the interface class of the module (cf.. fig. H.2). It is called such as the module, for example `BallLocator`[7]:

```
class BallLocator : public Module,
                    public BallLocatorInterfaces
{
public:
  BallLocator(BallLocatorInterfaces& interfaces)
    : BallLocatorInterfaces(interfaces)
  {}

  virtual ~BallLocator() {}
};
```

In the constructor, an instance of `BallLocatorInterfaces` is used to initialize base class `BallLocatorInterfaces`. It is surprising that this works, but it is the main trick of the module/ solution mechanisms. As all derived solutions only have to pass an instance of their interface class to their base class, only one file (in the example *BallLocatorInterfaces.h*) must be edited when the interface of a module changes.

All solutions of a module should reside in separate subdirectories and are derived from the common module base class. For example the solution `ATH2004BallLocator`[8] is derived from `BallLocator`:

```
class ATH2004BallLocator : public BallLocator
{
public:
  ATH2004BallLocator(BallLocatorInterfaces& interfaces);
```

[7]Definition in *Src/Modules/BallLocator/BallLocator.h* .

[8]Not in the 2004 code release, definition and implementation in *Src/Modules/BallLocator/ATH2004BallLocator/ATH2004BallLo* and *.cpp* .

```
    virtual void execute();

  private:
    ... // own class variables and class functions
  };
```

The virtual `execute()` function must be implemented and the constructor of the base class is initialized with an instance of the interface class:

```
  ATH2004BallLocator::ATH2004BallLocator(
    BallLocatorInterfaces& interfaces)
  : BallLocator(interfaces)
  {
    ..
  }
```

H.2.4 Selecting Solutions

For switching module solutions at run-time, it must be known which solutions exist for each module. Thereto, class `SolutionRequest`[9] contains enumerations for all modules and all solutions. A new module can be defined there by adding an ID to enum `ModuleID` and by specifying a string for the ID in `getModuleName(id)`. For a new solution, an ID is added to enum `ModuleSolutionID` and a name to `getModuleSolutionName(id)`.

The class `SolutionRequest` is used to represent, which solution shall be selected for which module. For example it is sent from the *Settings* dialog bar in *RobotControl* (cf. fig. H.3) to the robots.

A standard configuration is loaded from the file *modules.cfg* at startup. With the *Settings* dialog bar it is possible to save a standard configuration in this file.

All solutions of a module are encapsulated by a solution *selector* class. This class instead of single solutions is embedded into the process that uses the module. All these classes are derived from `ModuleSelector`[10]. To save memory, only the selected solution is created. When switching solutions, the previous one is deleted with `delete` before the new one is created.

Thereto, each selector class has the function `createSolution(id)`, which creates a new instance of a solution for a given ID. The `execute()` method automatically calls the `execute()` function of the selected solution (same with `handleMessage(..)`).

[9]Definition and implementation in *Src/Tools/Module/SolutionRequest.h* and *.cpp* .
[10]Definition and implementation in *Src/Tools/Module/ModuleSelector.h* and *.cpp* .

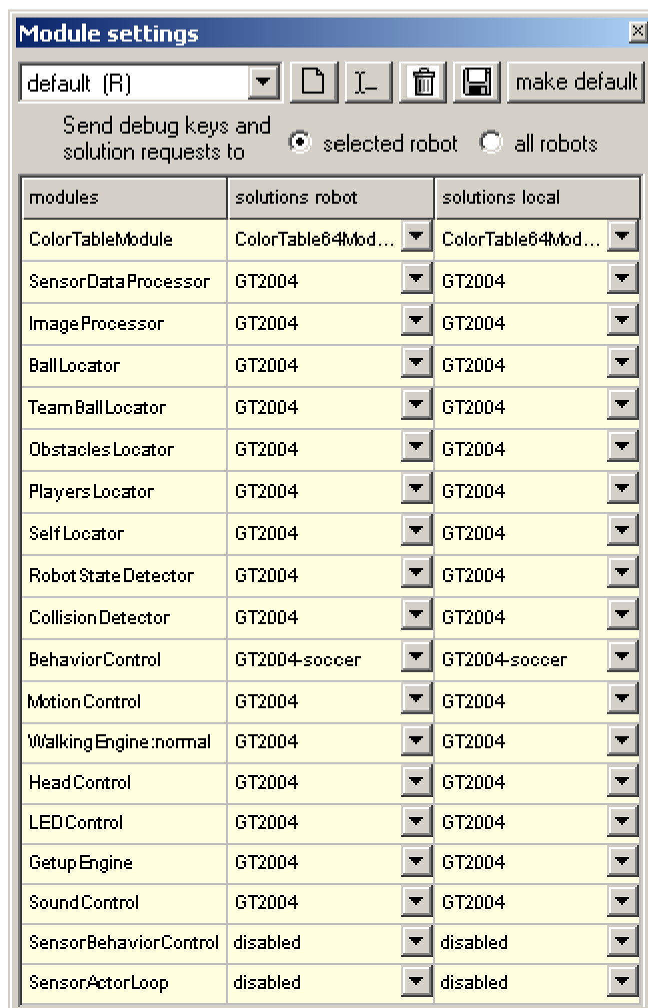


Figure H.3: With the *Settings* dialog bar of *RobotControl* it is possible to select between different module solutions or to switch off a module completely. On the left side are the selected solutions for physical, on the right for simulated robots.

The solution selector classes should be called such as the module + `Selector`. Thus, for the *BallLocator* module the class name should Be `BallLocatorSelector`[11]:

```
class BallLocatorSelector : public ModuleSelector,
                            public BallLocatorInterfaces
{
public:
  BallLocatorSelector(ModuleHandler &handler,
                      BallLocatorInterfaces& interfaces)
    : ModuleSelector(SolutionRequest::ballLocator),
      BallLocatorInterfaces(interfaces)
```

[11]Definition in *Src/Modules/BallLocator/BallLocatorSelector.h* .

```
    {
      handler.setModuleSelector(
               SolutionRequest::ballLocator, this);
    }

    virtual Module* createSolution(
      SolutionRequest::ModuleSolutionID id)
    {
      switch(id)
      {
      case SolutionRequest::pidSmoothedBallLocator:
        return new PIDSmoothedBallLocator(*this);

      case SolutionRequest::ath2004BallLocator:
        return new ATH2004BallLocator(*this);

      case ...:
         ...
      default:
        return 0;
      }
    }
  };
```

The interfaces of the module as well as a reference to a `ModuleHandler` (see below) are passed to the constructor of `BallLocatorSelector`. The constructor of base class `ModuleSelector` is initialized with the ID of the module (`SolutionRequest::ballLocator`) and the interfaces are passed to base class `BallLocatorInterfaces`. The module selector is registered at the module handler of the process with `moduleHandler.setModuleSelector(id, this)`. A new *BallLocator*-solution is created in `createSolution` dependent on the requested solution ID.

H.2.5 Administration of Modules

Class `ModuleHandler`[12] is the interface between the module selector classes and the processes. Each process has an instance of a `ModuleHandler` that handles incoming `SolutionRequest` messages and selects solutions according to this. Thereto, all module selectors register themselves at the module handler of their process.

[12]Definition and implementation in *Src/Tools/Module/ModuleHandler.h* and *.cpp* .

H.3 Modules and Processes

As already mentioned, modules "do not know each other". They communicate only via representations and the processes that embed the modules have to assure that all prerequisite representations are updated when a module is executed. A distribution of modules onto processes is called a *process layout*. Over the years, the CMD-layout (three processes: Cognition, Motion, and Debug) proved to be the best. The processes of such a layout reside in separate directories in *Src/Processes/*, thus for example in *Src/Processes/CMD*. In the remainder of this section, only the CMD layout is discussed.

H.3.1 Embedding Modules into Processes

For each module that is embedded into a process, a pointer to the module selector class is added as a member variable to the process. For example in class Cognition[13] there is this member variable:

```
BallLocatorSelector* pBallLocator;
```

In the constructor of the process, for each module the interface class and the module itself is created. For example in the constructor of Cognition the BallLocatorSelector is created such:

```
BallLocatorInterfaces ballLocatorInterfaces(
  theOdometryDataReceiver,
  cameraMatrix, ballPercept,
  thePackageCognitionMotionSender.robotPose,
  thePackageCognitionMotionSender.ballPosition,
  timeOfImageProcessing);

pBallLocator = new BallLocatorSelector(moduleHandler,
  ballLocatorInterfaces);
```

All representations that are accessed by the *BallLocator* module are passed to the constructor of BallLocatorInterfaces. Thereto, there must be an instance of each representation in the processes (see below).

Finally, the modules are executed in the main() method of the process:

```
pBallLocator->execute();
```

The order of execution results from their interfaces. All modules that produce input for a module have to be executed before that module.

[13] Definition and implementation in *Src/Processes/CMD/Cognition.h* and *.cpp* .

H.3.2 Representations in Processes

For all representations that are used by the modules of a process there must be instances in the process. References to them are passed to the module interface classes. In the normal case, these instances are simple member variables in the class of the processes.

For those representations that are exchanged with the robot operating system (`Image`, `MotorCommands`, etc.) there are already instances in the senders and receivers of this type. For example images are accessed through the variable `theImageReceiver` (declared with the macro `RECEIVER(Image)`).

Another exception are those representations that are used by modules in different processes. For example in the `Cognition` process the module `BehaviorControl`[14] writes its output into the representation `MotionRequest`[15], which is used in the `Motion` process by the module `MotionControl`[16]. Such representations must be transmitted between the processes via senders/ receivers. All representations that are sent from `Cognition` to `Motion` are combined in class `PackageCognitionMotion`[17]. Through the `SENDER(PackageCognitionMotion);` the package is sent to `Cognition`.

As members of these packages are real instances, they are accessed by the modules for example through `thePackageCognitionMotionSender.motionRequest`.

[14]Definition in *Src/Modules/BehaviorControl/BehaviorControl.h* .

[15]Definition and implementation in *Src/Representations/MotionRequest.h* and *.cpp* .

[16]Definition in *Src/Modules/MotionControl/MotionControl.h* .

[17]Definition and implementation in *Src/Processes/CMD/PackageCognitionMotion.h* and *.cpp* .

Appendix I

Programming RobotControl

The developed debugging mechanisms provide developers with the ease of preconfigured debugging techniques. The involved tasks are visualization and manipulation of debug data, as well as providing a generic debug switch. The mechanisms shown in the Section G would be meaningless, without a standard way of using them. This leads to the necessity of a combined debug application forming the frontend or graphical user interface to the debugging techniques. This application is RobotControl. Being a frontend to the debugging techniques allows RobotControl to be used for the debugging of robots, which use the GermanTeam debugging architecture (cf. Sect. 2.2.2). RobotControl depends on the debugging architecture not the robot architecture. There are however parts of RobotControl supporting the cognitive architecture like Modules (cf. Sect. 3). Note that the debugging techniques however do not depend on RobotControl. They were developed so any frontend, like a java graphical user interface and so on, can be used. The Simulator is another frontend to the debugging mechanisms.

RobotControl was developed using *C#* and the *.Net Framework*, making heavy use of new features like *Reflection*, *Delegates* and *Events*. It can be compiled with any standard Visual Studio .Net 2003 environment.

I.1 The Application

The main application of *RobotControl* is used as a frontend to the debugging techniques described in the earlier sections, as well as a frontend to control other parts of the GermanTeam architecture, like *Modules* and *Solutions*.

Visualization. All representations that are exchanged between the *Modules* of the GermanTeam architecture can be viewed in various user controls. Images can be shown, representations can be drawn into a virtual soccer field as well as in images where needed. There are textual user controls for representations not suitable to be shown in images or the soccer field. It is possible to trace values over time in generic graphing tools. Nearly any value can be tracked with special or user defined user controls. *Debug Drawings* can be drawn on images or soccer fields.

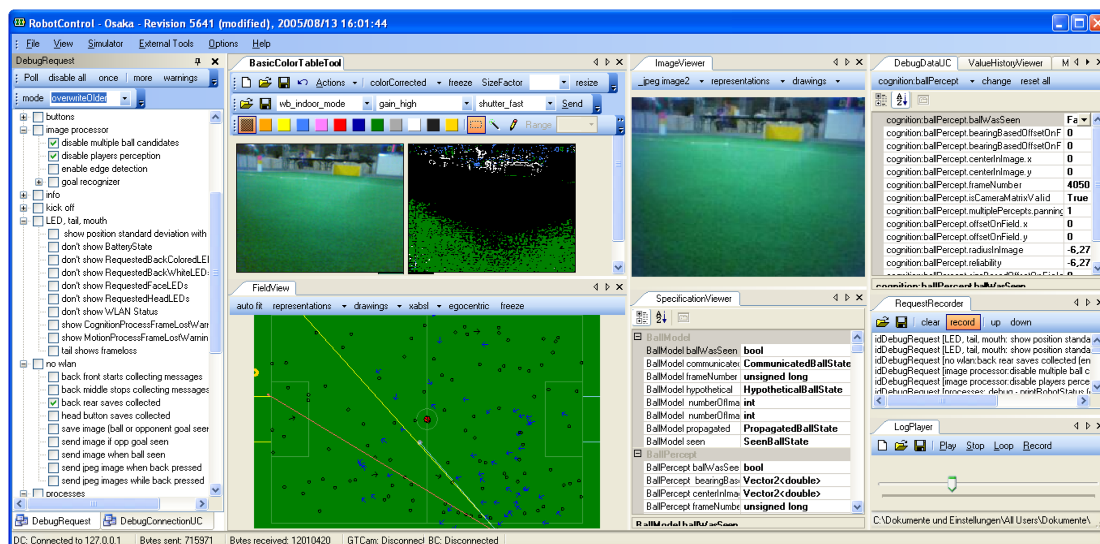


Figure I.1: The *RobotControl* application.

Modification and Manipulation of Data. There are standard dialogs and user controls for the manipulation of data. All representations shown, can be manipulated. There are standard ways for connecting manipulation devices, like joysticks, gamepads or other devices to modifiable data.

Control. There are dialogs for switching *Solutions*, changing parameters of the robot's built in camera, as well as operating the robot via joystick.

There has long been a close coupling of debug tools and robot code in the GermanTeam. Previous debug tools had robot code built in for simulation purposes. This principle has been overcome by *RobotControl*. For *RobotControl* it makes no difference whether it is connected to a simulated process or a real robot. One of the first steps in developing *RobotControl* was supporting to connect to the *Simulator*, which simulates a given number of robots as well as their environment, allowing for connections to the simulated processes as if they were real robots.

RobotControl is almost completely decoupled from the actual robot code. It solely relies on the architecture, especially the debugging architecture. This leads to the necessity of changing the debug tool in case the debugging mechanisms change for the robot, but not in the case these mechanisms are used more frequently or at different places, even on different platforms. This allows *RobotControl* to be always up to date. If for instance different developers have used different *Debug Requests* this will not require them to change *RobotControl* but rather allows *RobotControl* to manipulate either one of the control programs for the robot.

I.2 General Structure

RobotControl was developed with two principles in mind. The first is to solve the tasks almost everybody has in developing robot control software and debugging parts of the software. The second principle however is to leave the tool as extensible as possible for needs not identified in the development process of *RobotControl*. This is reflected in the general architecture of *RobotControl*. The application only provides a general framework. Specific tasks are handled by managers or user controls. Tasks that need to be solved for many user controls are grouped in managers, which are named after the concept they provide. A lot of tasks involved have been implemented already, only a part of it will be introduced here.

There are tasks that are defined in order to leave *RobotControl* not an empty building which could do anything, but help the developer to debug robot control applications. This involves handling Messages and MessageQueues, as well as a number of debugging techniques already introduced, like *Debug Requests*, *Debug Drawings* and so on. A lot of these tasks are packaged in managers. User Controls can be used to access features of managers. *RobotControl* is hierarchical in the sense that user controls do not know each other, but all managers are accessible to user controls as well as other managers. Every user control and every manager gets references to all managers during construction. The interfaces exposed by the manager and the protocol to communicate with a given manager are defined by the manager. Though most managers usually have event driven interfaces.

RobotControl has built in documentation mechanisms and allows for the scripting of dialogs.

I.3 Message Handling

RobotControl is a wrapper providing dialogs and user controls with a standard way of accessing messages from the robot. The most important service of *RobotControl* besides providing a graphical user interface is providing a message handling and distribution algorithm. Messages can come from physical robots as well as from log files. Messages can be delivered to simulated processes (running robot control software on a developer workstation).

A special and very important manager is the *Message Handler*. All debugging mechanisms introduced rely on the concept of *Messages* and *Message Queues*. All higher level debugging mechanisms use *Messages* and *Message Queues* as a means for communication with the debug tool. Either directly over wireless network or indirectly through log files. The importance of *Messages* and *Message Queues* is also underlined by the huge number of messages communicated in standard debug sessions. For instance *Debug Drawings* and their primitives communicated via *Messages* can generate more than 100 *Messages* per frame, resulting in more than 3000 messages per second. Messages can contain anything from jpeg encoded images to bitmaps, from representational data or data types to drawings. Not all messages are of interest to all user controls. Most of the time the order of arrival of messages plays a crucial role.

The class `MessageHandler` provides an event driven interface. A dialog or manager can register a function it wants to be notified if a message of a specified type arrives. The class `MessageHandler` keeps track of the running functions and also preserves the order of arriv-

ing messages for that dialog. A special concept termed *Realtime Message Handling* allows for the silent dropping of *Messages* in case there are too many messages arriving, slowing down the whole application.

I.4 Manager

Managers basically solve common tasks that either do not need a graphical user interface or are to be used by a number of user controls. There are managers handling special debugging techniques like *Debug Requests* or *Debug Drawings*. There are for instance two user controls using the *Debug Drawing Manager*, the *Image viewer*, showing images as well as *Debug drawings* of the type ”drawingOnImage”, as well as the *Field View*, showing drawings of the type ”drawingOnField”. The *Debug Drawing Manager* handles the receiving of messages, the request of drawings, as well as their collection and drawing. It provides connected user controls with a data structure that can be directly rendered to the screen, relieving the user control of the necessity of knowing and handling the underlying debug mechanisms.

I.5 User Controls

There are a number of user controls supporting the visualization or the modification of data. User controls should be derived from the class *RobotControlUserControl*. This allows member wise access to the interface of managers. User controls can hook into the message handling processing through the *Message Handler*, make use of drawings through the *Drawing Manager*, and so on. A number of user controls were developed. The initialization and loading of user controls makes heavy use of the .Net runtime, namely *Reflection*. User controls can simply be added to the project. Every user control derived from *RobotControlUserControl* will be loaded automatically. Since there are no interconnections between user controls this also allows for the easy removal of user controls.

References

[1] Ronald C. Arkin. Motor schema-based mobile robot navigation. *The International Journal of Robotics Research*, 8(4), 1989.

[2] Ronald C. Arkin. *Behavior-Based Robotics*. The MIT Press, 1998.

[3] AT&T. GraphViz homepage, 2000. http://www.research.att.com/sw/tools/graphviz/.

[4] Hynek Bakstein. A complete dlt-based camera calibration, including a virtual 3d calibration object. Master's thesis, Charles University, Prague, 1999.

[5] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies – A comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.

[6] Jean-Yves Bouguet. Camera calibration toolbox for matlab. http://www.vision.caltech.edu/bouguetj/calib_doc/.

[7] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. W3C recommendation: Extensible markup language (XML) 1.0 (second edition), 2000. http://www.w3.org/TR/REC-xml.

[8] Rodney A. Brooks. The behavior language; user's guide. Technical Report AIM-1127, 1990.

[9] 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.

[10] R. Brunn, U. Düffert, M. Jüngel, T. Laue, M. Lötzsch, S. Petters, M. Risler, Th. Röfer, and A. Sztybryc. GermanTeam 2001. In *RoboCup 2001 Robot Soccer World Cup V, A. Birk, S. Coradeschi, S. Tadokoro (Eds.)*, number 2377 in Lecture Notes in Computer Science, pages 705–708. Springer, 2001. More detailed in: http://www.tzi.de/kogrob/papers/GermanTeam2001report.pdf.

[11] Ronnie Brunn and Michael Kunz. A global vision system to support development in autonomous robotic soccer. 2005. Available online: http://www.sim.informatik.tu-darmstadt.de/publ/da/2005-Brunn-Kunz.pdf (Diploma thesis published in German language only).

[12] Jared Bunting, Stephan Chalup, Michaela Freeston, Will McMahan, Rick Middleton, Craig Murch, Michael Quinlan, Christopher Seysener, and Graham Shanks. Return of the nubots! - the 2003 nubots team report. Technical report, 2003.

[13] 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.

[14] 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.

[15] James Clark. W3C recommendation: XSL transformations (XSLT) version 1.0, 1999. http://www.w3.org/TR/XSLT.

[16] Sony Corporation. Open-r documentation - open-r internet protocol version 4. Technical report, 2004. Available online: http://openr.aibo.com/openr/eng/index.php4.

[17] Nick Barnes Daniel Cameron. Knowledge-based autonomous dynamic colour calibration. In *7th International Workshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2004. to appear.

[18] Uwe Düffert and Jan Hoffmann. Reliable and precise gait modeling for a quadruped robot. In *RoboCup 2005: Robot Soccer World Cup IX*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.

[19] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[20] David C. Fallside. W3C recommendation: XML schema part 0: Primer, 2001. http://www.w3.org/TR/xmlschema-0/.

[21] 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.

[22] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11):1203–1233, 1999.

[23] Matthias Hebbel, Ingo Dahm, Denis Fisseler, and Walter Nistico. Learning Fast Walking Patterns with Reliable Odometry Information for Four-Legged Robots. In *ISMCR'05, 15th International Symposium on Measurement and Control in Robotics*, 2005. to appear.

[24] J. Heikkilä and O. Silvén. A four-step camera calibration procedure with implicit image correction. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'97)*, pages 1106–1112, 1997.

[25] Bernhard Hengst, Darren Ibbotson, Son Bao Pham, and Claude Sammut. Omnidirectional locomotion for quadruped robots. In *RoboCup 2001 Robot Soccer World Cup V, A. Birk, S. Coradeschi, S. Tadokoro (Eds.)*, number 2377 in Lecture Notes in Computer Science, pages 368–373. Springer, 2002.

[26] Jan Hoffmann and Daniel Göhring. Sensor-Actuator-Comparison as a Basis for Collision Detection for a Quadruped Robot. In *8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.

[27] Jan Hoffmann, Matthias Jüngel, and Martin Lötzsch. A vision based system for goal-directed obstacle avoidance used in the RC 03 obstacle avoidance challenge. In *8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.

[28] Jan Hoffmann, Michael Spranger, Daniel Göhring, and Matthias Jüngel. Exploiting the unexpected: Negative evidence modeling and proprioceptive motion modeling for improved markov localization. In *RoboCup 2005: Robot Soccer World Cup IX*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.

[29] V. Jagannathan, R. Dodhiawala, and L. Baum. *Blackboard Architectures and Applications*. Academic Press, Inc., 1989.

[30] Matthias Jüngel. Using layered color precision for a self-calibrating vision system. In *8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2005.

[31] 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.

[32] Kurt Konolige. COLBERT: A language for reactive control in Sapphira. In *KI-97: Advances in Artificial Intelligence – Proceedings of the 21st Annual German Conference on Artificial Intelligence, G. Brewka, C. Habel, and B. Nebel (Eds.)*, number 1303 in Lecture Notes in Artificial Intelligence, pages 31–52. Springer, 1997.

[33] Cody Kwok and Dieter Fox. Map-based multiple model tracking of a moving object. In *8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2005.

[34] Tim Laue and Thomas Röfer. A Behavior Architecture for Autonomous Mobile Robots Based on Potential Fields. In *RoboCup 2004: Robot Soccer World Cup VIII*, Lecture Notes in Artificial Intelligence. Springer, 2005.

[35] Tim Laue, Kai Spiess, and Thomas Röfer. SimRobot - A General Physical Robot Simulator and Its Application in RoboCup. In *RoboCup 2005: Robot Soccer World Cup IX*, Lecture Notes in Artificial Intelligence. Springer, 2006. to appear.

[36] 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)*, 2002.

[37] Martin Lötzsch. DotML Documentation, 2003. http://www.martin-loetzsch.de/DOTML.

[38] Martin Lötzsch. XABSL web site, 2003. http://www.ki.informatik.hu-berlin.de/XABSL.

[39] Martin Lötzsch. XABSL - a behavior engineering system for autonomous agents. Diploma thesis. Humboldt Universität zu Berlin, 2004. Available online: http://www.martin-loetzsch.de/papers/diploma-thesis.pdf.

[40] 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.

[41] D. MacKenzie, R. Arkin, and J. Cameron. Multiagent mission specification and execution. *Autonomous Robots*, 4(1):29–52, 1997.

[42] Jonathan Marsh and David Orchard. W3C candidate recommendation: XML inclusions (XInclude) version 1.0, 2002. http://www.w3.org/TR/xinclude/.

[43] R. Mohr and B. Triggs. Projective geometry for image analysis, 1996.

[44] David Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.

[45] Walter Nistico and Thomas Röfer. Improving Percept Reliability in the Sony Four-Legged League. In *RoboCup 2005: Robot Soccer World Cup IX*, Lecture Notes in Artificial Intelligence. Springer, 2006. To appear. Online: `http://www.tzi.de/kogrob/papers/rc06-vision.pdf`.

[46] 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.

[47] Th. Röfer, H.-D. Burkhard, U. Düffert, J. Hoffmann, D. Göhring, M. Jüngel, M. Lötzsch, O. v. Stryk, R. Brunn, M. Kallnik, M. Kunz, S. Petters, M. Risler, M. Stelzer, I. Dahm, M. Wachter, K. Engel, A. Osterhues, C. Schumann, and

J. Ziegler. GermanTeam RoboCup 2003. Technical report, 2003. Available online: http://www.robocup.de/germanteam/GT2003.pdf.

[48] Th. Röfer, H.-D. Burkhard, U. Düffert, J. Hoffmann, D. Göhring, M. Jüngel, M. Lötzsch, O. v. Stryk, R. Brunn, M. Kallnik, M. Kunz, S. Petters, M. Risler, M. Stelzer, I. Dahm, M. Wachter, K. Engel, A. Osterhues, C. Schumann, and J. Ziegler. GermanTeam RoboCup 2004. Technical report, 2004. Available online: http://www.robocup.de/germanteam/GT2004.pdf.

[49] Thomas Röfer. SimRobot homepage. http://www.tzi.de/simrobot.

[50] Thomas Röfer. An architecture for a national robocup team. In *RoboCup 2002 Robot Soccer World Cup VI, Gal A. Kaminka, Pedro U. Lima, Raul Rojas (Eds.)*, number 2752 in Lecture Notes in Artificial Intelligence, pages 417–425. Springer, 2003.

[51] Thomas Röfer. Evolutionary gait-optimization using a fitness function based on proprioception. In *8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.

[52] Thomas Röfer, Time Laue, and Dirk Thomas. Particle-filter-based self-localization using landmarks and directed lines. In *RoboCup 2005: Robot Soccer World Cup IX*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.

[53] L.G. Roberts. Machine perception of three dimensional solids. *Optical and Electro-Optical Information Processing*, pages 159–197, 1968.

[54] S. Russel and P. Norvig. *Artificial Intelligence, a Modern Approach.* Prentice Hall, 1995.

[55] D. Schulz and D. Fox. Bayesian color estimation for adaptive vision-based robot localization. In *Proceedings of IROS*, 2004.

[56] Stephen M. Smith and J. Michael Brady. SUSAN - A New Approach to Low Level Image Processing. *Int. Journal Computer Vision*, 23(1):45–78, 1997.

[57] 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.