GERMANTEAM
RoboCup 2004

Thomas Röfer
Tim Laue

Hans-Dieter Burkhard
Jan Hoffmann
Matthias Jüngel
Daniel Göhring
Martin Lötzsch
Uwe Düffert
Michael Spranger
Benjamin Altmeyer
Viviana Goetzke

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

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
Max Risler
Maximilian Stelzer
Dirk Thomas
Stefan Uhrig

Uwe Schwiegelshohn
Ingo Dahm
Matthias Hebbel
Walter Nisticó
Carsten Schumann
Michael Wachter

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

Computer Engineering Institute,
Fakultät Elektrotechnik,
University of Dortmund,
Otto-Hahn-Strasse 4,
44221 Dortmund, Germany

Abstract

The GermanTeam is a joint project of four German universities in the Sony Legged Robot League. This report describes the software developed for the RoboCup 2004 in Lisbon. 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

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, and 2004 in Paderborn. In 2004, the Microsoft Hellhounds (from Dortmund) also participated in the American Open, Australian Open, and Japan Open. 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), and Lisbon (2004).

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).

The GermanTeam won the RoboCup World Championship 2004. It 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. 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.

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 wheeld 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. In 2004 the Darmstadt Dribblers participated in the Humanoid-League for the first time.

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 the investigation of a fast self-localization algorithm combining Monte-Carlo and single landmark approaches, low-level behavior algorithms using potential fields and an improved walking engine. For the competitions in 2004 a Kalman Filter has been implemented to enable the goal keeper as well as the field players to anticipate the velocity and direction of a ball moving in the robot's direction. Using this method the ball holding capabilities of the players, especially of the goal keeper, could be improved significantly.

1.2.3 Universität Bremen

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

The research project focuses on plan recognition detection of agents in general. However, the RoboCup is an ideal test-bed for the methods to be developed. This project is also part of the priority program "Cooperating teams of mobile robots in dynamic environments" funded by the Deutsche Forschungsgemeinschaft (German Research Foundation).

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

1.2.4 Universität Dortmund

The team of the Universität Dortmund focuses its research interests in 2005 on the wide fields of learning algorithms, modeling, and behavior control with applications in robot soccer.

To improve our existing solutions and approaches, we will realize a measuring environment: An external camera, which is mounted at the ceiling directly over the robots playing field. Using this camera, we will analyze the exact difference and variance between estimated object positions and real-world data. Based on this, we will investigate, how error models can be further improved. If we know more about estimation errors, we assume to increase the reliability and accuracy of our world model (esp. ball model, players model, and robot self-localization).

The external robot observation shall also be used to learn walking parameters for omnidirectional walking and odometry calibration of the walk. Our target is to implement an even smoother and faster walk than before. Thereto, we use the ceiling camera as an external sensory device of the robot itself and fuse it with the onboard acceleration sensors. From our point of view, here we can profit from the developed virtual-robot framework.

Due to the change of rules, the border of the playing field will be removed in the next season. Thus, kicking the ball exactly becomes more crucial than ever before. For this reason, we plan to investigate, how kicking the novel ball can be learned automatically.

As handling the ball becomes more difficult, we assume, that we have to focus the ball more often than we did in the past. Consequently, this effects localization, as we cannot afford to search for landmarks that frequently. Hence, we try to extract and use more features of the field (e.g. intersecting lines and the center circle). This should help to localize with a high reliability.

Finally, we have to use all extracted information to establish a collective behavior which addresses the problem of controlling a team of multiple soccer robots coherently. We will investigate further, how our developed virtual-robot approach can be improved by an automatic situation analysis.

1.3 Contributing Team Members

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

1.3.1 Aibo Team Humboldt (Humboldt-Universität zu Berlin)

“Diplom” Students. Benjamin Altmeyer, Uwe Düffert, Daniel Göhring, Viviana Goetzke, Martin Lötzsch, Michael Spranger.

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

Professor. Hans-Dieter Burkhard.

1.3.2 Darmstadt Dribbling Dackels (Technische Universität Darmstadt)

"Diplom" Students. Ronnie Brunn, Marc Dassler, Michael Kunz, Sebastian Petters, Max Risler, Michael Schmitt, Marcus Schobbe, Patrick Stamm , Dirk Thomas, Holger Tronnier, Stefan Uhrig.

PhD Student. Max Stelzer.

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

1.3.3 Bremen Byters (Universität Bremen)

"Diplom" Students. Holger Dick, Martin Fritsche, Jessica Marrufo.

PhD Student. Tim Laue.

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

1.3.4 Microsoft Hellhounds (Universität Dortmund)

"Diplom" Students. Arthur Cesarz, Damien Deom, Jörn Hamerla, Mathias Hülsbusch, Jochen Kerdels, Thomas Kindler, Hyung-Won Koh, Tim Lohmann, Manuel Neubach, Claudius Rink, Andreas Rossbacher, Frank Roßdeutscher, Bernd Schmidt, Carsten Schumann, Pascal Serwe, Michael Wachter.

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 solution used in the "Open Challenge".

Only 60% of the approximately 330,000 lines of code that were written by the GermanTeam for the RoboCup 2004 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 2004 and giving an outlook on the future perspectives of the GermanTeam in 2005 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 and D describe the usage of the simulator and RobotControl, the two main tools of the GermanTeam. Appendix E contains a detailed documentation of the behavior engine used by the GermanTeam in Lisbon. Afterwards, the GermanTeam's abstraction of *processes*, *senders*, and *receivers* is presented in Appendix F, followed by Appendix G on *streams*, Appendix H on the debugging support, and Appendix I on the way how the GermanTeam supports different implementations for a single task in parallel. The three final appendices describe how the main debugging tool *RobotControl* works (Appendix J) and how it can be extended with new dialogs (Appendix K) and toolbars (Appendix L).

1.5 Innovations in 2004

For those who already read our team report from 2003 [46], pointers to the main innovations achieved in 2004 are given here.

Image Processing was revised. It compensates for the blue shade produced by the camera of the ERS-7 near the image border, as well as for geometric distortions (cf. Section 3.2.3), as well as for angular errors resulting from taking images while the head is moving fast (cf. Section 3.2.10). The ball specialist used in 2002 was re-established and improved (cf. Section 3.2.5). The search for flags (cf. Section 3.2.1) and the specialist for recognizing them was also improved (cf. Section 3.2.6). Points on lines are now augmented with the orientation of the corresponding line in field coordinates (cf. Section 3.2.4).

World Modeling. The self-locator uses the directedness of points on field lines to distinguish between horizontal and vertical lines on the field (cf. Section 3.3.2.3), which resulted in a very good localization of the goalie. The ball's position and speed are modeled using a Kalman filter (cf. Section 3.4).

Behavior Control. The behaviors are now described using XABSL version 2 (cf. Section 3.8). XABSL is also used to describe the different motions of the head of the robot (cf. Section 3.9.3). Developing behaviors using XABSL is supported by the *XABSL profiler* (cf. Section E.5.4), a tool that does statistics on activation paths and state changes in the XABSL graph.

Several new and improved basic behaviors such as turning with the ball, turning behind the ball, and approaching the ball were implemented (cf. Section 3.8.1). In general, it was tried to

be able to play soccer without kicks, and kicks are only used in selected situations (cf. Section 3.8.1.2). The decision on which kick to use in a certain situation is now solved by the *Kick Selection Table* (cf. Section 3.8.1.4), i. e. a table that maps current ball position and desired kicking direction to an adequate kick. Several basic behaviors are now based on a more flexible and XML-based implementation of potential fields (cf. Section 3.8.2.3).

Motion Control. All the walks used by the GermanTeam were optimized using Evolutionary Algorithms (cf. Section 3.9.1.6). The walking engine is now able to interpolate between different walks for different directions (cf. Section 3.9.1). Some of the kicking motions were developed using inverse kinematics (cf. Section 3.9.2). The head control now supports active vision, i. e. it tries to point the head in directions relevant for both ball recognition and self-localization (cf. Section 3.9.3).

Infrastructure. The communication between the robots of a team is now based on UDP. The robots of a team automatically detect each other with the help of the *Dog Discovery Protocol* (cf. Section 2.2.3.2). The communication between a robot and the debug tools of the GermanTeam now uses direct TCP without the TCP-gateway. The simulator is based on OpenGL now and produces more realistic images (cf. Section 5.1). In addition to the ERS-210, it also simulates the ERS-7.

Chapter 2

Architecture

The GermanTeam is an example of a national team. The members participated as separate teams in the German Open 2002, 2003, and 2004, but formed a single team at the RoboCup in Fukuoka, Padova and Lisbon. 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. 5.1). Although the Open-R SDK is now publicly available, there is no reason for the GermanTeam to remove the platform-independent encapsulation from their code.

2.1.2 Realization

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

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

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

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

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

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

2.1.3 Supported Platforms

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

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

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

Open-R Emulator under Cygwin. The environment was also implemented on the so-called Open-R emulator that allows parts of the robot software to be compiled and run under Linux and Cygwin.

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 H. 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:

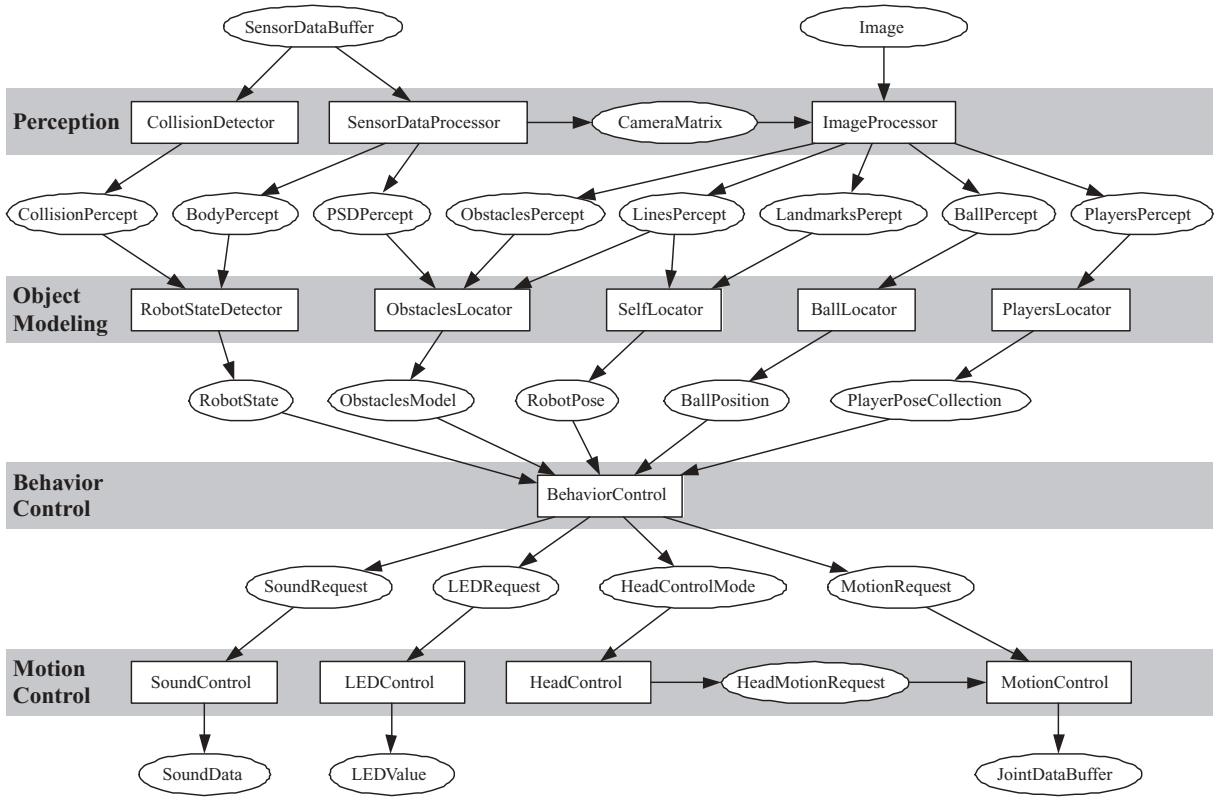


Figure 2.1: The tasks identified by the GermanTeam 2004 for playing soccer.

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 system manages two sets of information, the current state of all *debug keys*, and the currently active solutions. Debug keys work similar to C++ preprocessor defines, but they can be toggled at runtime (cf. Sect. H.1.3). A special infrastructure called *message queues* (cf. Sect. H.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 keys and to switch between different solutions. The message queues are also used to transmit other kinds of data between the robot(s) and the debugging tool on the PC (cf. Sect. 5.2). For example, motion requests can directly be sent to the robot, images, text messages, and even drawings (cf. Sect. H.3.1) can be sent to the PC. This allows visualizing the state of a certain module, textually and even graphically. These techniques work both on the real robots and on the simulated ones (cf. Sect. 5.1).

2.2.3 Process-Layouts

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

2.2.3.1 Communication between Processes

In the robot control program developed by the GermanTeam 2001 for the championship in Seattle, the different processes exchanged their data through a shared memory [9], i. e., a blackboard architecture [28] 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.

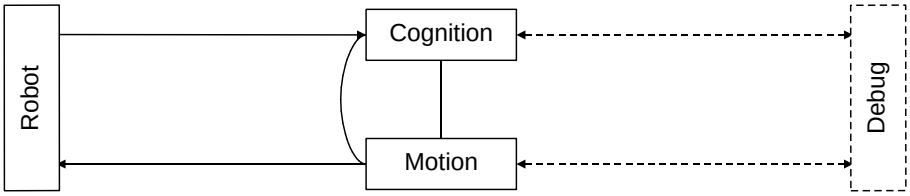


Figure 2.2: The process layout of Humboldt 2002, since RoboCup 2002 used by the whole GermanTeam.

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* [15]. 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 entry-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* [46], 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. F.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. H.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 and chosen way is to send single-cast packets. In this solution every robot sends out a separate packet to every other robot in the team.

Therefore the data to send is also streamed to a memory buffer by the use of *NetSender*s and *NetReceiver* 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.

Since every robot needs the IP-address of each other robot in his team, the "Dog-Discovery-Protocol" (DDP) was developed. 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. If these parameters match its own parameters, the robot puts the IP-address of the teammate in a list of teammates and starts sending team communication data to the robot. If for some reason the teammate did not receive any packet from the robot for 10 seconds, it stops sending to it and removes it 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.

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 and 2004, a simple speed optimized pre-processor called *Depend* (*GT2004/Src/Depend*) (cf. Sect. 5.5)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 (*$PDIR*, located in *GT2004/Build*) to avoid conflicts between different builds as well as the compulsion for a complete rebuild after changing the process layout, build variant, or compiler. Each such *$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. 5.5) is used to completely generate all dependencies for the chosen build target in *GT2004/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 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 *GT2004/Make/Dsp_generation/*. Since 2004 project files for Visual Studio 2003.Net can be generated too.

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 GT2004

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 I.2.4.

This chapter describes most of the modules that were implemented. For some solutions only the chosen approach for the competition in Lisbon 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 scanning using a wider spaced grid.

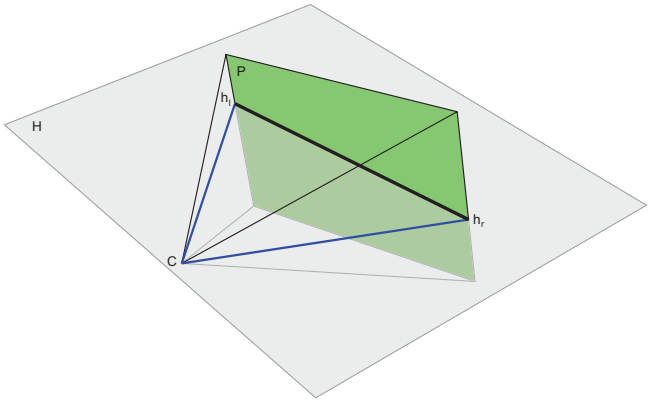


Figure 3.1: Construction of the horizon

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

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

with only z_l and z_r unknown. Let

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

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

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

with the rotation matrix

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

leads to

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

In the same way follows

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

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

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

To speed up the object detection and to decrease the number of false positives, essentially three 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 borders, field lines, and the lower borders of the goals (cf. Fig. 3.2a). A set of grid lines

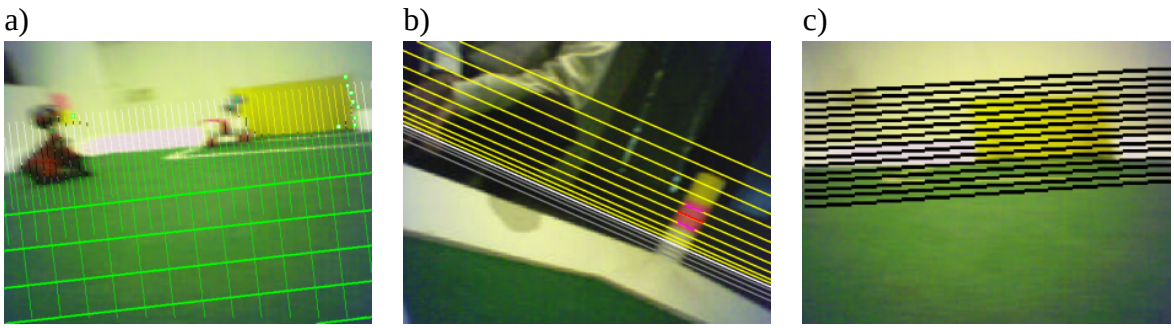


Figure 3.2: Different scanlines and grids. a) The main grid which is used to detect objects on the field. b) The grid lines for beacon detection. c) The grid lines for goal detection.

parallel to and in most parts over the horizon is used to detect the pink elements of the beacons (cf. Fig. 3.2b). The goal detection is also based on horizontal grid lines (cf. Fig. 3.2c)

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

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 [29], [53]), 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 similar to the approach described in [35], but in contrast to it, this technique here is not used to perform a semi-automated calibration from a set of samples, but to extend the validity of a manually generated one. Thus, a color table is processed in the following way:

- Each point, which is assigned to a color class, irradiates it's influence to the whole color space, with an influence factor which decreases exponentially with the distance:

$$I_i(p_1, p_2) = \begin{cases} \lambda^{|p_1 - p_2|} & i = c(p_2) \\ 0 & \forall i \neq c(p_2) \end{cases} \tag{3.7}$$

  where p_1, p_2 are two arbitrary points in the color map, $\lambda < 1$ is the exponential base, $I_i(p_1, p_2)$ is the influence of p_2 on the (new) color class $i \in \{red, orange, yellow, \cdots\}$ of p_1, and $c(p_2)$ is the color class of p_2,

- Since the cost of the influence calculation is $O(n^2)$, where n is the number of elements of the color table (2^{18} in our case), instead of the euclidean distance, a manhattan distance is used:

$$|p_1 - p_2|_{manhattan} = |p_{1y} - p_{2y}| + |p_{1u} - p_{2u}| + |p_{1v} - p_{2v}| \tag{3.8}$$

- For each point in the new color table, the total influence for each color class is computed:

$$I_i(p_0) = B_i \cdot \sum_{p \neq p_0} I_i(p_0, p) \tag{3.9}$$

  where $B_i \in (0..1]$ is a bias factor which can be used to favor the expansion of one color class over its neighbors

- Then the color class that has the highest influence for a point is chosen, if:

$$\frac{max(I_i)}{I_{bk} + \sum_i I_i} > \tau \tag{3.10}$$

  where τ is a confidence threshold, I_{bk} is a constant value assigned to the influence of the background (noColor) which prevents an unbounded growth of the colored regions to the empty areas, and $i \in \{red, orange, yellow, \cdots\}$ again represents the color class.

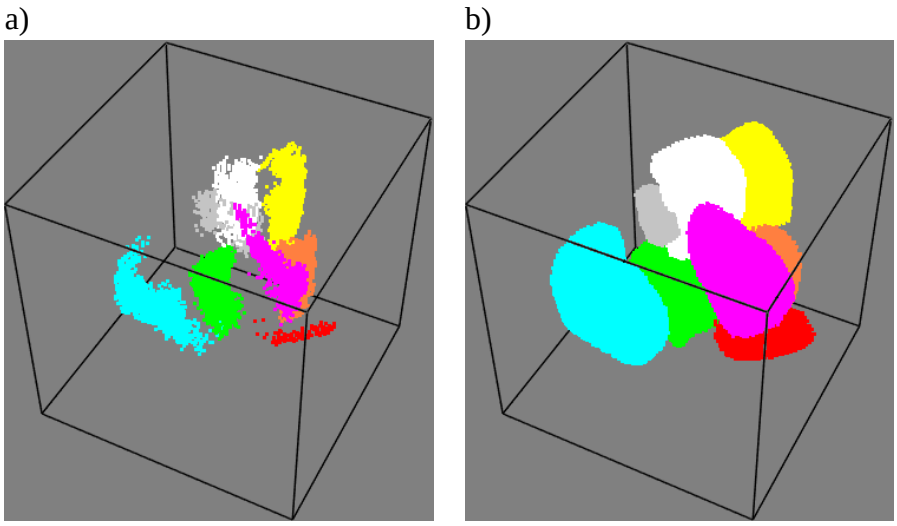


Figure 3.3: Appearance of the color table after the exponential generalization: (a) original, (b) optimized.

The parameters λ, τ, B_i, I_{bk} control the effects of the generalization process, and we have implemented 3 different settings: one for conservative generalization, one for aggressive expansion, one for increasing the minimum distance among neighbouring regions. The time required to apply this algorithm, on a 2^{18} elements table, is $\approx 4-7$ minutes on a 2.66GHz Pentium4 processor, while for a table of 2^{16} elements, this figure goes down to only 20-30 seconds.

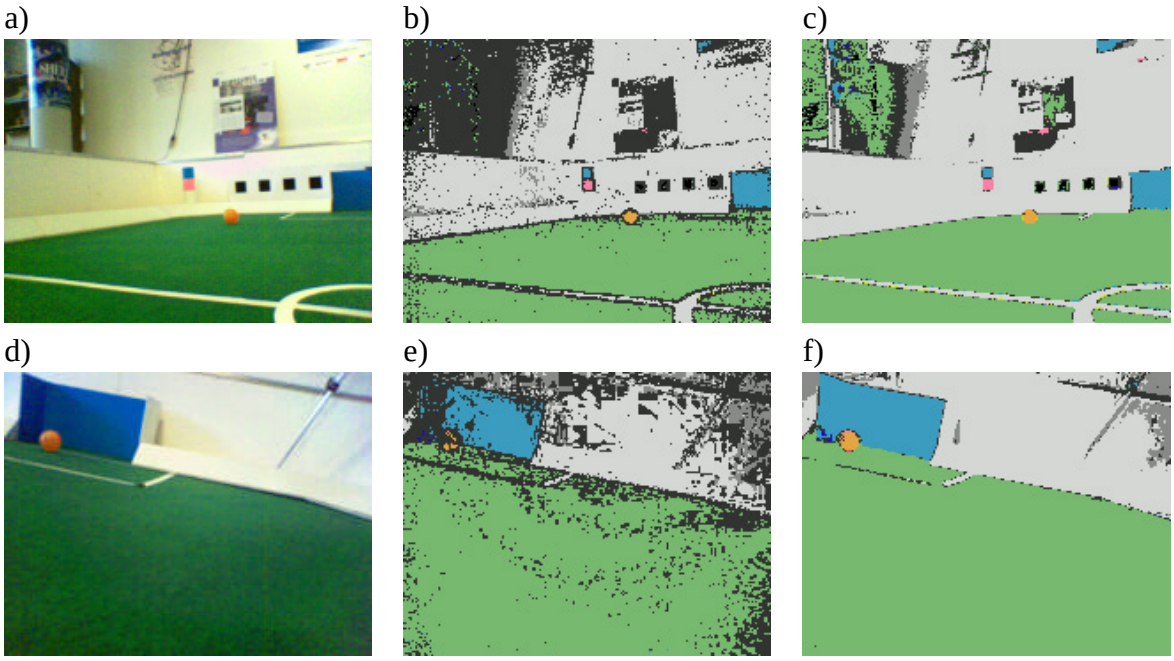


Figure 3.4: Effects of the exponential generalization. (a) and (d) represents images taken from the same field, but the lighting conditions differ due to an increased amount of sunlight in image (d): this can be seen by the blueish cast on white objects such as the walls. (b) and (e) are classified using the original color table, calibrated for the conditions found in (a); it can be noticed that (e) is not satisfactory, as the ball is hard to detect and the goal appears largely incomplete. (c) and (f) are classified using the generalized table, and it shows that it can gracefully accomodate to the new lighting conditions (f).

3.2.3 Camera Calibration

With the introduction of the ERS7 among the available platforms for the 4-Legged League, an analysis of the camera of the new robot was required to adapt to the new specifications. While the resolution of the new camera is $\approx 31\%$ higher compared to the previous one, preliminary tests revealed some specific issues which weren't present in the old model:

- lower light sensitivity, which can be countered 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. 3.5).

Geometric camera model. In the previous years, the horizontal and vertical opening angles of the cameras were used as the basis for all the measurements calculations, following the classical "pinhole" model; however this year we decided to use a more complete model which would also take into account 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

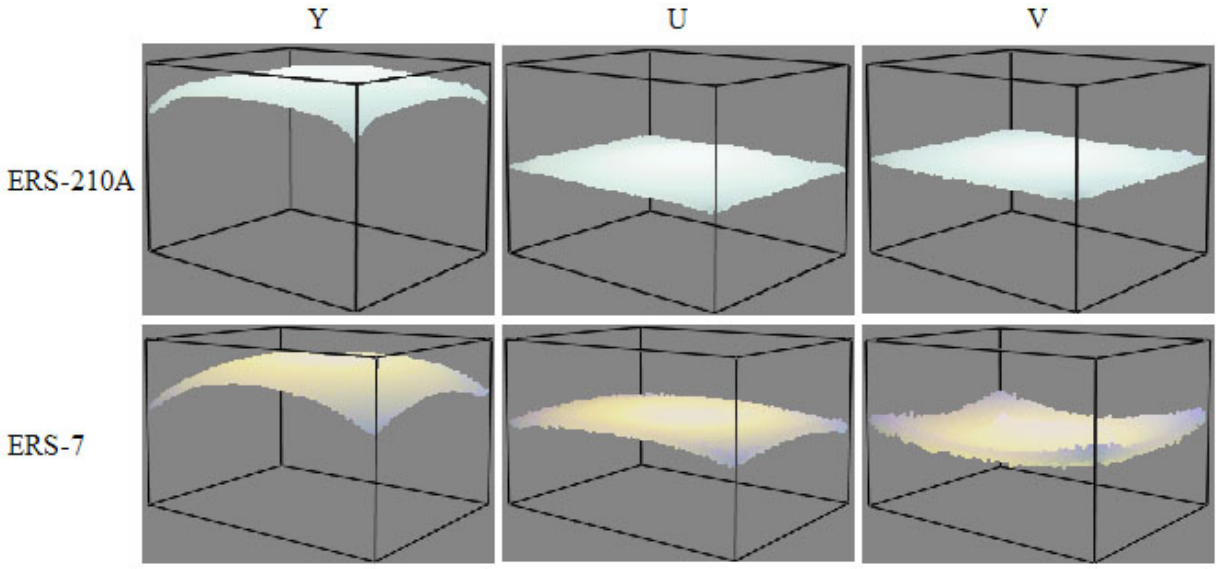


Figure 3.5: Comparison of the image of a white wall, captured by the ERS210 and ERS7 cameras. Y represents the brightness, U and V are the color bands of the image, which is provided by the camera in the YUV color space. The ERS7 images have a chromatic distortion at the corners; they also appear darker even if in this example, the camera gain was set to *high*, while in case of the ERS210A the setting was *low*. See [47] for details.

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:

$$p = \begin{bmatrix} x_w & y_w & z_w & 1 \end{bmatrix} \tag{3.11}$$

- image coordinates:

$$q = \begin{bmatrix} f \cdot x_i & f \cdot y_i & f \end{bmatrix} \tag{3.12}$$

- intrinsic parameters:

$$K = \begin{bmatrix} \sigma_x & \sigma_\theta & u_0 \\ 0 & \sigma_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.13}$$

- extrinsic parameters:

$$M = \begin{bmatrix} \ddots & \vdots & \iddots & \vdots \\ \cdots & R & \cdots & T \\ \iddots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.14}$$

- then the result is:

$$q = K \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot M \cdot p \tag{3.15}$$

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 [25] and [42]:

$$\begin{aligned} x' &= x + \rho_1 \bar{x} r^2 + \rho_2 \bar{x} r^4 + \rho_3 \bar{x} r^6 + \tau_1 (2\bar{x}^2 + r^2) + 2\tau_2 \bar{x}\bar{y} \\ y' &= y + \rho_1 \bar{y} r^2 + \rho_2 \bar{y} r^4 + \rho_3 \bar{y} r^6 + \tau_2 (2\bar{y}^2 + r^2) + 2\tau_1 \bar{x}\bar{y} \end{aligned} \tag{3.16}$$

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 an individual calibration for all the robots in the German Team at this time wasn't feasible. As far as the non-linear distortion is concerned, the

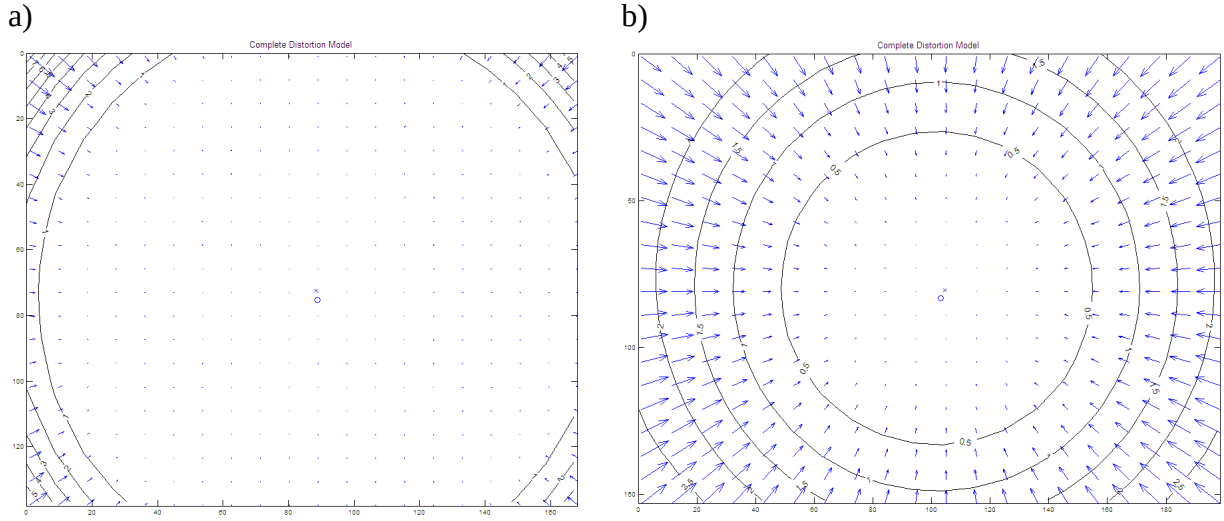


Figure 3.6: Complete camera distortion model. The blue arrows show the direction of the pixel displacement, while the magnitude is illustrated by the black iso-lines. a) ERS210: while most of the image exhibits negligible distortion, in the corners the error arrives up to 7 pixels of displacement b) 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 3.6 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. To be able to observe the characteristics of this vignetting effect, we wanted to capture from the robot's camera images of uniformly colored objects, lit by a diffuse light source (in order to minimize the effects of shadows and reflections). The blue and yellow goals from the RoboCup official field seemed to be very good candidates, as these 2 colors are opposite in the UV plane, and they are close to the high and low extremes values which can be reached in the U color channel, the one which turned out to be affected by the highest distortion. As can be seen in Figure 3.7, the radiometric distortion d_i for a given spectrum i of a reference

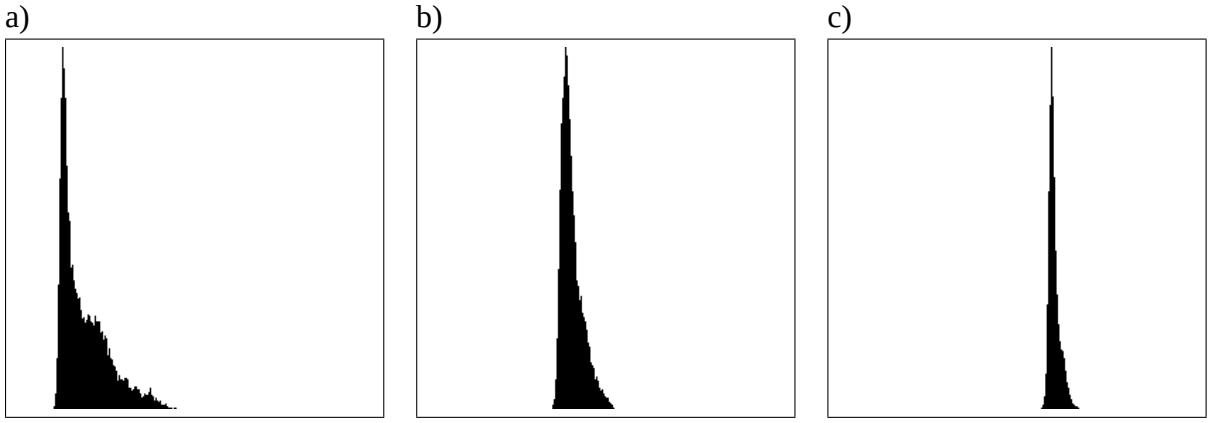


Figure 3.7: Histograms of the U color band for uniformly colored images: yellow (a), white (b) and skyblue (c). In case of little or no vignetting effect, each histogram should exhibit a narrow distribution around the mode.

color I is dependent on it's actual value (*brightness component*):

$$d_i(I) \propto \lambda_i(I_i) \tag{3.17}$$

Moreover, the chromatic distortion that applies on a certain pixel (x, y) appears to be also dependent on its distance from a certain point, center of distortion (u_d, v_d), which lies approximately close to the optical center of the image, the principal point; so, let $r = \sqrt{(x-u_d)^2 + (y-v_d)^2}$, then (*radial component*, cf. Fig. 3.8):

$$d_i(I(x,y)) \propto \rho_i(r(x,y)) \tag{3.18}$$

Putting it all together:

$$d_i(I(x,y)) \propto \rho_i\left(r(x,y)\right) \cdot \lambda_i\left(I_i(x,y)\right) \tag{3.19}$$

Now, we want to derive $\rho_i, \lambda_i, \forall i \in \{Y, U, V\}$ from a set of sample pictures; since both sets of functions are non-linear, we decided to use a polynomial approximation, whose coefficients can be estimated using least-square optimization techniques:

$$\begin{aligned} \rho_i\left(r\right) &= \sum_{j=0}^{n} \varrho_{i,j} \cdot r^j \\ \lambda_i\left(I_i\right) &= \sum_{j=0}^{m} l_{i,j} \cdot I_i^j \end{aligned} \tag{3.20}$$

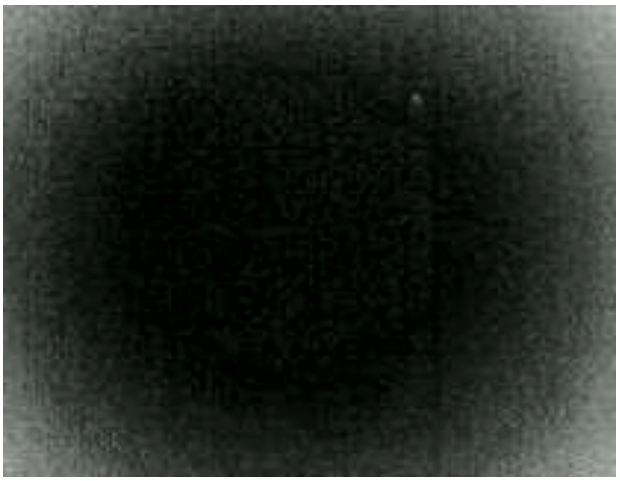


Figure 3.8: Brightness distribution of the U color band for a uniformly yellow colored image.

In order to do so, we have to create a log file containing reference pictures which should represent different points belonging to the functions that we want to estimate, hence we chose to use uniform yellow, blue, white and green image taken under different lighting conditions and intensities. Then, the log file is processed in the following steps:

- For each image, a reference value is estimated for the 3 spectra Y, U, V, as the modal value of the corresponding histogram, hence the value with the highest number of occurences
- The reference values are clustered into classes, such that series of images representing the same object under the same lighting condition have a single reference value; this is achieved using a first order linear Kalman filter to track the current reference values for the 3 image spectra, and a new class is generated when:

$$\exists j \in \{Y, U, V\} : \left\langle \left| r^m_{j,k} - r^p_{j,k-1} \right| > \vartheta \right\rangle \tag{3.21}$$

  where $r^m_{j,k}$ is the reference (for spectrum j) measured at frame k, $r^p_{j,k-1}$ is the reference predicted by the Kalman filter at frame $k-1$, and $\vartheta = 40$ is a confidence threshold.

- Simulated annealing ([52]) is used to derive the coefficients $\varrho_{i,j}$, $l_{i,j}$ in a separate process for each color band $\forall i \in \{Y, U, V\}$
- In each step, the coefficients $\varrho_{i,j}, l_{i,j}$ are mutated by the addition of zero mean gaussian noise, the variance is dependent on the order of the coefficients, such that high order coefficients have increasingly smaller variances than low order ones
- The mutated coefficients are used to correct the image, as:

$$I'_i(x, y) = I_i(x, y) - \rho_i \left(r(x, y) \right) \cdot \lambda_i \left(I_i(x, y) \right) \tag{3.22}$$

- For each image $I_{i,k}$ in the log file (i is the color band, k the frame number), given its reference value previously estimated $r_k(i)$, the current "energy" E for the annealing process is calculated as:

$$E_i = \sum_{(x,y)} \left(I'_{i,k}(x,y) - r_{i,k} \right)^2 \tag{3.23}$$

- The "temperature" T of the annealing is lowered using a linear law, in a number of steps which is given as a parameter to the algorithm to control the amount of time spent in the optimization process; the starting temperature is normalized relative to the initial energy
- The correction function learned off-line is stored in a look-up table for a fast execution on the robot.

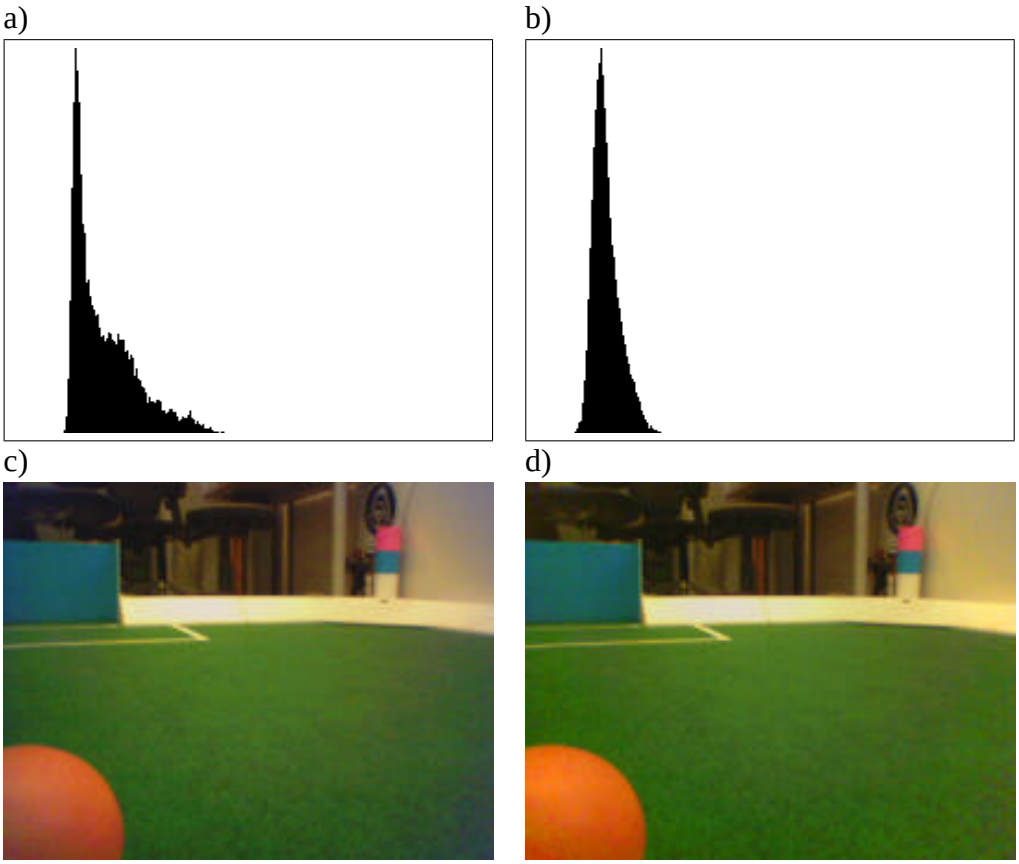


Figure 3.9: Color correction in practice: histograms of the U color band of a uniformly yellow colored image before correction(a), and after (b); actual image taken from a game situation, before correction (c) and after (d)

Figure 3.9 shows some examples of corrections obtained after running the algorithm on a log file composed of 8 image classes (representing different colors at different lighting conditions) of 29 images each, decreasing the temperature to 0 in 100 steps, for a total optimization time of 7 minutes (Java application, non speed optimized, Pentium4 2.66GHz).

3.2.4 Detecting Points on Edges

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 and borders 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 the border is a bit more complicated. In most cases, the border has a bigger size in the image than a field line. But a far distant border might be smaller than a very close field line. 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 a border. The projection of the pixels on the field plane is also used to determine their relative position to the robot.

If less than three points of a certain edge type are detected in the image, these points are ignored to reduce noise. With the introduction of the ERS7 in the league another problem appeared. The white body and the legs of a robot could lead to the detection of points classified as field lines or border. Therefore another operation for filtering those points was necessary.

For every point classified as field line or border, the gradient of the y-channel is computed (cf. Fig. 3.10b,c). This gradient is based on the values of the y-channel of the edge point and three neighbouring pixels, using a Roberts operator ([51]):

$$\begin{aligned} s &= I\left[\begin{array}{cc} x+1 & y+1 \end{array}\right] - I\left[\begin{array}{cc} x & y \end{array}\right] \\ t &= I\left[\begin{array}{cc} x+1 & y \end{array}\right] - I\left[\begin{array}{cc} x & y+1 \end{array}\right] \\ |\nabla I| &= \sqrt{s^2+t^2} \\ \angle\nabla I &= \arctan\left(\tfrac{s}{t}\right) - \tfrac{\pi}{4} \end{aligned} \tag{3.24}$$

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

In the next step, all points of a certain type are compared pairwise, if they could belong to the same line based on the position and the gradient. The segment with the most points is accepted to be a valid line segment. The rest of the points are again grouped and the next segment is determined. This continues until the resulting segments consist of less than three points. With this operation, single incorrectly detected points are skipped and not used in any further step.

3.2.5 Detecting the Ball

While scanning along the grid, the longest run of orange pixels is detected. Starting at the middle of this run, the *ball specialist* scans the image in horizontal, vertical and both diagonal directions for the edge of the ball (cf. Fig. 3.10a). Each of these eight scanlines stops, if one of the following conditions is fulfilled:

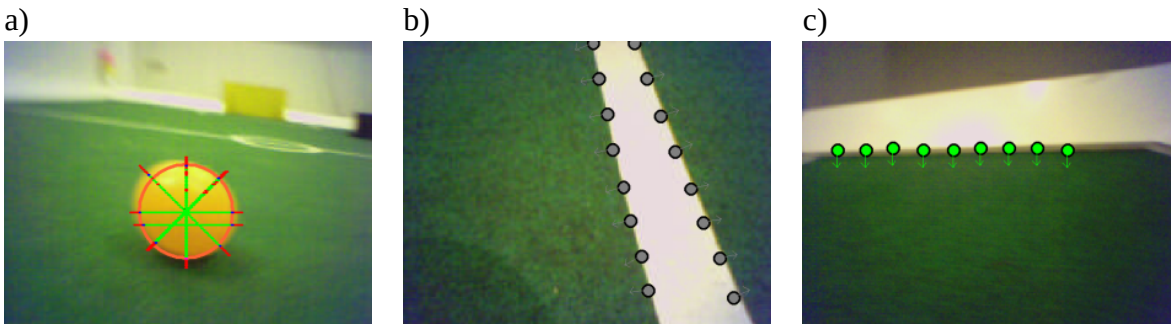


Figure 3.10: Percepts: a) The ball percept along with the scanlines of the ball specialist. b) Points on the edges of a line. c) Points on the edge of a border. The images b) and c) also show the computed gradients of the edges.

- 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. [11]) 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.

Compared to the ball detection used in 2003, the new method improved the recognition of balls in various situations. The ball is still detected at greater distances to the robot, balls with highlights and shadows can be detected in most cases and the centerpoint and radius of balls partially hidden behind other objects (e. g. robots) is computed correctly instead of detecting a small ball only covering the visible part of the real ball. The distance to the ball can now be calculated using the radius of the circle recognised in the image. This proves much more accurate than the projection of the line of sight to the ground, which depends on the error-prone sensor data measuring the positions of the head-joints.

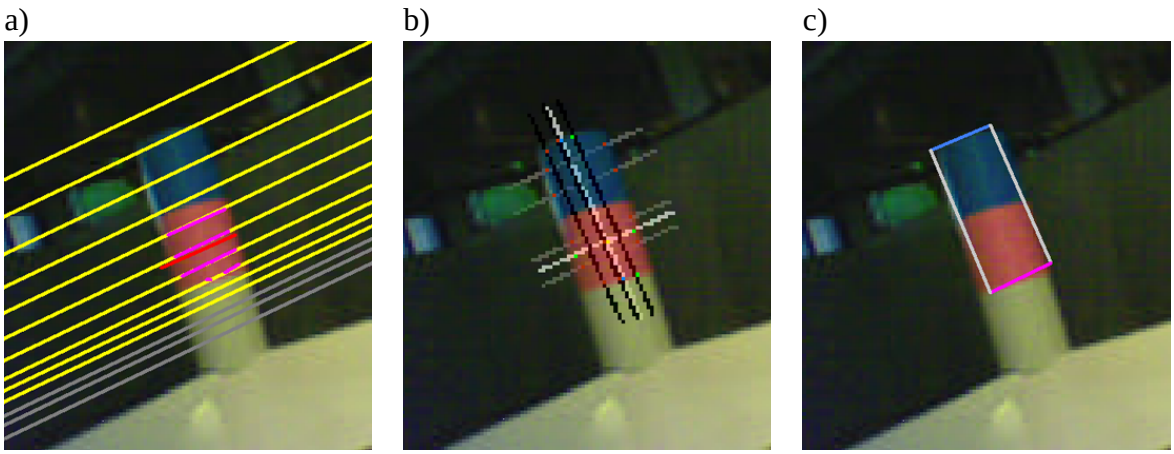


Figure 3.11: Three steps in beacon detection: a) Scanlines searching for pink runs. The pink segments are the detected pink runs, the red segment is the result of clustering. b) The specialist detects the edges of the beacon. c) The generated percept.

3.2.6 Detecting Beacons

To scan and detect the beacons, a *beacon specialist* is used. This module generates its own scanlines (which are looking only for pink) parallel to the horizon, with a density that decreases as the distance to the horizon increases; the pink runs found this way are clustered depending on their horizontal and vertical proximity, resulting in a projected scanline which passes close to the middle of the pink part of the flag (cf. Fig. 3.11a). 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 [55]); depending on the number of edge points found and the sanity checks which are met, like the ratio between the horizontal and vertical length of each part of a beacon or the consistency of the flag type hypotheses of all the scanlines, a reliabilty value is computed and compared with a confidence threshold to decide whether the object in question is a beacon or not. In case such a check is successfull, the center of the pink part is calculated, and it's passed, along with the flag type (PinkYellow, YellowPink, PinkSkyBlue, SkyBluePink) to the *flag specialist*, which will use this information as a starting point. From the initialization pixel the image is scanned for the border of the flag to the top, right, down, and left where top/down means perpendicular to the horizon and left/right means parallel to the horizon. This leads to a first approximation of the size of the flag. Two more horizontal lines are scanned in the pink part and if the flag has a yellow or a sky-blue part, two more horizontal lines are also scanned there. To determine the height of the flag, three additional vertical lines are scanned (cf. Fig. 3.11b). The leftmost, rightmost, topmost, and lowest points found by these scans determine the size of the flag (cf. Fig. 3.11c). The angles to the four edges of the flag are written into the *PerceptCollection*.

[1] This trimmed down version doesn't perform the non-maxima suppression and binary thinning.

a) b)

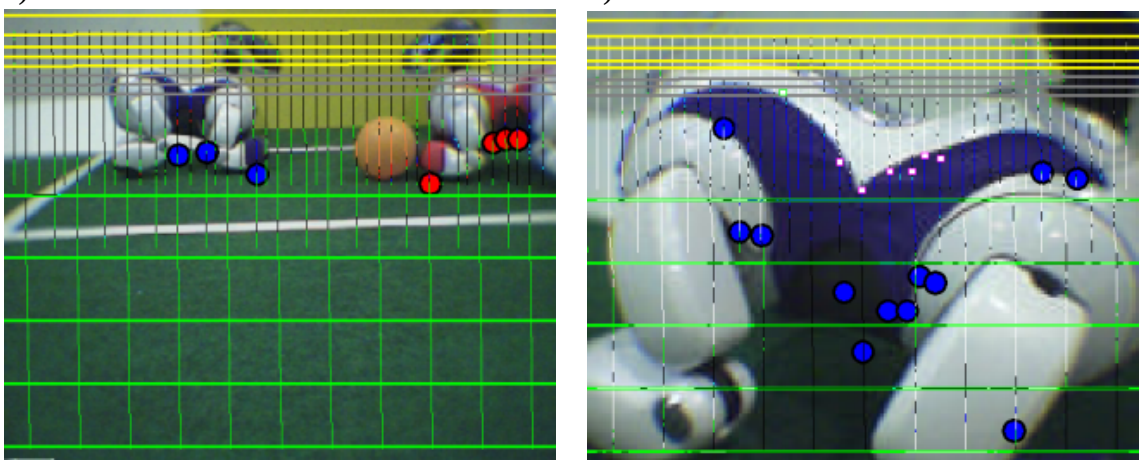


Figure 3.12: Recognition of other robots. a) Several foot points for a single robot are clustered (shown in red and blue). b) Close robots are recognized based on the upper border of their tricot (shown in pink).

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

3.2.7 Detecting Goals

A *goal specialist* measures the height and the width of a goal. The image is scanned for the borders of the goal from the left to the right (cf. Fig. 3.2c) and, if any indications for a goal have been found, from the top to the bottom, where again top/down means perpendicular to the horizon and left/right parallel to the horizon.

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

3.2.8 Detecting Robots

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

- If the number of pixels in tricot color found on a scan line is above a certain threshold, it is assumed that the other robot is close. In that case, the upper border of its tricot (ignoring the head) is used to determine the distance to that robot (cf. Fig. 3.12b). As with many other percepts, this is achieved by intersecting the view ray through this pixel with a plane that is parallel to the field, but on the “typical” height of a robot tricot. As the other robot

a) b)

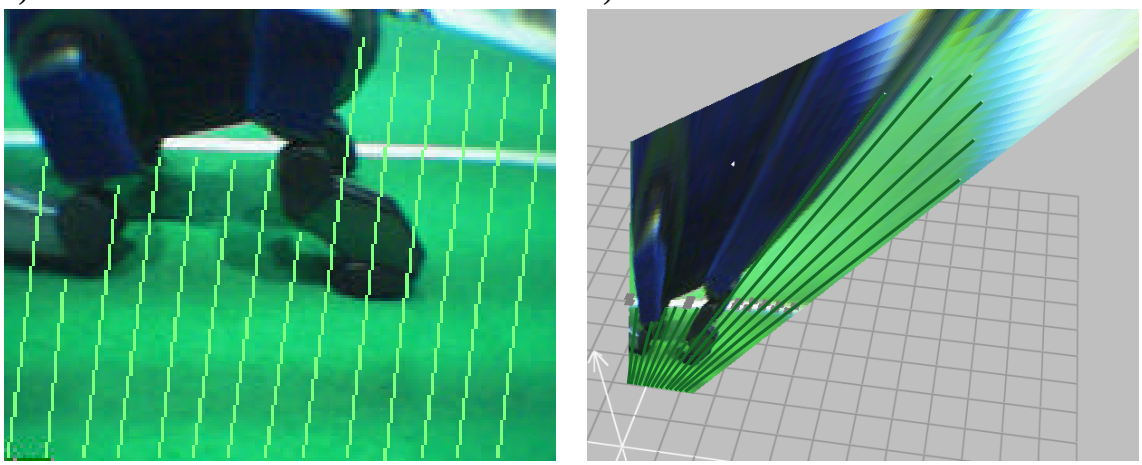


Figure 3.13: Obstacles Percept. a) An image with an obstacle. Green lines: projection of the obstacles percept to the image. b) The projection of the image to the ground. Green lines: obstacles percept.

is close, a misjudgment of the "typical" tricot height does not change the result of the distance calculation very much. As a result, the distance to close robots can be determined.

- If the number of pixels in tricot color found is smaller, it is assumed that the other robot is further away. In that case, the scan lines are followed until the green of the field appears (cf. Fig. 3.12a). Thus the *foot points* of the robot are detected. From these foot points, the distance to the robot can be determined by intersecting the view ray with the field plane. As not all foot points will actually be below the robot's feet (some will be below the body), they are clustered and the smallest distance is used.

During RoboCup 2004, the GermanTeam did not use the detection of other robots. This was achieved through leaving the color classes *red* and *blue* void. The behavior was solely based on detected obstacles.

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 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. 3.13a) 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. 3.13b). If the part of the projection of

a) b)

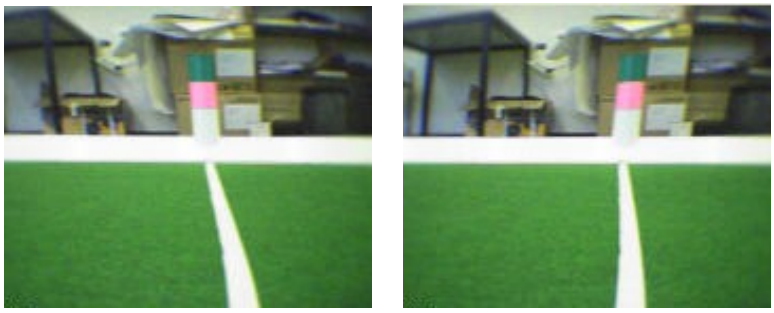


Figure 3.14: Images taken while the head is quickly turning. a) Left. b) Right.

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 CCD chip. This has an impact on the images if the camera is moved while an image is taken. For instance in Figure 3.14 it can be seen, that the flag is slanted in different directions depending on whether the head is turning left or right. In experiments it was recognized that the timestamp attached to the images by the operating system of the Aibo corresponds to the time when the lowest row of the image was taken. Therefore, features in the upper part of the image were recorded significantly earlier. It is assumed that the first image row is recorded shortly after taking the previous image was finished, i. e. 10% of the interval between two images, so 90% of the overall time are spent to take the images. For the ERS-7, this means that the first row of an image is recorded 30 ms earlier than the last row. If the head, e. g., is rotating with a speed of 180°/s, this results in an error of 5.4° for bearings on objects close to the upper image border. Therefore, the bearings have to be corrected. Since this is a quite time-consuming operation, it is not performed as a preprocessing step for image processing. Instead, the compensation is performed on the level of percepts, i. e. recognized flags, goals, edge points, and the ball. The compensation is done by interpolating between the current and the previous camera positions (the so-called camera matrices) depending on the y image coordinate of the percept.

However, this compensation is not inverted when percepts are projected back to the image for the means of visualization. As a result, percepts sometimes appear next to the pixel information that generated them instead of covering it, but nevertheless their recognition worked fine.

3.3 Self-Localization

The *GT2004 Self-Locator* implements a Markov-localization method employing the so-called Monte-Carlo approach [23]. 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. 3.17a). Each particle can be seen as the hypothesis of the robot being located at this posture. Therefore, such particles mainly consist of a robot pose, i. e. a vector representing the robot's x/y-coordinates in millimeters and its rotation θ in radians:

$$pose = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} \tag{3.25}$$

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

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

3.3.1 Motion Model

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

$$\Delta_{error} = \begin{pmatrix} 0.1d \times \text{random}(-1 \ldots 1) \\ 0.02d \times \text{random}(-1 \ldots 1) \\ (0.002d + 0.2\alpha) \times \text{random}(-1 \ldots 1) \end{pmatrix} \tag{3.26}$$

In equation (3.26), d is the length of the odometry offset, i. e. the distance the robot walked, α is the angle the robot turned.

For each sample, the new pose is determined as

$$pose_{new} = pose_{old} + \Delta_{odometry} + \Delta_{error} \tag{3.27}$$

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

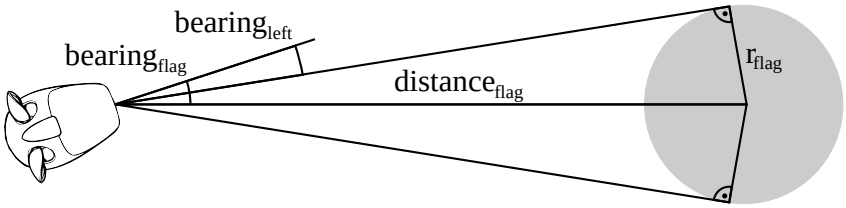


Figure 3.15: Calculating the angle to an edge of a flag.

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 flags 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 flags and goals delivered by the flag/goal specialist (cf. Sect. 3.2.6 and 3.2.7), and points on edges between the field and the field lines, the field wall, and the goals. These have to be related to the assumed bearings from hypothetical postures. To determine the expected bearings for flag and goal edges, 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 flags, edges of goals, and edge points:

3.3.2.1 Flags

The calculation of the bearing on the center of a flag is straightforward. However, to determine the angle to the left or right edge, the bearing on the center $bearing_{flag}$, the distance between the assumed camera pose and the center of the flag $distance_{flag}$, and the radius of the flag r_{flag} are required (cf. Fig. 3.15):

$$bearing_{left/right} = bearing_{flag} \pm \arcsin(r_{flag}/distance_{flag}) \tag{3.28}$$

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, field wall, yellow goal, or blue goal), and by projecting it on the field, a relative offset from the body center of the robot is determined. Note that the calculation of the offsets is prone to errors because the pose of the camera cannot be determined precisely. In fact, the farther away a point is, the less precise the distance can be determined. However, the precision of the direction to a certain point is not dependent on the distance of that point.

Lines only provide localization information perpendicular to their orientation. Therefore, the four edge types provide very different information: *The field lines* are mostly oriented across the field, but the side lines of the penalty area also provide important information. The field lines are seen less often than the field wall. *The field wall* is surrounding the field. Therefore it provides information in both Cartesian directions, but it is often quite far away from the robot. Therefore, the distance information is less precise than the one provided by the field lines. The field wall is seen from nearly any location on the field. *Goals* are the only means to determine the orientation on the field, because the field lines and the field wall are mirror symmetric. The goals are seen only rarely. It turned out that the vision system is reliably able to determine the orientation of field lines, while the orientation of the edge between field wall and field is not as stable. Therefore, it is distinguished between field lines along the field and field lines across the field. This especially improves the localization of the goalie, because it sees both types of lines surrounding the penalty area.

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

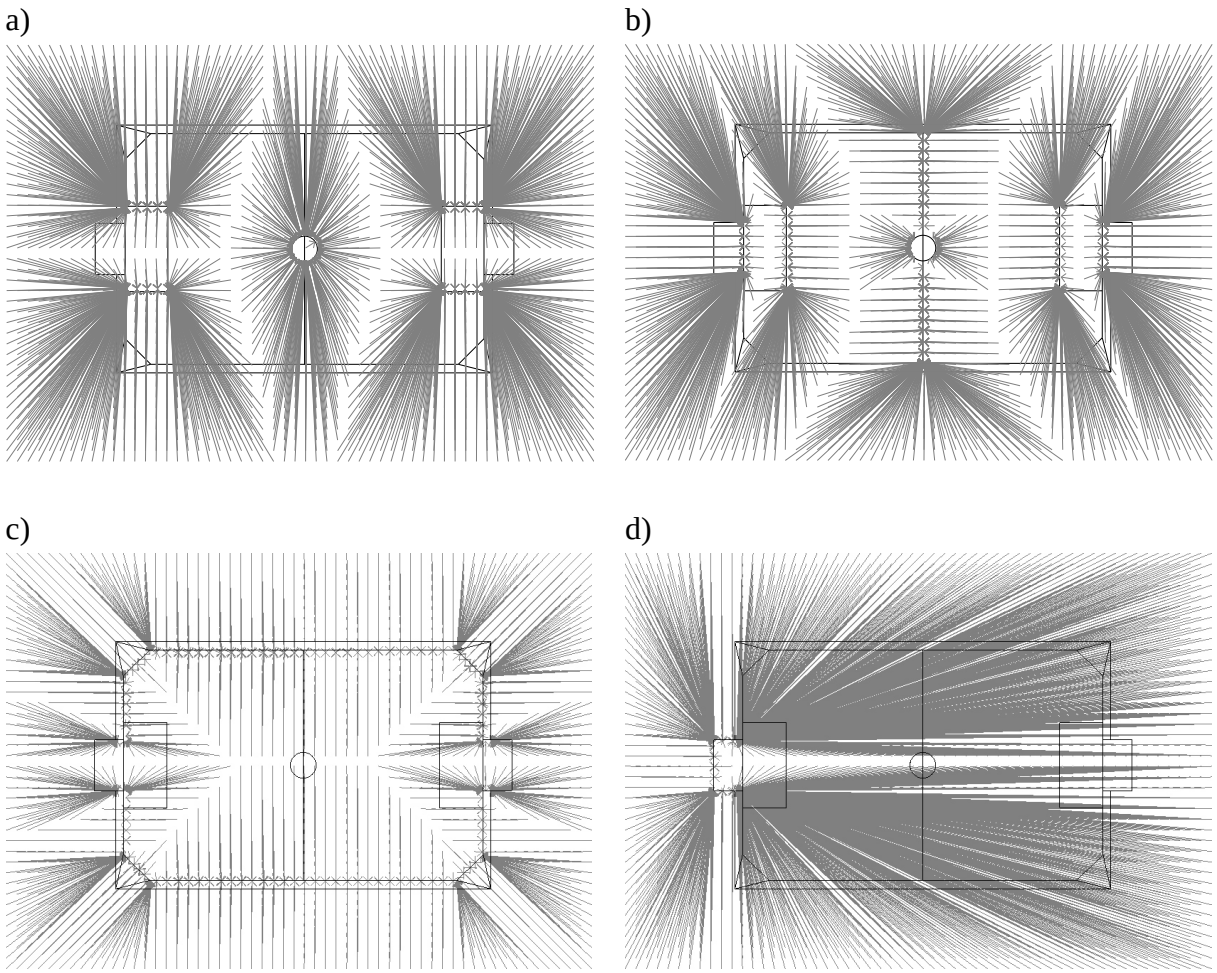


Figure 3.16: Mapping of positions to closest edges. a) Field lines along the field. b) Field lines across the field. c) Field wall. c) A goal.

Closest Model Points. The projections of the pixels are used to determine the three probabilities of each sample in the Monte-Carlo distribution. As the positions of the samples on the field are known, it can be determined for each measurement and each sample, where the measured points would be located on the field if the position of the sample was correct. For each of these measured points in field coordinates, it can be calculated, where the closest point on a real field line of the corresponding type is located. Then, the horizontal and vertical angles from the camera to this model point are determined. These two angles of the model point are compared to the two angles of the measured point. The smaller the deviations between the model point and the measured point from a hypothetic position are, the more probable the robot is really located at that position. Deviations in the vertical angle (i. e. distance) are judged less rigidly than deviations in the horizontal angle (i. e. direction).

Calculating the closest point on an edge in the field model for a small number of measured points is still an expensive operation if it has to be performed for, e. g., 100 samples. Therefore,

the model points are pre-calculated for each edge type and stored in two-dimensional lookup tables with a resolution of 2.5 cm. That way, the closest point on an edge of the corresponding type can be determined by a simple table lookup. Figure 3.16 visualizes the distances of measured points to the closest model point for the four different edge types.

3.3.2.4 Probabilities for Flags and Goals

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

$$s(\omega_{measured}, \omega_{expected}) = \begin{cases} \mathrm{e}^{-50d^2} & \text{if } d < 1 \\ \mathrm{e}^{-50(2-d)^2} & \text{otherwise} \end{cases} \quad \text{where } d = \frac{|\omega_{measured} - \omega_{expected}|}{\pi} \tag{3.29}$$

The probability $q_{landmarks}$ of a certain particle is the product of these similarities:

$$q_{landmarks} = \prod_{\omega_{measured}} s(\omega_{measured}, \omega_{expected}) \tag{3.30}$$

3.3.2.5 Probabilities for Edge Points

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:

$$s(\omega_{seen}, \omega_{exp}, \sigma) = \mathrm{e}^{-\sigma(\omega_{seen} - \omega_{exp})^2} \tag{3.31}$$

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

$$q_{edge\ type} = s(\alpha_{seen}, \beta_{seen}, \alpha_{exp}, \beta_{exp}) = s(\alpha_{seen}, \alpha_{exp}, 10 - 9\frac{|v|}{200}) \cdot s(\beta_{seen}, \beta_{exp}, 100) \tag{3.32}$$

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

Calculating the probability for all points on edges found and for all samples in the Monte-Carlo distribution would be a costly operation. Therefore, only three points of each edge type

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

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

$$q'_{new} = \begin{cases} q'_{old} + \Delta_{up} & \text{if } q > q'_{old} + \Delta_{up} \\ q'_{old} - \Delta_{down} & \text{if } q < q'_{old} - \Delta_{down} \\ q & \text{otherwise.} \end{cases} \tag{3.33}$$

For landmarks, $(\Delta_{up}, \Delta_{down})$ is $(0.1, 0.05)$, for edge points, it is $(0.01, 0.005)$

3.3.2.6 Overall Probability

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

$$p = q'_{landmarks} \cdot q'_{field\ lines} \cdot q'_{field\ walls} \cdot q'_{goals} \tag{3.34}$$

3.3.3 Resampling

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

3.3.3.1 Importance Resampling

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

3.3.3.2 Drawing from Observations

In a second step, some samples are replaced by so-called candidate postures. This approach follows the *sensor resetting* idea of Lenser and Veloso [34], and it can be seen as the small-scale version of the Mixture MCL by Thrun *et al.* [56]: 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 flags seen. Estimated distances to these landmarks and odometry are used to update the bearings on these memorized flags when the robot moves. Bearings on goal posts are not inserted into the buffer, because their distance information is not reliable enough to be used to compensate for the motion of the robot. However, the calculation of the current posture also integrates the goal posts, but only the ones actually seen. So from the buffer and the bearings on goal posts, all combinations of three bearings are used to determine robot postures by triangulation.

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

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.

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

$$\frac{rnd}{n}\sum_{i}^{n} p_i > p_j \tag{3.35}$$

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

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

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 sample is, the less it is moved. This can be seen as a probabilistic random search for the best position, because the samples that are randomly moved closer to the real position of the robot will be rewarded by better probabilities during the next observation update steps, and they will therefore be more frequent in future distributions.

The samples are moved according to the following equation:

$$pose_{new} = pose_{old} + \begin{pmatrix} 100(1-p') \times \text{random}(-1 \ldots 1) \\ 100(1-p') \times \text{random}(-1 \ldots 1) \\ 0.5(1-p') \times \text{random}(-1 \ldots 1) \end{pmatrix} \tag{3.36}$$

3.3.4 Estimating the Pose of the Robot

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

3.3.4.1 Finding the Largest Cluster

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

3.3.4.2 Calculating the Average

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

$$\theta_{robot} = \text{atan2}(\sum_i \sin\theta_i, \sum_i \cos\theta_i) \tag{3.37}$$

3.3.4.3 Certainty

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

$$c = \frac{m}{n} \cdot \frac{1}{m} \sum_i p'_i = \frac{1}{n} \sum_i p'_i \tag{3.38}$$

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.

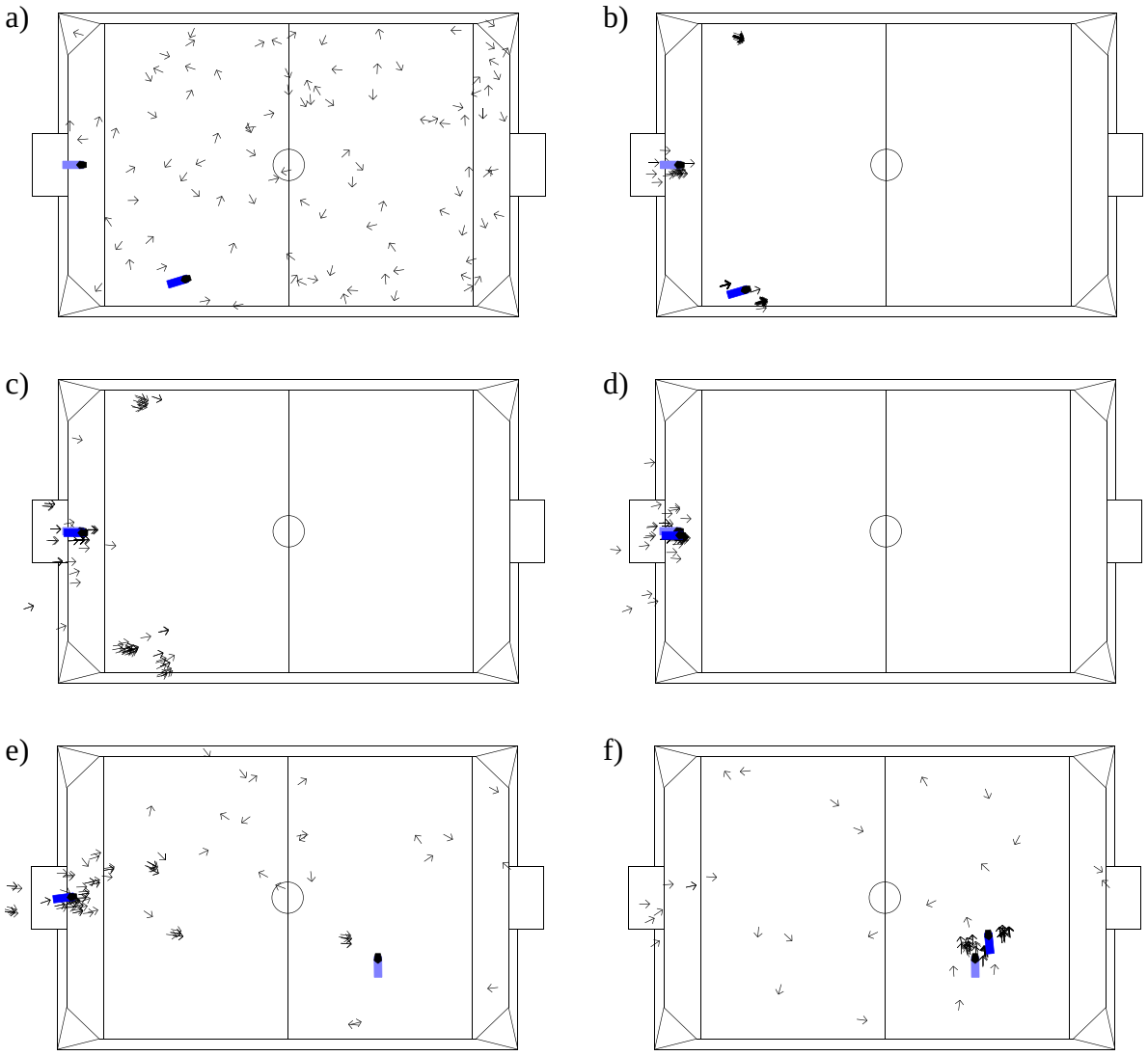


Figure 3.17: Distribution of the samples during the Monte-Carlo localization while turning the head. The bright robot body marks the real position of the robot, the darker body marks the estimated location. a) After the first image processed (40 ms). b) After eight images processed (320 ms). c) After 14 images (560 ms). d) After 40 images (1600 ms). e) Robot manually moved to another position. f) 13 images (520 ms) later.

3.3.5 Results

Figure 3.17 depicts some examples for the performance of the approach using 100 samples. The experiments shown were conducted with the simulator (cf. Sect. 5.1) and an older version of the self-locator that only used flags and goals for localization on the field used in 2002. 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, and Lisbon, the method also proved to work very well

on real robots. In 2002, the GermanTeam was the only team that supported all features of the RoboCup Game Manager that allows the referee to give instructions to the robots. This includes automatic positioning on the field, e. g. for kickoff. For instance, the robots of the GermanTeam were just started somewhere on the field, and then—while still many people were working on the field—they autonomously walked to their initial positions. Even in 2004, the majority of the teams are not able to do this. In addition, the self-localization worked very well on fields without an outer barrier, e. g. on the practice field in Fukuoka. In 2003, it was also demonstrated that the GermanTeam can play soccer without the flags.

3.4 Ball Modeling

3.4.1 Ball Position and Ball Speed

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

The ball's position is derived geometrically from the "ball percept" taking into account the robot's pose. The ball speed is calculated from the current and the last ball position perceived.

3.4.2 Kalman Filtering of Ball Percepts

To reduce the measurement's noise a Kalman filter is applied to ball position and speed derived from the "ball percept". The implemented Kalman filter uses a discrete process model of the form

$$\mathbf{x}_{t+\Delta t} = \mathbf{A} \cdot \mathbf{x}_t$$

where $\mathbf{x}_t$ is the actual state of the ball and $\mathbf{x}_{t+\Delta t}$ is the predicted state of the ball after time Δt.

The implementation uses a constant speed model. Therefore state vector $\mathbf{x}$ and matrix $\mathbf{A}$ are chosen as

$$\mathbf{x} = \begin{pmatrix} p_x \\ p_y \\ v_x \\ v_y \end{pmatrix} \qquad \mathbf{A} = \begin{pmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where p_x and p_y specify the position and v_x and v_y specify the speed of the ball.

The Kalman filter algorithm consists of two steps. The first step is the time update (prediction):

$$\mathbf{x}^-_{t+\Delta t} = \mathbf{A}\mathbf{x}_t \tag{3.39}$$

$$\mathbf{P}^-_{t+\Delta t} = \mathbf{A}\mathbf{P}_t\mathbf{A}^T + \mathbf{Q} \tag{3.40}$$

Variables indexed with t are the results of the previous cycle of the algorithm, variables with index $t + \Delta t$ are the results of the current cycle. Matrix $\mathbf{Q}$ is the process covariance matrix

that specifies the process model noise that is expected. Matrix $\mathbf{P}$ is the internal filter covariance matrix that is maintained by the filter and should be initialized with high values.

The second step is the measurement update (correction):

$$\mathbf{K} = \mathbf{P}^{-}_{t+\Delta t} \left(\mathbf{P}^{-}_{t+\Delta t} + \mathbf{R} \right)^{-1} \tag{3.41}$$

$$\mathbf{x}_{t+\Delta t} = \mathbf{x}^{-}_{t+\Delta t} + \mathbf{K} \left(\mathbf{z}_{t+\Delta t} - \mathbf{x}^{-}_{t+\Delta t} \right) \tag{3.42}$$

$$\mathbf{P}_{t+\Delta t} = (\mathbf{E} - \mathbf{K}) \mathbf{P}^{-}_{t+\Delta t} \tag{3.43}$$

Matrix $\mathbf{R}$ is the measurement covariance matrix that specifies the measurements noise that is expected. Finally $\mathbf{x}_{t+\Delta t}$ contains the estimated state of the ball.

It is important to choose the parameters of the covariance matrices carefully because they are essential for the quality of the filter results. However, it turned out that it's quite hard to find the optimal parameters for the process and measurement covariance matrices used by the Kalman filter. These difficulties are caused by the frequent changes of the ball state. If the ball is kicked by a robot or bounces against a robot or the field border the real ball state and the ball state predicted by the process model differ. The filter cannot determine if the ball really changed its state or if the measured ball state change is caused by measurement noise.

So if values of the measurement covariance matrix are chosen too high and the ones of the process covariance matrix are chosen too low the ball state calculated by the filter becomes "smoother" but the robot is unable to detect real sudden ball state changes and is less "reactive". In the reverse case (values of measurement covariance matrix too low and values of process covariance matrix too high) the ball becomes "jumpy" because the filter is not able to filter out the measurement noise.

Because it is impossible to find the optimal static values the measurement covariance matrix is made dependent on the distance of the robot to the ball and on the speed of the head. The nearer the ball is to the robot the better are the results of the measurements and the faster the head (and as a result the camera) is moved the worse are the results of the measurements.

So if the ball is far away the measurement noise is high and it is not very important for the robot to be reactive to sudden ball changes. As a result high values are chosen for the measurement covariance matrix (the filter "trusts" the process model prediction more than the measurements). If the ball is near to the robot, the measurement noise is quite low. In this case low values are chosen for the measurement covariance matrix (the filter "trusts" the measurements more than the process model prediction).

The following strategy was used to adapt the measurement covariance matrix $\mathbf{R}$ each cycle of the filter (see figure 3.18):

1. The variances (the diagonal elements of the matrix, squares of parameters a and b in the figure) are calculated by a quadratic polynomial dependent on the distance of the ball to the robot (parameter r in the figure). The remaining elements are set to zero.

2. These variances are increased proportional to the head speed if the speed of the head (and with it the camera) exceeds a threshold.

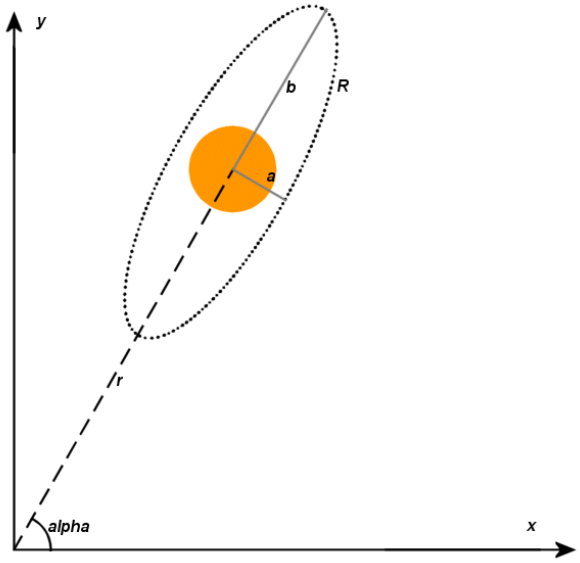


Figure 3.18: Idea of measurement covariance adaption

3. Finally the 2x2 submatrices specifying the position and speed variances are rotated accordingly to the angle of the direction from the robot to the ball (“alpha” in the figure):

$$\mathbf{R}_{sub} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} a^2 & 0 \\ 0 & b^2 \end{pmatrix} \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix}$$

The calculated ball speed was quite good. However, in many cases the ball position calculated by the filter did not reflect the real position of the ball when the ball was manipulated by players. On the one hand that reduced the reactiveness of the robots using the filter and on the other hand it raised a problem with the head motion control that was trying to detect the ball at the position that was calculated by the filter. So finally only the speed calculated by the filter was used, the ball position was adopted directly from the ball percept.

There is still work to be done. The filter cannot be configured easily using RobotControl, it has to be adapted directly in the code. Also the results could probably be improved by doing some fine-tuning. Additionally a dynamic adaptation of the process covariance matrix could lead to further improvements.

3.4.3 Communicated Information about the Ball

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

- The way the robot moves its head: it performs a periodic left-right scanning motion, scanning its surroundings for the ball, other players, and landmarks. Due to the small opening

angle of the robot's camera, the ball cannot be seen by the robot during some intervals of the scanning motion even if the robot is relatively close to the ball.

- The different errors of the ball measurements: while a robot is able to perceive with sufficient accuracy (ball position in coordinates relative to the robot) where the ball is, communicating the ball's position from one robot to another requires the use of a global system of coordinates. Since the robots are only localized within a certain accuracy, the localization errors of both robots accumulate and deteriorate the quality of the information communicated.

If the robot 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.

3.5 Obstacle Model

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. 3.19). 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.

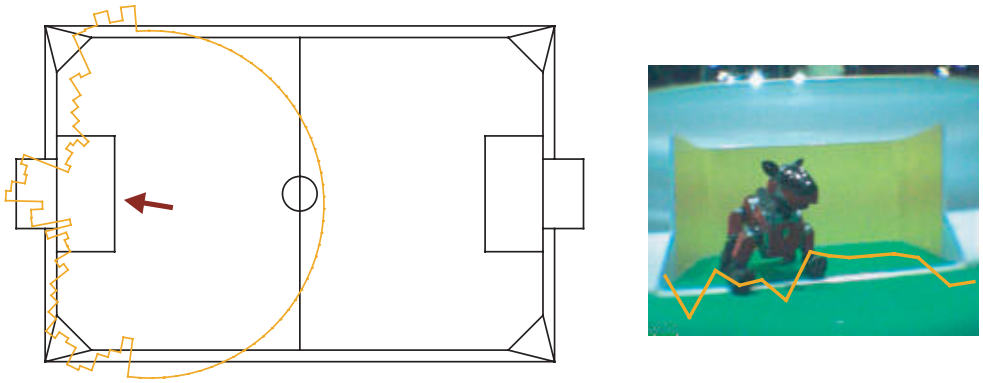


Figure 3.19: The obstacle model as seen from above and projected into the camera image. The robot is in front of the opponent goal.

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.

3.5.1 Updating the Model with new Sensor Data

The robot performs a scanning motion with the camera. The sectors which are within opening angle of the camera can be updated. Image processing can yield two points, the first corresponding to the lower image boundary and the second corresponding to either the distance of 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. 3.20).

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

3.5.2 Updating the Model Using Odometry

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

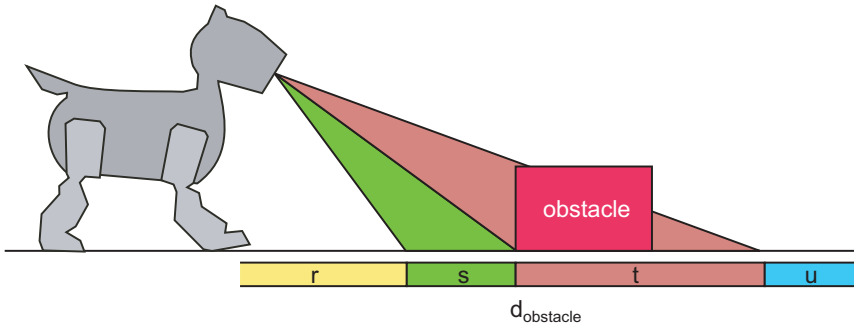


Figure 3.20: The above diagram depicts the robot looking at an obstacle. The robot detects some free space in front of it (s) and some space that is obscured by the obstacle (t). The obstacle model is updated according to the diagram (in this case the distance in the sector is set to $d_{obstacle}$ unless the distance value stored lies in r).

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 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.

3.6 Collision Detector

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

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

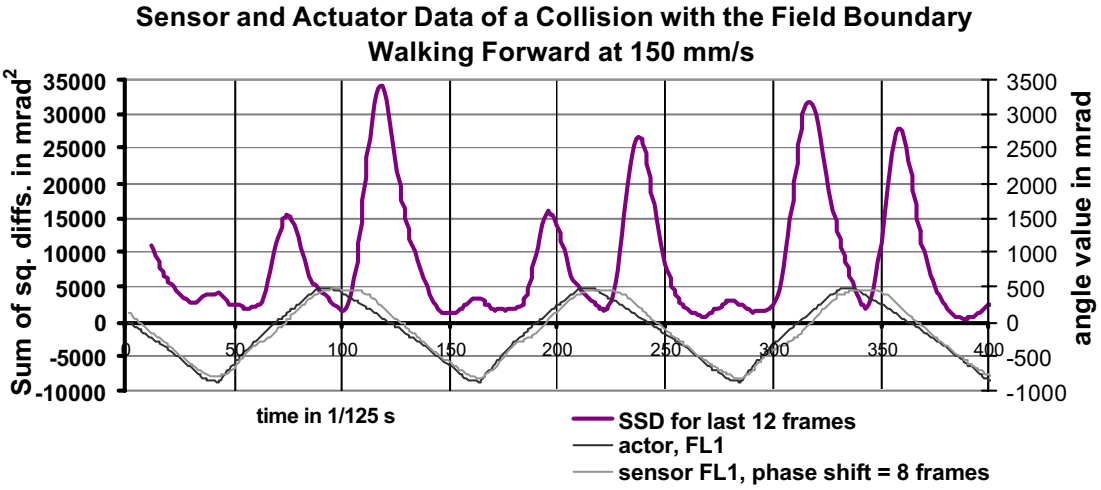


Figure 3.21: The graph shows the actuator and sensor curves and the sum of the squared differences (SSD). Peaks in the SSD curve correspond to collisions.

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. 3.21).

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

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

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

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

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

Usage in the Competitions. For the competitions in Lisbon only the communicated positions of the team members were used. The colortable used for imageprocessing was manipulated, so no percepts were generated for own or opponent players.

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* [38, 39] since 2002 and improved it largely in 2003 and 2004. Appendix E gives an introduction into this system and a complete documentation can be found at the *XABSL* web site [37].

For the German Open 2004, each of the four universities of the GermanTeam used *XABSL* for behavior control and continued the behaviors that were developed by the German Team for Padova 2003. They all followed different approaches:

The Aibo Team Humboldt from the Humboldt-Universität zu Berlin won the German Open 2004. They kept the high-level behaviors from Padova nearly unchanged and mainly focused on ball handling skills. New methods for dribbling and ball grabbing as well as kick selection tables were introduced.

The Darmstadt Dribbling Dackels from the Technische Universität Darmstadt reached the second place at the German Open 2004, also by mainly fine-tuning the GermanTeam's 2003 behaviors.

The Microsoft Hellhounds from the University of Dortmund reached the fourth place with a additional decision making module on top of the *XABSL* system, the *dynamic team tactics*.

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

After the German Open 2004 the behaviors of the teams could be easily merged into a common solution that was continued until the RoboCup 2004 in Lisbon. This section describes in detail the implemented strategies and behaviors of the GermanTeam 2004. Those who are not familar with the *XABSL* language should probably read appendix E first.

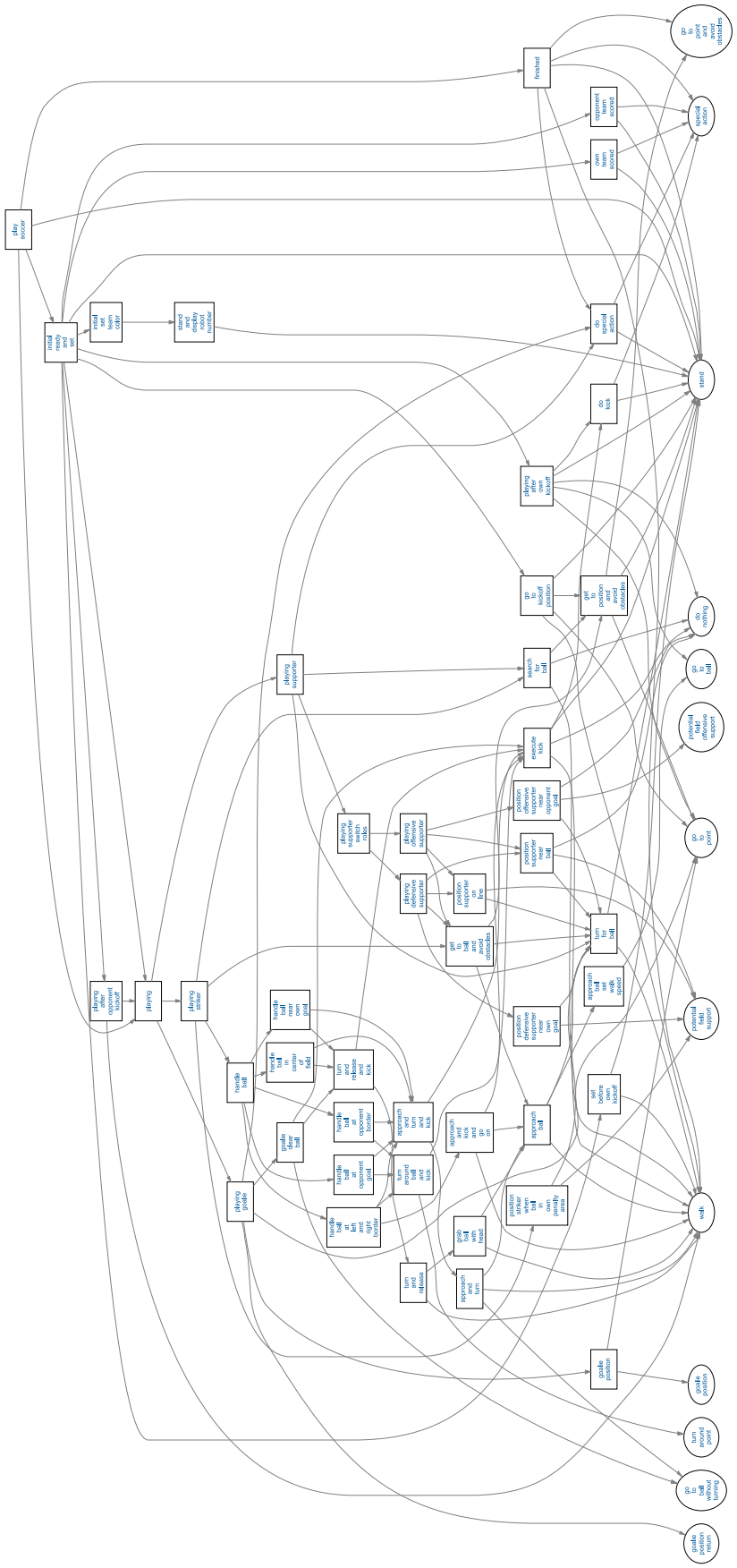


Figure 3.22: The option graph of the soccer-related behaviors of the *GermanTeam*.

The behaviors which the *GermanTeam* developed in *XABSL* for the RoboCup championships 2004 in Lisbon are distributed among about 60 options. Figure 3.22 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 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/gt2004/*. It is recommended to use this site as an additional source to this Section.

3.8.1 Ball Handling

The *GermanTeam* won the 2004 RoboCup world championships due to – besides other things – its sophisticated well tuned ball handling behaviors. They are composed from 18 options and 7 basic behaviors, which looks much. But this section will show how step by step the whole behavior is composed from simple options in a clear and straight forward way.

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. 3.23). 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

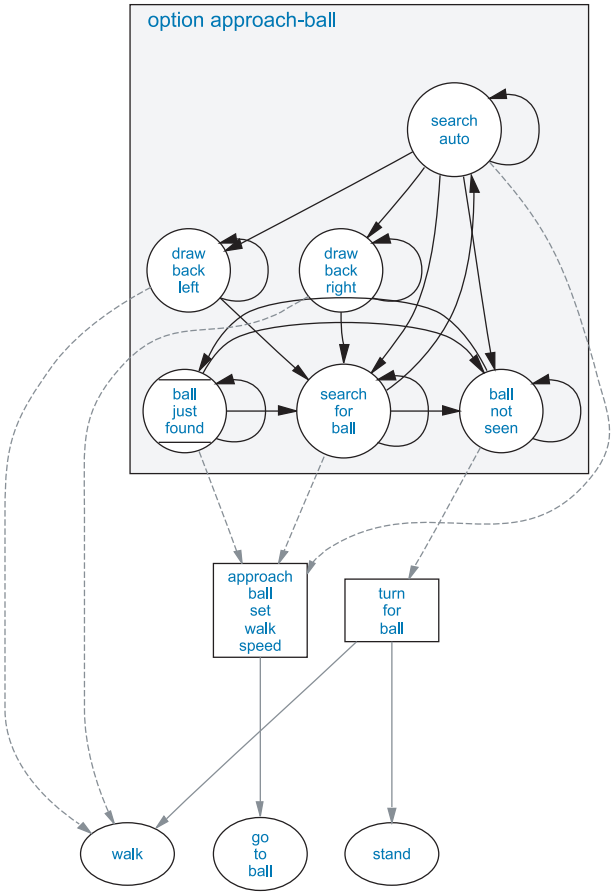


Figure 3.23: Option *"approach-ball"* controls the head movements while approaching the ball and handles collisions and ball losses. The complete documentation of the option can be found at *http://www.ki.informatik.hu-berlin.de/XABSL/examples/gt2004/option.approach-ball.html*.

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 robot is far away from the ball (in state *"search-auto"*) and there is a collision with another robot (detected as described in [26]), in the states *"draw-back-left"* and *"draw-back-right"* the robot walks sideways for a short time to uncouple from the other robot. There is no transition from *"search-for-ball"* to the draw back states in order to give opponent robots no advantage near the ball.

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"*.

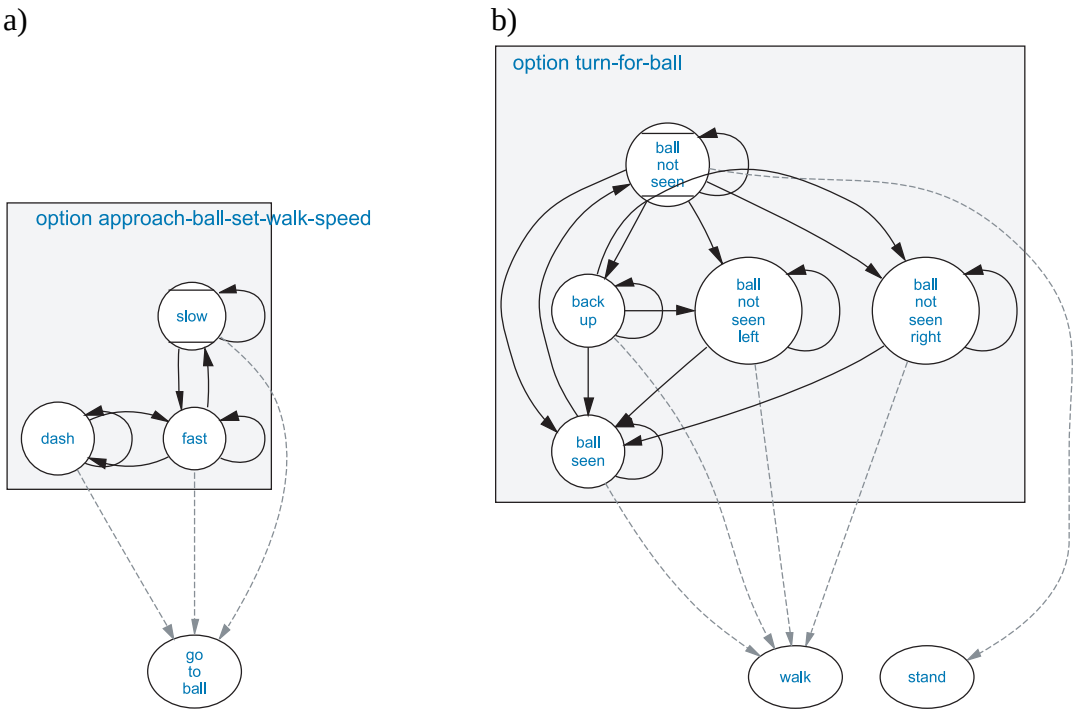


Figure 3.24: a) Option *"approach-ball-set-walk-speed"* controls the speed of ball approaching. b) Option *"turn-for-ball"* tries to redetect a previously lost ball.

Option *"approach-ball-set-walk-speed"* (cf. fig. 3.24a) 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.

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. 3.24b) 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 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

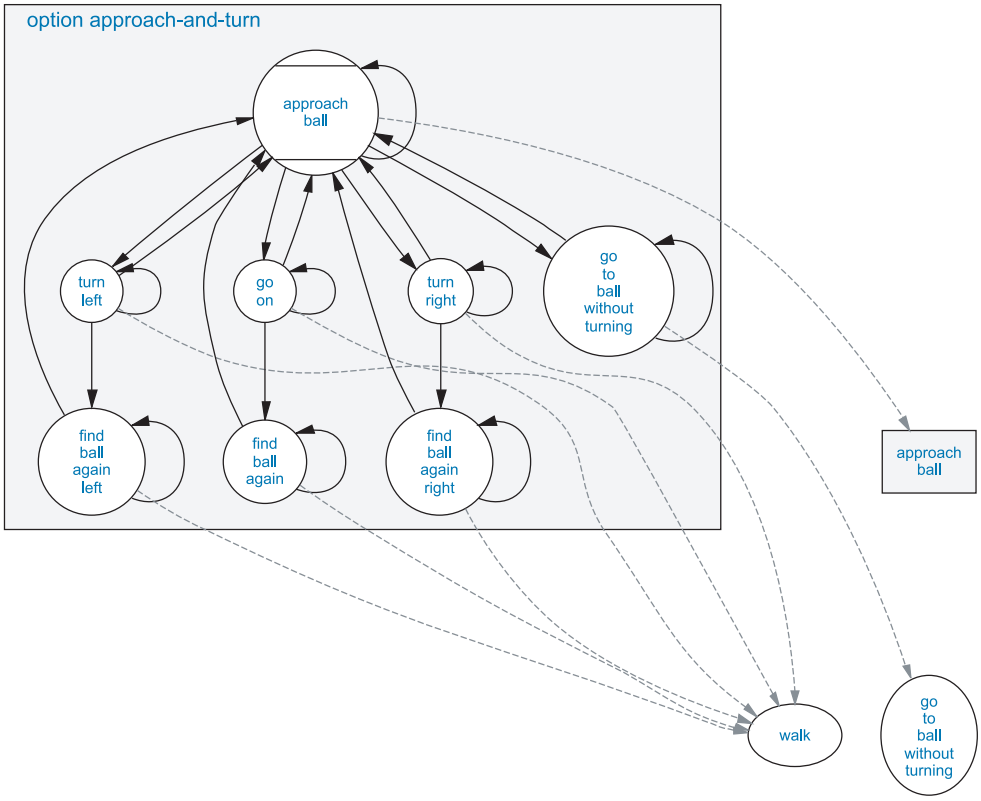


Figure 3.25: Option *"approach-and-turn"* dribbles the ball into given direction by pushing the ball with the chest or pulling it around with the front legs.

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. 3.25) 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 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.

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 and Pushing Backward

The dribbling with *"approach-and-turn"* is 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.

The behavior for ball grabbing is encapsulated in a separate option, *"grab-ball-with-head* (cf. fig. 3.26). 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 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.

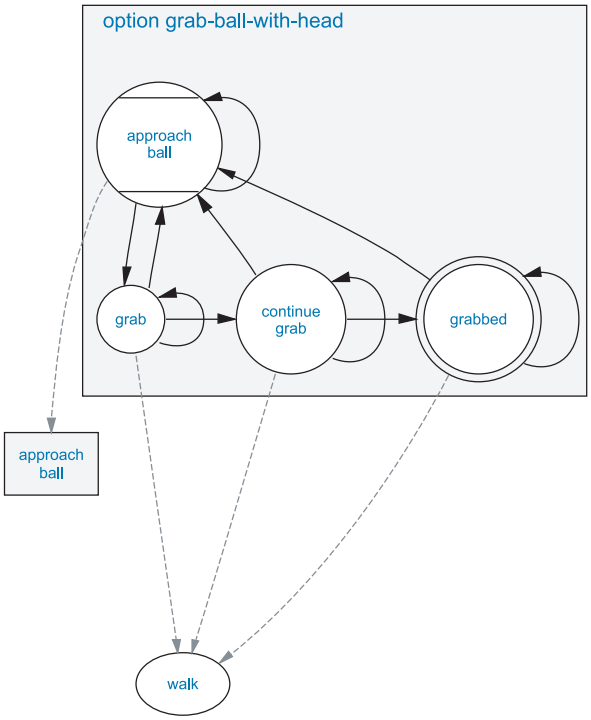


Figure 3.26: Option *"grab-ball-with-head"* grabs the ball with the head.

In the initial state *"grab"* of option *"turn-and-release"* (cf. fig. 3.27), 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 *"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.

3.8.1.4 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.

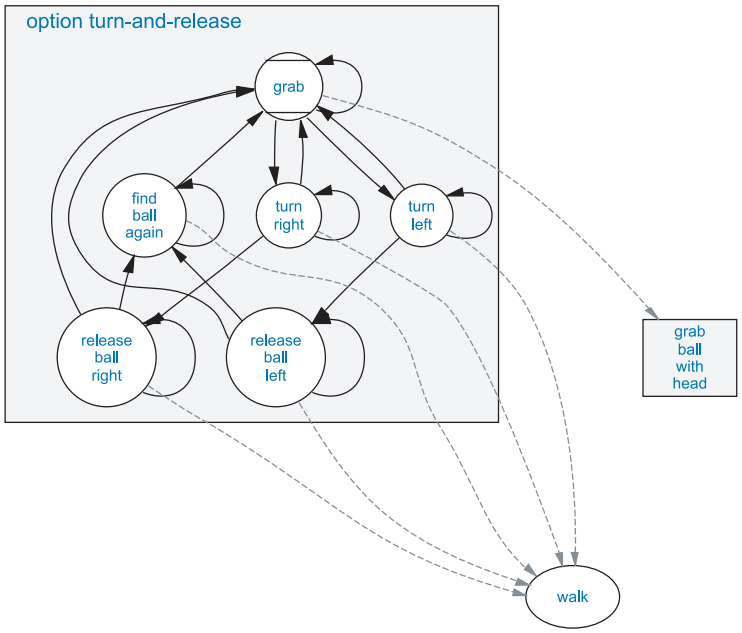


Figure 3.27: Option *"turn-and-release"* grabs the ball and pulls it around. The actual job of lifting the head in the right moment is done in the head control.

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 3.28a) 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 3.28b) 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

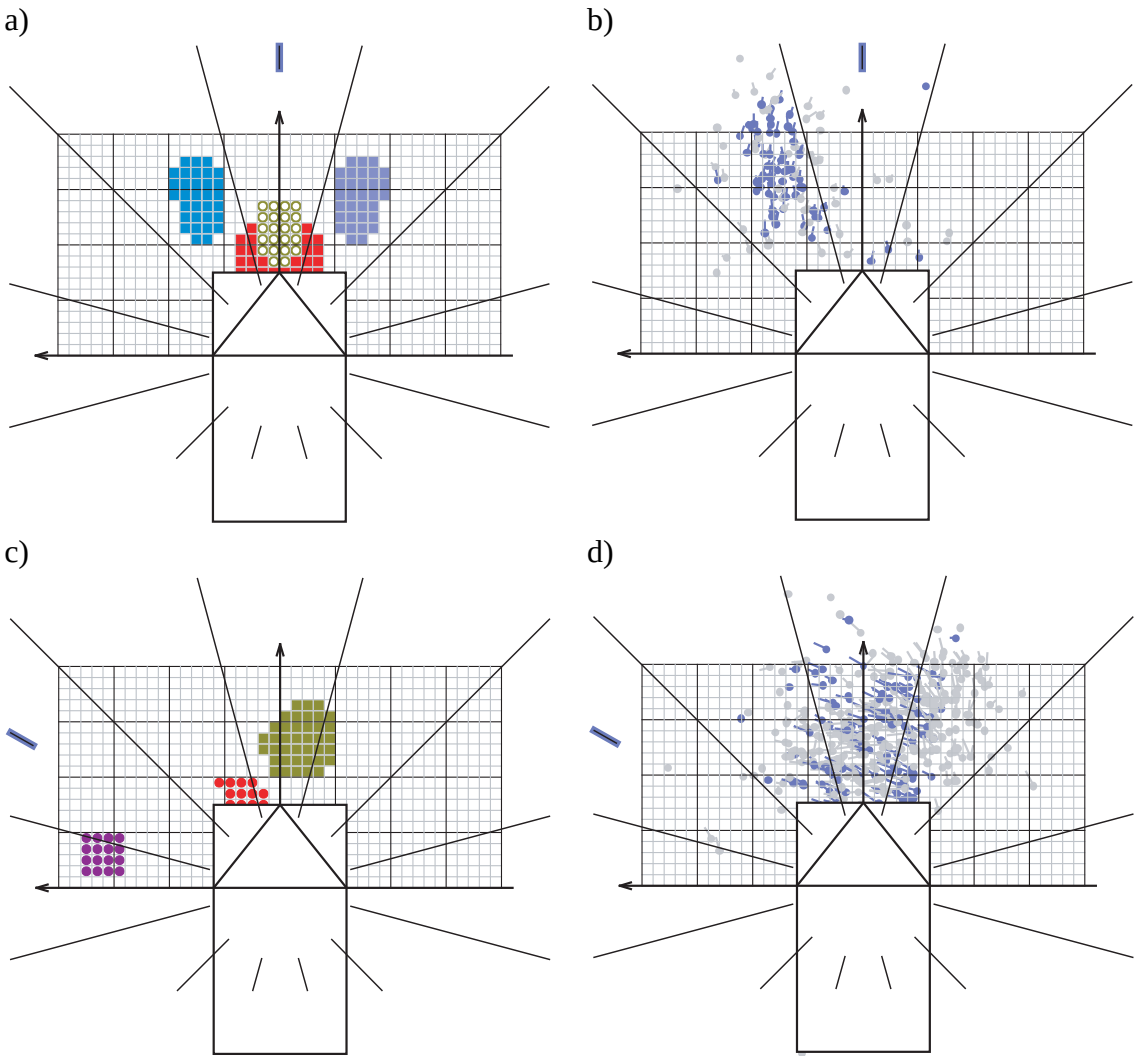


Figure 3.28: a) The kick selection table for the goalie when the desired kick direction is “forward” (in the sector between -15 and 15 degrees). If the current position of the ball is in the outer blue areas, the *“left-paw”* or *“right-paw”* kick is selected, in front of the robot (red area), kick *“chest-strong”*, 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.

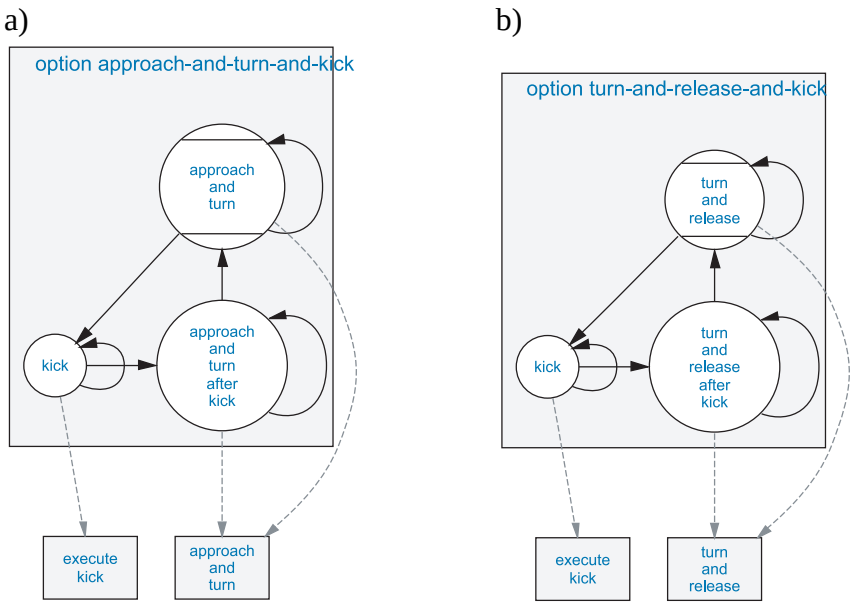


Figure 3.29: Both a) option *"approach-and-turn-and-kick"* and b) option *"turn-and-release-and-kick"* execute a behavior that is able to play the ball without kicking and only perform a kick if by chance it is applicable.

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. 3.8.1.5).

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"*.

Option *"approach-and-turn-and-kick"* (cf. fig. 3.29a) is an example for a behavior that uses kicks. It is composed from a behavior that is able to play the ball without kicking (*"approach-and-turn"*) and the kick execution option *"execute-kick"*. In the initial state *"approach-and-turn"*, the kick selection table is always queried whether a kick is possible. If so, the kick is executed in the state *kick*. After it finished (the option *"execute-kick"* reached its target state), option *"approach-and-turn-and-kick"* remains for 2.5 seconds in the state *"approach-and-turn-after-kick"*, which executes the same behavior as state *"approach-and-turn"* but makes sure that there elapse at least 2.5 seconds between two successive kicks.

Similarly, the option *"turn-and-release-and-kick"* (cf. fig. 3.29b) is composed of *"turn-and-release"* and *"execute-kick"*. In the option *"turn-around-ball-and-kick"*, the basic behavior *"turn-around-ball"* turns the robot behind the ball, which is needed at the borders of the field (*"turn-and-release"* does not work there). Option *"approach-and-kick-and-go-on"* uses only *"approach-ball"* but has an additional state *"go-on"*, similar to in *"approach-and-turn"*.

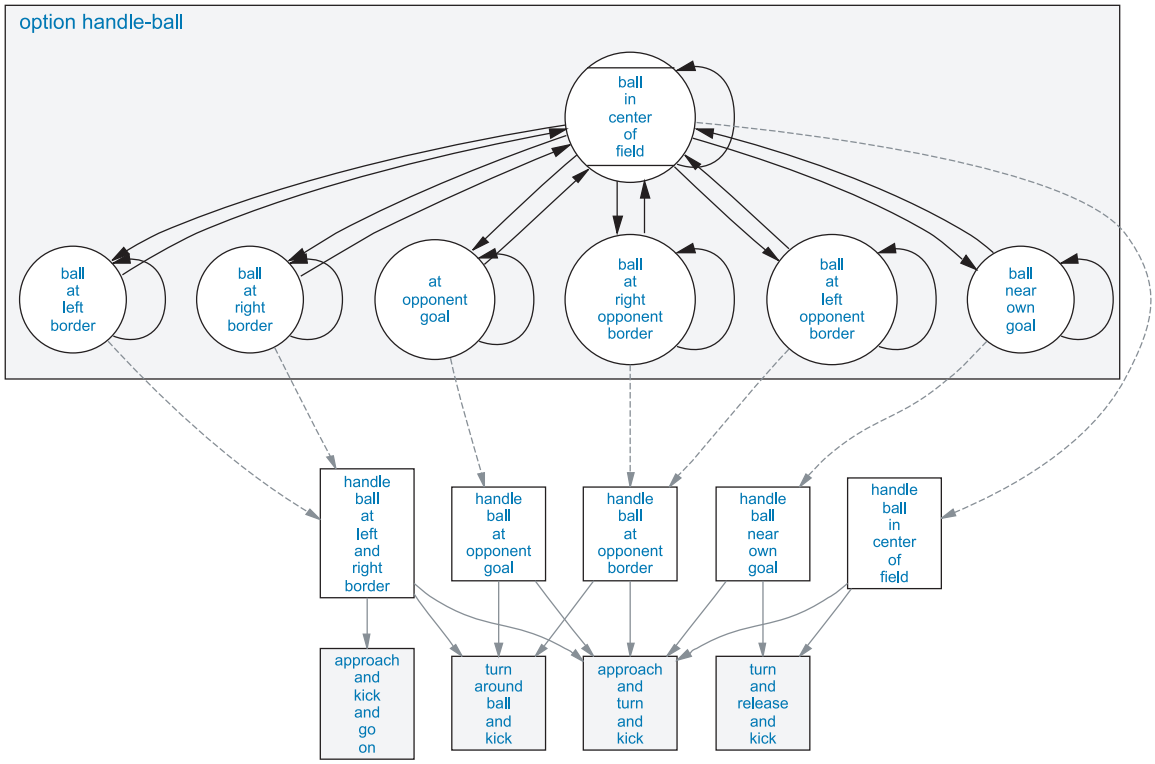


Figure 3.30: Option *"handle-ball"* selects between different behaviors for different zones on the field.

3.8.1.5 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 borders, it is often not possible to grab the ball with the head. Instead, if the robot is in front of the ball, it has to turn behind the ball before it can be dribbled or kicked. 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. 3.30) 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. For instance, there is a transition from *"ball-in-center-of-field"* to *"ball-at-left-border"* when the y position of the ball is greater than 1250 mm. If then the y position of the ball gets less than 1100 mm, there is a back transition to *"ball-in-center-of-field"*.

In the center of the field, option *"handle-ball-in-center-of-field"* selects only between *"turn-and-release-and-kick"* and *"approach-and-turn-and-kick"*, depending on whether the robot is behind the ball or not. As the desired direction of play the *"best-angle-to-opponent-goal"* is passed. This angle is mostly the direct angle to the opponent goal. If there are obstacles on the

way there, the angle is bended to the bigger gap in the obstacles. If there are obstacles everywhere in the direction of the goal, the angle to the next team mate is chosen, which sometimes results in a pass.

Near the own goal, option *"handle-ball-in-center-of-field"* uses the same options as in the center of the field, but passes a different angle: the *"best-angle-away-from-own-goal"* is not directed to the opponent goal but away from the own goal.

At the opponent goal, option *"handle-ball-at-opponent-goal"* combines *"approach-and-turn-and-kick"* with *"turn-around-and-ball-and-kick"*, using the angle *"angle-to-point-behind-opponent-goal"*. The turning behind the ball is indeed slower than when doing a kick to the side, but it is safer when opponent players such as the goalie are involved.

At the left and right border, option *"handle-ball-at-left-and-right-border"* chooses between *"approach-and-turn-and-kick-and-go-on"* if the robot is completely behind the ball, *"approach-and-turn-and-kick"* if the robot is almost behind the ball, and *"turn-around-ball-and-kick"* if not. The average distance to the ball over two seconds is used to decide whether the robot got stuck to other robots. If so, the kick selection table *"when-stuck"* containing quite imprecise but strong kicks is used. If not, less kicks are performed.

Option *"handle-ball-at-opponent-border"* has a very similar structure but uses less aggressive kicks to dribble the ball securely along the border into the opponent goal.

3.8.1.6 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.

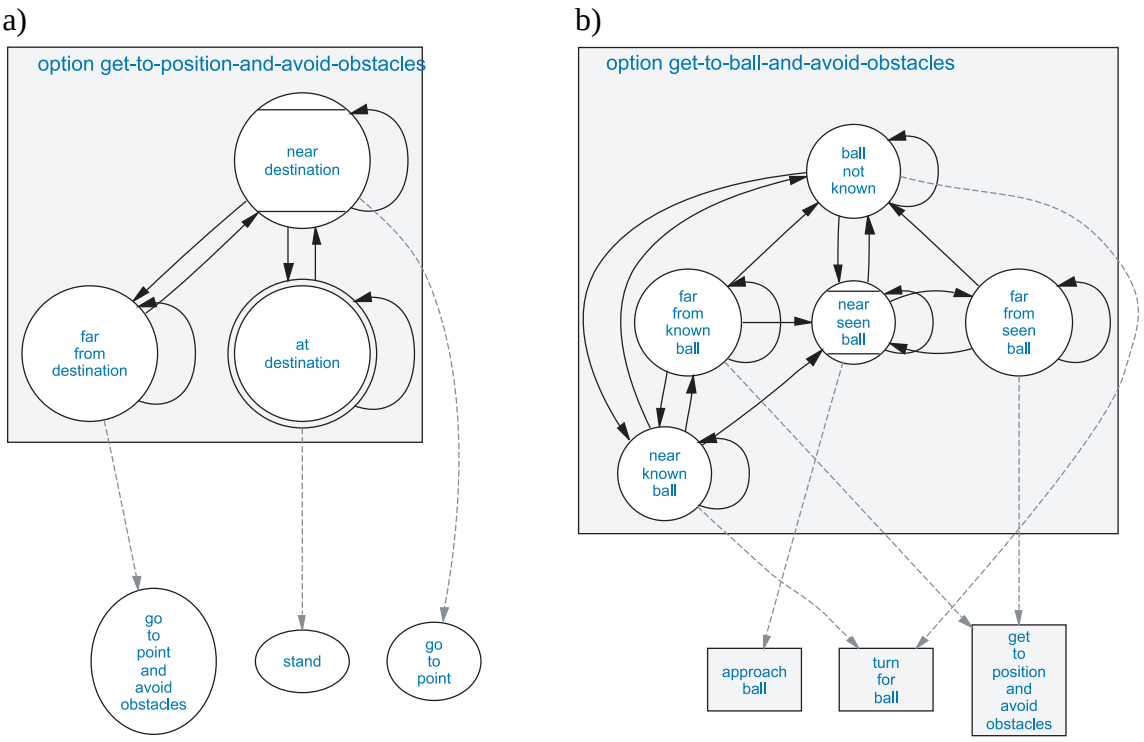


Figure 3.31: a) Option *"get-to-position-and-avoid-obstacles"* walks to a position avoiding obstacles on the way there. b) On top of that, *"get-to-ball-and-avoid-obstacles"* walks to the ball.

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. 3.31a) 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. 3.31b) 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 [33]. 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. 3.32) 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,

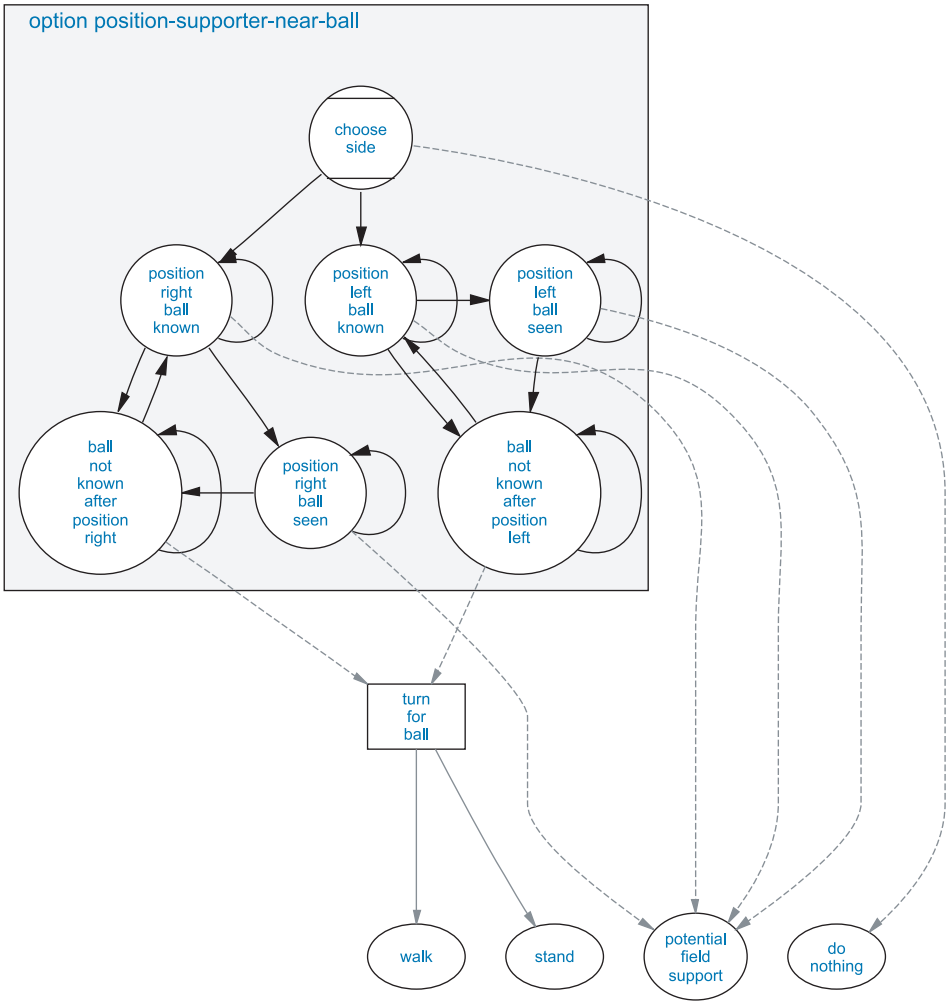


Figure 3.32: Option *"position-supporter-near-ball"* positions the robot near the ball. The speed is controlled depending on the reliability of the ball position.

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. 3.33). 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. 3.8.1.5) 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 *“playing-offensive-supporter”* (cf. fig. 3.34a). 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-supporter-on-line*, waiting for the pass. 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 supporter positions at the opposite corner of the penalty area using option

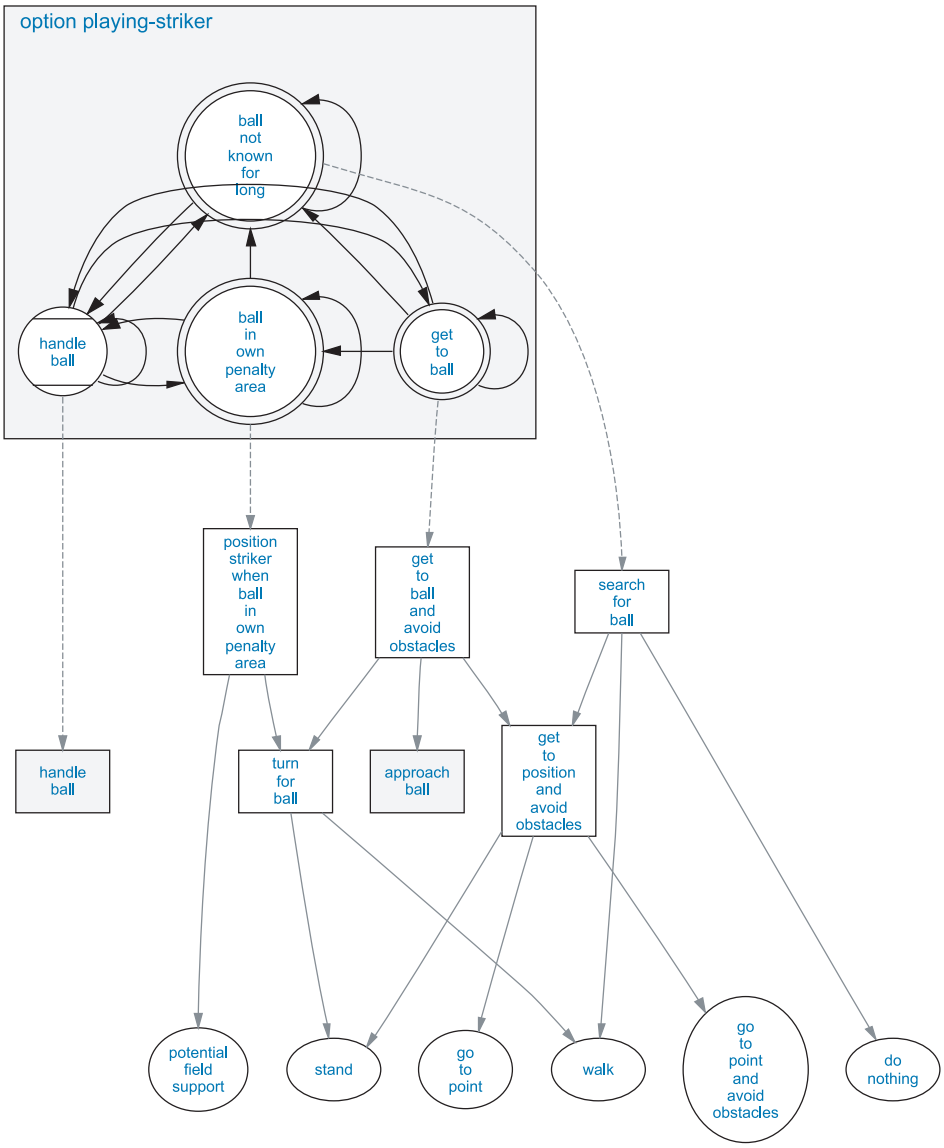


Figure 3.33: Option *"playing-striker"* implements a complete striker.

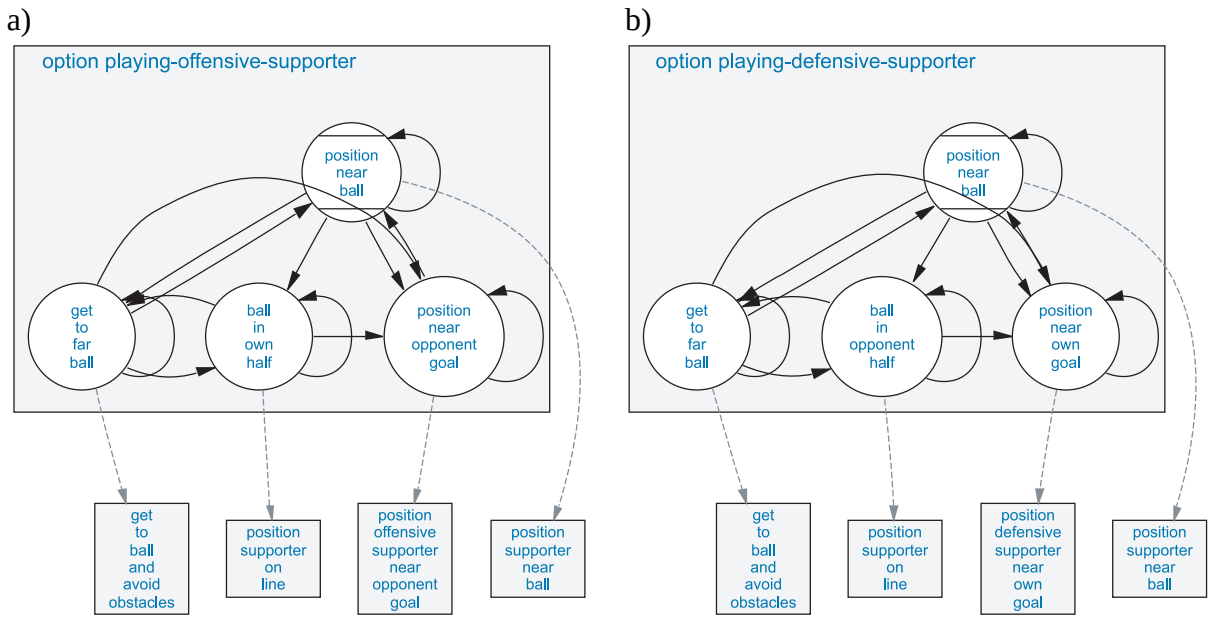


Figure 3.34: a) Option *"playing-offensive-supporter"* and b) *"playing-defensive-supporter"* decide where to position the robot.

"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 *"playing-defensive-supporter"* (cf. fig. 3.34b) mostly stays behind the ball. When the ball is in the opponent half, in state *"ball-in-opponent-half"* positions in the middle of the own half at the y position of the ball using option *"position-supporter-on-line"*. 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 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"* (cf. fig 3.35) 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. The actual analysis whether this could be successful 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-*

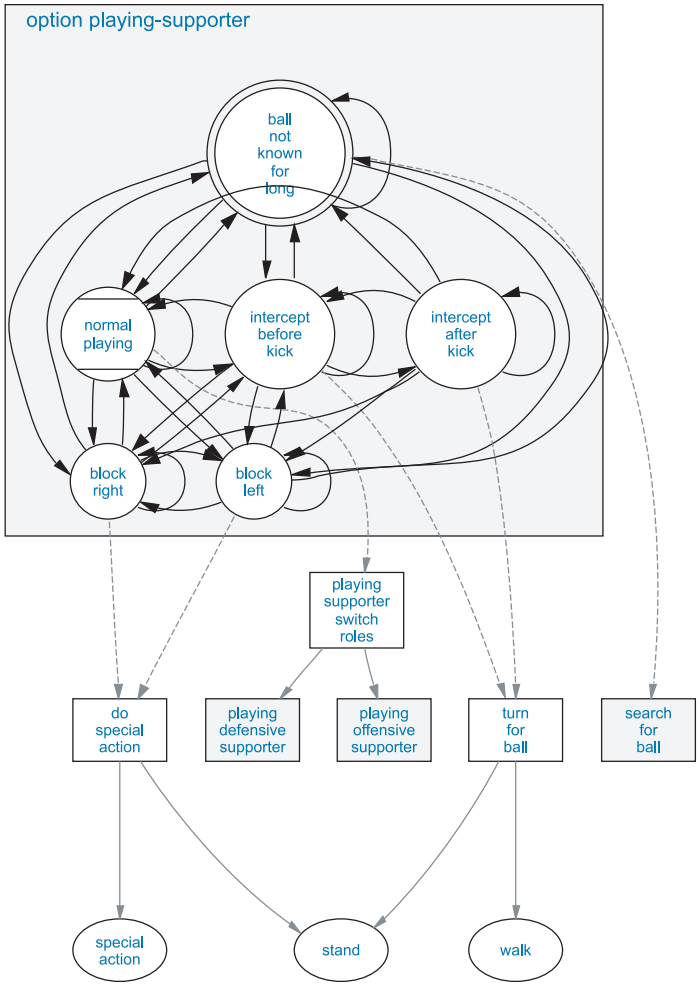


Figure 3.35: Option *"playing-supporter"* intercepts kicks from the own team and blocks kicks of the other team.

performing-a-kick" becomes true, in state *"intercept-before-kick"* the supporters stop positioning 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 2004 tournament, receiving only 8 goals compared to 65 goals scored by the team. 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

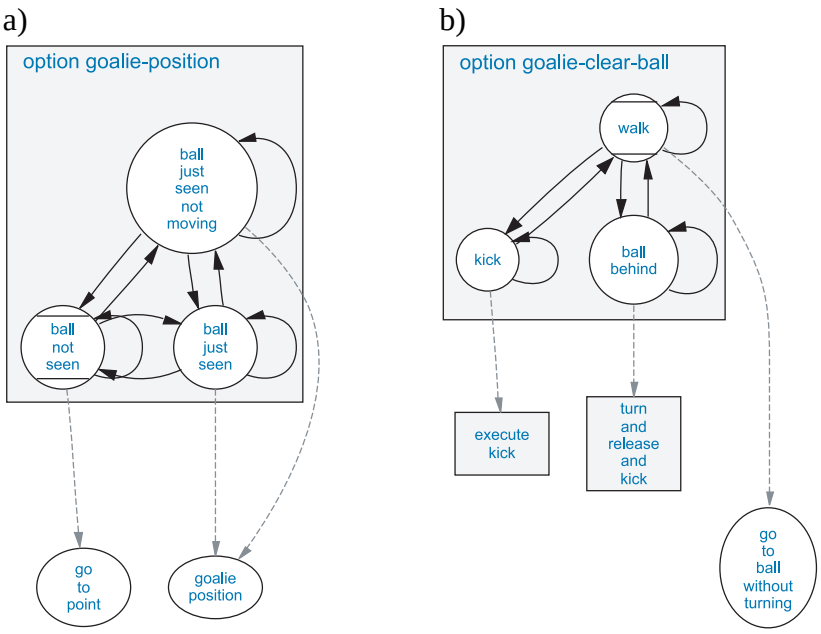


Figure 3.36: a) Option *"goalie-position"* positions the robot inside the goal. b) Option *"goalie-clear-ball"* tries to get the ball out of the penalty area.

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 3.36a) 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 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 3.36b) 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. 3.8.1.4), in state *"kick"* a

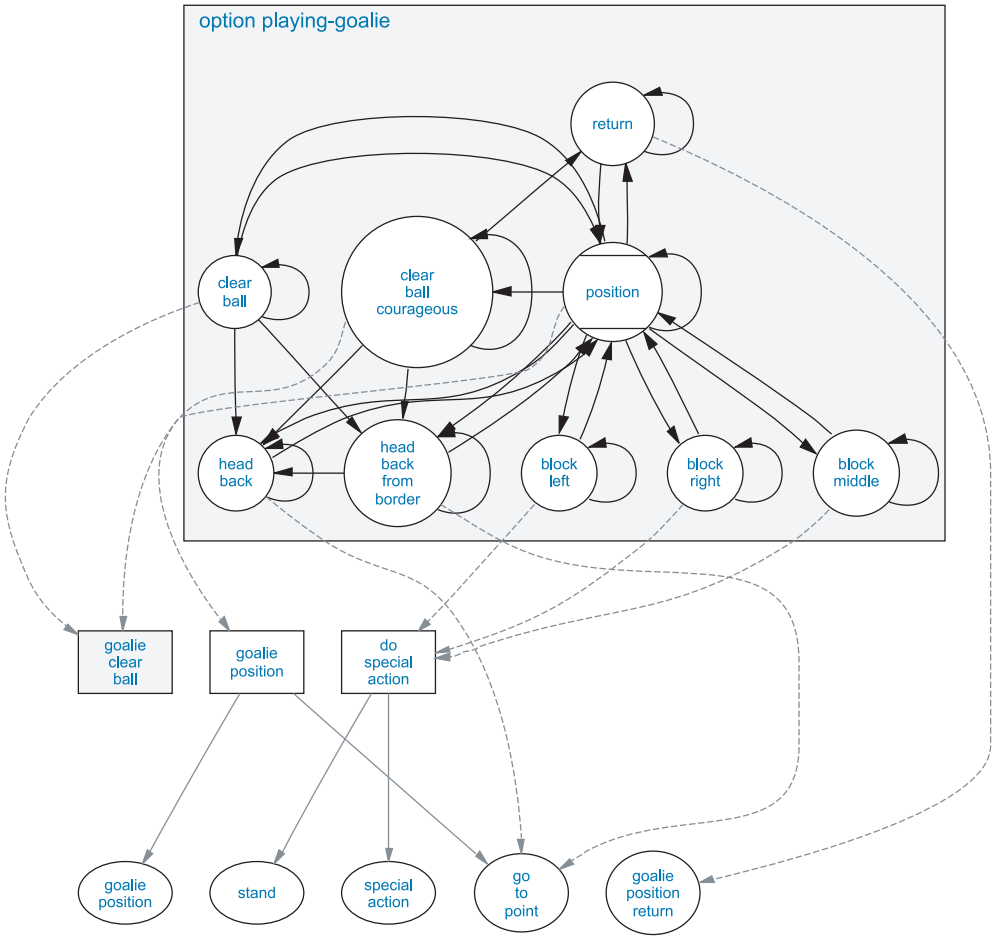


Figure 3.37: Option *"playing-goalie"* implements the goal keeper behavior.

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. 3.8.1.4) is used to clear the ball.

The complete goal keeper behavior is implemented in option *"playing-goalie"* (cf. fig. 3.37). 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 it 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"*, *"block-left"*, and *"block-right"*).

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

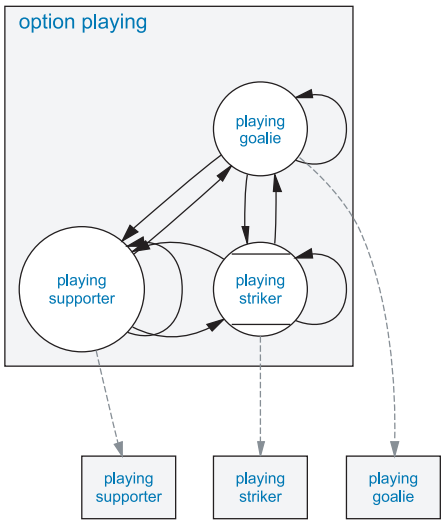


Figure 3.38: Option *"playing"* selects between the different roles.

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. 3.38) 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 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

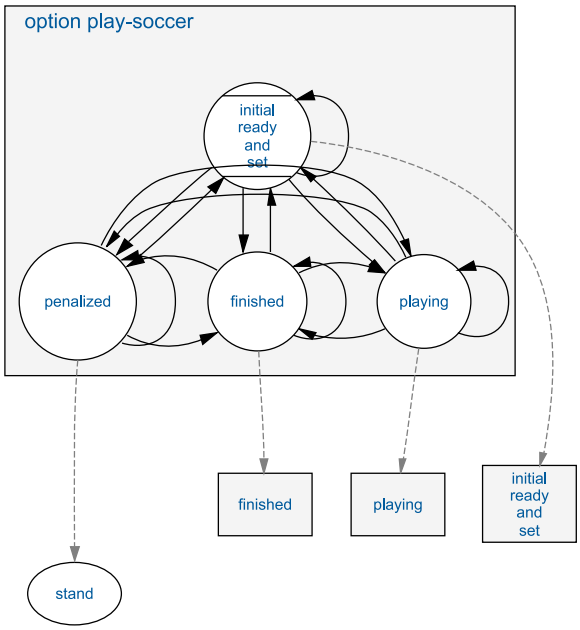


Figure 3.39: Option *"play-soccer"* is the root option of the option graph.

"initial-set-team-color".

The option *"play-soccer"* (cf. fig. 3.39) 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"* (cf. fig. 3.40) 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

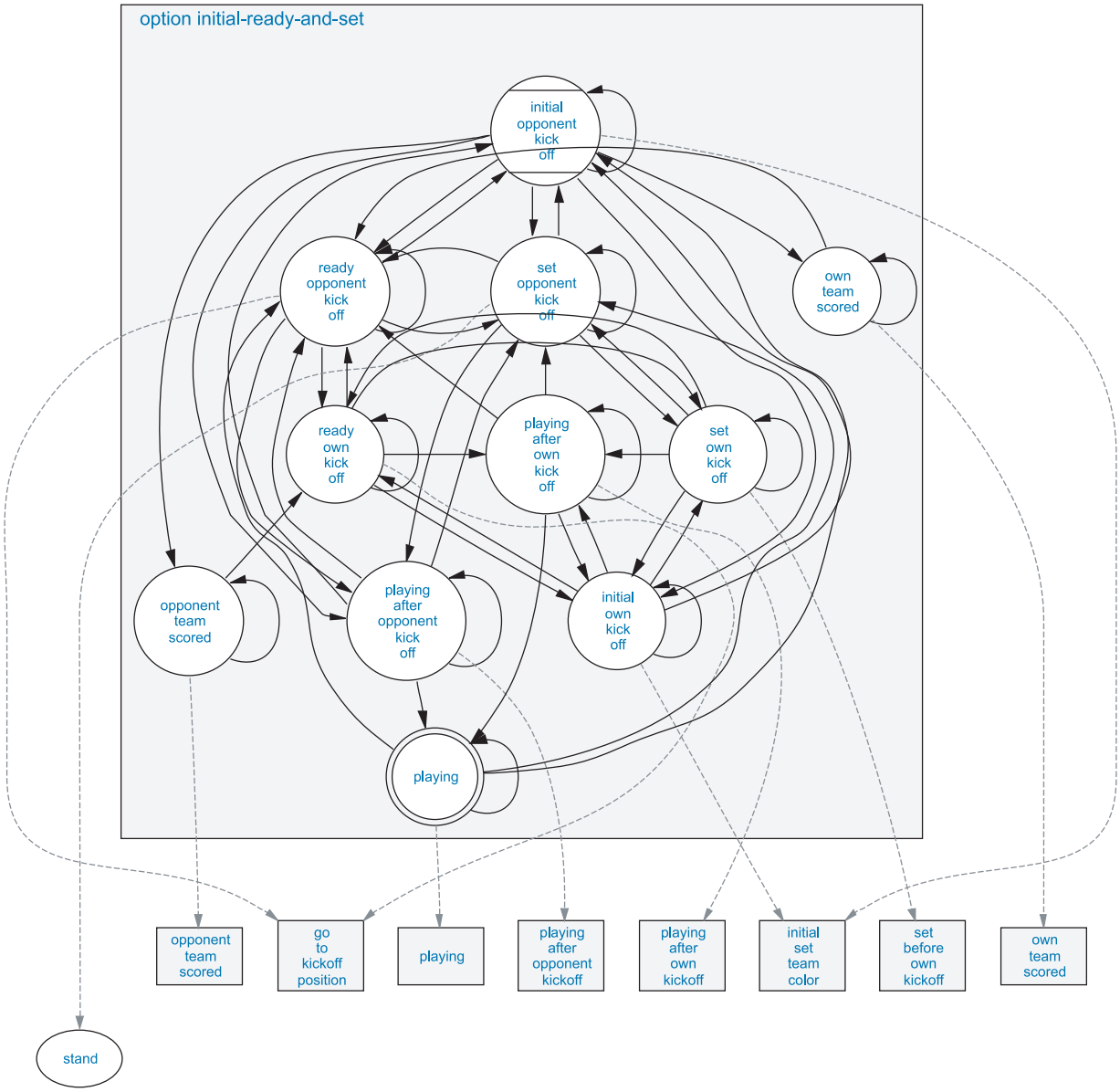


Figure 3.40: Option *"initial-ready-and-set"* is probably the most complicated looking one.

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. 3.41a) 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.

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. 3.41b).

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

a) b)

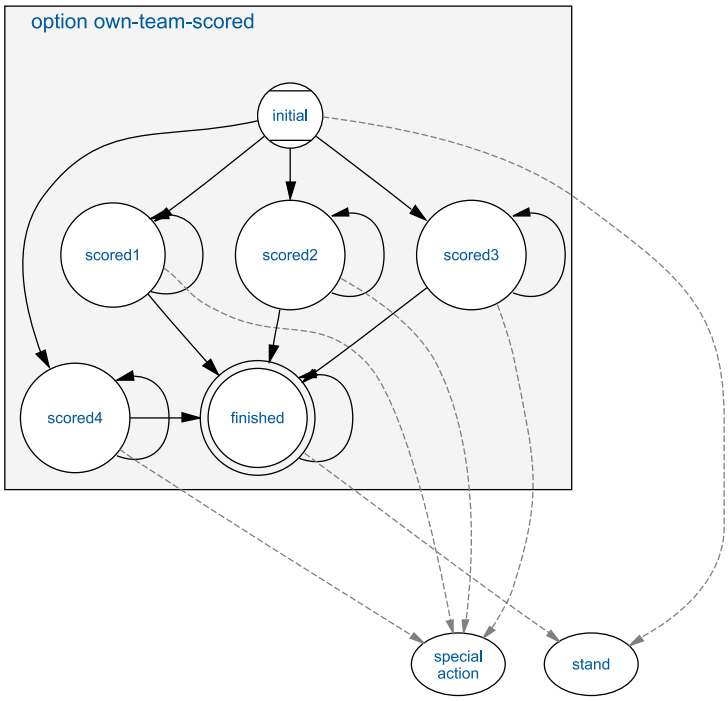


Figure 3.41: a) Option *"own-team-scored"* only chooses one out of four *"scored"* states, executes them for a few seconds, and then terminates. b) At the end of option *"finished"*, all robots walk to the center of the field for being also on the winner photo.

3.9 Motion

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

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

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.

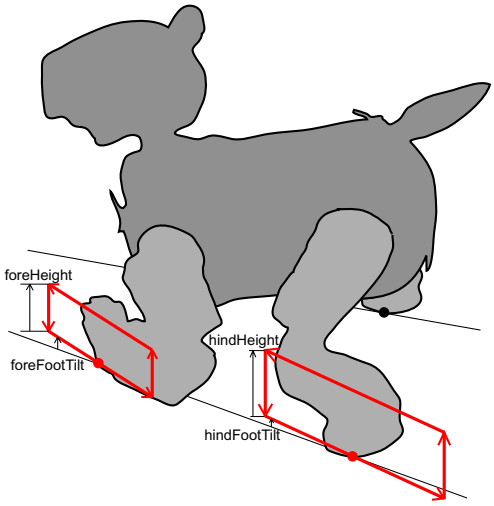


Figure 3.42: Walking by moving feet in rectangles

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.

MotionControl keeps track of the current position in the walk cycle, to smooth transitions between different walk types, assuming that the periods of the walk types are somewhat comparable.

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 first is called *InvKinWalkingEngine* and was used by the GermanTeam for several years. A main feature 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 the set of parameter values is given to the constructor

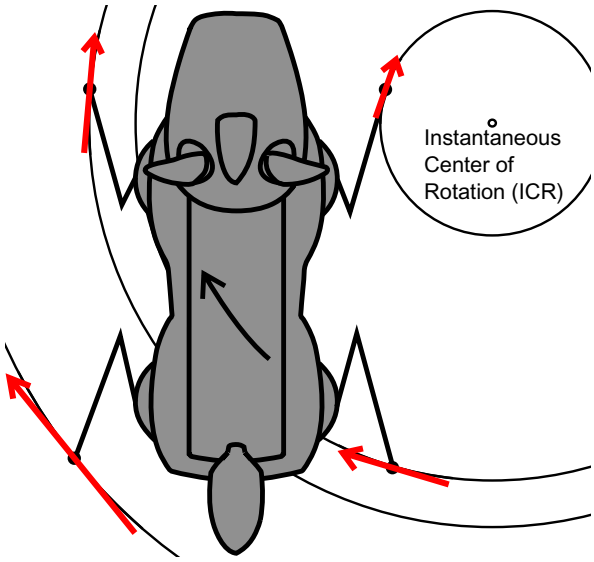


Figure 3.43: Principle of treating legs as wheels. Walking and turning combined results in a rotation around an Instantaneous Center of Rotation (ICR). Instead of really using wheels the robot makes steps (red) with the same direction and speed a wheel would have.

of the engine. Therefore it is possible to have different instances with different parameter sets. It is even possible to transmit new parameters via the wireless network from RobotControl to test them at runtime (cf. Sect. D.6.3).

3.9.1.1 Approach

The general idea is to calculate the position of the feet relative to the body while they move in parallelograms around their center position (cf. Fig. 3.42). The necessary joint angles to reach the foot position are calculated by inverse kinematics.

For the direction of walking the four legs are more or less treated as wheels. Seen from above, the rectangles are rotated to the desired walking direction (cf. Fig. 3.43). 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 rectangles. 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.

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

Additionally the walking engine receives a position specifying the current point in the walk execution cycle, smoothing transitions between different parameter sets.

3.9.1.2 Parameters

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

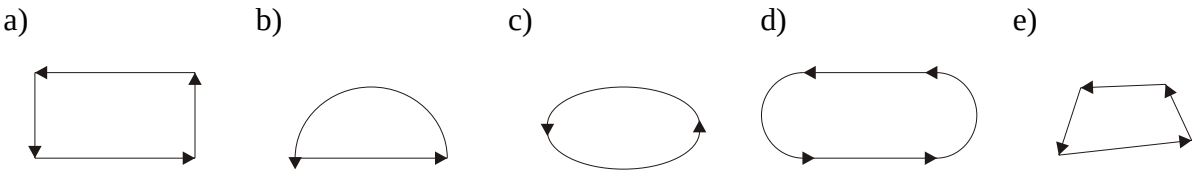


Figure 3.44: Possible modes of foot movement a) rectangle b) semi-ellipse c) ellipse d) oval e) arbitrary quadruple.

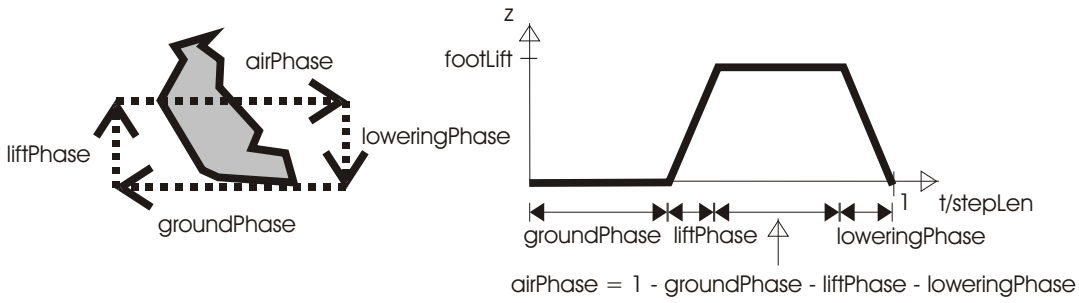


Figure 3.45: Timing of one step cycle

footMode. This parameter selects how the feet will be moved while in the air. Besides the rectangular shape mentioned above, it is also possible to have the feet move in different shapes, e. g. a semi-circle like it was the case in our walking engine for RoboCup 2001 (cf Fig. 3.44). This parameter was not used much since the rectangular shape seemed to provide best general performance. It was mainly included for increasing flexibility.

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

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

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

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

stepLen. This is the time for one complete step cycle (cf. Fig. 3.45).

groundPhase, liftPhase, loweringPhase. These values define the timing of the step cycle (cf. Fig. 3.45). *groundPhase* defines how much time of the step cycle the foot will be on the ground. *liftPhase* defines how fast the foot will be lifted, *loweringPhase* how fast it will be lowered. There are two values each, one for the fore and one for the hind legs.

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

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

3.9.1.3 Combining several optimized parameter sets

In the last years most teams tried to use a walking model that is as universal as possible, i. e. enables a robot to walk omnidirectional with one set of parameters. Although that is a good starting point, much better parameters exist for special task like walking forward or backward only. Therefore the GermanTeam used two or three parameter sets and switched between them until 2003. As RoboCup is a highly dynamic environment, we decided to use a more continuous model this year. For every motion request (every combination of desired walk speed, walk direction, and turn speed) we use a certain parameter set interpolated from several fixed, optimized parameter sets around, as described in [19].

Many additional walking parameters have been introduced in the last years, most if them were designed to increase the speed or robustness of walking in a certain direction. Interpolating between those parameters is only useful if their influence on omnidirectional walking is known or can be estimated. Furthermore continuously increasing walk speeds without those parameters were published. Because using several parameter sets increases the effort for measuring, calibrating, and fine-tuning, we tried to minimize the number of parameters per parameter set (see above) and used only those, that proved to have a positive influence on omnidirectional walking.

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.

Using several optimized parameter sets in 2004 gave us the possibility to calibrate each such parameter set on its own without any influence on other parameter sets. Therefore it was possible to use (and know of) different maximum walk speeds for different walk directions leading to faster walking in certain directions than before. At the same time the accuracy of this faster walking was increased by separate calibration. The optimization and calibration was automated as far as possible [19] to compensate for the increased number of used parameter sets.

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:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = Rot_y(-q_1)Rot_x(q_2)Trans \begin{pmatrix} 0 \\ 0 \\ -l_1 \end{pmatrix} Rot_y(-q_3)Trans \begin{pmatrix} 0 \\ 0 \\ -l_2 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad (3.44)$$

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

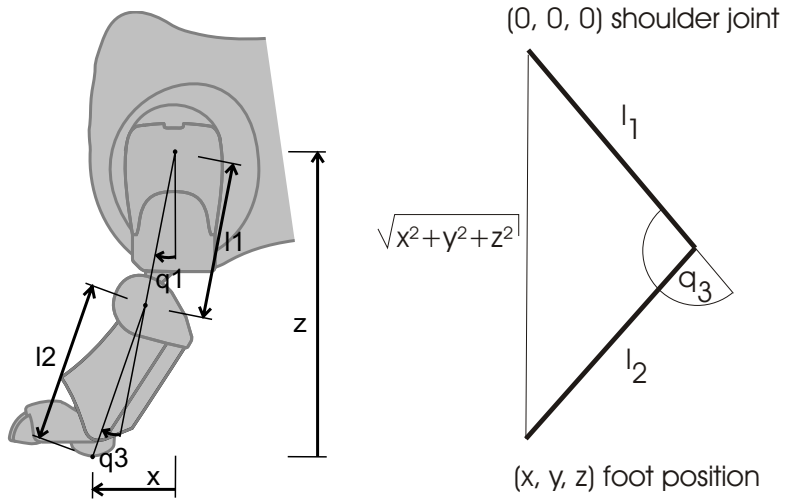


Figure 3.46: leg side view, calculation of knee joint q_3 via law of cosine

This is equivalent to:

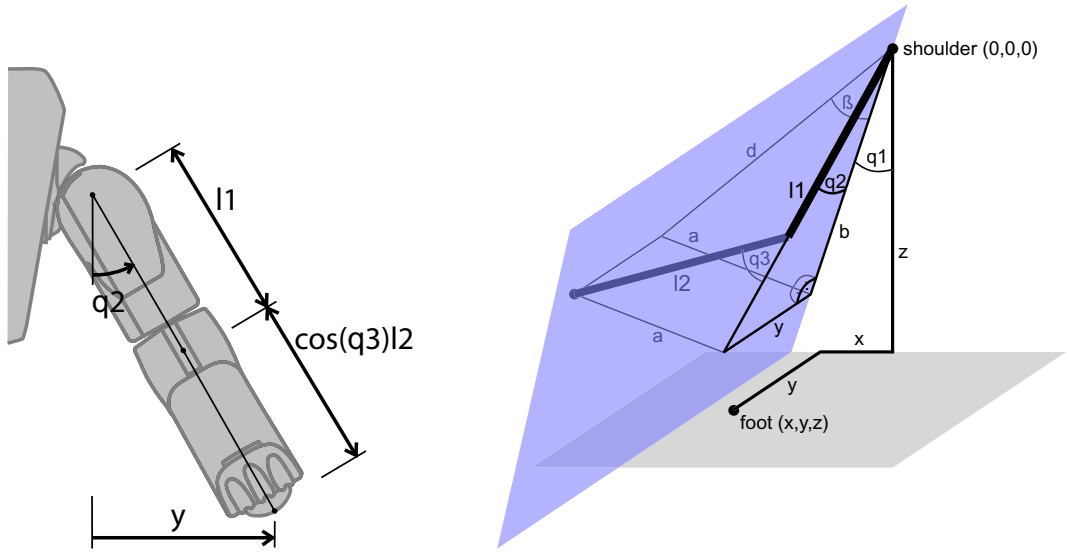
$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(q_1) & 0 & -\sin(q_1) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(q_1) & 0 & \cos(q_1) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(q_2) & -\sin(q_2) & 0 \\ 0 & \sin(q_2) & \cos(q_2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -l_1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$\begin{pmatrix} \cos(q_3) & 0 & -\sin(q_3) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(q_3) & 0 & \cos(q_3) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -l_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{3.45}$$

Matrix multiplication results in

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(q_1)\sin(q_3)l_2 + \sin(q_1)\cos(q_2)\cos(q_3)l_2 + \sin(q_1)\cos(q_2)l_1 \\ \sin(q_2)l_1 + \sin(q_2)\cos(q_3)l_2 \\ \sin(q_1)\sin(q_3)l_2 - \cos(q_1)\cos(q_2)\cos(q_3)l_2 - \cos(q_1)\cos(q_2)l_1 \\ 1 \end{pmatrix}. \tag{3.46}$$

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.



a) b)

Figure 3.47: a) leg front view, calculation of shoulder joint q_2 b) leg model, calculation of shoulder joint q_1 with several helping variables

According to the law of cosine (cf. Fig. 3.46)

$$\cos\alpha = \frac{l_1^2 + l_2^2 - x^2 - y^2 - z^2}{2l_1 l_2} \tag{3.47}$$

with upper limb length l_1 and lower limb length l_2.

With

$$|q_3| = |180^\circ - \alpha| = \arccos\frac{x^2 + y^2 + z^2 - l_1^2 - l_2^2}{2l_1 l_2} \tag{3.48}$$

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 (3.46) (geometrically apparent cf. Fig. 3.47a)

$$\begin{aligned} y &= \sin(q_2)l_1 + \sin(q_2)\cos(q_3)l_2 \\ &= \sin(q_2)\left[l_1 + \cos(q_3)l_2\right]. \end{aligned} \tag{3.49}$$

Consequently

$$q_2 = \arcsin\left(\frac{y}{l_2\cos(q_3) + l_1}\right). \tag{3.50}$$

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 (3.46) (cf. Fig. 3.47b)

$$\begin{aligned} x &= \cos(q_1)\sin(q_3)l_2 + \sin(q_1)\cos(q_2)\cos(q_3)l_2 + \sin(q_1)\cos(q_2)l_1 \\ &= \cos(q_1)\sin(q_3)l_2 + \sin(q_1)\left[\cos(q_2)\cos(q_3)l_2 + \cos(q_2)l_1\right]. \end{aligned} \tag{3.51}$$

When defining

$$a := \sin(q_3)l_2, \tag{3.52}$$

$$b := -\cos(q_2)\cos(q_3)l_2 - \cos(q_2)l_1 \tag{3.53}$$

and

$$\beta := \arctan\left(\frac{a}{b}\right), \tag{3.54}$$

$$d := \sqrt{a^2 + b^2} \quad \left(= \frac{b}{\sin(\beta)}\right), \tag{3.55}$$

so that

$$a = d\cos(\beta), \qquad b = d\sin(\beta), \tag{3.56}$$

equation (3.51) simplifies to

$$\begin{aligned} x &= \cos(q_1)a - \sin(q_1)b \\ &= d\left[\cos(q_1)\cos(\beta) - \sin(q_1)\sin(\beta)\right] \end{aligned} \tag{3.57}$$

which can be transformed to

$$x = d\cos(q_1 + \beta). \tag{3.58}$$

Hence

$$|q_1 + \beta| = \arccos\left(\frac{x}{d}\right). \tag{3.59}$$

The sign of $q_1 + \beta$ can be obtained by checking the z-component of equation (3.46). As in equations (3.51)-(3.58) this results in:

$$\begin{aligned} z &= \sin(q_1)\sin(q_3)l_2 - \cos(q_1)\cos(q_2)\cos(q_3)l_2 - \cos(q_1)\cos(q_2)l_1 \\ &= \sin(q_1)\sin(q_3)l_2 - \cos(q_1)\left[\cos(q_2)\cos(q_3)l_2 + \cos(q_2)l_1\right] \\ &= \sin(q_1)a + \cos(q_1)b \\ &= d\left[\sin(q_1)\cos(\beta) + \cos(q_1)sin(\beta)\right] \\ &= d\sin(q_1 + \beta). \end{aligned} \tag{3.60}$$

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.

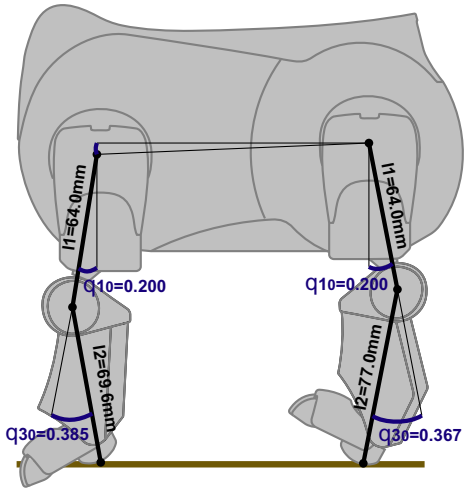


Figure 3.48: Sending an angle of 0 degrees to all joint motors results in vertical legs but not in angles of 0 degrees between the joints.

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. 3.48 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. Two different approaches to optimize the gait parameters were used in the GermanTeam:

Localization-Based Fitness. The major problem in autonomous learning approaches for walking is to find a way to measure the speed of the resulting walk reliably. This approach uses the self-locator of the robot to determine the walking speed.

First the robot stands on the field and localizes itself. After being well localized, the robot walks for a fixed period of time into a specified direction with maximum speed. Then it stops and localizes it self again. By calculating the distance between the starting and ending point on the field, the robot can calculate the speed of the walk. To increase the reliability of this kind of measurement, we let the robot walk three times with the same parameter set and take the average of the three measured speeds.

For the optimization process we used a simple $(1+1)$ Evolution Strategy [5]. An individual is represented by a set of walking parameters. Its fitness is discovered by the corresponding speed of the robot while walking on the field. Since this approach does not need any external hardware it is easily possible to adapt the walking parameter set to the actual environmental conditions.

With this approach we found a stable walk for the ERS-7 walking with 35 cm/s. Slight modifications in the PID settings increased the walking speed to 38 cm/s. These walking parameters were used by the GermanTeam in the competitions for the “normal” walk.

Odometry-Based Fitness. The other approach to gait optimization employs a probabilistic evolution scheme [50]. It separates the typical crossover-step of the evolutionary algorithm into an interpolating step and an extrapolating step, which allows for solving optimization problems with a small population, which is an essential for robotics applications. In contrast to other approaches, real odometry is used to assess the quality of a gait. The motion of the robot is estimated from the trajectories of the rear legs while they have contact to the ground. The main advantage is that gaits can be assessed very quickly (every 5 seconds a new one), so learning is very fast. The only drawback of this approach is that it can only learn gaits in which the ground contact sensors of the rear feet touch the ground. With this approach, a maximum forward speed of 40 cm/sec and a turning speed of 195°/sec were reached. This gait was only used to cover large distances on the field.

This approach is implemented in the *BB2004WalkingEngine*. It loads gait parameter sets from the file *Config/Walking.cfg*. Currently, four different sets are supported both for the ERS-7 and the ERS-210: *forward* (normal forward motion including rotations), *fastForward*, *backward*, and *stand* (same body posture as *forward*, but motionless). When the debug key *learnWalking* is activated, the evolution starts (in the code release, the *forward* set is selected for evolution). However, the behavior control has to request the motion that has to be learned, e. g. walking straight as fast as possible or following the ball. It is important that the different parameter sets tested receive a comparable feedback, e. g., it is not recommended that the robot has collisions while assessing some of the parameter sets while not colliding when assessing others. The winning parameter set of each generation is printed to the message window. When the gait is fast enough or after the battery is replaced, the best set can be copied to the file *Config/Walking.cfg*, so that it either can be used or it can be the starting point for further evolution. Please note that the ground contact sensors of the rear legs must touch the ground, otherwise the evolution will not have a useful direction.

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 5.4. 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 an-

gles 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 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).

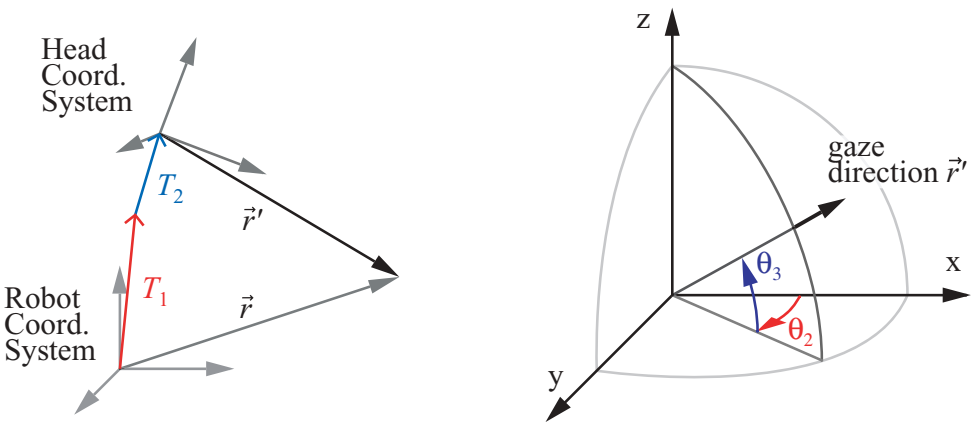


Figure 3.49: *Left:* the target point is transformed from the robot coordinate system to the shoulder coordinate system (T_1) to the head coordinate system (T_2). *Right:* transformation from cartesian into spherical polar coordinates yields the desired joint angles.

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. 3.49):

$$\theta_1 = \theta_c, \qquad \theta_2 = \arctan\frac{y'}{x'}, \qquad \theta_3 = \arctan\frac{\sqrt{x'^2 + y'^2}}{z'} \tag{3.61}$$

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. 3.50 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.

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,

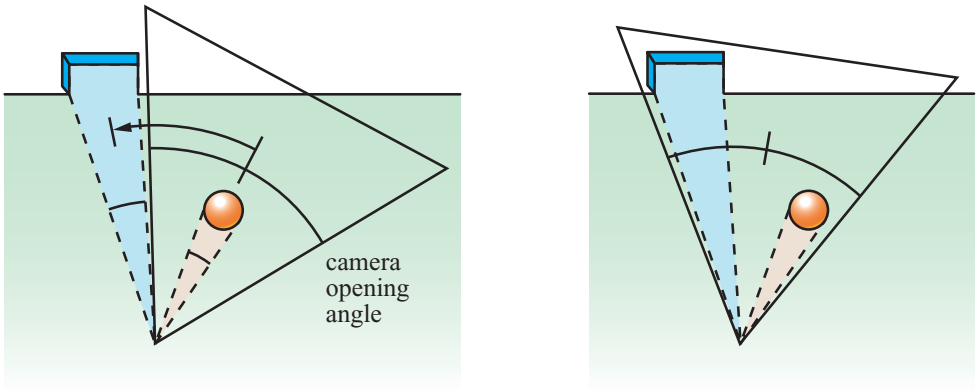


Figure 3.50: Top down view of the field of view of the robot. In the left diagram, the robot's field of view is centered on the ball. The right diagram shows the optimized gaze direction which enables the robot to perceive the goal too.

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 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.

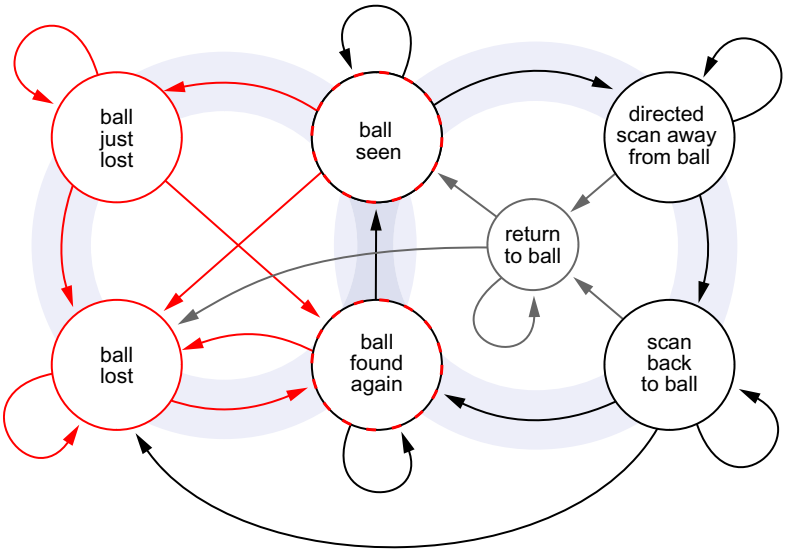


Figure 3.51: State machine used for tracking the ball. On the right side of the diagram, states are shown related to “intentionally not looking at the ball but at landmarks instead”. To the left (highlighted in red) the states are shown that the robot passes through/visits when it has lost sight of the ball.

The state machine is shown in figure 3.51. 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 *HeadPath-Planner* or set head joints directly.

Chapter 4

Open Challenge

Besides the soccer competitions in the Sony 4-legged Robot League the teams also are invited to take part at annual technical challenges. One of these challenges is the so-called *Open Challenge*. There, teams can present and demonstrate parts of their research in a creative and entertaining way. We decided to create a scenario in which our research on cooperative behavior could be demonstrated and came up with the following idea:

A robot should kick a goal with a ball from the RoboCup Mid-Size League. To accomplish this task, one robot has to cooperate with four more robots of his team and the whole has to act as a "virtual Mid-Size League robot".

For this transformation we built a cart with rails at the sides and a ramp, enabling a robot to climb on the cart using a ramp. Then, this randomly selected robot stays on top of the cart, meanwhile the four other robots search for the cart. When they found it, they stay at the sides of the cart and bite the rails. In this manner we literally visualized the virtual robot metaphor used by us so far: All five robots together form a virtual Mid-Size robot!

The robot on the top of the cart localizes itself and searches for the orange ball. As all other robots are blind, because they look directly toward the sides of the cart, they require localization information from the non-blind one on top of the cart. On the other hand, the robot on top can not move himself. For this reason, he sends walking requests to the four cart-moving robots. In this situation they are simply representing actuators of the virtual robot.

The low level behavior like walking to positions or performing certain tasks were realized in XABSL (see Sec. E). On top of this behavior we used the Dynamic Team Tactics (DTT) that was first introduced at the GermanOpen 2004 in Paderborn, Germany. DTT was developed to comply with the needs of sharing robots as a whole and sensory information as well. Additionally, tasks can be scheduled to all active robots of the team.

Let's assume there are τ different tasks $T_1 \dots T_\tau$ which can be executed on at least one individual R_j of the team. When there are more tasks than robots $(\tau > \rho)$, we have to deal with a matching problem. To solve it efficiently, priorities must be assigned to all tasks. Further, we have to find out which resources are occupied by executing a task. Hence, we distinguish between sensor tasks, processing tasks, and action tasks [16].

The next step is to obtain knowledge about the actual situation and the environmental conditions:

4.1 Classification

Typically, for each robot R_i the environment at time t can be described by its own world model $M_i(t)$ [17]. Without loss of generality, $M(t)$ can be classified into a set of situations s_k with $1 \leq k \leq \sigma$.

In case of robot soccer, there is nothing more important than scoring, especially if the agent is in front of the opponent goal. Hence, in this example the task "scoring" has a higher priority than "defend own goal". The situation changes if the opponent moves the ball in the other half and consequently gets an advantageous position to score a goal. Therefore, we assume that in every situation s_k exists an optimal priority-rating of the tasks T_i.

Let $\vec{v} = (v_1, v_2, \dots v_\tau)$ be a τ-dimensional weight vector, where the importance (priority) of task $T_1, \dots, T_\tau$ is stored. If there are σ different situations, then there will be σ corresponding weight vectors $\vec{v}^1 \dots \vec{v}^\sigma$, too.

Thus, after classification of the actual world-state, finding the corresponding optimal rating is a matching problem [57].

4.2 Matching

To find the best solution of the matching problem, we suggest to estimate the ability w_j^i of robot R_i to solve task T_j. For the following, this value is called task-rating. Further, the current world-state belief $M_i(t)$ of robot i has to be classified into a situation $s_{j(M_i)}$. Then, finding the optimal task distribution means to maximize the dot-product of task-rating vector and priority-vector according to the actual situation $v(s_j)$.

This problem can be illustrated by a bipartite graph: A number of ρ robots is connected to τ tasks, whereas every robot R_i has an edge of weight w_j^i to task T_j. The problem to find the combination of task assignments that maximize the global weight can be solved efficiently by applying the so-called "Hungarian Algorithm" [43].

4.3 Estimation

If all robots had the same information about their environment (i.e. if they shared the same world model), every robot would find the same solution of the matching problem.

The matching would be identical and unique, resulting in a deterministic problem solution. However, due to the physical limits of communication speed and inaccuracies in measurements, the world models of the robots differ from each other. Thereto, it is necessary to estimate the decisions of all teammates and address the hazards which might occur.

4.4 Arbitration

Hazards can occur whenever reality and estimate differ from each other [54]. A serious problem is a non-unique task assignment: first, this can lead to allocation problems; second, it blocks robot

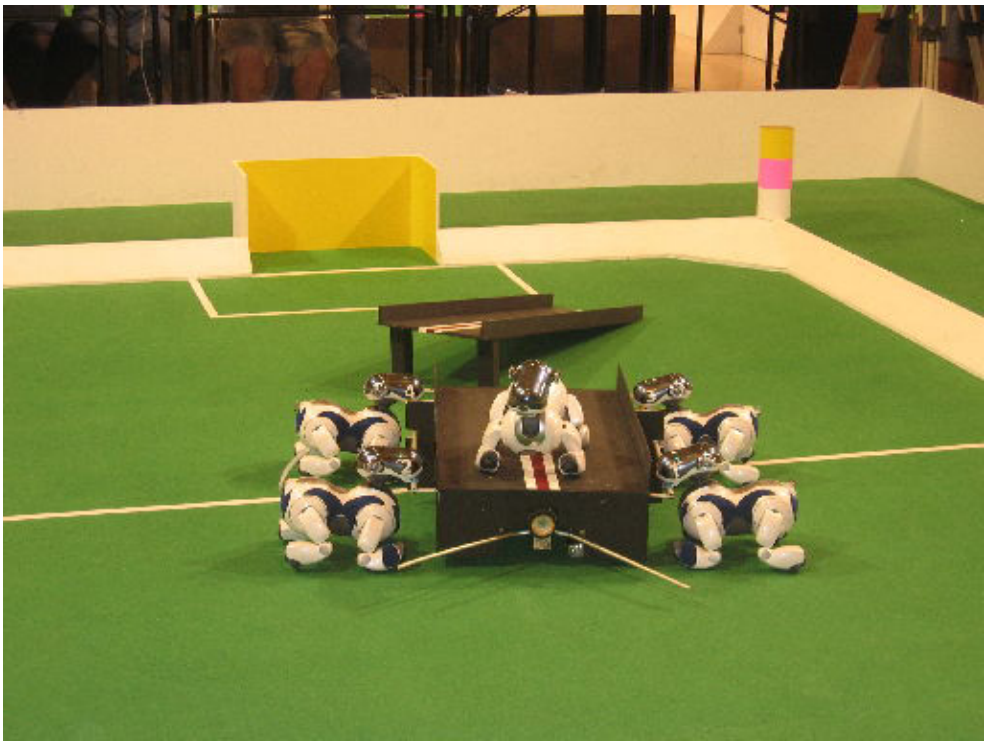


Figure 4.1: Frontal view of the virtual robot: Four robots work as actualors at the groud. Another robot works as sensor on top.

resources which should be utilized more efficiently. As a result, an efficient arbitration is required to avoid such conflicts. As the robotic system works in real-time, it's required an approach that fits this additional constraint.

4.5 Conclusion

Almost every team in Lisbon which saw this demonstration was truly impressed and entertained. Accordingly we achieved the first place in the open challenge competition by a great margin. More details about behavior, image processing and dynamic resource sharing can be found in [16]. An overview of the shown performance is given by Figure 4.1.

Chapter 5

Tools

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

In section 5.1 and 5.2 two very similar programs are described: the simulator based on SimRobot and RobotControl. They both have in common:

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

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

The *Emon Log Parser* (cf. Sect. 5.6) was used to get as much information as possible out of the log files produced by the Open-R SDK Emergency Monitor.

5.1 Simulator

SimRobot is a kinematic robotics simulator that was developed at the Universität Bremen [45]. It is written in C++ and is distributed as public domain [48]. 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 is an intermediate release that is only available as part of the GermanTeam 2004 code release. The official release, called SimRobot 2005, will include a physical simulation as well as a new generic description language for robot simulations, and will be released in spring 2005. 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 F 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.

5.1.1 Simulation Kernel

The kernel of SimRobot models the environment, simulates sensor readings, and executes commands given by the controller. A simulation scene is described textually as a hierarchy of objects. Objects are bodies, 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 as well as the embedding into other application, e. g. *RobotControl* (cf. section 5.2).

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. 5.1) 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.

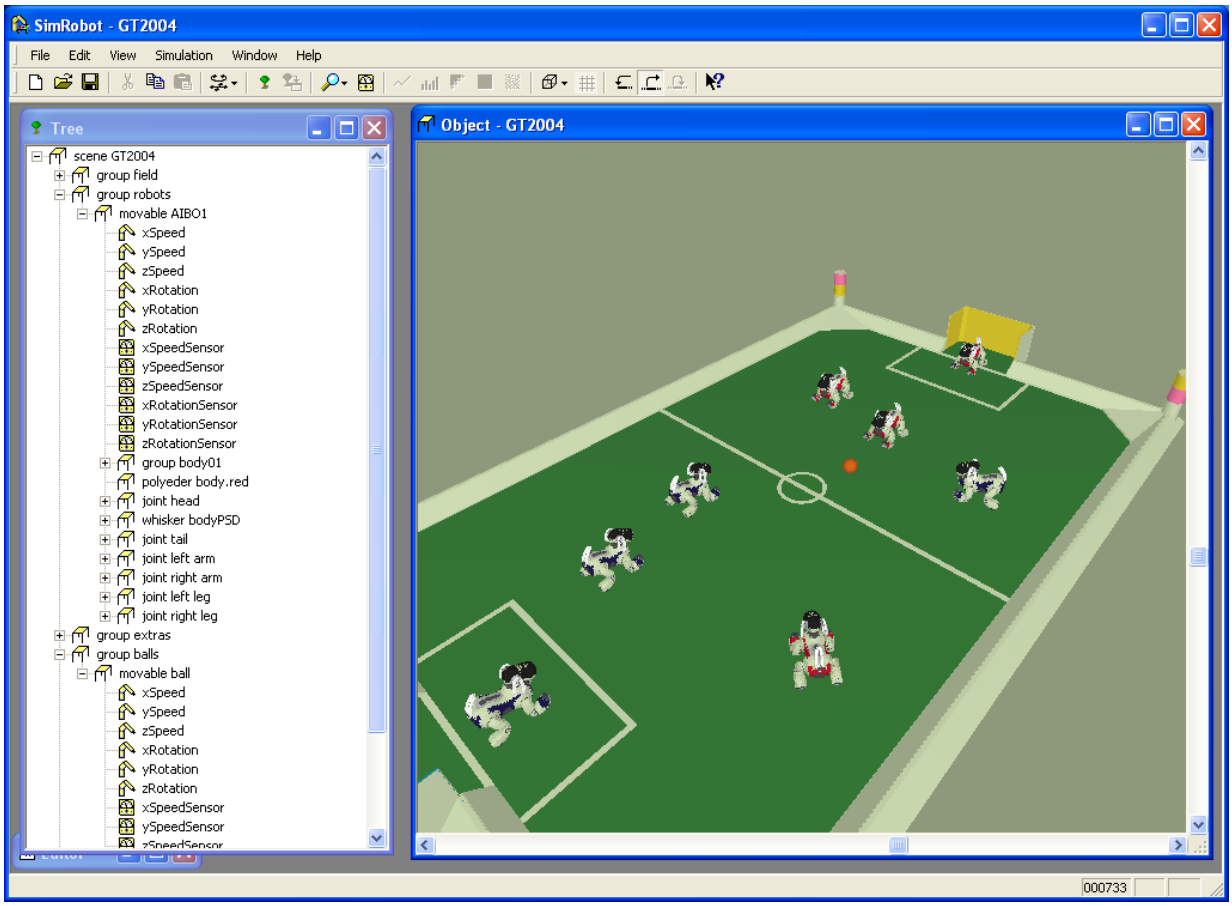


Figure 5.1: SimRobot simulating the GermanTeam 2004.

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 GT2004 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:

- 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.

5.1.2 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. 5.1, 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. 5.2, 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.

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.

5.1.3 Controller

The controller implements the sense-think-act cycle; it reads the available sensors, plans the next action, and sets the actuators to the desired states. Then, SimRobot performs a simulation step and calls the controller again. Controllers are C++ classes derived from a predefined class *CONTROLLER*. Only a single function must be defined in such a controller class that is called

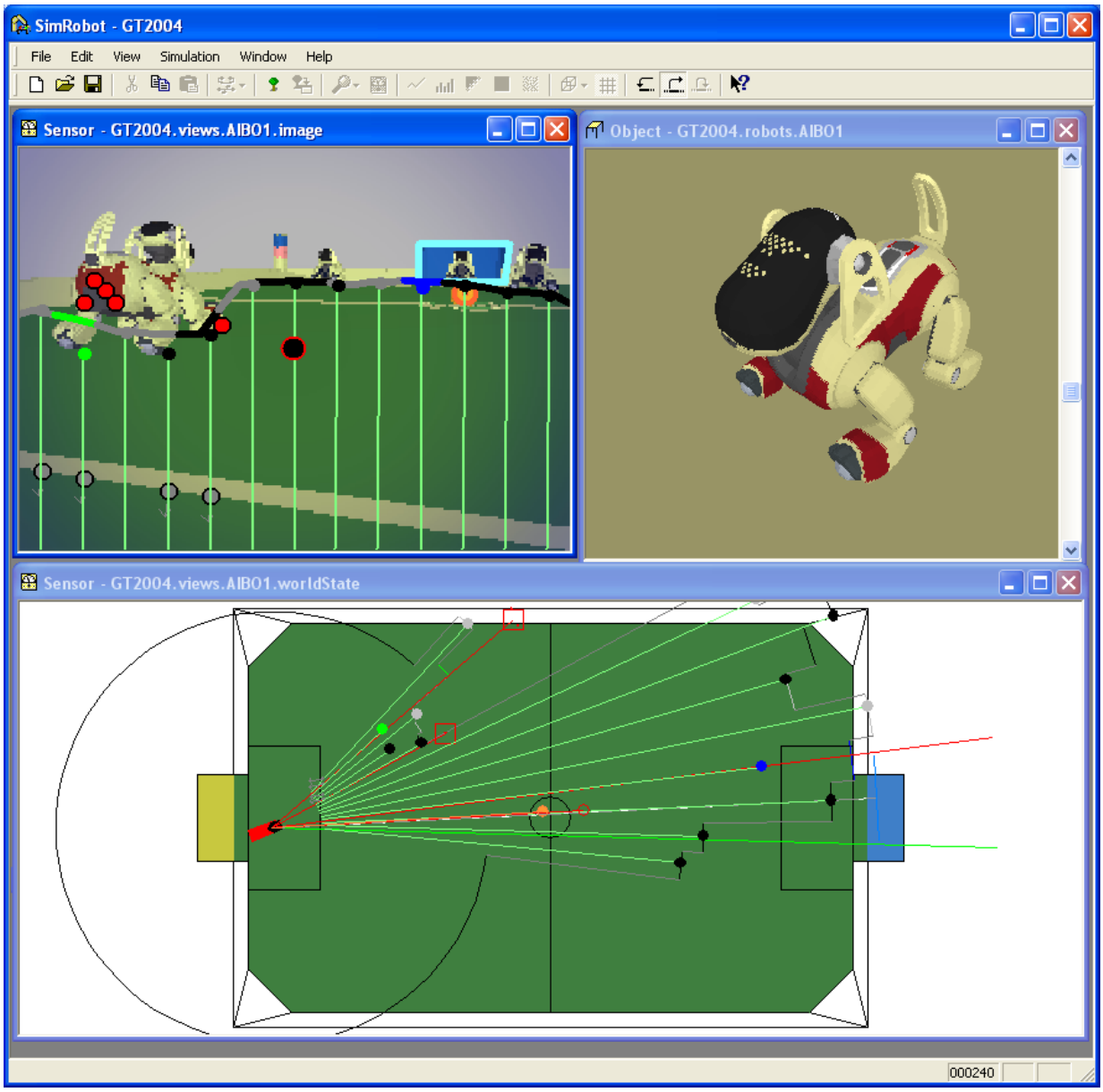


Figure 5.2: SimRobot displaying an image from a simulated camera, a single robot, and a user-defined view.

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 5.2 is a view that contains the soccer field overlaid by a visualization of the robot's percepts and the currently estimated world state.

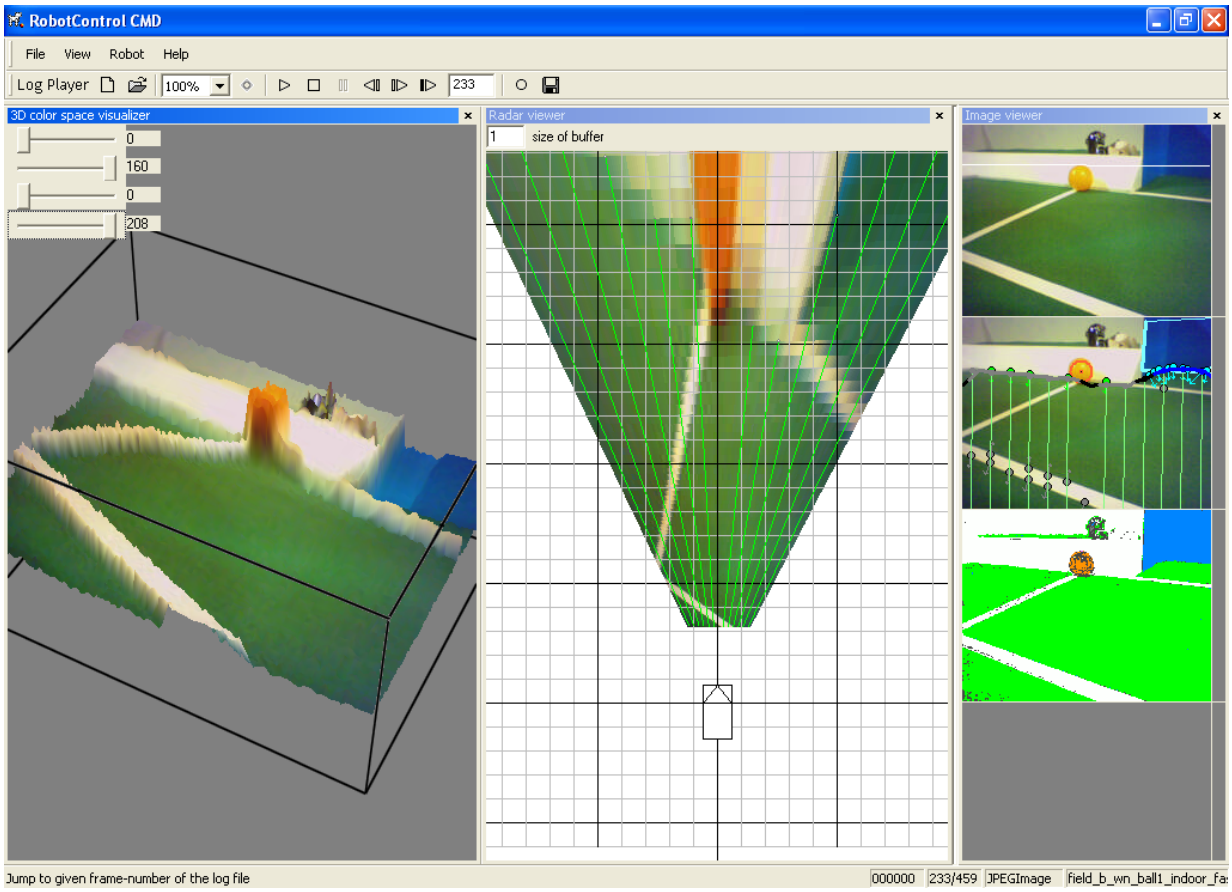


Figure 5.3: The RobotControl application

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

5.2 RobotControl

In contrast to the pure simulator SimRobot (see previous section), RobotControl (cf. Fig. 5.3) was initially intended to be a general support tool that should help to increase the speed and comfort of the software development process.

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

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

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

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

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

This is also one of the two main differences between RobotControl and 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. As many tasks require a graphical user interface, e. g. creating color tables, the Simulator provides only a small portion of the functionality of RobotControl.

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

5.3 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 hopefully CMPack'04 is supported).

5.3.1 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 28 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

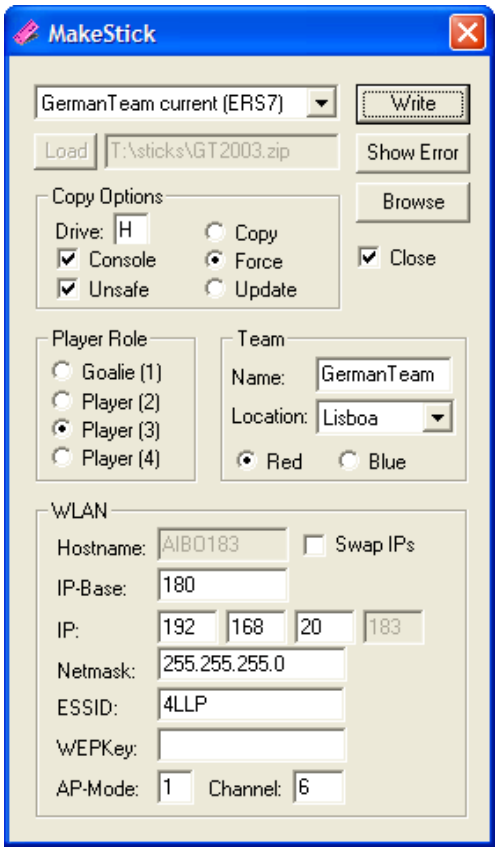


Figure 5.4: The MakeStick tool

the default handler in the properties dialog of the memory stick drive), and MakeStick can be selected as default handler.

5.3.2 Usage

Figure 5.4 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 2004 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*, 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*, and *CMPack* code is supported.

5.3.2.1 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. 5.6) 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 decides whether MakeStick is closed after *Write*, *Show Error*, or *Browse* were successfully executed.

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

5.3.2.2 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.

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.

5.3.2.3 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.

5.3.2.4 Team

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.

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.

Red/Blue specifies the color of the team.

5.3.2.5 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.

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.

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

5.4 Universal Resource Compiler

The *Universal Resource Compiler* (formerly knows as *Motion Net Code Generator*) parses a set of motion specifications described in a special language and compiles it into a C data structure. 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 an xml 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.

For interactively testing motions and their transitions, RobotControl provides a dialog to request specific motions. The requested motion and the transition from the current one will be executed immediately (cf. Sect. D.5.1). Furthermore, a second dialog is provided to transmit new motion descriptions to the robot and execute them without the need to recompile anything (cf. Sect. D.5.3).

5.4.1 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 typical data line looks like this:

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

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

5.5 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 *GT2004/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 and 2004. 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 *GT2004* 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.

5.6 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 *GT2004/Bin/emonLogParser.pl* uses *mipsel-linux-readelf* and *mipsel-linux-objdump* to output an assembler dump around the crashing opcode and around the caller of that routine. The crashing line is highlighted. This is especially useful if the crash happened in

unoptimized binaries with debug symbols. Furthermore the call stack is analyzed to give an idea of the order of calling methods.

The script has to be called by:

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

So, the path to the emon.log of the crash has to be provided as well as the name of the configuration of GT2004 that caused the crash, e. g. *Debug*, *Release* or *DebugNoDebugDrawing*. This will find the correct **.nosnap.elf* in the build directories. It can easily be modified to be used with binaries of other teams. Of course it is only useful to provide the binaries that caused the crash.

Chapter 6

Conclusions and Outlook

The GermanTeam now exists for more than three years. Over the years, the results achieved in the RoboCup competition got better and better, culminating in winning the world championship in 2004. The general architecture developed for RoboCup 2002 has proven to be sustainable, still satisfying our needs. 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]. 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 2004, two other teams (the Hamburg Dog Bots and the Dutch Aibo Team) were also using the GermanTeam code base. The Hamburg Dog Bots even reached the quarter final. Also from a scientific point of view, the GermanTeam provides a good foundation for doing research. In 2003 and 2004, it was the team with most publications at the RoboCup Symposium, e.g., every sixth talk of the RoboCup Symposium 2004 was given by a member of the team [26, 29, 33, 50, 58], and besides there were three poster presentations [20, 27, 59].

6.1 The Competitions in Lisbon

The GermanTeam scored very well at the past RoboCup world championship. During round-robin, we became winner of our Group D. (4:2, 13:0, 6:1, 12:0, 7:0). During quarterfinals, we defeated CMPack with a 9:0 score. We won against NuBots by scoring 9:2. Finally, we got world champion as we bet UTS in the finals 5:3.

[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.

Round Robin	
GermanTeam – rUNSWift	4:2
GermanTeam – Team Chaos	13:0
GermanTeam – ASURA	6:1
GermanTeam – Georgia Tech Yellow Jackets	12:0
GermanTeam – Baby Tigers	7:0
Quarter Final	
GermanTeam – CMPack'04	9:0
Semi Final	
GermanTeam – NuBots	9:2
Final	
GermanTeam – UTS Unleashed!	5:3

Table 6.1: The results of the GermanTeam in Lisbon

As every year, teams were invited to take part at the so called "Technical Challenge". Each team has to demonstrate, that it is able to solve a number of scientific problems with the robots. In 2004, we took part at the so-called "Open Challenge" and scored best (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 German Open in April 2005, 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" [13] 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.

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.

This year we will invest a certain amount of time to develop a new debug tool that enables us to validate our vision system, the modeling methods and the behaviors.

6.2.2 Technische Universität Darmstadt

Until 2001 the GermanTeam was able to measure the true positions of the robots on the (at that time smaller) field with the global view of a web camera mounted above the field at the ceiling of the lab. Currently such a mechanism is newly developed in Darmstadt and will be reintegrated in GT2005 with a much higher accuracy and reliability than the previous approach. This will enable us 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 2005 are to improve the ball modeling based on the Kalman Filter, the recognition and modeling of the other players, the ball passing and cooperation of players, the ball controling 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

Due to a lack of time, the implementation of a fully probabilistic world model was postponed till 2005. Instead, the current localization method was improved, and ball modeling based on the Kalman filter approach was added by team members from Darmstadt. However, a fully integrated model is still required, because the larger field and the lack of field walls will require more precision in game play. Kwok and Fox [32] have already laid the foundation for a more integrated world model. However, their approach ignores the possibility of communicating information between the different robots in a team. While the integration of Kwok's and Fox's approach into the framework of the GermanTeam has already started, further research is required on integrating the perceptions of different robots, because they result from points in time.

Another research topic will be the elimination of manual color calibration. Although several approaches have already been presented [18, 30, 29], non 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.

In 2004, the OpenGL-based version of SimRobot was used to simulate both the ERS-7 and the ERS-210. It is only a kinematic simulator. Currently, SimRobot 2005 is developed. It integrates the *Open Dynamics Engine* (ODE) to perform a physical simulation. This will result in a rather realistic simulation of collisions, ball movement, and walking as well as kicking motions of the robots. The interface to the code of the GermanTeam will not change, so it can be used immediately when it is available.

6.2.4 Universität Dortmund

The University Dortmund will contribute to the GermanTeam code by adapting it to the official changes of the rules (field size enlargements, changes of ball, removal of the border). Further, we will open our ceiling cam resources to the public. We try to help migrating the current Robot-Control to a modular approach that uses the .NET framework. Additionally, we will give our best to place in at least one of the annual challenges.

Other work at the code is already cover by our scientific goals which are described in Section 1.2.4.

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 2004 for travel support. The team members from Berlin, Bremen, and Dortmund thank the Deutsche Forschungsgemeinschaft (DFG) for funding parts of their respective projects. The team from Darmstadt thanks Deutscher Akademischer Austauschdienst (DAAD[1]) for travel support to Lisbon, as well as Vereinigung von Freunden der Technischen Universität zu Darmstadt e.V. [2], Siemens [3] and Fachbereich Informatik of TU Darmstadt [4] for financial support. Allied Vision Technologies [5] supplied Darmstadt with the ceiling camera. The team members from Dortmund thank Microsoft MSDNAA and Lachmann & Rink GmbH for their effective cooperation and sponsoring.

The members of the GermanTeam 2004 also want to thank the members of the GermanTeam 2002 and 2003 for creating the foundation for the continuing success of the GermanTeam and for writing the previous year's team reports [12, 46] 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.
- 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.daad.de
[2] http://www.tu-darmstadt.de/freunde
[3] http://www.ct.siemens.de, http://www.siemens.com
[4] http://www.informatik.tu-darmstadt.de
[5] 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 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**GT2004*. 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 *GT2004.dsw* (Visual Studio 6) and *GT2004.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. G), 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. 5.5) 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 2033.Net) project files from these dependencies generated by *Depend* (cf. Sect. 2.2.4.4).

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

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

SimRob95 contains parts of the old version of the kinematic robotics simulator SimRobot that is used in RobotControl (cf. Sect. 5.2).

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

A.3 The Developer Studio Workspace GT2004.dsw/.sln

The Developer Studio Workspace GT2004.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 *_GT2004*, *_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.

_GT2004 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 WLAN configuration dialog of RobotControl (cf. Sect. D.2.4) to ensure that the same configuration parameters are used in RobotControl and on the memory stick.

RobotControl can be built in different configurations. The executable will be copied to *GT2004\Bin*. The *Win32 CMD Release* configuration uses statically linked MFC libraries, the Debug configurations dynamically linked ones. The configuration *Win32 Debug Optimized* uses $G6$ optimization, but does not support *Edit and Continue*. *Win32 Debug* supports it but is not optimized.

Simulator currently only supports a single configuration. The executable, together with the help file, will be copied to *GT2004\Bin*.

The other projects generate libraries (*GUI*, *GuiLib*, *SimRobotCore*, *SimRobotForRobotControl*), source code (*SpecialActions* by executing the universal resource compiler *URC* cf. Sect. 5.4)), tools (*Depend* cf. Sect. 5.5), or preprocessed behavior (*Xabsl2* 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 *Xabsl2*. It produces a pretty good documentation of the behavior.

Appendix B

Getting Started

If you have installed the required software and the source code we suggest you to follow the introduction to GermanTeam's code given in this section. Step by step it is explained how to 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 *GT2004\Config* on the PC. In the code release, the file contains the name "Lisboa", 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 binary file consists of three values (each four bytes and little endian):

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

B.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 GT2004
```

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 *Debug Message Generator Dialog* (cf. App. D.6.3) 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 *GT2004SelfLocator* 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 [48], 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 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, the simulator uses a new simulation kernel that employs OpenGL for rendering, and therefore, is much faster. 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. 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 requires a lot less synchronization, which also makes the simulator faster than RobotControl. On the other hand, there are a lot of things that cannot be done with the simulator, e. g. creating color tables and all kinds of OpenGL visualizations, etc. And, in fact, if very different tasks have to be performed in a row as, e. g., during a contest, the mouse-enabled interface of RobotControl is much more comfortable.

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

C.2 Getting Started

The simulator can either be started directly from the Windows Explorer (from *GT2004\Bin*), from Microsoft Developer Studio, or by starting a scene description file[1]. In the first case, a scene description file has to be opened manually, whereas it will already be loaded in the latter two cases. When a simulation is started for the first time and no layout has been patched into the Windows registry, only the editor window will show up in the main window. Select *Simulation|Start* to run the simulation. The *Tree View* will appear. A *Scene View* showing the soccer field can be opened by double-clicking *scene GT2004*. 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

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

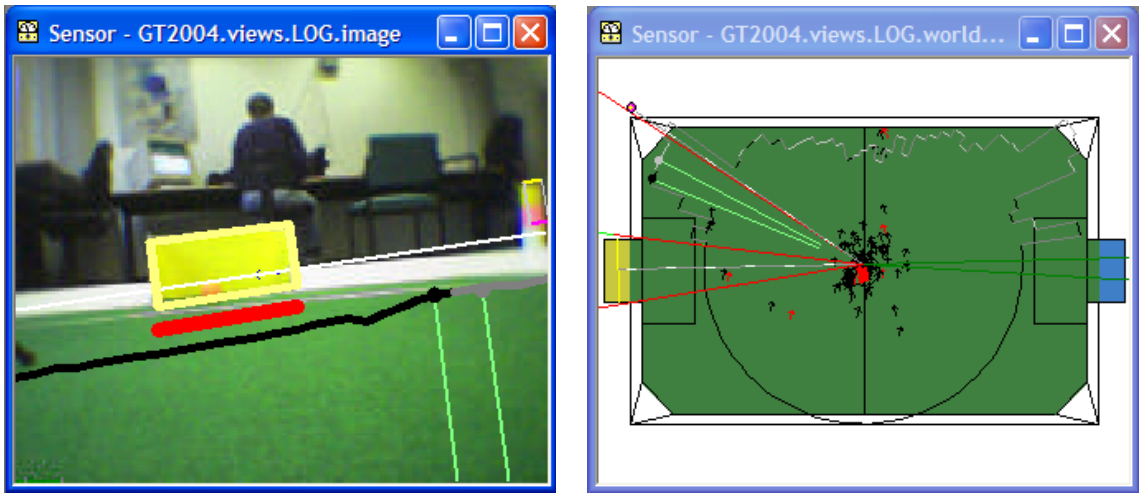


Figure C.2: Image view and field view with several debug drawings

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.

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*. In the simulator, the views are defined in the source code. They are instantiated separately for each robot. There are five kinds of views related to information received from robots: *image views*, *field views*, *Xabsl views*, *sensor data views*, and *timing views*. Field and image views display debug drawings received from the robot, whereas the views visualize 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 *GT2004\Src\Platform\Win32\SimRobot\RobotConsole.cpp*.

C.4.1 Image Views

An image view (cf. left of Fig. C.2) 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 *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
dk send_selfLocator_drawing every 100 ms
```

C.4.2 Field Views

A field view (cf. right of Fig. C.2) 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,
```

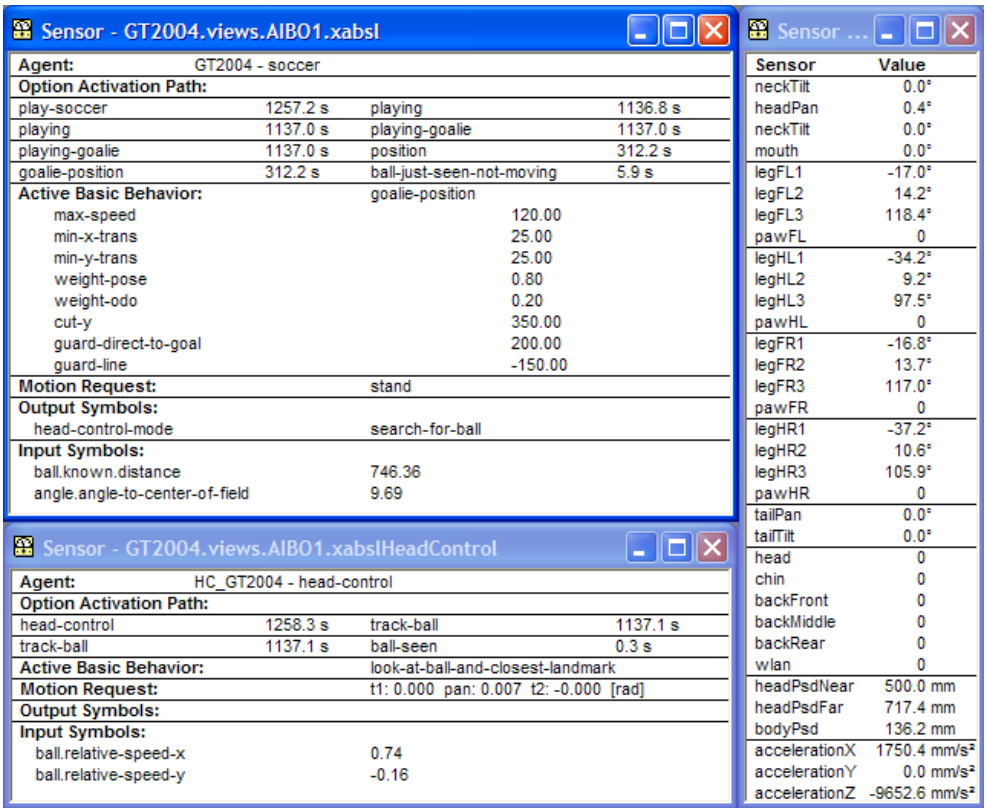


Figure C.3: XABSL views and sensor data view in the simulator

```
  Drawings::worldState,
  Drawings::percepts_ballFlagsGoalsField
END_VIEW(worldState)
```

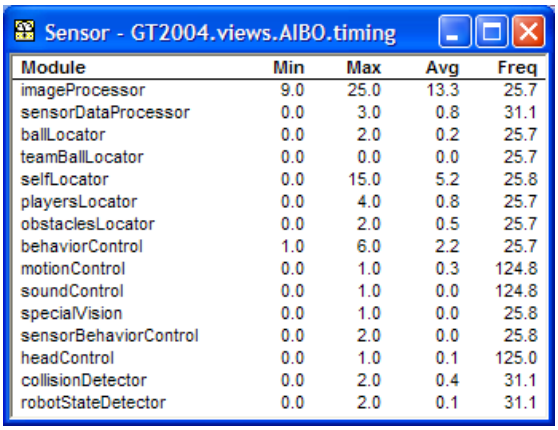
To display all this information, console commands (cf. Sect. C.6) such as the following are also required:

```
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.4.3 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



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.4: The timing view in the simulator

been set, and a Xabsl behavior that matches the one loaded by the console command *xlb* (cf. Sect. C.6.3) must be active on the robot.

```
# set behavior control solution to GermanTeam 2004
sr BehaviorControl GT2004-soccer

# load behavior of the GermanTeam 2004
xlb gt04

# 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.4.4 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 *sendSensorData* must have been executed (cf. right view in Fig. C.3).

C.4.5 Timing View

The timing view displays statistics about the speed of certain modules (cf. Fig. C.4). 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 *GT2004.scn*, *Demo.scn*, and *Match.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 *GT2004\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, *GT2004\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 *GT2004\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 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

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).** 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.

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** | **clear** | **save** <**file**>**.** Records a log file. *log start* starts or continues recording all data received from the robot. *log stop* stops the recording. *log clear* removes all recorded data from memory. *log save* stores the data recorded to the log file with the name specified. If the file already exists, it will be replaced. If no path is given, *GT2004\Config\Logs* is used as default. Otherwise, the full path is used. *.log* is the default extension of log files. It will be automatically added if no extension is given.

mr ? [<**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, *GT2004\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 ballHolding | **keeperCharged** | **playerCharged** | **illegalDefender** | **illegalDefense** | **obstruction** | **pickup** | **continue.** Penalize robot. The command sends one of the seven 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 off | **on.** Switch sending of *oracled world states* on or off. *Oracled world states* are normally sent to all processes. This allows the modules calculating the world state to be switched off without a failure of the robot. However, the option can produce confusing results if parts of the world state are only sometimes calculated by the robot. Then, the world state sometimes results from the robot's own calculations and sometimes from the simulator. Therefore, sending oracled world states to the robots can be switched off. By default, it is switched on. Note that this command only has an effect on simulated robots.

sr ? [<**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.

xlb [**hc**] **(?** [<**pattern**>] | <**name**> **).** Load a Xabsl behavior. The command loads the symbols for the specified behavior and will send the compiled version of the behavior to the robot. The command must be executed before any other Xabsl command and the *Xabsl View* will work. Type *xlb ?* to list all available behaviors. The resulting list can be shortened by specifying a search pattern after the question mark. Please note that the behavior loaded has to match the solution for *behavior control* selected on the robot. To use the *Xabsl View*, the corresponding debug key has to be set, i. e. *dk sendXabslDebugMessagesForBehaviorControl on* for the behavior control and *dk sendXabslDebugMessagesForHeadControl on* for the head control. 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.

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 GT2004

# 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 GT2004ImageProcessor. It instantiates a robot named *LOG1* that is fed by the data stored in the log file *GT2004\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 GT2004
sr SelfLocator GT2004
sr BallLocator GT2004
sr PlayersLocator GT2004
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

RobotControl Usage

This chapter describes how to use the RobotControl application. As RobotControl is a very complex system, not all features will be described. But it will help to get an overview about the capabilities of the program.

D.1 Starting RobotControl

Requirements. RobotControl needs at least version 4.0 of the Microsoft Internet Explorer installed on the system to work properly. In addition, it is important that RobotControl is started from a directory under *GT2004*.

After the First Start The application looks a little bit strange. No child windows appear and all tool bars are pushed together. But the toolbars can be moved with the mouse to be distributed over more rows. To switch toolbars on or off, right-click on the toolbar area and select the visible toolbars from the pop-up menu. Dialogs can be opened by using the “View” menu. Note that the window layout will not be restored during the next start of the application unless it is saved using the “Screen Layout” from the “View” menu.

D.2 Application Framework

The following toolbars and dialogs form the framework of the application.

D.2.1 The Debug Keys Toolbar

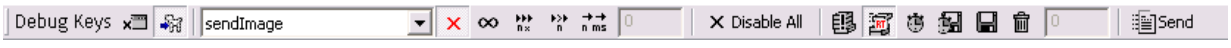


Figure D.1: The Debug Keys Toolbar

The *Debug Keys Toolbar* (cf. Fig. D.1) is used to switch debug keys on or off. Each debug key can be parameterized in four different ways describing how often and in which frequency it will be enabled.

The combo box contains all available debug keys. To edit the properties of a debug key, select the key from the list and use on of these buttons:

Disabled. The debug key is disabled.

Always. The debug key is always enabled.

n **times.** The debug key is enabled for n times, i. e., it will return *true* the next n frames, and *false* afterwards. n has to be entered into the edit control before the button is pressed.

Every n **times.** The debug key is enabled every n-th frame, i. e., it will return *true* every n-th call, and *false* in between.

Every n **ms.** The debug key is enabled every n milliseconds, i. e., it will return *false* until at least n milliseconds passed since the last time it returned *true*.

Disable All. Disables all debug keys.

There are five buttons to select how the outgoing message queue is treated on the robot:

Immediately. All outgoing messages are sent immediately via the wireless network.

Realtime. This mode allows dropping messages if there is not enough time to transmit. This is useful, e. g., for sending as many images as possible without slowing down the robot.

Send after n **seconds.** The transmission of outgoing messages is delayed for n seconds. The value of n is set with the attached edit field.

Save to stick after n **seconds.** Instead of transmitting outgoing messages via the wireless network, they are stored on the robot's memory stick after a delay of n seconds. A log file is created on the memory stick which afterwards can be replayed using the *Log Player Toolbar* (cf. Sect. D.2.3). The value of n is set with the attached edit field.

Reject all. All outgoing messages are dropped.

There are two debug key tables in RobotControl, one for a physical robot connected via the wireless network, and one for the selected simulated one. With the buttons *Edit table for robot* and *Edit table for local processes* one can select which of them is edited. *Send* sends both debug key tables to the proper destinations by putting them into message queues, i. e. nothing will change as long as *Send* has not been pressed.

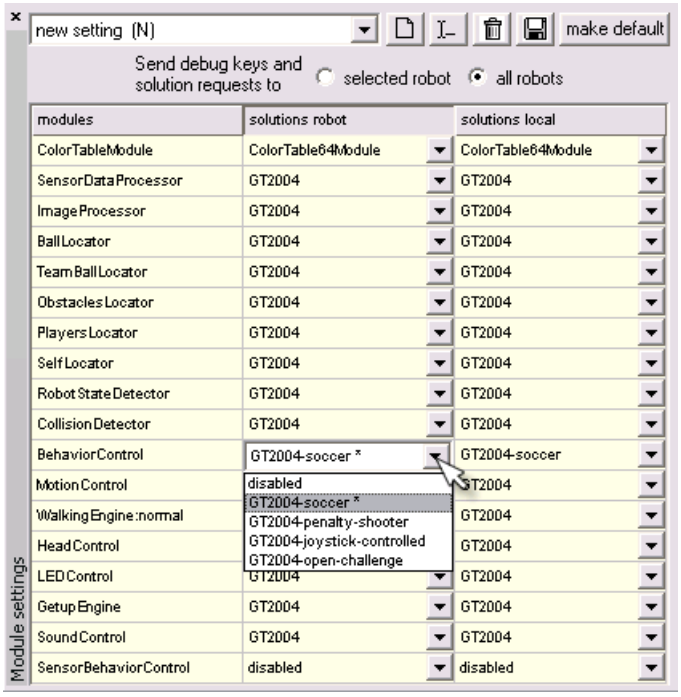


Figure D.2: The Settings Dialog: RobotControl's most often used dialog

D.2.2 The Settings Dialog

With the *Settings Dialog* (cf. Fig. D.2), solutions for modules (cf. Sect. 3) running on the robot or on the PC can be switched. A certain combination of solutions is called a setting and can be stored. All settings are stored in *GT2004\Config\Settings*. The default solution for each module is marked with an asterisk.

D.2.3 The Log Player Toolbar

Figure D.3: The Log Player Toolbar

In RobotControl, log files can store a set of messages of any kind used to communicate between modules or between dialogs or toolbars and modules, e. g. it is possible to record pictures sent by a robot in a log file, and then play that log file several times to test different kinds of image processing with exactly the same input data.

The *Log Player Toolbar* (cf. Fig. D.3) is used to record, play, and modify such log files. Its buttons should be known from other players, e. g. CD-players. *Play*ing a log file sends all messages in it to all running modules in RobotControl as well as to all dialogs that can handle that type of message. *Step forward* and *Step backward* do the same with single messages.

*Record*ing appends all messages sent from a robot via the wireless network to the actual log file (in memory) that can be *save*d afterwards.

D.2.4 WLan Toolbar

Figure D.4: The WLan Toolbar

The *WLan Toolbar* (cf. Fig. D.4) is used to create, edit, and switch between different wireless network configurations. It also allows connecting to (and of course disconnecting from) all enabled robots in the current wireless network configuration.

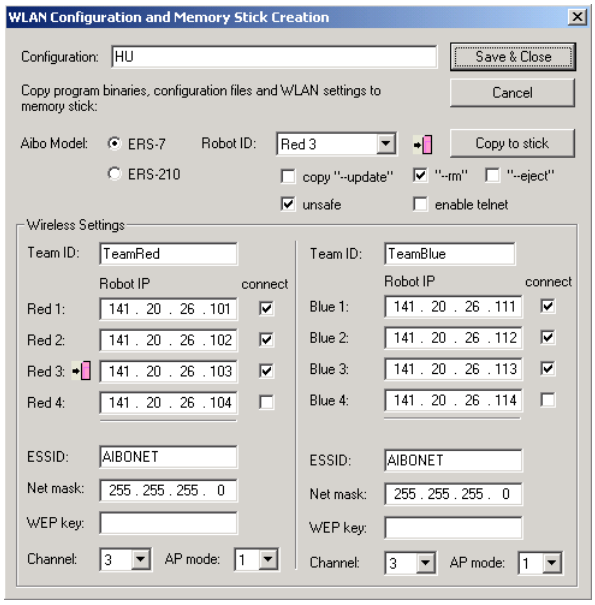


Figure D.5: The WLan Connection Parameters Dialog

Creating new and modifying existing wireless network configurations opens the dialog shown in figure D.5. The dialog allows all relevant parameters to be specified: the IP addresses of the robots as well as their TeamID (for Dog Discovery Protocol) and the hardware type of the robots.

Furthermore settings such as the wireless network *ESSID*, the *Netmask*, the *APMode*, the *WepKey*, and the *Channel* can be edited. Pushing the *Copy to stick*-button calls *copyfiles.bash* with all parameters of the currently selected robot (marked with a memorystick symbol) in the dialog. So by using this dialog it is possible to keep the settings written to memory sticks and the settings used to connect to robots consistent.

Figure D.6: The Game Toolbar

D.2.5 Game Toolbar

With the *Game Toolbar* (cf. Fig. D.6) the game control data can be generated and sent. So the RoboCup Game Controller is not needed for testing behaviors. However, this dialog does not support all functions of the Game Controller.

The slider at the right adjusts the *game speed* (from 0.1 to 1). This allows for testing behaviors in "slow motion". The value is multiplied to the translation vector of the motion request processed by the *MotionControl* module.

D.3 Vision Related Tools

D.3.1 Image Viewer and Large Image Viewer

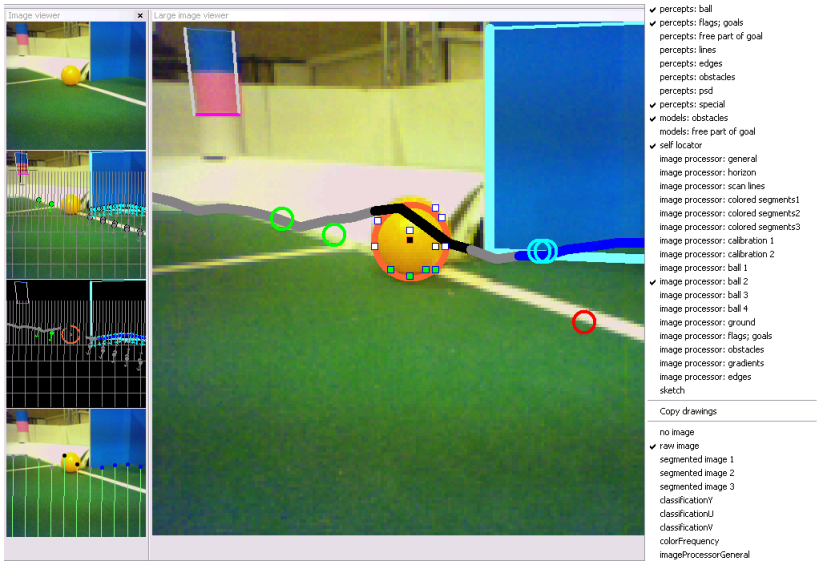


Figure D.7: Image Viewer and Large Image Viewer Dialog

The two image viewer dialogs (cf. Fig. D.7) display images and debug drawings from the *queueToGUI* (cf. Fig. J.1) so images from the robot, the log player, or the simulator are displayed. The *Image Viewer* has space for eight images with a fixed size. The *Large Image Viewer* shows only one image and is sizable. With the context menu different types of images and different debug drawings can be selected. The context menu also contains *Copy drawings* and *Copy image and drawings* which copies only the debug drawings as a vector graphic or the image with the draw-

ings as a bitmap to the clipboard. Important: to see a debug drawing created in a special solution of some module (cf. Sect. 3), this solution has to be selected (cf. Sect. D.2.2).

D.3.2 Field View and Radar Viewer

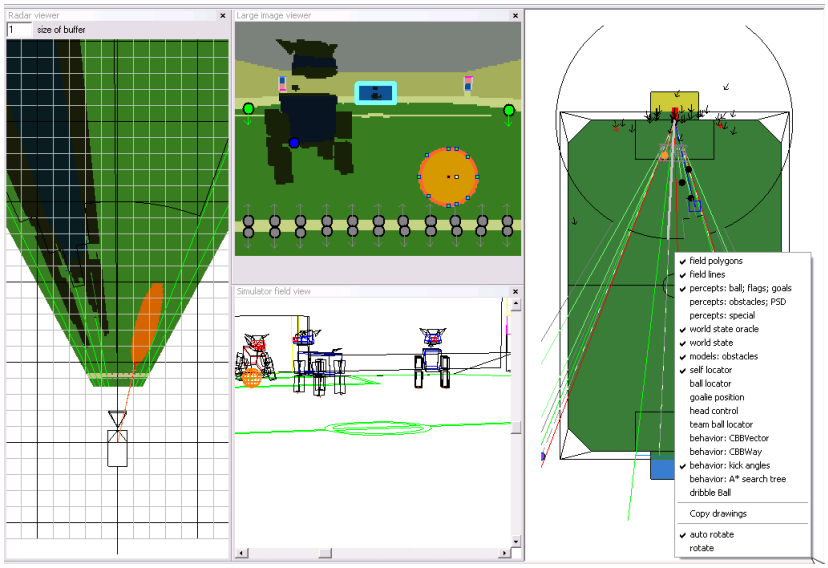


Figure D.8: RobotControl's Field View and the Radar Viewer Dialog

The *Field View* and the *Radar Viewer* (cf. Fig. D.8) both display percept drawings. The *Radar Viewer* displays the percepts relative to the robot. The *Field View* draws the percepts based on the robot's localization on the field. The field view is also used for world state and localization drawings. In both dialogs the drawings can be selected with the context menu. *Copy drawings* copies the drawings as a vector graphic to the clipboard.

D.3.3 Radar Viewer 3D

The *Radar Viewer 3D* (cf. Fig. D.9) provides a 3D visualization for percepts, but also for the image. The percepts are displayed in the 3D space after a transformation of the percepts into the robot's system of coordinates of using the camera matrix. The sliders can be used to adjust the position of the camera.

D.3.4 Color Space Dialog

The *Color Space Dialog* (cf. Fig. D.10) visualizes how an image uses the YUV color space, and it displays either the y, u, or v channel as a height map. By dragging with the left mouse button the 3-D scene can be rotated. With the context menu the type of view can be selected.

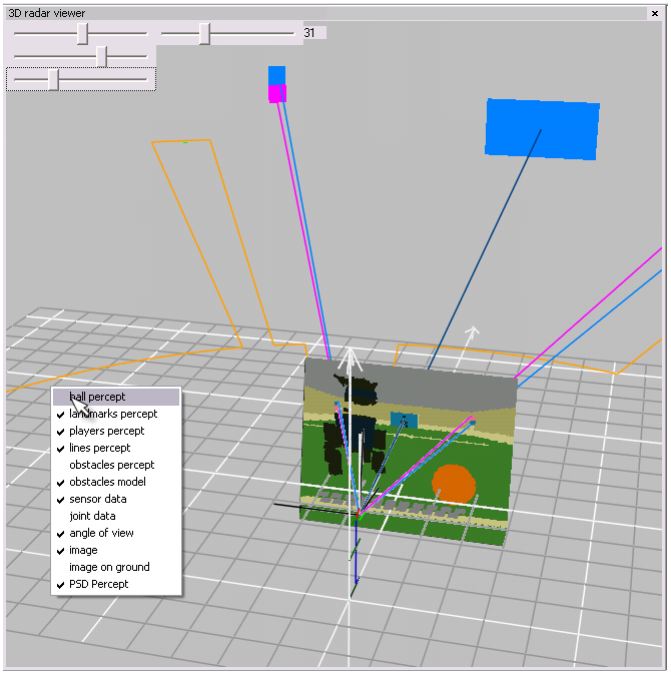


Figure D.9: The Radar Viewer 3D

D.3.5 The Color Table Dialog

The *Color Table Dialog* (cf. Fig. D.11) is used to create color tables for image processing. The current image from the simulator, a logfile, or the robot's camera (via WLAN) is shown top left (cf. Fig. D.11). Beneath the same image can be seen, segmented with the current color table. Choosing a color class and clicking with the left button on a pixel in one of the two images will assign the color class to the $4 \times 4 \times 4$ cube in YUV color space according to the color value of the pixel. The result takes effect immediately and the segmented image will be updated. Clicking with the right button on a pixel will remove the according color class assignment. There are also functions to undo the last assignment, to clear all assignments for a whole color class and to reset the complete table.

To speed up the process of creating a color table, pixels with no neighbors of the same color class can be removed by pressing the *remove* button. For large areas of the same color class, it is useful to add pixels with at least a given count of neighbors and a maximum distance in the color space. This can be done by pressing the *median* button. Both optimizations can work with the selected color class or with all colors (*all colors* checkbox selected). If existing assignments should not be overridden, *allow reassign* checkbox should not be selected to prevent unwanted changes.

With *auto-remove/median* enabled, the optimizations are triggered with every incoming image. This is useful if logfiles with specific colors have been recorded. The color table can be created without many manual assignments.

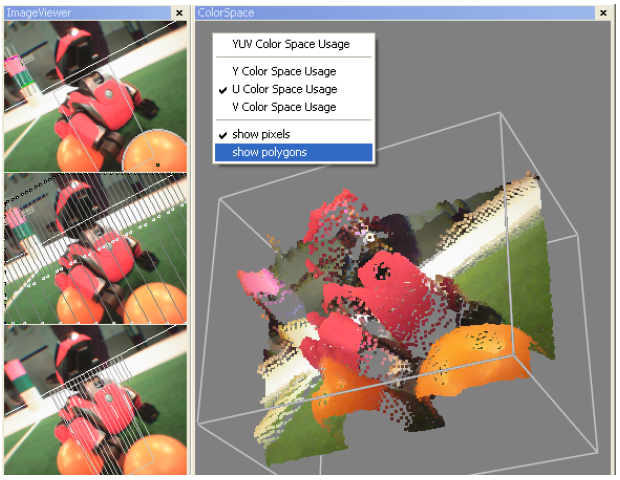


Figure D.10: The Color Space Dialog showing the u-channel of an image as a height map.

After having assigned all colors needed, the table can be sent to the robot via WLAN or saved to a file. The GT2004 vision modules need a file named *coltable.c64*. It is also possible to load an existing file and to modify it.

D.3.6 HSI Tool Dialog

This tool uses the HSI color space to create a color table. It allows the user to specify the minimum and maximum values for hue, saturation, and intensity for each color class using sliders, and it gives an instant feedback by real time color classification of four images at the same time. These images are directly taken from the memory of *RobotControl* and can be held as long as needed. It is also possible to update an image simultaneously with the one in the memory of *RobotControl*. The tool saves the color table both in HSI and YUV format. By selecting an image there is another dialog with a zoomed view (cf. Fig. D.13) of it and the belonging color classified image. In this dialog the color class can simply be edited by selecting single pixels. There is also an undo function for several steps.

This HSI approach leads to good results very quickly by defining big sectors in the color space and it is more tolerant against changing light conditions. The tool can be combined with the other color table tool using YUV color space. First, the HSI tool is used to quickly generate a color table and the YUV tool can be employed for fine tuning if required.

Dialog Structure. The main dialog (cf. Fig. D.12) consists of two parts. In the upper half are spaces for four RGB images and below them the corresponding color classified images will be displayed. Below each space there are buttons for capturing an image, and at the left is a check box for an automatic update of the image above it.

In the lower half of the dialog most of the controls are located. There is a combo box with entries for all color classes used by the image processing module, a button for loading HSI color tables, a button for saving the HSI and the corresponding YUV color table, and six sliders for

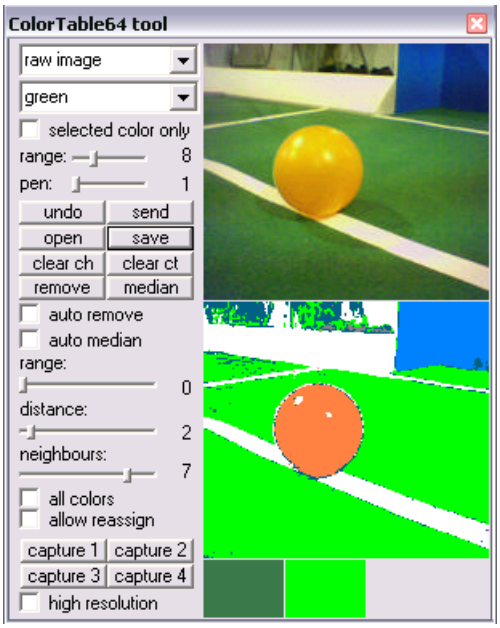


Figure D.11: The Color Table Dialog

determining the range of the selected color class in the HSI color system. An HSI color class consists of a minimum and a maximum value for hue (H), saturation (S), and intensity (I).

Main Dialog. With the *capture* buttons in the upper half of the main dialog (cf. Fig. D.12) the actual image that *RobotControl* has in memory can be saved. It can be done for four images. When the box before *update* below the left place is checked, this image is updated automatically, when *RobotControl* gets a new image from the robot or a log file. The color classified image will be updated automatically when you change the range of a color class.

With the combo box at the lower left of the dialog the color class to be edited can be selected. The sliders show the minimum and maximum values of this color class. The ranges for H, S, and I can be modified by changing the positions of the sliders. While moving a slider the color classified image is permanently updated.

The range of values for the hue of a color class goes from 0 to 360, and for saturation and for intensity from 0 to 100. Because the value for hue in HSI color system lies on a circle, it is possible that the maximum value for this range is smaller than the minimum value. For a red color, e. g., the minimum of the hue range could be 350 and the maximum could be 15. For saturation and intensity the minimum value should be below the maximum value.

The color table can be saved by pressing the *Save* button. It appears a file dialog where the destination and the name of the color table can be selected. The suffix *.hsi* will be added automatically. A YUV color table converted from the HSI color table will also be saved with the same name and the suffix *.c64*.

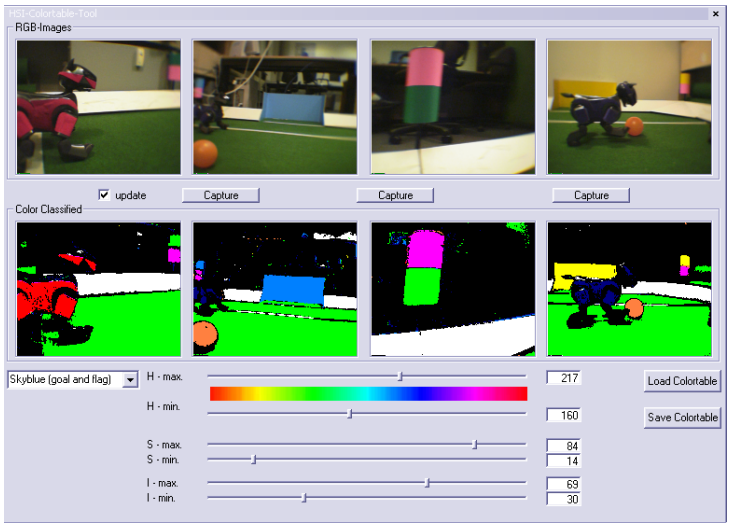


Figure D.12: The HSI Tool Dialog

An HSI color table can be load by pressing the *Load* button. It can be selected in the appearing dialog. It is only possible to load HSI color tables. YUV color tables are not supported yet by the HSI tool.

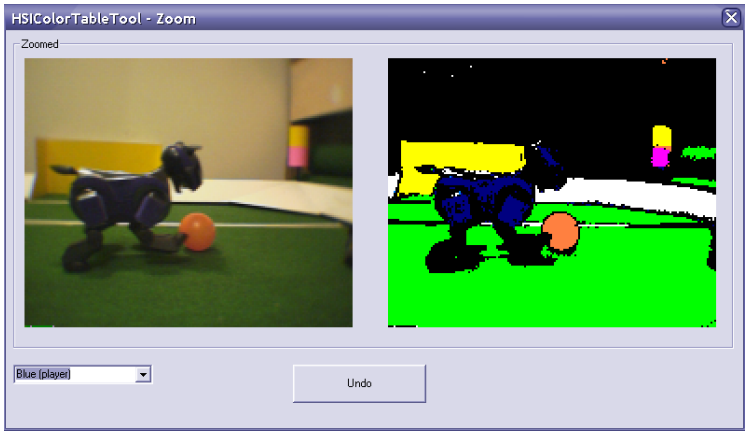


Figure D.13: The HSI Tool Zoom Dialog

Zoom Dialog. By performing a click with the left mouse button on one of the RGB images, another dialog window with a zoomed view (cf. Fig. D.13) of the selected image and the belonging color classified image will appear. In this dialog it is possible to improve the precision of the ranges for color classes by selecting single pixels. At the lower left the combo box with the color classes to edit is located. There is also an *Undo* button for the last six changes.

By pressing the left or the right mouse button on a pixel in one of the zoomed images the selected color class will be modified. If the left mouse button has been pressed and the color of

the selected pixel lies outside the color class, the range will be enlarged until the values for hue, saturation, and intensity are inside this range. If the right mouse button has been pressed and the selected pixel lies inside the range of the color class, the range will be reduced until the values for hue, saturation, and intensity are outside of it.

By selecting single pixels, it is possible to determine which color should belong or not belong to the chosen color class. Thus precision of the class can be improved.

D.3.7 The TSL Color Segmentation Dialog

The main idea of color segmentation is to partition the chrominance space into subspaces, where each subspace represents exactly one color. The algorithmic performance of color classification depends on the chosen chrominance space. The reason for that is, that the complexity of color segmentation strongly depends on the shape of the subspaces. If the normal vectors of all bounding hyperplanes are parallel to the unit vectors of the chrominance space coordinate system, then the color assignment can be done at highest efficiency. Therefore, we define a new chrominance space where all relevant colors can be extracted by using a set of thresholds $t_1^c, \ldots, t_6^c$ per color c. For our purpose, a classification is needed to distinguish between the main nine RoboCup colors (green, sky-blue, yellow, orange, pink, dark blue, red, black and white). The so-called TSL* chroma-space based color-segmentation is more robust against luminance variation than similar YUV-based algorithms.

The TSL Dialog is a powerful tool for creating a set of thresholds for a robust color classification. On the left side of the dialogbox, two images are displayed. The upper images contains the raw data (either camera data or simulator output). The lower images shows the result of the color classification.

The user can select a color (e.g. ”yellow” in the *Select color* list in Fig. D.14) and a corresponding region in the image by clicking on the upper left and lower right corners of a rectangle. Then, a histogram of each color component (T’, S’, and L’) is calculated in real-time and displayed immediately. By pressing the *Auto* button, threshold values are set automatically. Using the sliders, the user can adjust and optimize the thresholds (in Fig. D.14: $t_1^{yellow}, \ldots, t_6^{yellow}$) manually. The influence of parameter values on the color classification can be seen in the lower left image. The robustness of the classification can be checked by adding noise using the *Noise* slider to the original image.

If areas overlap, a classification is not unique. Thus, the order of color classifications is important. With the *Up* and *Down* buttons, the priority of a selected color can be changed. The priority is displayed in the *TSL Order* list. Settings can be saved and reloaded by using the *Load* and *Save* buttons. The *Save YUV* button stores the settings in the conventional YUV color table format. A set of threshold values can be send to a connected robot by clicking the *Send* button.

D.3.8 Camera Toolbar

The *Camera Toolbar* (cf. Fig. D.15) is used to set the parameters that are provided by the robot’s camera. These parameters are set with the combo boxes. White balance may be set to indoor,

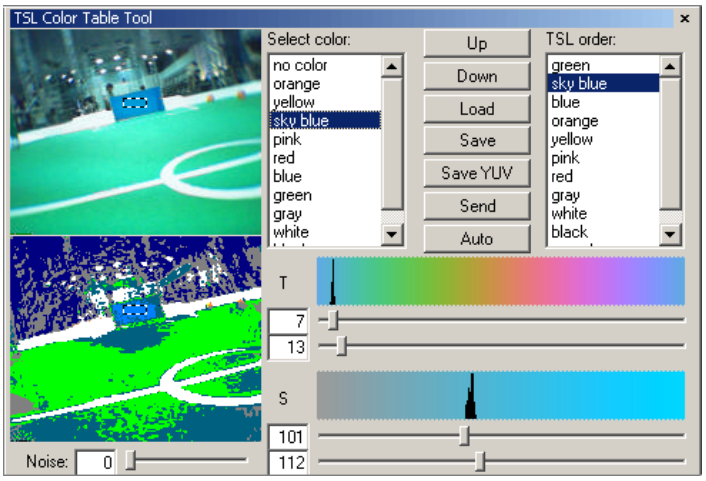


Figure D.14: The TSL Color Table Tool.

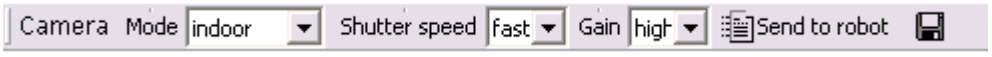


Figure D.15: The Camera Toolbar

outdoor, or FL mode. Shutter speed may be selected from slow, medium, or fast. Finally low, medium, or high camera gain can be chosen.

The *send to robot* button sends the selected parameters via the wireless network to the robot. The new settings are applied immediately. When viewing the camera pictures with the image viewer (cf. Sect. D.3.1) the effects of different camera settings can be observed.

The *save* button writes the settings to a file named *camera.cfg*. This file is loaded at the start of RobotControl to initialize the *Camera Toolbar* with its contents. But more important this file is copied to the memory stick and loaded when booting the robot. The settings from this file are used on the robot unless different parameters are sent with the toolbar.

D.4 Behavior Related Tools

D.4.1 Xabsl2 Behavior Tester

The *Xabsl2 Behavior Tester* is a debugging interface to the behavior modules derived from *Xabsl2BehaviorControl* (cf. Sect. 3.8) and the HeadControl. One can view almost all internal states of the engine. *Options* and *Basic Behaviors* and *Output Symbols* can be selected manually for separate testing.

With the check box *test on robot* in the right upper corner one can set whether the behavior shall be tested on the robot or in the simulator.

In the topmost combo box an option or basic behavior can be selected. If an option is selected it is used as the root option. The execution of the option tree starts from that option. If a basic behavior is selected, only this behavior gets executed without any option tree being traversed.

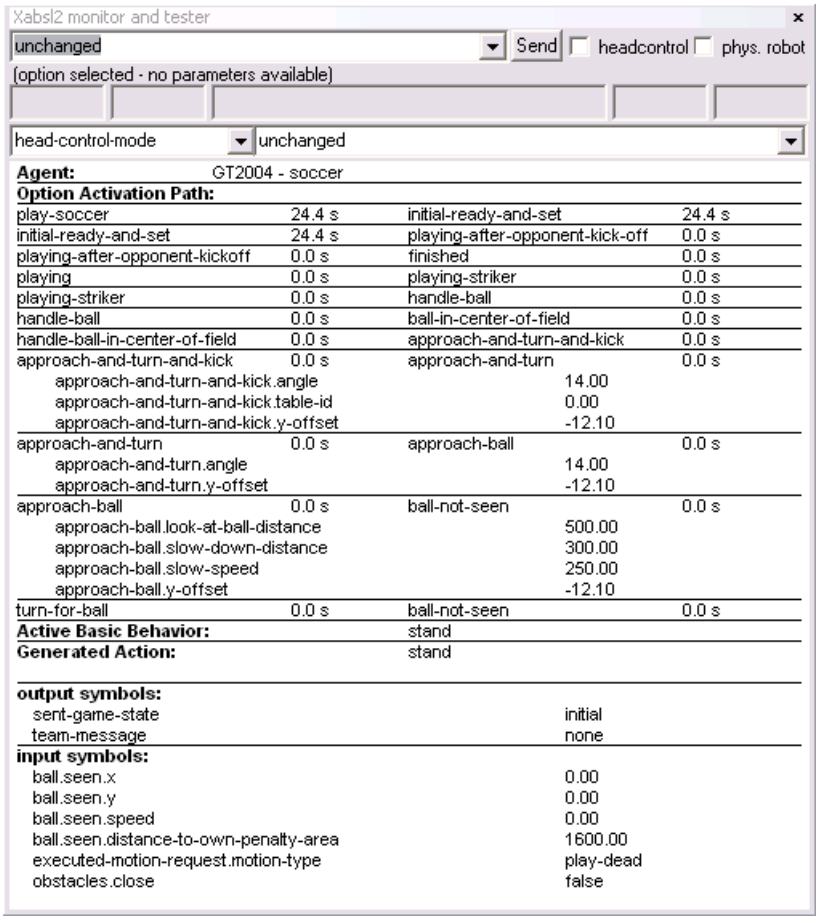


Figure D.16: The Xabsl2 Behavior Tester Dialog

This allows testing single options and basic behaviors separately. If the option or basic behavior selected has parameters these can be entered into the edit fields below.

The following two combo boxes allow altering output symbols manually. The left box selects an output symbol while the right one sets its value. If this is done the value generated through the engine by traversing the option activation path is ignored and overwritten with the set value. This is useful for testing output symbols separately.

At the top of the white area *Agent* shows the name of the active Xabsl2 agent. Next the *Option Activation Path* is displayed. The first column shows all the activated options. The second column shows the time how long these options are already activated. Then in the third column the active state of each option, and in the fourth column the time how long the state is already active, are displayed. If an activated option is parameterized its current parameter values are shown below.

Then the *Active Basic Behavior* shows, which basic behavior is currently activated along with its parameter values. The *motion request* shows the motion request that resulted from the execution of the basic behavior.

The *Input Symbols* section shows the current values of the input symbols selected. The selection, which symbols shall be displayed, can be done with the context menu of the dialog. The same applies to *Output Symbols*.

At last, in the context menu exists an entry *Reload Files*. The dialog rereads the debugging symbols and sends the intermediate code to the robot and to the local processes, where the engine is newly created. That allows the testing of changes in the behavior without rebooting the robot or restarting *RobotControl*.

D.5 Motion Related Tools

D.5.1 Motion Tester Dialog

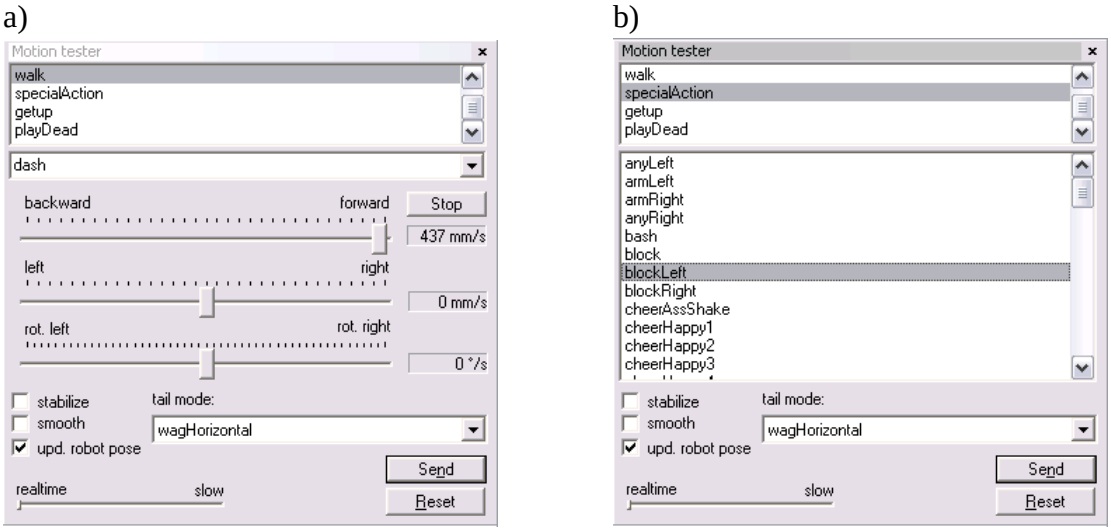


Figure D.17: The Motion Tester Dialog with a) walking and b) special actions

With the *Motion Tester Dialog* (cf. Fig. D.17) it is possible to send *MotionRequests* from *RobotControl* to the robot.

In the upper area of the dialog the different modes stand, getup, walk, or special action can be chosen. In walk mode the velocities in x and y direction and the rotation speed can easily be set by sliders. In special action mode the different special actions (i. e. kicks or funny actions) can be chosen from the select box. In the lower area of the dialog the movement of the tail can be set.

To send the *MotionRequest* to the robot, you have to push the *send* button.

D.5.2 Head Motion Tester Dialog

The *Head Motion Tester Dialog* (cf. Fig. D.18) is handy to test the head control module. It is possible to send *HeadMotionRequests* or to set the *HeadControlMode* from *RobotControl*.

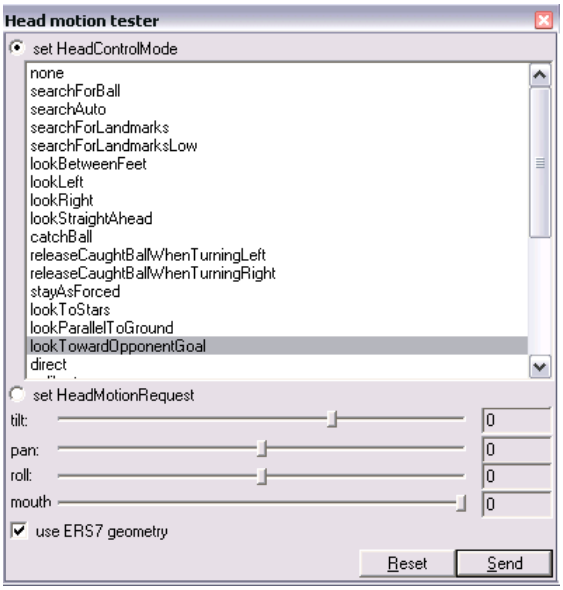


Figure D.18: The Head Motion Tester Dialog

The desired *HeadControlMode* is selectable from a list of all available modes. Some modes require additional parameters (i. e. the coordinates of a point to look at). These can be set in the appearing input elements.

In *HeadMotionRequest* mode, the desired joint values can be set by sliders.

D.5.3 Mof Tester Dialog

The *Mof Tester Dialog* (cf. Fig. D.19) is used to write and test new motions. Motions are specified in a description language (cf. Sect. 5.4.1). Joint data lines from these descriptions may be entered into the input field of the dialog and can be sent to the robot via the wireless network at runtime.

For this dialog to work it is necessary that the module *DebugMotionControl* is running on the robot instead of the default motion control module. The debug module will not execute normal motion requests from behavior control but rather wait for debug messages sent from the *MofTester* dialog. It is activated on the robot by switching module solutions with the *Settings Dialog* (cf. Sect. D.2.2).

The *execute* button parses the input field for lines containing joint data information and sends the sequence to the robot. If the *loop* check box is activated when pressing *execute* the sequence will be executed repeatedly.

The *stop* button stops any sequence currently being executed.

The *read* button provides a very handy tool when creating new motions. It reads the robot's current joint angles from sensor data and puts them into a new line in the input field. This is extremely useful in combination with the stay-as-forced motion mode the *DebugMotionControl* module provides while not executing joint data sequences. In stay-as-forced mode all motors are controlled with feedback from sensor input, and therefore they always maintain their current

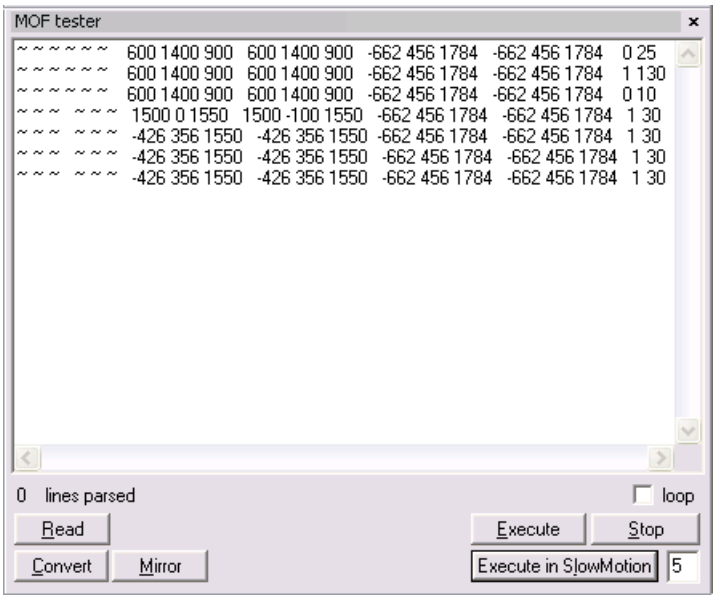


Figure D.19: The Mof Tester Dialog

position. In this mode joints may be moved manually and the resulting joint angles can be read into the *Mof Tester Dialog*.

D.5.4 Joystick Motion Tester Dialog

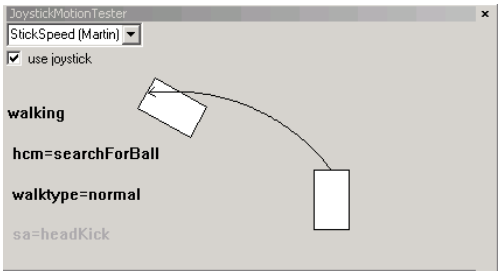


Figure D.20: The Joystick Motion Tester Dialog

The *Joystick Motion Tester Dialog* (cf. Fig. D.20) is used to control a robot that is connected via the wireless network to a joystick. Using such an analog input device is e. g. useful for testing new walking engines or parameters or just walking stability. The primary aim is not a complete remote control, but only moving the robot around (and optionally moving its head) while all other modules keep running autonomously.

This dialog is only tested with an MS Sidewinder Precision 2 USB joystick, but should work with any joystick providing three axes, an accelerator, and eight buttons. After the joystick has been attached, the *use joystick* box has to be checked to start using this dialog. If no joystick seems to be connected, the dialog will refuse to work.

The joystick controls the walking engine of a robot connected with RobotControl via the wireless network. The direction the robot should move, according to the state of the joystick, is visualized in the dialog. Red text in *Joystick Motion Tester Dialog* signals that current changes have not been executed yet, black text shows the actual states or commands, and gray text visualizes states or commands that were sent last but are inactive at the moment.

The buttons 1 to 4 can be used for different kicks, button 7 to start the *getup* motion and button 5 to switch from walking control to head control. As long as button 5 is pressed, the joystick will control the head instead of the walking. That will be visualized by the dialog, too.

To ease the use of the dialog for different people, different schemes for joystick control were implemented. One is called *StickSpeed*, in which the walking speed is completely controlled by the three axes of the joystick, the accelerator is used to switch between head control modes, the walking type can be changed with button 8, and all special actions can be generated by holding button 6 pressed and moving the accelerator.

Another scheme for *Joystick Motion Tester Dialog* is called *AcceleratorSpeed*: the accelerator is used to control the forward walking speed, the axes are only used for sideways speed and direction, and walk types can be changed with button 6.

Commands will only be sent via the wireless network if they differ from the previous command and at most every 300 ms, because the wlan throughput and especially the response times do not allow much more. So whirling around the joystick will definitely not encourage the robot to do the same.

D.6 Sensing and Debugging

D.6.1 Value History Dialog

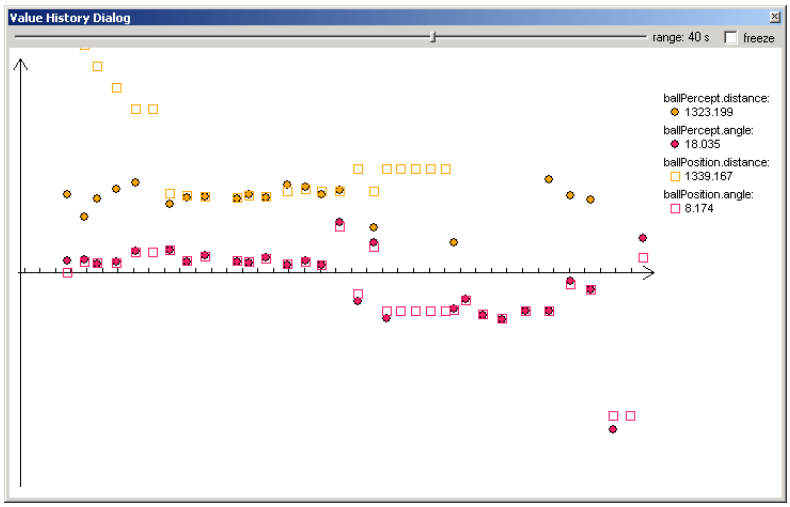


Figure D.21: The Value History Dialog

The *Value History Dialog* (cf. Fig. D.21) allows displaying different values over time. This helps, e. g., to check the stability of a ball modeling algorithm. The values that shall be traced can be selected from the context menu. The time range that is displayed can be changed using the slider at the top of the dialog.

D.6.2 Time Diagram Dialog

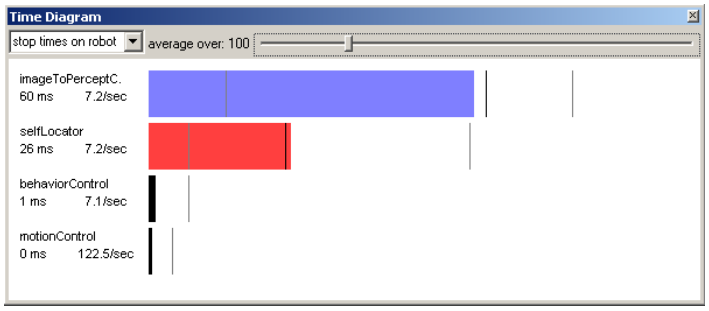


Figure D.22: The Time Diagram Dialog

The *Time Diagram Dialog* (cf. Fig. D.22) visualizes the times which different modules need for their execution in terms of bars. The values next to each bar show the measured time in milliseconds and the frequency (in times per second).

Times can be measured on the robot by selecting *stop times on robot*. If the simulator is used (cf. Sect. D.7), the times can be measured on the computer by selecting *stop times local*. The option *view log files* displays the measured times of recorded log files. Since the times for the execution of the modules can vary very much from one measurement to the next one, the motion of the time-indicators can be smoothed by using average values. The average can be chosen between 2 and 500 measurements. Clicking the right mouse button in the dialog opens a context menu in which the modules of interest can be selected. This menu also offers the option to export the values to a file in a comma-separated format.

The design of the dialog varies, depending on its size and the number of the selected modules.

D.6.3 Debug Message Generator Dialog

In addition to the *Test Data Generator*, the *Debug Message Generator Dialog* (cf. Fig. D.23) can be used to generate less common debug messages for which no special dialog exists. The *Test Data Generator* is usually easier to use and offers more features. The need to use the *Debug Message Generator Dialog* might arise if more than 10 parameters of a module need to be updated in one go. The combo box is used to select what type of debug message is generated. When pressing the *send* button a message will be generated by parsing the input in the text field. The dialog may be extended easily by adding the code to parse the text input into a debug message. Therefore it allows generating new debug messages with *RobotControl* without having to create a new dialog. It is used, e. g., to send a new set of parameters to the walking engine. By

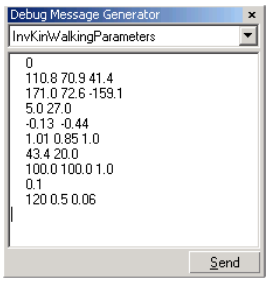


Figure D.23: The Debug Message Generator Dialog

this it is possible to change the walking style at runtime and therefore develop new walks very quickly. Another application is to test playing acoustic messages on the robot by sending the corresponding debug message.

D.7 The Simulator

The simulator is a very powerful extension for RobotControl. It is based on the previous version of *SimRobot* (cf. Sect. 5.1). The simulator offers a lot of possibilities to develop, test, and debug new algorithms or alternative solutions for modules without using a robot.

Figure D.24: Toolbar of the Simulator

As shown in Figure D.25, all relevant objects for robot-soccer are included in the simulation: the field (including landmarks, goals, lines, etc.), players, and the ball. Other objects (e. g. for challenges) can be added with ease. The image created depends on the position of the robot and also on the current angles of the head and the leg joints.

For developing modules which result in movement of the robots, e. g. behavior, the object viewer (cf. Fig. D.25) shows the complete field with all simulated objects. It can be activated by the button *Object Viewer* of the simulator toolbar (cf. Fig. D.24). The vantage point of the observer is variable and can be changed by moving the bars under and beneath the scene displayed. The zooming level, detail level, and the perspective distortion can be adjusted by using the appropriate buttons in the toolbar. Besides these options, the toolbar contains buttons to start and reset the simulation, and to force a step-by-step mode. The touch sensors at the back and the head of the robot can be “virtually pressed” by the buttons marked with an arrow at the according position. One of those buttons pretends the robot to be fallen aside.

A very helpful feature of the simulator is the oracle. It lets the robot know everything of its environment exactly. This can help to develop modules without being dependent on other modules. For example a behavior can be implemented and tested without a self-locator. The button *send oracle* activates this function.

a) b)

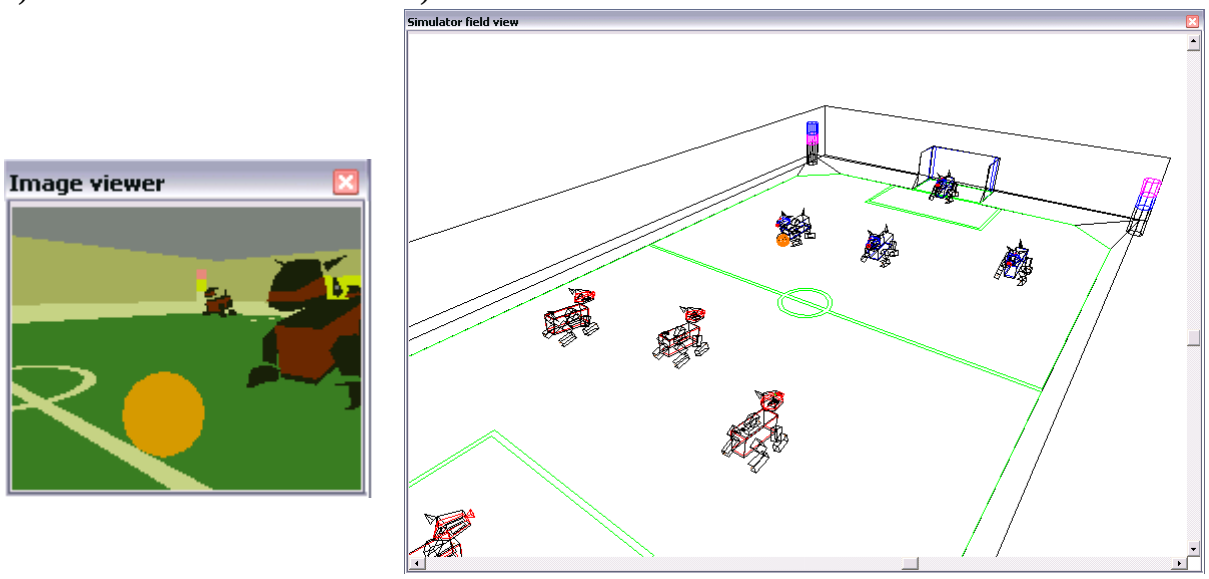


Figure D.25: a) Simulated image and b) Object Viewer of the simulator

Up to four robots of one team can be simulated at the same time. To send commands or receive information from a robot, connect to it by choosing it out of the list at *Robots* in the menu bar. This menu also includes the option to generate images for the connected robot or all simulated robots. Be aware, that generating images for all robots needs a lot of computing power. An entry in the status line of RobotControl shows the currently connected robot.

Appendix E

Extensible Agent Behavior Specification Language

The *Extensible Agent Behavior Specification Language (XABSL)* [38, 39] 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 [37].

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

E.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.

E.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

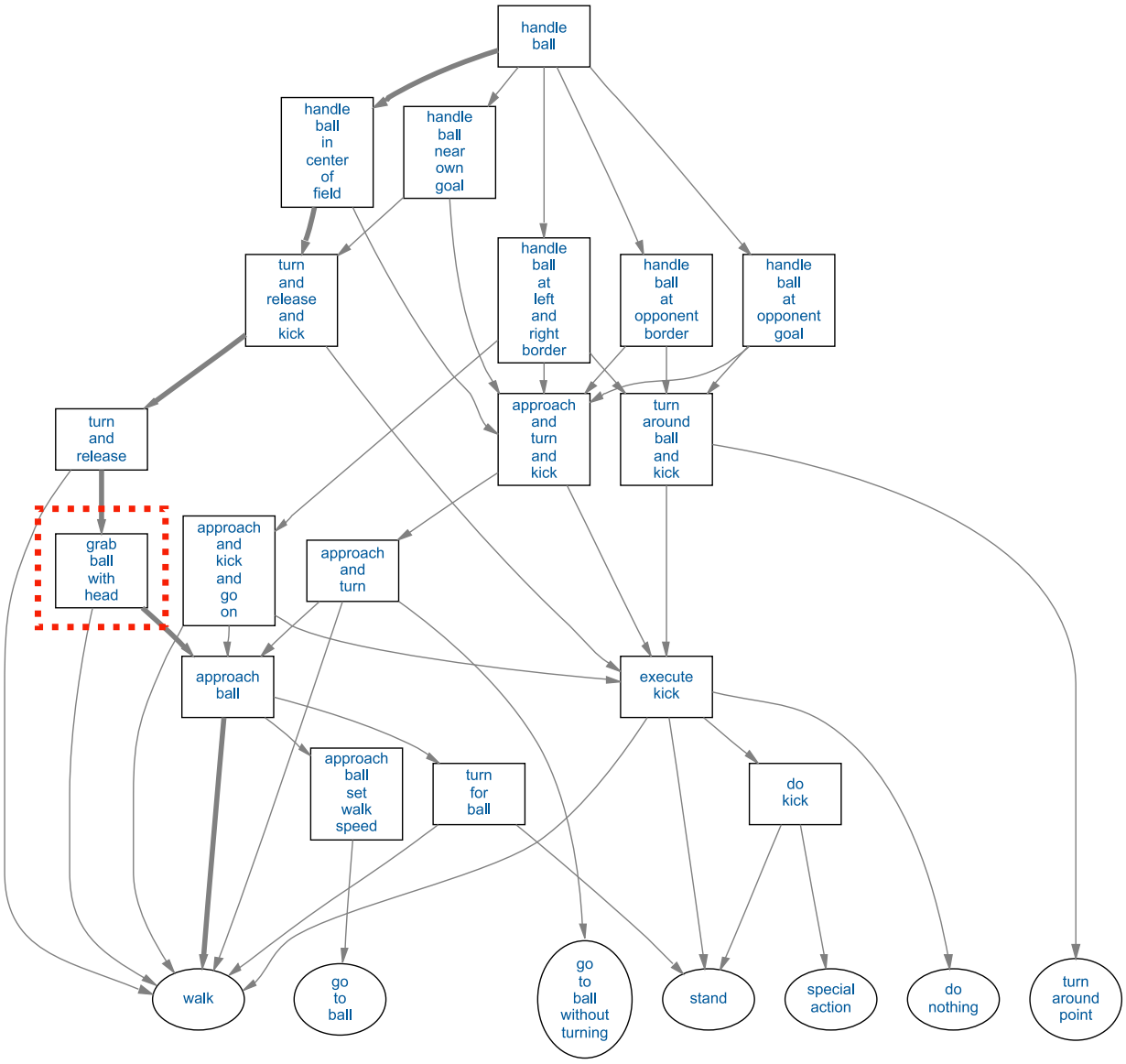


Figure E.1: An example for an option graph from the robot soccer domain (the ball handling part of the *GermanTeam*'s soccer behaviors for the world championships 2004 in Lisbon). Boxes denote options, ellipses denote basic behaviors. The edges show which other option or basic behavior can be activated 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 E.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 E.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 E.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.)

E.1.2 State Machines

Within options, the activation of subordinated behaviors is done by finite state machines. Figure E.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 E.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.

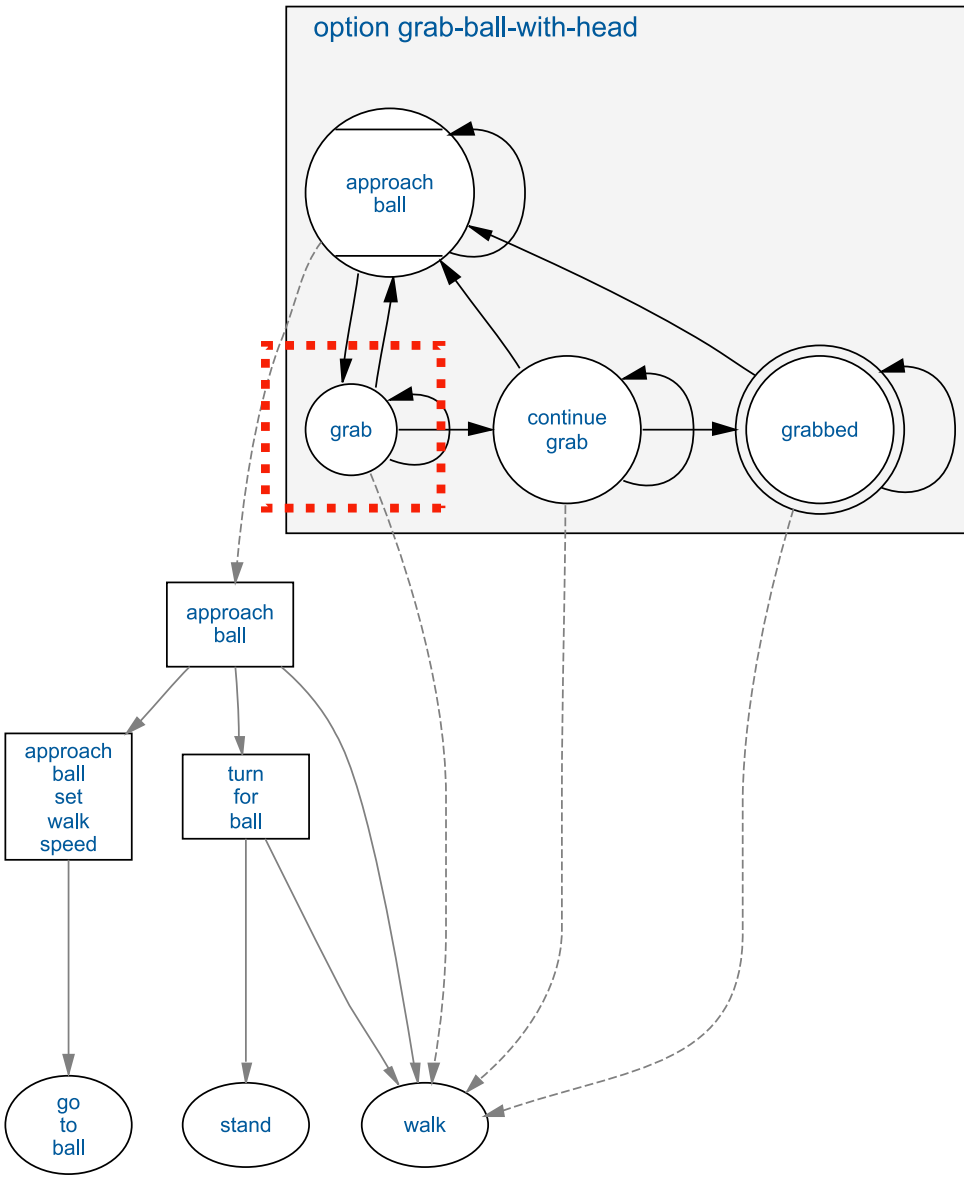


Figure E.2: An example for an option's internal state machine (the option *"grab-ball-with-head"* from the example in figure E.1). Circles denote states, the circle with the two horizontal lines denotes the initial state, the double circle denotes a target state. An edge between two states indicates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

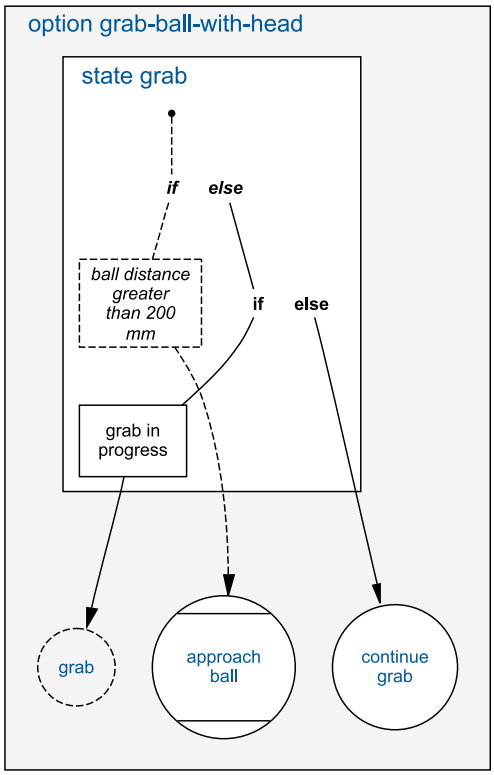


Figure E.3: An example for a decision tree of a state (state *"grab"* of option *"grab-ball-with-head"* in figure E.2). The leaves of the tree are transitions to other states. The dashed circle denotes a transition to the same state. The pseudo code of that decision tree is shown in figure E.4.

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

E.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.

E.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. E.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. E.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).

E.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 E.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 E.5: An example for an *XABSL* XML notation: a source code fragment for the state *"grab"* (cf. fig. E.3) of option *"grab-ball-with-head"* (cf. fig. E.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 E.1, E.2, E.3, and E.4 were generated automatically from the XML source in figure E.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.

E.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. E.6) instead of textual representations such as EBNF or others. A complete language reference can be found at the *XABSL* web site [37].

E.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. E.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. E.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. E.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.

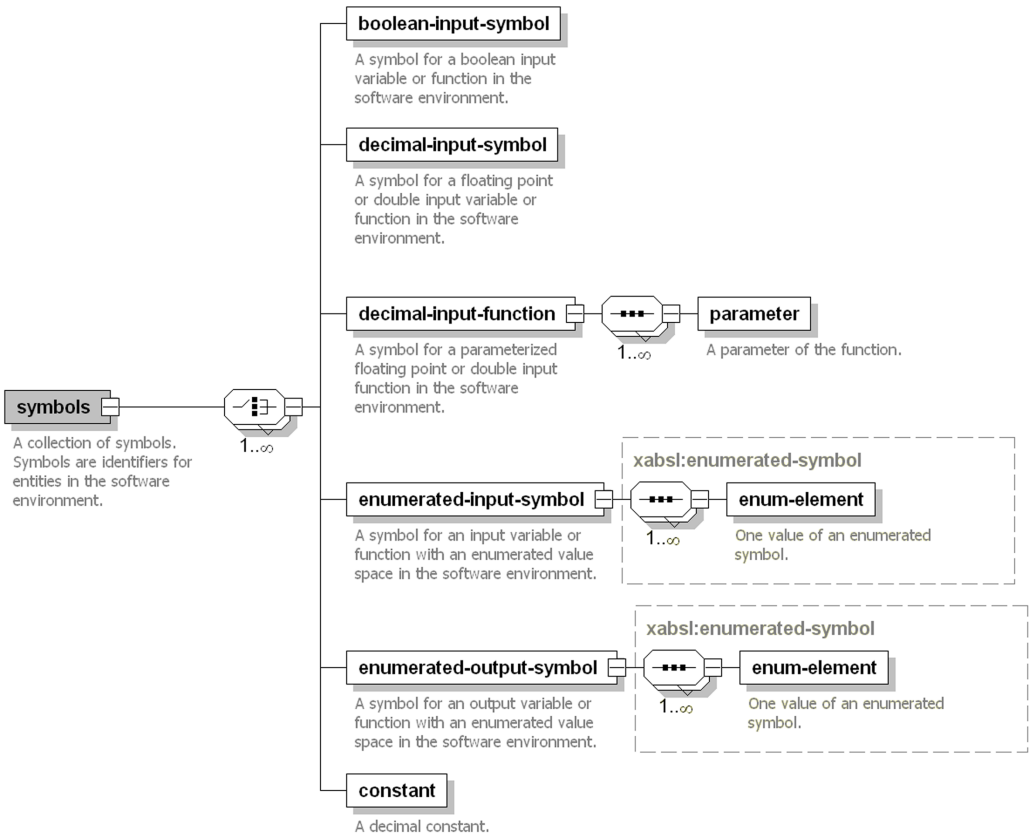


Figure E.6: The syntax of the element *"symbols"*.

E.3.2 Options and States

The root element of an option file is the *"option"* element (cf. fig. E.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. E.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 E.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

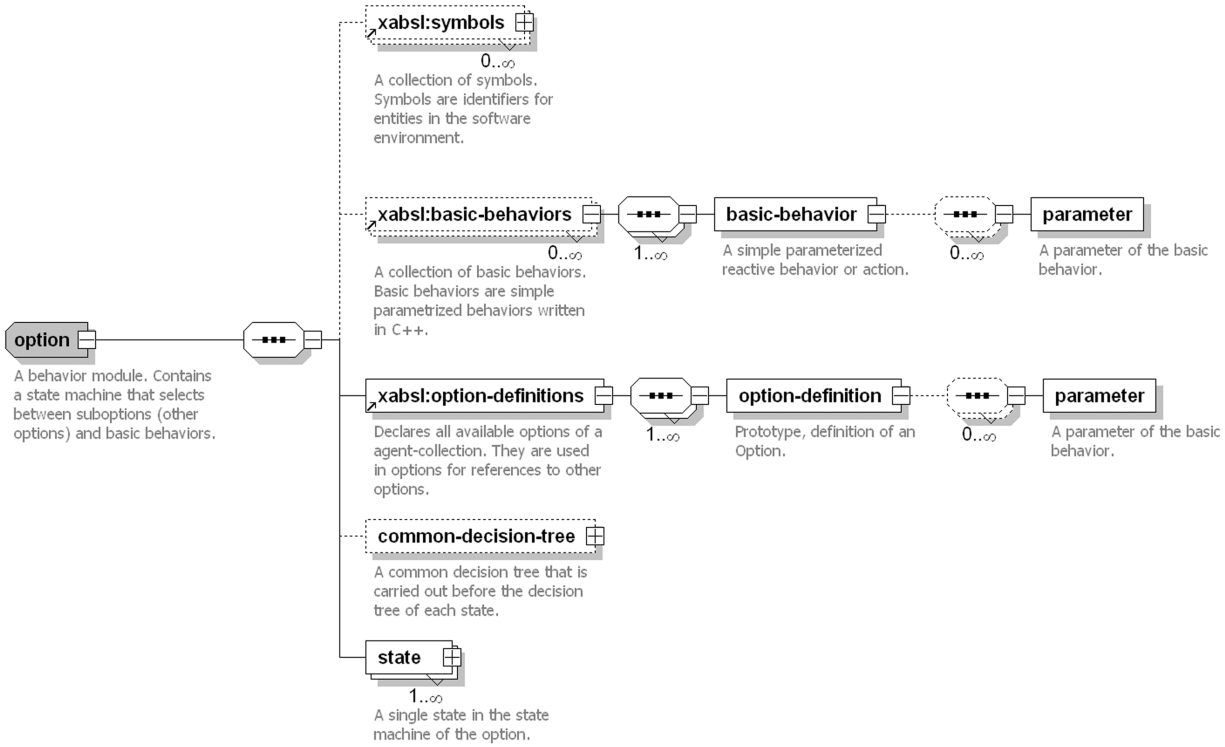


Figure E.7: The syntax of the element *"option"*.

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. E.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.

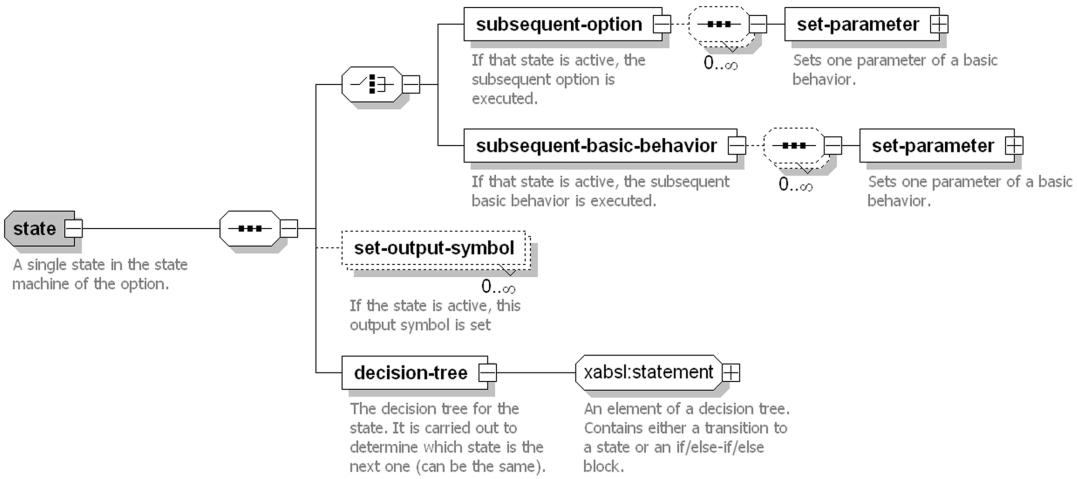


Figure E.8: The syntax of the element *"state"*.

E.3.3 Boolean and Decimal Expressions

A *"boolean-expression"* can be one of the elements shown in figure E.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. E.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.

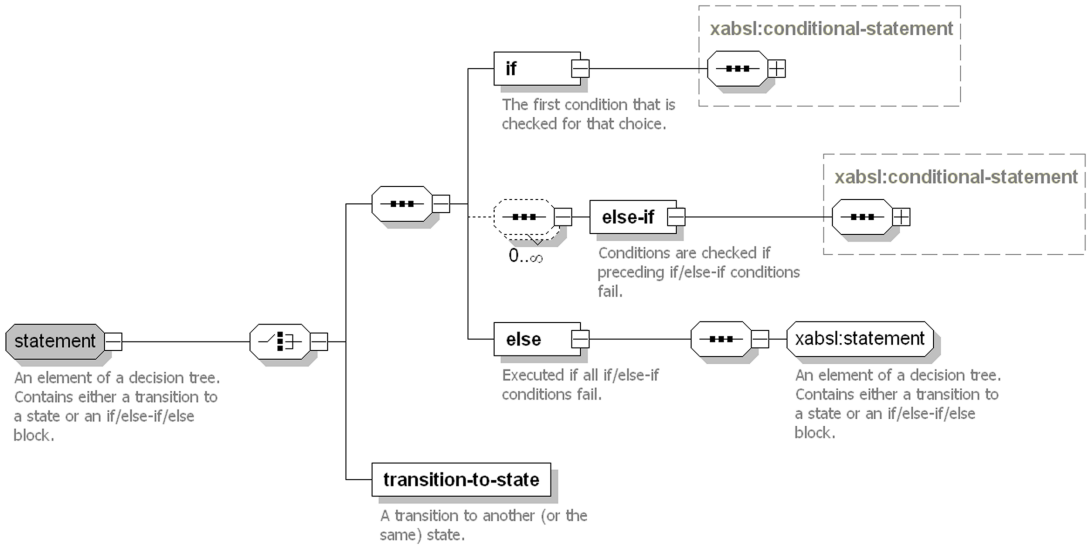


Figure E.9: The syntax of the group *"statement"*. Amongst others, the element *"decision-tree"* is of this type.

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.

E.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 E.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.

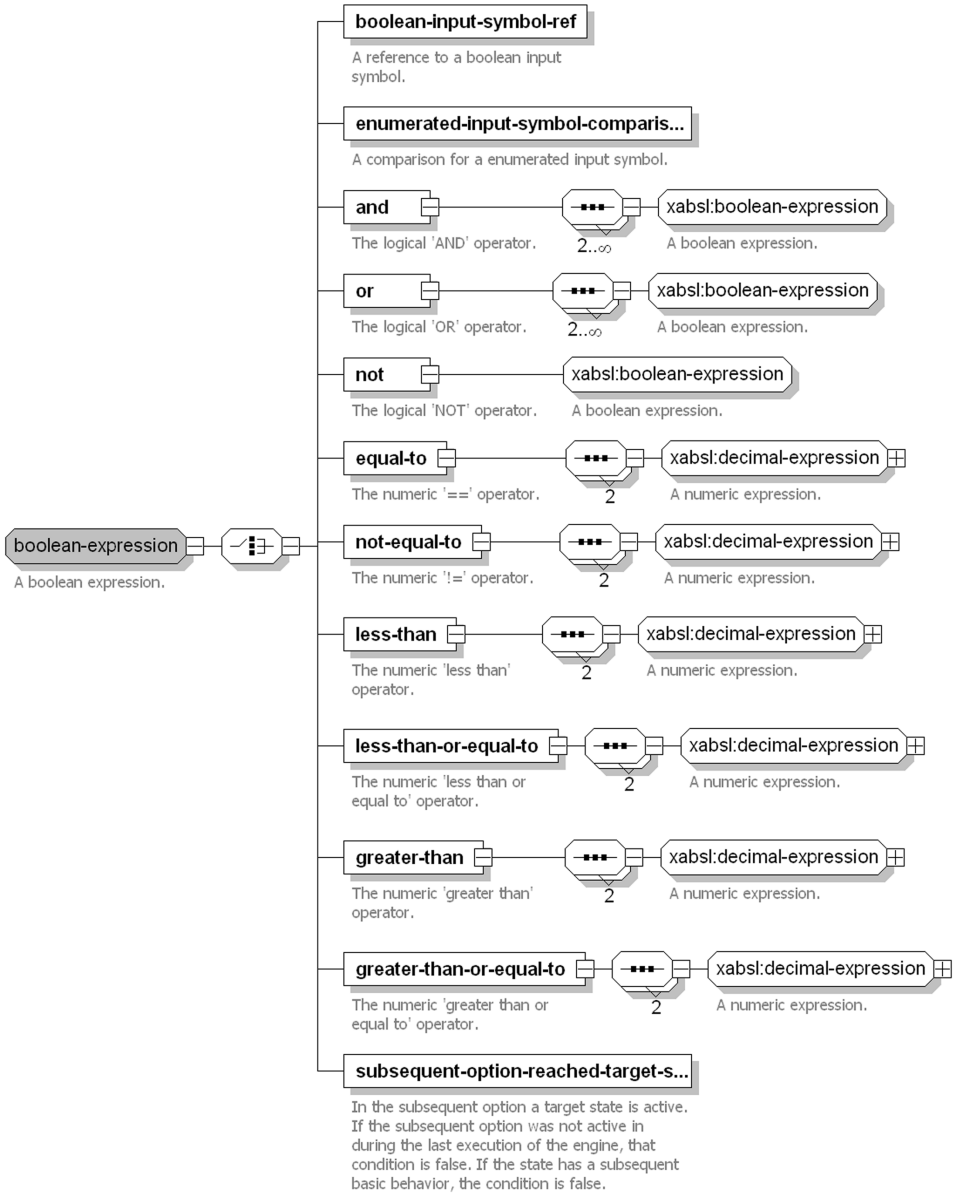


Figure E.10: The syntax of the group "*boolean-expression*". Elements from this group are used inside conditions of decision trees.

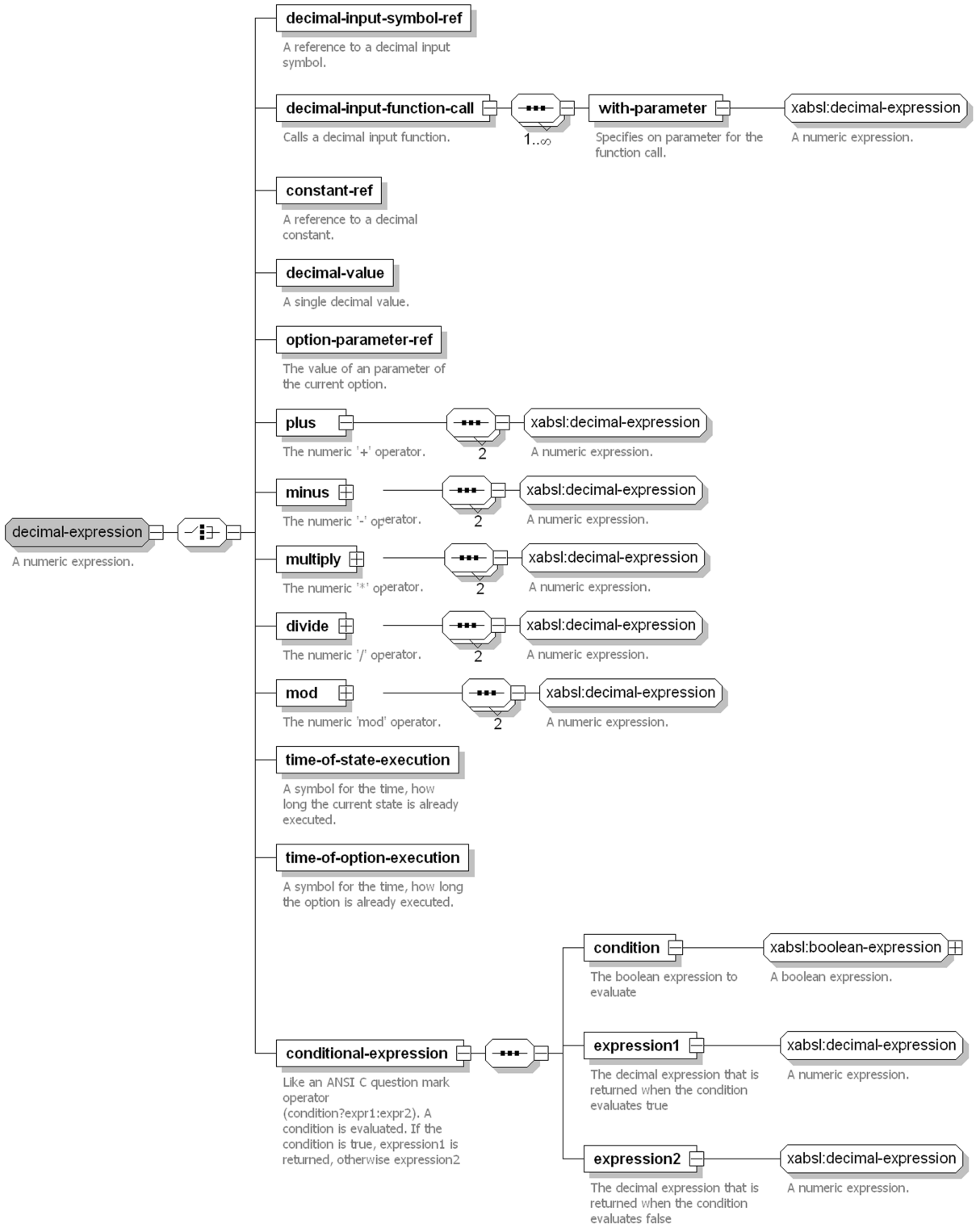


Figure E.11: The syntax of the group *"decimal-expression"*.

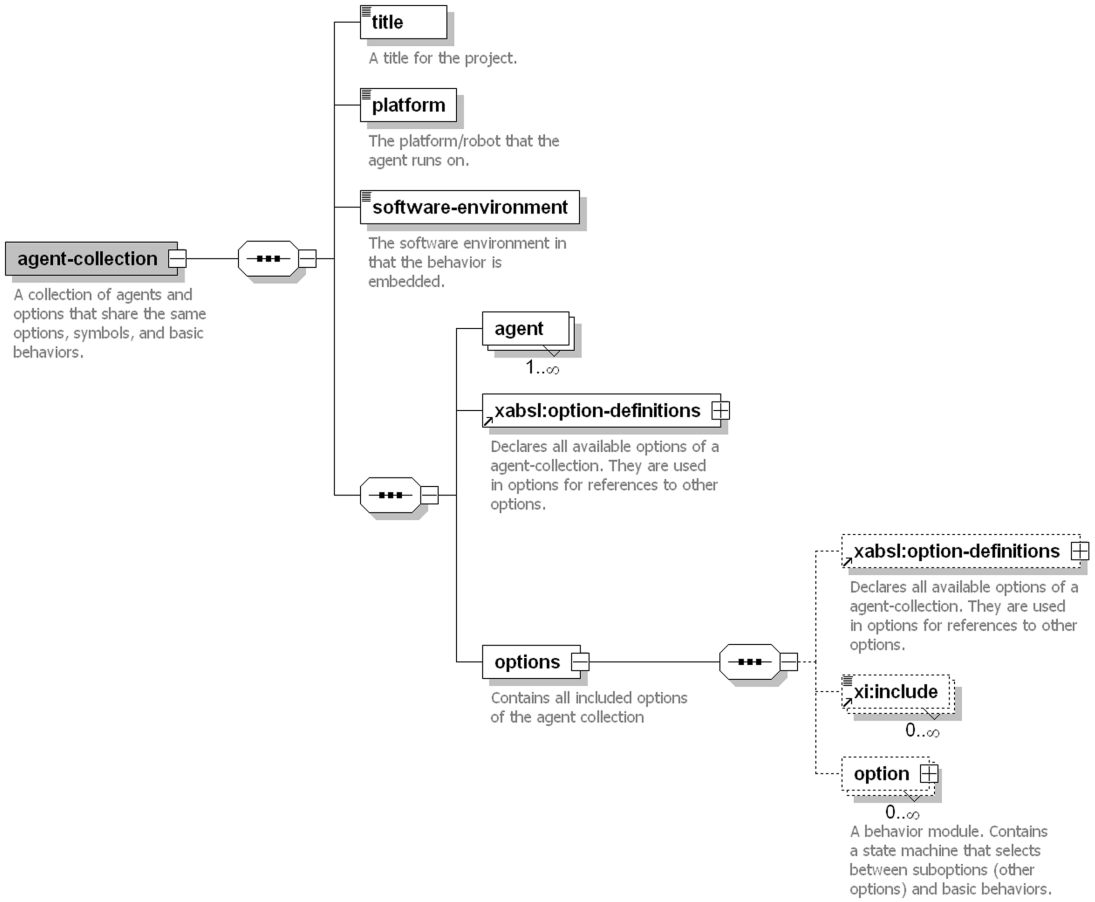


Figure E.12: The syntax of the element *"agent-collection"*.

E.4 Mechanisms and Tools

XABSL is an *XML 1.0* [7] dialect that is specified in *XML Schema* [22]. 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 [37].

E.4.1 File Types and Inclusions

Figure E.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

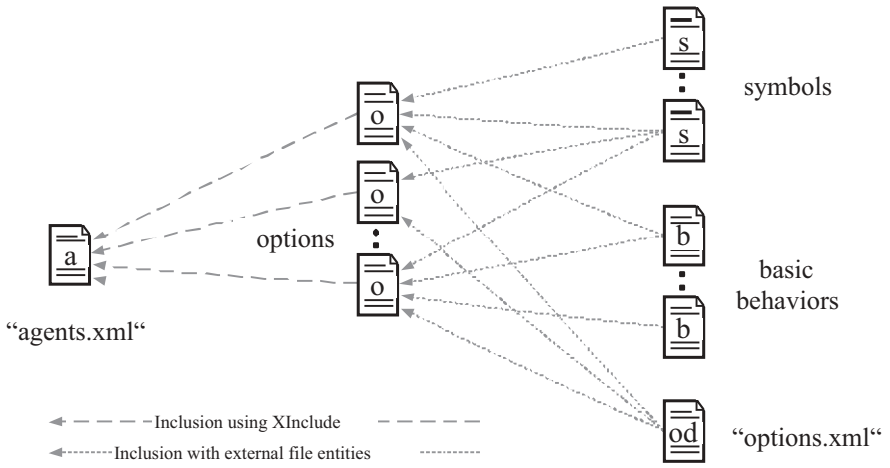


Figure E.13: Different file types of an *XABSL* specification and include mechanisms.

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* [41] 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.

E.4.2 Document Processing

Standard XSLT [14] 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. E.16) was integrated into the *RobotControl* application, the general debug tool of the *GermanTeam*.

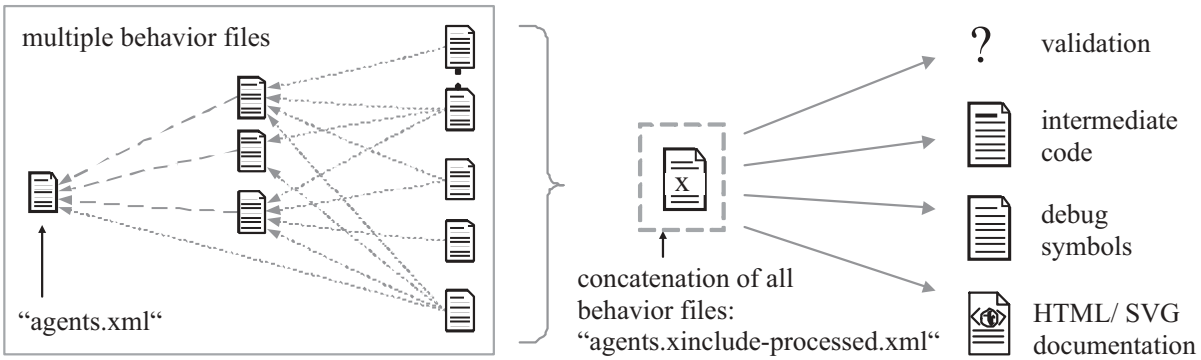


Figure E.14: Document generation in *XABSL*

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 [24, 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) [36] was developed. Note that the figures E.1, E.2, and E.3 were generated automatically from *XABSL* documents with DotML and "dot".

Figure E.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 [37].

E.5 The XabslEngine Class Library

The *Xabsl2Engine* is the *XABSL* runtime system. It is written in plain ANSI C++ [21] 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 [37].

E.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()* .

E.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.

E.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

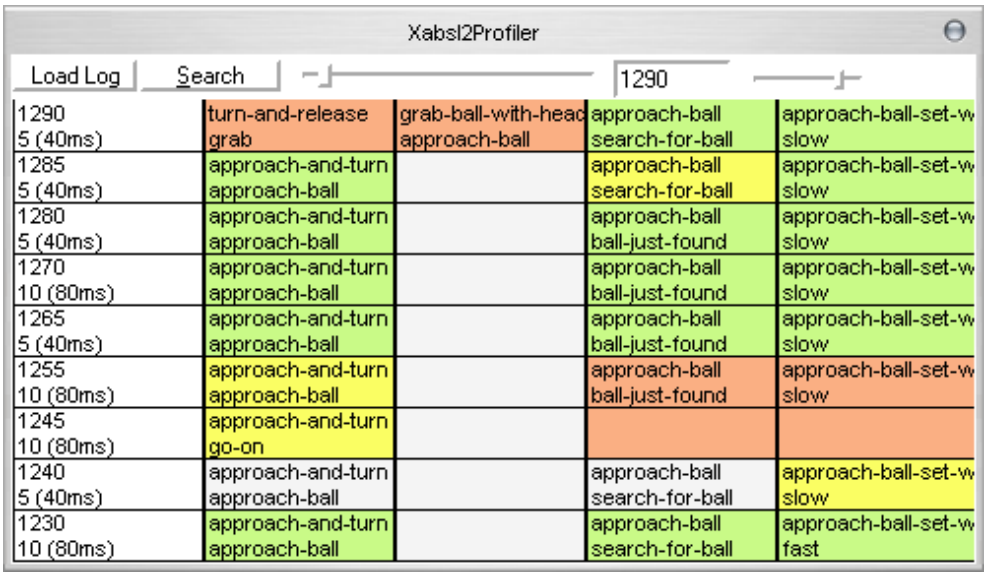


Figure E.15: The *Xabsl2 Profiler* allows to analyze changes in the behaviors over time. Each line reports a change in the state of the *XABSL* system. In the left column, a timestamp and the number of frames with no change of state is displayed. The other columns show the corresponding option and state 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(..)*.

E.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* [49] application, the general debug tool of the *GermanTeam*. First, the *Xabsl Behavior Tester* (cf. fig. E.16) allows tracing the option activation path, the parameters and execution times of options, states,

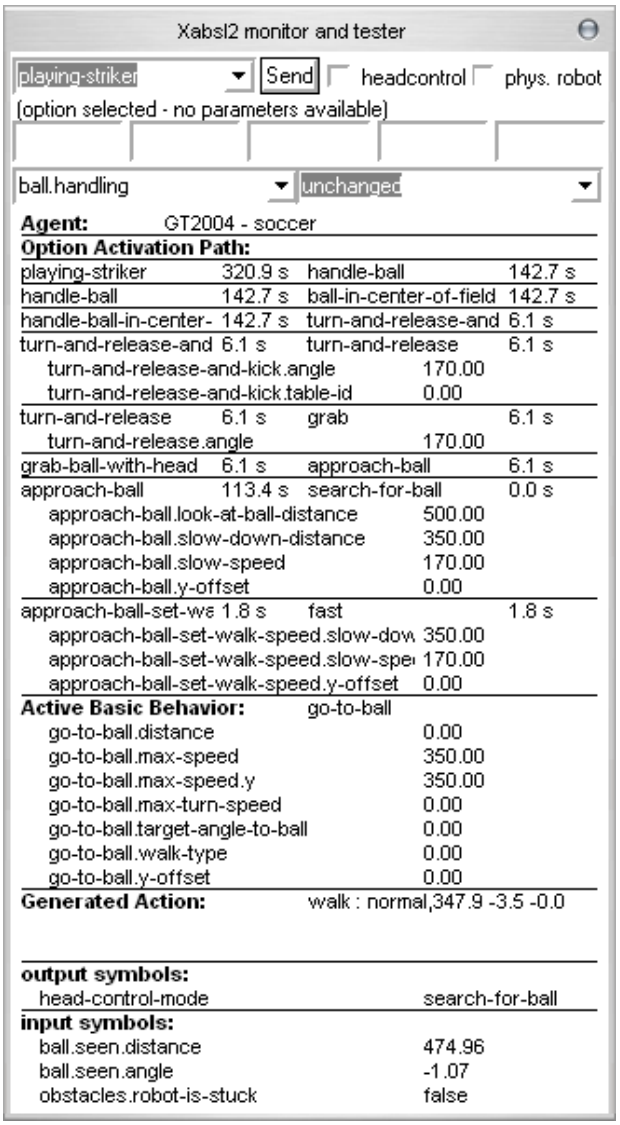


Figure E.16: The *Xabsl2 Behavior Tester*, a part of the *RobotControl* application, makes use of the debugging interfaces of the *XabslEngine*.

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. E.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.

E.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 [52], *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* [31], or *CDL/ MissionLab* [40], 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.

Appendix F

Processes, Senders, and Receivers

F.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.

F.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 *GT2004\Src\Processes*. Process layouts are named after the processes that exist in them, and in fact, in 2003 and 2004 there was only one layout, namely *CMD* that consists of the processes

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

- insert a new line into *GT2004**Src**Processes*\processLayout*object.cfg*, following the format */MS/OPEN-R/APP/OBJS/*shortName.*bin*,
- insert a line in *GT2004**Src**Processes*\processLayout\processLayout.*ocf* starting with *# objectmapping* followed by the *short name* and the *full name*,
- create a new *object* line in the same file using the *short name*,
- create a *.cpp* file in *GT2004**Src**Processes*\processLayout with the full name,
- and, insert that source file in the GT2004 project both under *GT2004**Processes*\processLayout and *RobotControl**SharedCode**Processes*\processLayout in the Microsoft Developer Studio.

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

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

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

MAKE_PROCESS(Example);
```

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

```
      return 500;
```

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

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

```
        return -500;
```

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

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

F.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 G). 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()*.

F.3.1 Packages

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

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

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

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

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

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

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

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

F.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!

F.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 G

Streams

G.1 Motivation

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

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

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

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

G.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

G.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);
```

G.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.

G.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;
  }
}
```

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

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

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

Out& operator<<(Out& stream,const Sample& sample)
```

```
{
  stream.write(sample,sizeof(Sample));
  return stream;
}

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

This approach has its pros and cons. On the one hand, the implementations of the streaming operators need not to be changed if member variables in the streamed class are added or removed. On the other hand, this approach does not work for dynamic members. It will corrupt pointers to virtual method tables if classes or their base classes contain virtual functions. Last but not least, the structure of an object is lost (not the data) when it is streamed to a text file, because in the file, it will look like a memory dump, which is not well readable for humans.

G.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 H

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.

H.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.

H.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 (see the end of section J.2.1) 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.

H.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.

H.1.3 Requesting Messages With Debug Keys

There is a wide range of messages that can be sent from physical or simulated robots to the PC. As it is not possible to send all the messages at once, *debug keys* are used to toggle the output of these messages. They are also transmitted to the robot using message queues. The keys are defined and stored in class `DebugKeyTable`[10]. Each key can have one of following states:

Disabled. No output is sent.

Send always. The output is always sent.

Send n times. The output is sent a total of n times (n cycles of the sending process).

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

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

The *DebugKeys* tool bar of *RobotControl* (cf. fig. H.1) can be used to send debug key tables to the robots.

Each process contains an instance of `DebugKeyTable`[11]. With the static function `getDebugKeyTable()`[12] this instance can be accessed from any piece of code running on the

[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] Definition and implementation in *Src/Tools/Debugging/DebugKeyTable.h* and *.cpp* .

[11] See *Src/Tools/Process.h* .

[12] Definition and implementation in *Src/Tools/Debugging/Debugging.h* and *.cpp* .

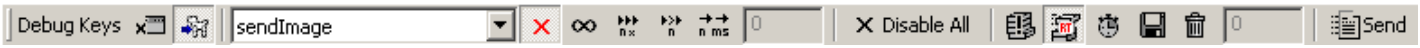


Figure H.1: The *DebugKeys* tool bar in *RobotControl*.

robot. The `DebugKeyTable` -member `isDebugKeyActive(key)` determines whether a message shall be sent dependent on the state of a given key:

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

A new debug key is declared by adding an ID for the key to the enum `debugKeyID` and a name to `getDebugKeyName(debugKeyID aID)` (in class `DebugKeyTable`).

H.1.4 Debug Macros

To simplify the access to outgoing message queues and to the appropriate debug key table, three macros are defined[13]:

- `OUTPUT(type, format, data);` stores `data` in a certain `format` and a certain message type in the outgoing queue of the process.
- `INFO(key, type, format, data);` works similar to `OUTPUT`, but only, when the debug key `key` is active.
- `WATCH(key, type, format, data);` works as `OUTPUT` on the Windows platform and as `INFO` on Open-R.

For example

```
OUTPUT(idText, text, "Could not load file " << filename);
```

outputs an text message independent from any debug keys. The statement

```
INFO(sendFunnyNumbers, idFunnyNumbers, text,
  "i: " << i << ", j: " << j);
```

sends two variables as a text message of the type `idFunnyNumbers` when the debug key `sendFunnyNumbers` is active. Finally,

[13] In */Src/Tools/Debugging/Debugging.h* .

```
WATCH(sendImages,idImage,bin,theImageReceiver);
```

sends images binary formatted with the message type `idImage`. On the physical robots, this message is only sent when the debug key `sendImages` is active. On the simulated robots on the Windows platform this message is sent automatically.

To safe processing time, the macros are ignored in the *Release* configuration.

H.2 Message Queues and Processes

Each process[14] has two message queues: `debugOut` for outgoing and `debugIn` for incoming messges. Debug messages are transmitted between processes such as normal packages. Thereto, with the macro `DEBUGGING`[15] the receiver `theDebugReceiver` for and the sender `theDebugSender` are defined for `debugIn` and `debugOut`.

The static function `getDebugOut()`[16] is used by the debugging macros to access `debugOut` from an arbitrary position in the source code.

H.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);
```

[14] Class `Process`, definition and implementation in *Src/Tools/Process.h* and *.cpp* .
[15] Definition in *Src/Tools/Process.h* .
[16] Definition and implementation in *Src/Tools/Debugging/Debugging.h* and *.cpp* .

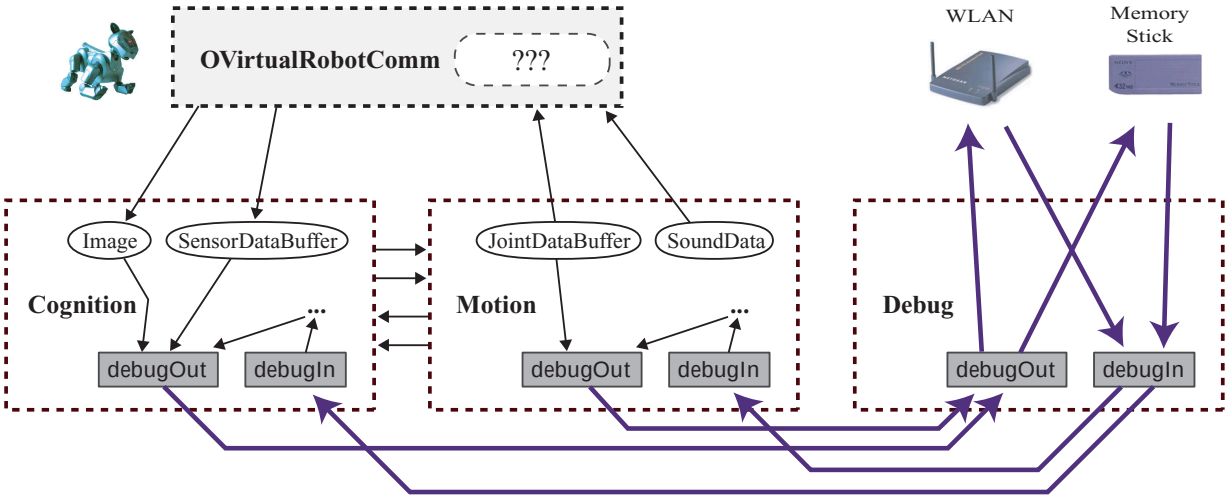


Figure H.2: Data flow between processes in the CMD process layout.

```
    }
  }
```

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. I.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.

H.2.2 The Process Debug

The process `Debug`[17] 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. H.2).

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 *RobotControl* into the file *requests.dat* on the MemoryStick (cf. sect. J.2.1). The `Debug` process reads this file at startup and puts

[17] Definition and implementation in *Src/Processes/CMD/Debug.h* and *.cpp* .

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`[18] to the `Debug` process (for example with the *DebugKeys* tool bar of *RobotControl*, cf. fig. H.1). 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.
- `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.

H.3 Common Debug Mechanisms

Based on message queues, debug keys, and message handlers, the *GermanTeam* developed a rich variety of module-specific debugging mechanisms. Two common high-level debug tools are introduced in this section.

H.3.1 Debug Drawings

Debug drawings can be sent from every piece of code that runs on the robot. There exist two types of drawings: *imageDrawings* are in pixel coordinates and will be displayed on images,

[18] Definition and implementation in *Src/Tools/Debugging/QueueFillRequest.h* und *.cpp* .

whereas *fieldDrawings* are in the system of coordinates of the field and will be shown in field and radar viewers.

Debug drawings are requested with debug keys and are transmitted to the PC as messages of the type `idDebugDrawing`. In *RobotControl*, the *debug drawing manager* automatically sends the debug keys for the drawings that shall be displayed (that were selected in the context menu of the drawing papers).

To use these mechanisms, one first has to declare an ID for a drawing in class `Drawings`[19]. For field drawings, an element is added to the enum `FieldDrawing` and a description string (used by the context menu of *RobotControl*) is added to `Drawings::getDrawingName(fieldDrawing)`. Images drawings are added to enum `ImageDrawing` and labeled in function `Drawings::getDrawingName(fieldDrawing)`.

Drawings are requested with debug keys. Thereto, for each drawing ID a corresponding key must be added to class `DebugKeyTable` (see section H.1.3). For automated processing with macros, the debug key ID should be composed such: `send_{drawing-id}_drawing`. For example for the drawing `sketch`, the corresponding debug key must be `send_sketch_drawing`. Finally, the debug key is assigned to a drawing ID in `Drawing::getDebugKeyID(..)`.

To draw and send drawings from inside any peace of source code, there exist a variety of macros for geometric primitives[20]:

- `LINE(id, x1, y1, x2, y2, penWidth, penStyle, penColor)` draws and sends a line. Parameter `id` is the id of the drawing, `penWidth` the thickness of the line in pixels. These values can be set for `penStyle`: `ps_solid`, `ps_dash`, `ps_dot`, and `ps_null` (invisible). Colors can be `red`, `green`, `blue`, `yellow`, `orange`, `pink`, `skyblue`, `white`, `light_gray`, `gray`, `dark_gray`, `black`, or `yellowOrange`.

- `RECTANGLE(id, x1, y1, x2, y2, penWidth, penStyle, penColor)` draws a rectangle.

- `CIRCLE(id, center_x, center_y, radius, penWidth, penStyle, penColor)` draws a circle.

- `DOT(id, x, y, penColor, fillColor)` draws a dot.

Macros for some more shapes can be found in *Src/Tools/Debugging/DebugDrawing.h*.

A drawing can be composed from many different shapes. To notify that a drawing is finished, after the last shape `DEBUG_DRAWING_FINISHED(id)` is stated. The following examples illustrates the macros:

[19] Definition in *Src/Tools/Debugging/DebugDrawings.h* .

[20] For that, include *Src/Tools/Debugging/DebugDrawings.h* .

```
// paint to the drawing
CIRCLE(sketch, 100, 100, 50, 2,
       Drawings::ps_solid, Drawings::red);
LINE(sketch, 50, 50, 150, 150, 2,
       Drawings::ps_dashed, Drawings::blue);
DOT(sketch, 100, 100, Drawings::black, Drawings::black);

// finish the debug drawing
DEBUG_DRAWING_FINISHED(sketch);
```

A red circle, a blue line, and a black dot are sent with the drawing ID `sketch`, provided that the debug key `send_sketch_drawing` is active. `DEBUG_DRAWING_FINISHED(sketch)` notifies the visualization mechanisms on the PC that all parts of the drawing arrived and that it can be displayed.

Note that if messages are requested in *real-time* mode (standard setting in *RobotControl*, cf. sect. H.2.2), only the last shape arrives on the PC, as all others are deleted from the outgoing message queue of the robot.

To save processing time, these macros are ignored in build configurations *Release* and *Debug no DebugDrawings*.

H.3.2 Stopwatch

To track down waste of time in the code, there are two macros for measuring execution-time of source code fragments[21]:

- `STOP_TIME(expression)` measures the system time before and after the execution of `expression` and outputs the difference as a text.
- `STOP_TIME_ON_REQUEST(eventID, expression)` measures the execution time of `expression`, provided that the measurement for the `enventID` was requested with a debug key. The message is sent with the ID `idStopwatch` to the PC.

For example

```
STOP_TIME(
  for(int i = 0; i < 1000000; i++)
  {
    double x = sqrt(i);
    x *= x;
  }
);
```

[21] In *Src/Tools/Debugging/Stopwatch.h* .

measures and outputs as text the execution time of the loop statement. Instead,

```
STOP_TIME_ON_REQUEST(imageProcessor,
                     pImageProcessor->execute(); );
```

measures the execution time of the whole module `ImageProcessor`.

New event IDs are declared by adding an element to enum `StopWatchEventID` of class `Stopwatch`, a corresponding debug key to `DebugKeyTable`, and a assignment from the id to a debug key (in `Stopwatch::getDebugKeyID(..)`) and to a name (in `Stopwatch::getStopwatchEventIDName(..)`).

Appendix I

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.

I.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. I.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/* .

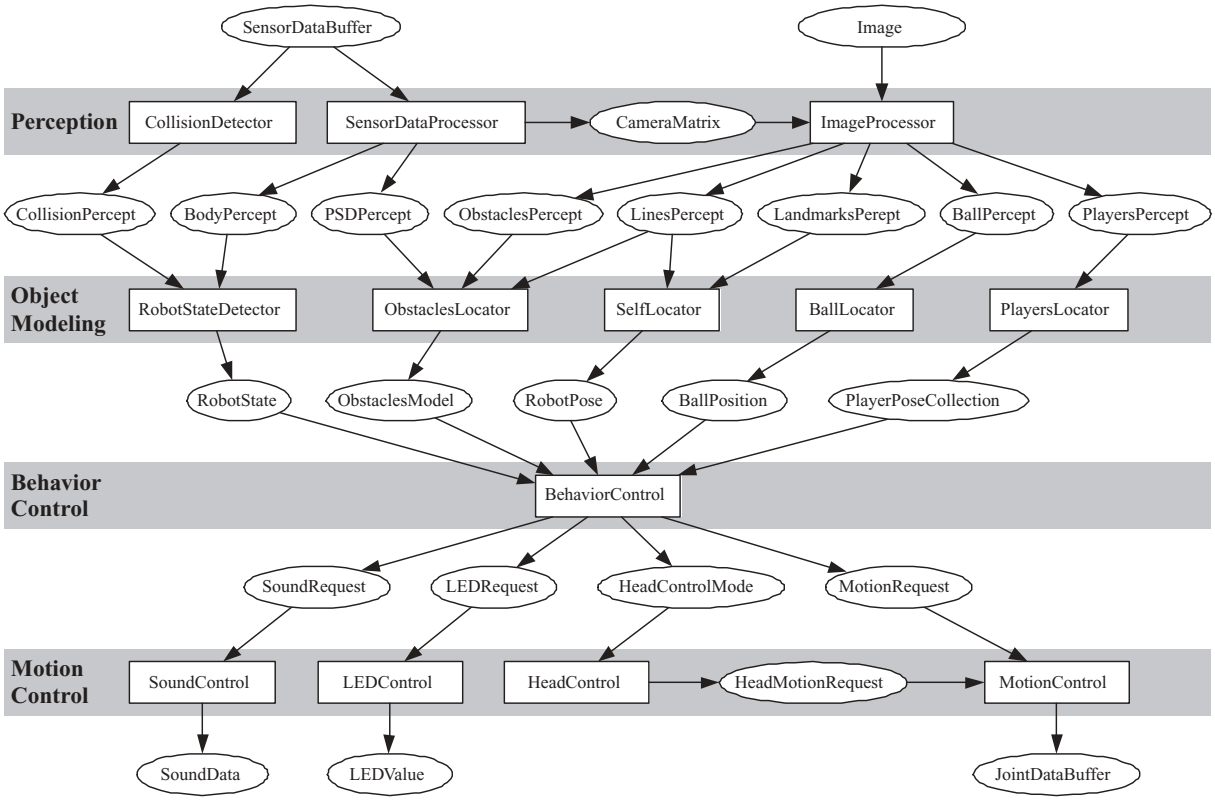


Figure I.1: Simplified graph of the *GermanTeam*'s modules (boxes) and representations (ellipses).

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*) .

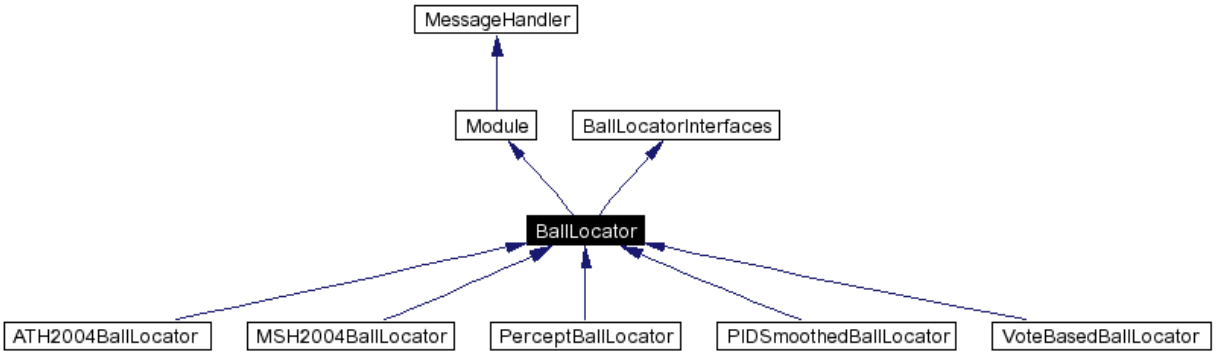


Figure I.2: The class hierarchy of module *BallLocator* (as in January 2004).

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.

I.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. I.2), but this section will show that this structure is needful.

I.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. H.1.2), it is possible to overwrite the `handleMessage(..)` function and to receive messages in a solution.

I.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.

I.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. I.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)
{
  ..
}
```

I.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. I.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* .

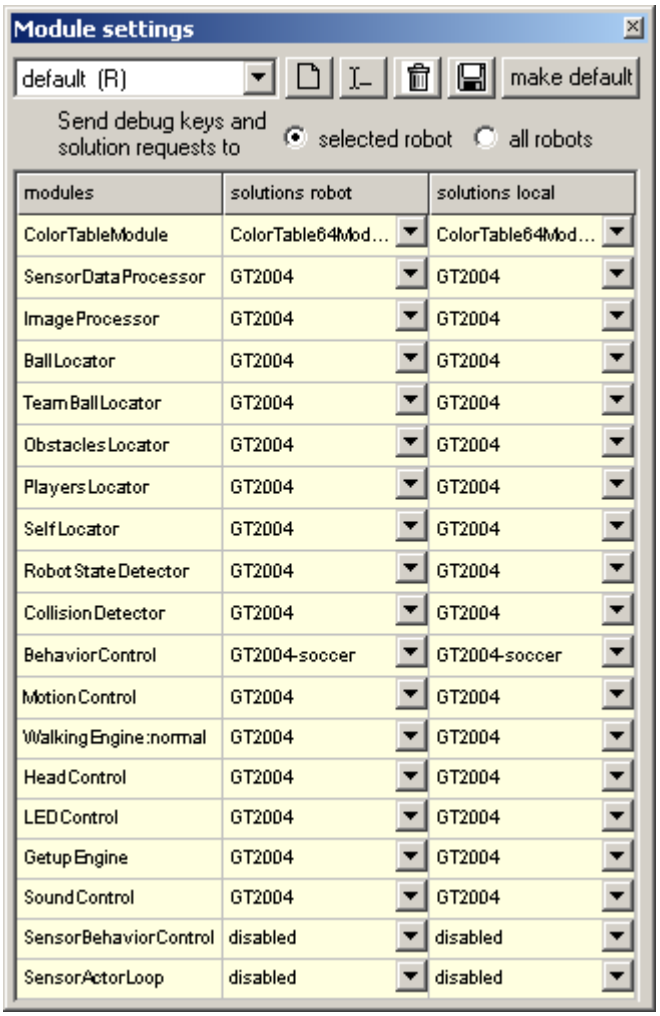


Figure I.3: With the *Settings* dialog bar of *RobotControl* it is possible to select between different module solutions or to switch off a module completely. On the left side are the selected solutions for physical, on the right for simulated robots.

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.

I.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* .

I.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.

I.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* .

I.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 J

Programming RobotControl

RobotControl runs exclusively under Windows and is compiled with the Microsoft Visual C++ compilers[1]. It uses the Microsoft Foundation Classes (MFC) and some other libraries for the graphical user interface.

J.1 General Structure

RobotControl has a strict modular structure. The main components are encapsulated each in separate classes so that it is easy to replace single components by others.

The different components exchange data almost only via message queues – the main program logic is defined by the message distribution mechanisms. Visualization components do not access shared representations but are notified on the arrival of certain messages. Thus many synchronization problems do not appear as the data arrive in the right order in message queues. This message based approach also ensures economical use of processing resources. In *RobotControl* components are only invoked when new data arrive for it or on user input.

It was emphasized to keep the dependencies between source code files low – small local changes should force only a few files to recompile.

J.2 Message Queues and Message Distribution

A variety of message queues (cf. sect. H.1.1) form the backbone of *RobotControl*. The most important ones[2] are combined in class `CRobotControlQueues`[3]. Through the global defined function `getQueues()`[4] one can access these queues.

[1] Visual C++ 6.0 and Visual C++ .Net

[2] As messages can be exchanged only in a synchronized way between threads, there are some more in the interfaces to the simulated and physical robots.

[3] Definition and implementation in *Src/RobotContol/RobotControlQueues.h* and *.cpp*, Instantiation in `CRobotControlApp` (*Src/RobotContol/RobotControl.h*) .

[4] Definition in *Src/RobotContol/RobotControlQueues.h* .

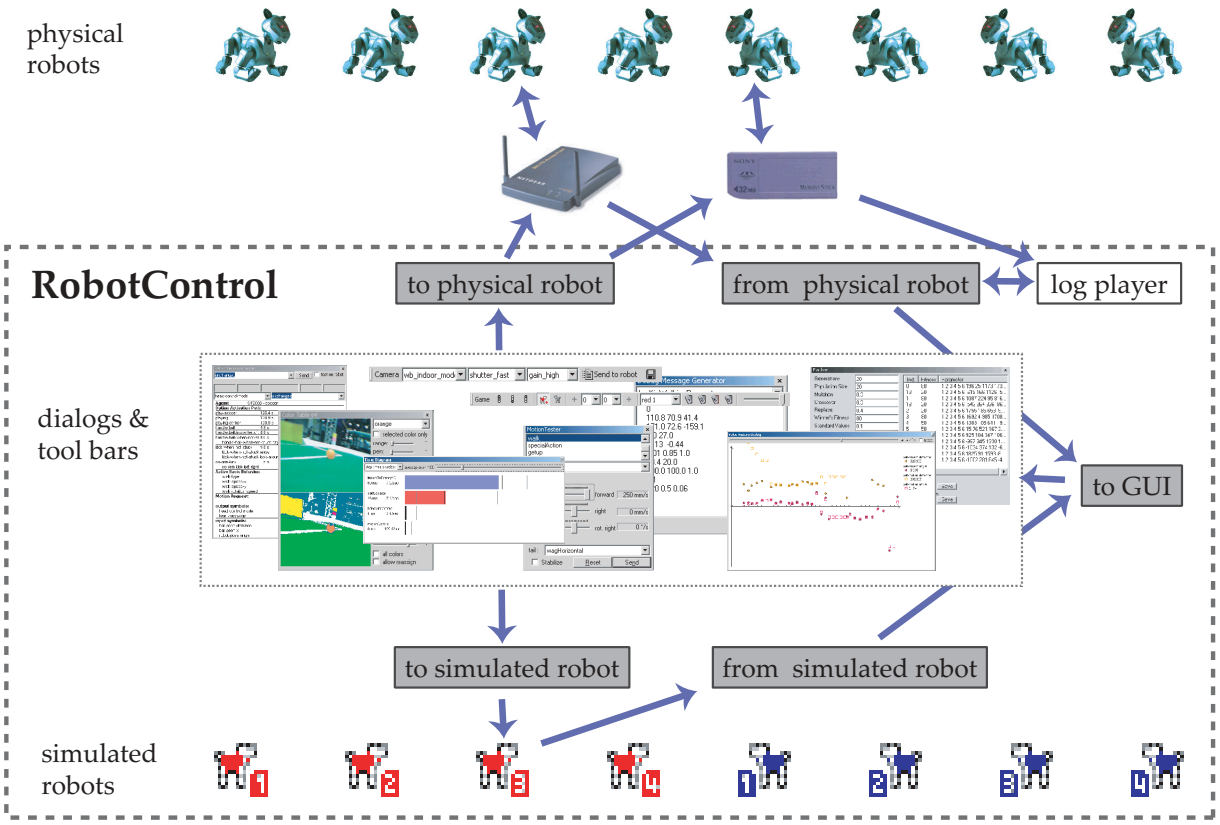


Figure J.1: The main components and message queues of *RobotControl*.

J.2.1 Sending Messages to Robots

For both the simulated and physical robots there are message queues to a specific robot ($0 \ldots 7$), the *selected* robot, and all robots.

Developers of dialog bars and tool bars access messages queues as follows:

`getQueues().toSimulated.selectedRobot. ..`
Messages to the selected simulated robot.

`getQueues().toSimulated.allRobots. ..`
Messages to all simulated robots.

`getQueues().toSimulated.robot[n]. ..`
Messages to a specific simulated robot (n..0-7).

`getQueues().toPhysical.selectedRobot. ..`
Messages to the selected physical robot.

`getQueues().toPhysical.allRobots. ..`
Messages to all physical robots.

`getQueues().toPhysical.robot[n]. ..`
Messages to a specific physical robot (`n`..0-7).

For example with

```
getQueues().toPhysical.selectedRobot.out
           .bin << motionRequest;
getQueues().toPhysical.selectedRobot.out
           .finishMessage(idMotionRequest);
```

the *MotionTester* dialog bar[5] sends a `MotionRequest`[6] to the selected physical robot. With

```
getQueues().toSimulated.allRobots.out
           .bin << forSimulatedRobots;
getQueues().toSimulated.allRobots.out
           .finishMessage(idDebugKeyTable);
```

the central debug key table of *RobotControl* (cf. sect. J.6.1) is sent to all simulated robots. **Attention!** The function `getQueues()` may only be called from the context of the graphical user interface as the message queues are accessed unsynchronized (see also section J.7.2).

The class `CRobotControlSimulatedRobots` (cf. sect. J.4.3) is responsible for transmitting the messages to the simulated robots. If a message was put into the queue to a not active robot, it is discarded.

In `CRobotControlPhysicalRobots` (cf. sect. J.3) the messages to the physical robots are transmitted. In contrast to the simulated robots, messages to not connected physical robots are kept until the robot is connected (or *RobotControl* closes).

This allows to communicate with a physical robot without WLAN. With the *RobotControl* menu entry “Extras” – “Save message queue to physical robots” one can save the message queue to the selected physical robot on a memory stick.

To receive images from the robot despite a not functioning WLAN, one first sends to all simulated robots a debug key table that requests for instance every 10th image (using the DebugKeys tool bar, cf. sect. H.1.3). Additional, with the same tool bar, a `QueueFillRequest` (cf. sect. H.2.2) is sent, specifying that all messages on the physical robot shall be written to the memory stick after 30 seconds. After saving the message queue to the physical robot as

[5] Definition and implementation in *Src/RobotControl/Bars/MotionTesterDlgBar.h* and *.cpp*, instantiation in `CRobotControlMainFrame` (*Src/RobotControl/RobotControlMainFrame.h*).

[6] Definition and implementation in *Src/Representations/Motion/MotionRequest.h* and *.cpp*, instances mainly in the robot processes.

Figure J.2: The *Log Player* tool bar allows to record and play log files.

described above and after booting a robot with it, the `Debug` process on the robot reads these messages and distributes them to the other processes. After 30 seconds, the collected images are written by the `Debug` process into a log file on the memory stick, which can be opened and played in *RobotControl* with the log player (see next section). Note that these mechanisms only work if the source code for the robot is compiled with the configuration "Debug no WLAN".

The communication between the physical robots and *RobotControl* takes place optionally with WLAN or the memory stick, whereas the further processing is identical in both methods.

J.2.2 Log-Player

A special type of message queues is the class `LogPlayer`[7]. Its instance `logPlayer` in `CRobotControlQueues` can record and save the data of the message queue `fromPhysicalRobots` (see next section). These log files as well as log files recorded on physical robots later can be played again, whereas the time intervals between the messages are kept.

All messages played by `logPlayer` are put into the queue `fromPhysicalRobots` – there is no difference for the further processing whether the messages come from a log file or directly via WLAN from a physical robot.

The *Log Player* tool bar[8] is the graphical user interface to the `logPlayer` (cf. fig. J.2).

J.2.3 Distribution of Incoming Messages

In the message queue `toGUI`[9] arrive those messages from simulated and physical robots that shall be displayed or processed by the graphical user interface. But also dialog bars and tool bars use this queue to place text messages in the *MessageViewer* dialog bar[10]:

```
getQueues().toGUI.out.text << "Error in dialog xy";
getQueues().toGUI.out.finishMessage(idText);
```

[7] Derived from class `MessageQueue`, definition and implementation in *Src/Tools/MessageQueue/LogPlayer.h* and *.cpp* .

[8] Definition and implementation in *Src/RobotControl/Bars/LogPlayerToolBar.h* and *.cpp*, instantiation in `CRobotControlMainFrame` (*Src/RobotControl/RobotControlMainFrame.h*).

[9] As `fromSimulatedRobots` and `fromPhysicalRobots` member of class `CRobotControlQueues`.

[10] Definition and implementation in *Src/RobotControl/Bars/MessageViewerDlgBar.h* and *.cpp*, instantiation in `CRobotControlMainFrame` (*Src/RobotControl/RobotControlMainFrame.h*).

Messages from all physical robots and messages from log files at first arrive in the queue `fromPhysicalRobots`, messages from all simulated robots arrive in `fromSimulatedRobots`. These both message queues may be only accessed in a synchronized way, as *RobotControl* otherwise crashes (cf. sect. J.7.2).

The message distribution of these three queues is done in three classes derived from `MessageHandler`[11]:
`CMessageHandlerForQueueFromPhysicalRobots`,
`CMessageHandlerForQueueFromSimulatedRobots`, and
`CMessageHandlerForQueueToGUI` . All three are defined together in a common file[12].

The `CMessageHandlerForQueueFromSimulatedRobots` distributes the messages from the queue `fromSimulatedRobots`. At the moment, it is programmed such:

```
bool CMessageHandlerForQueueFromSimulatedRobots
       ::handleMessage(InMessage& message)
{
  message >> getQueues().toGUI;
  return true;
}
```

All messages from the simulated robots are directly put into the queue to the graphical user interface.

The `CMessageHandlerForQueueFromPhysicalRobots` distributes the messages from physical robots and log files (queue `fromPhysicalRobots`). First, for each message from that queue the `logPlayer` is notified. If this is recording at the moment, it appends the message to its internal queue.

As the simulated robots process sensor data both from physical robots as well as the simulator, some messages from `fromPhysicalRobots` are sent to the selected simulated robot. This happens only if a simulated robot is active. Additionally, it is possible to switch off the simulated robots with the button “Disable simulated robots” on the *Simulator* tool bar (cf. fig. J.6) so that all messages from `fromPhysicalRobots` are directly put into `toGUI`.

All messages that are not sent to a simulated robot are also put into `toGUI`:

```
bool CMessageHandlerForQueueFromPhysicalRobots
       ::handleMessage(InMessage& message)
{
  getQueues().logPlayer.handleMessage(message);
```

[11] Definition in *Src/Tools/MessageQueue/InMessage.h* , see section H.1.2 .
[12] *Src/RobotControl/RobotControlMessageHandler.h* and *.cpp* .

```
    if (getSimulatedRobots().getSelectedRobot() == -1
    || getSimulatedRobots().getSimulatedRobotsAreDisabled())
    {
      message >> getQueues().toGUI;
      return true;
    }
    else
    {
      switch(message.getMessageID())
      {
      case idImage:
      case idWorldState:
      case idPercepts:
      case idSpecialPercept:
      case idJPEGImage:
      case idOdometryData:
        message >> getQueues().toSimulated.selectedRobot;
        return true;
      default:
        message >> getQueues().toGUI;
        return true;
      }
    }
  }
```

The `CMessageHandlerForQueueToGUI` finally distributes all messages that shall be displayed in the user interface. For each single data type it is specified, which dialog bars are notified on the message.

With `mainFrame.handleMessageInDialog(id,message)`, with `id` as the ID of the dialog bar[13], a specific dialog bar is notified on a message. To notify a second dialog bar on the same message, the read position of the message must be reset with `message.resetReadPosition()`.

The queue `toGUI` has to handle messages of up to eight simulated and physical robots. On the other hand, most dialog bars are able to display data from a single robot only. Thus, the message queue can be queried with `getQueues().isFromSelectedOrUndefinedRobot(message)`, whether a message originates from the selected physical or simulated robot.

This example shows, how in the `CMessageHandlerForQueueToGUI` messages of the type `idJointData` are distributed to the dialog bars:

```
  bool CMessageHandlerForQueueToGUI
       ::handleMessage(InMessage& message)
```

[13] Defined in *Src/RobotControl/resource.h* .

```
{
  MessageID messageID = message.getMessageID();

  switch(messageID)
  {
  ...

  case idJointData:
    if (getQueues().
      isFromSelectedOrUndefinedRobot(message))
    {
      mainFrame.handleMessageInDialog(
        IDD_DIALOG_BAR_JOINT_VIEWER,message);
      message.resetReadPosition();
      mainFrame.handleMessageInDialog(
        IDD_DIALOG_BAR_RADAR_VIEWER_3D,message);
    }
    return true;

  ...
  }
  return false;
}
```

The message is first sent to the *Joint Viewer* dialog bar[14]. Then the read position is reset and the *Radar Viever 3D* dialog bar[15] also the message. All that only happens when the message comes from the selected robot, as both dialog bars can only process messages from one robot at the same time.

J.2.4 Example

To illustrate the mechanisms of the last section, we explain, how an images is transfered from a physical robot to the graphical user interface of *RobotControl*.

After connecting to a physical robot via WLAN, images are requested using the *Debug Keys* tool bar (cf. fig H.1). The modified `DebugKeyTable` is put into the message queue `toAllPhysicalRobots`. The class `CRobotControlPhysicalRobots` (cf. sect. J.3) sends the message to all connected physical robots. The message arrives in the `Debug` process

[14]Definition and implementation in *Src/RobotControl/Bars/JointViewerDlgBar.h* and *.cpp*, instantiation in `CRobotControlMainFrame` (*Src/RobotControl/RobotControlMainFrame.h*).

[15]Definition and implementation in *Src/RobotControl/Bars/RadarViewer3DDlgBar.h* and *.cpp*, instantiation in `CRobotControlMainFrame` (*Src/RobotControl/RobotControlMainFrame.h*).

Figure J.3: The *WLAN* tool bar is the graphical user interface to class `CRobotControlPhysicalRobots`.

of the robot. As it has the ID `idDebugKeyTable`, it is both sent to the `Cognition` and `Motion` process.

According to the request, in the `main()` method of `Cognition` images are put into `debugOut`, from where it is sent to the `Debug` process. This manages the transmission of the data back to the PC.

In *RobotControl* the images arrive in the message queue `fromPhysicalRobots`. If the simulated robots are not disabled and if at least one of them is active, the image is put into `toSimulated.selectedRobot` (if not, it is directly sent to the GUI). Shortly after that, the class `CRobotControlSimulatedRobots` passes the message to the process `OVirtualRobotComm` (cf. sect. J.4.1) of the selected simulated robot. Through the `Debug` process the image reaches the process `Cognition` of the simulated robot.

In `Cognition` all modules are executed as on the physical robot. The image as well as calculated percepts, the world model, and several drawings are automatically put one after each other into `debugOut` and sent via `Debug` and `OVirtualRobotComm` back to `CRobotControlSimulatedRobots`. From there the messages get through `fromSimulatedRobot` to the message queue `toGUI`. Finally, the image is passed to several dialog bars and visualization components, which display it together with the generated percepts, world states, and drawings.

J.3 Physical Robots

Class `CRobotControlPhysicalRobots`[16] encapsulates the WLAN communication with the physical robots. It has 8 members of the type `CRobotControlDebugConnection`[17], which implements a direct TCP connection to a physical robot.

Amongst the connected robots, the variable `selectedRobot` specifies the number of the "selected" robot. Dialog bars that can only process data from one robot at the same time are notified only for messages from this robot. Furthermore only this robot gets the messages from `toPhysical.selectedRobot`. Counting starts from 0 (red 1) and ends at 7 (blue 4).

The number of the selected robot is queried with `getSelectedRobot()`. If no robot is connected, -1 is returned. With `setSelectedRobot(int robot)` a physical robot is selected. To avoid selecting not connected robots, it can be queried with `isConnected(int robot)` whether the specified robot is connected.

[16] Definition and implementation in *Src/RobotControl/RobotControlPhysicalRobots.h* and *.cpp*, instantiation in `CRobotControlApp` (*Src/RobotControl/RobotControl.h*).

[17] Definition and implementation in *Src/RobotControl/RobotControlDebugConnection.h* and *cpp* .

On `connect(CRobotControlWLANConfiguration&)`, the class builds up connections to all robots given in a `CRobotControlWLANConfiguration`[18]. With `disconnect()` all these connections are disconnected again.

The *WLAN* tool bar (cf. fig. J.3) uses all these functions. With it a user can edit and connect WLAN configurations and define one of the robots as the selected robot. Through the globally defined function `getPhysicalRobots()` the *WLAN* tool bar accesses the instance of `CRobotControlPhysicalRobots` in *RobotControl*.

J.4 Simulated Robots

As already mentioned, the complete programs for the robots are also compiled and linked to *RobotControl*. This is possible because in the GT architecture all platform specific functionality was encapsulated in platform independent wrapper classes which were implemented for both the Open-R and Windows platform. For the robot programs there is no difference whether they run on a physical robot or on the PC. This allows to test and debug a method first in *RobotControl*, before it is compiled for the physical robots and tested on the field.

J.4.1 Replication of The Robot Operating System

To let the robot programs run on inside *RobotControl* as on a physical robot, all used functionality of the Open-R operating system had to be reimplemented for the Windows platform. The biggest difference is in the implementation of the system processes. Whereas for Aperios/ Open-R only the functionality of the Open-R API was encapsulated, the process framework on the Windows/VC platform is a complete reimplementation of the Open-R processes and its inter-process communication[19]. Processes are implemented using threads, that's why *RobotControl* runs only very unstable on systems such as Windows 95 or Windows 98.

On the Open-R platform the system process *OVirtualRobotComm*, which is only provided as a binary, manages the control of the sensors and actuators. (cf. fig. H.2). It feeds the *Cognition* process with images and other sensor data and receives motor commands from the *Motion* process. In order to have the same connections between processes on the physical robots and in *RobotControl*, the functionality of the Open-R process *OVirtualRobotComm* was reimplemented in class `OVirtualRobotComm`[20].

[18] Definition and implementation in *Src/RobotControl/RobotControlPhysicalRobots.h* and *.cpp*, instantiation in `CWLANToolBar` (*Src/RobotControl/Bars/WLANToolBar.h*).

[19] See class `PlatformProcess` and template `ProcessBase` in *Src/Platform/Win32/ProcessFramework.h* and *.cpp* .

[20] Definition and implementation in *Src/Platform/Win32/ForRobotControl/OVirtualRobotComm.h* and *.cpp*, instantiation at start and reset of SimRobot.

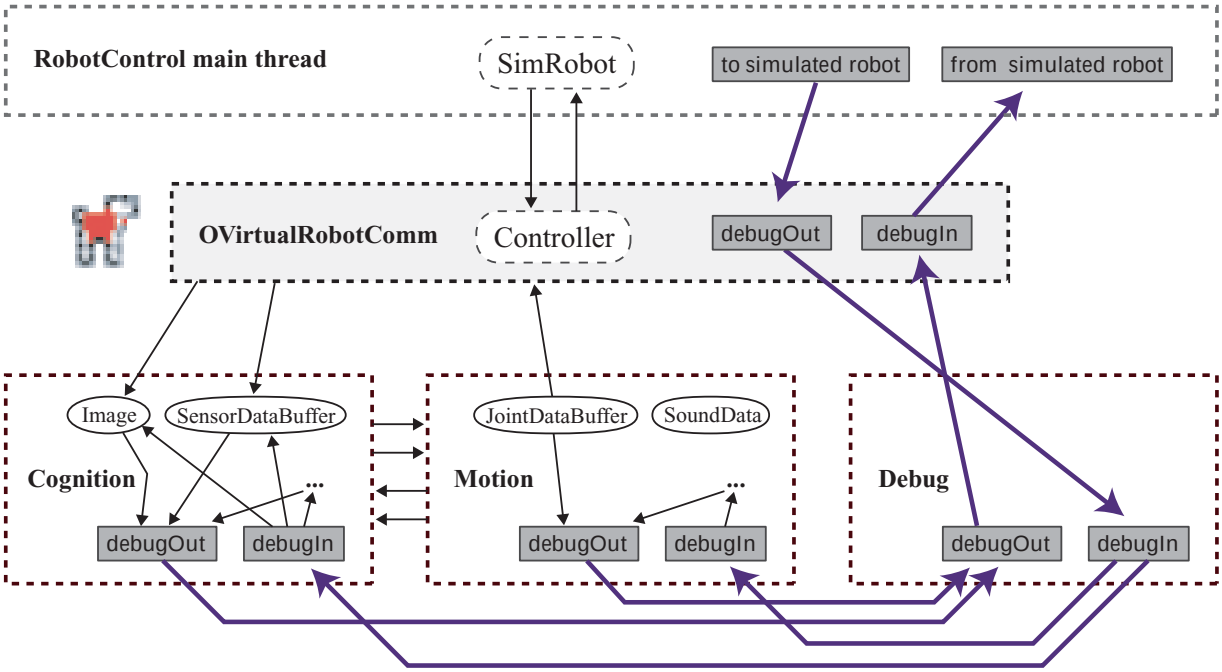


Figure J.4: The main data streams in simulated robots in the CMD process layout (see also fig. H.2.)

Like on the Open-R platform, this process is responsible for feeding the robot programs with sensor readings and receive the generated actions (cf. fig. J.4). Additionally, it transmits debug messages between the `Debug` process and *RobotControl*.

J.4.2 Integration of SimRobot

Additionally, the class `OVirtualRobotComm` is an interface between the simulated robots and the simulator *SimRobot*. This is actually an independent program with an own graphical user interface, but a small part of (the simulation and a visualization of the world, see fig. J.5) was integrated into *RobotControl* [21]. As there is already a newer version of *SimRobot*, which will possibly be integrated into *RobotControl*, this section only focuses on some general mechanisms.

At the start of *RobotControl*, the world description file *Config/Scenes/RobotControl.scn* is parsed. Different from native *SimRobot*, the number of robots to be used is not specified in the scene file but can be set from within *RobotControl*. The class `CSimRobotDocument` holds the world and is responsible for the simulation. The *Simulator Object Viewer* dialog bar[22] (cf. fig. J.5) is used to display the current state of the world.

[21] The files for the simulation part of *SimRobot* are in *Src/SimRob95/SimRobot*. They are compiled into the optimized library *SimRobotForRobotControl.lib* and linked with *RobotControl*. In directory *Src/RobotControl/SimRobot* are the files needed for the integration into *RobotControl*.

[22] Definition and implementation in *Src/RobotControl/Bars/SimulatorObjectViewerDlgBar.h* and *.cpp*, instantiation in `CRobotControlMainFrame` (*Src/RobotControl/RobotControlMainFrame.h*).

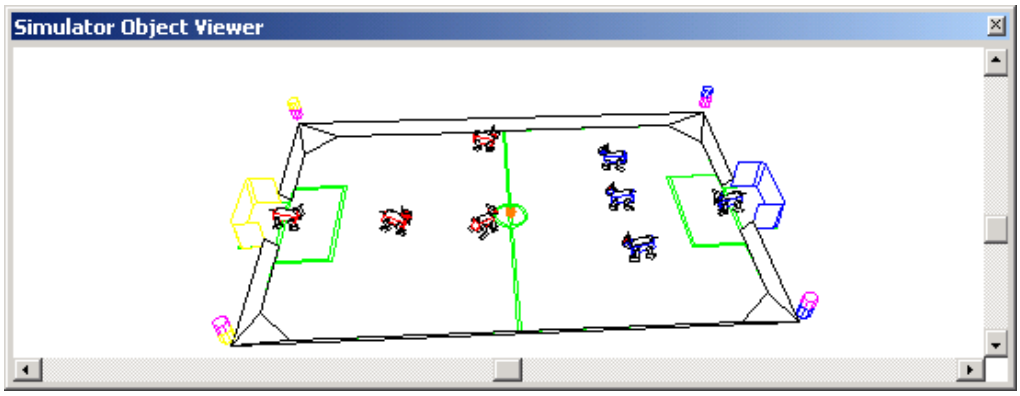


Figure J.5: With the *Simulator Object Viewer* dialog bar the state of the simulation can be viewed and objects can be moved with the mouse.

Figure J.6: The *Simulator* toolbar is the graphical user interface to the class `CRobotControlSimulatedRobots`.

In function `OVirtualRobotComm::update()` the motor commands calculated by the motor control programs are passed to the simulation and the calculation of simulated images and sensor data is requested. In `OVirtualRobotComm::main()` these data together with debug information are sent to the robot processes.

As the calculation of simulated images lasts very long, it is possible to switch the image simulation off. Instead, for example to test only the behaviors, the true world model can be sent to the robot processes, which is calculated by class `Oracle`[23] based on the state of the simulation.

J.4.3 Interface to RobotControl

The class `CRobotControlSimulatedRobots`[24] embeds the simulated robots and the simulation into in *RobotControl*.

`OnSimStart()` starts and stops the simulation. `OnSimStep` performs a single simulation step. `OnSimReset()` resets everything – all simulated robots are deconstructed and newly created.

[23] Definition and implementation in *Src/Platform/Win32/Oracle.h* and *.cpp* . No instantiation, all members are static.

[24] Definition and implementation in *Src/RobotControl/RobotControlSimulatedRobots.h* and *.cpp*, instantiation in `CRobotControlApp` (*Src/RobotControl/RobotControl.h*).

The user can use the (cf. fig. J.6) to define which robots shall be simulated how. This configuration is stored in variable `state[8]` of class `CRobotControlSimulatedRobots`. There are these states:

- `disabled`: The robot is neither simulated nor displayed on the field.
- `passive`: The robot is on the soccer field but is not simulated.
- `activeWithoutImages`: The robot is simulated, but no images and joint angles are calculated. Instead the robot directly receives a world state calculated by the `Oracle` (see previous section).
- `activeWithImages`: The robot is simulated completely.

If a user changes this configuration, the simulation is stopped and must be restarted manually (reset). As the connections between the robots are set up at the start of the processes, it is not possible to add or remove robots dynamically.

The variable `selectedRobot` defines, which of the active simulated robots is the "selected" robot. This means that messages from the queue `toSimulated.selectedRobot` are sent only to this robot and dialogs which can only process data from one robot are only notified on data from this robot. Numbering again is from 0 (red1) to 7 (blue4).

The class `CRobotControlSimulatedRobots` also manages the instantiation of the simulation and the message exchange between the simulated robots and *RobotControl*. As already mentioned, it is possible to switch of the simulated robots all at once with `setSimulatedRobotsAreDisabled(true)` to route data from physical robots directly to the GUI (cf. sect. J.2.3). Thereby the simulated robots are not deconstructed. They just are not triggered until they are switched on again with `setSimulatedRobotsAreDisabled(false)`.

With the globally defined function `getSimulatedRobots()`[25] the graphical user interface (particularly the *Simulator* tool bar) accesses the simulated robots.

J.4.4 Processing Data from Physical Robots

The simulated robots work not only on data calculated by the simulator but can also be fed with data from physical robots or log files. This for example helps to test algorithms for images processing or ball modeling without a physical robot but on real data.

Images, joint angles, percepts, and even world models are sent via message queues to the `Cognition`[26] process of the selected simulated robot. There the instances of the representations are overwritten with the data from the messages.

[25] Definition in *Src/RobotControl/RoboControlSimulatedRobots.h* .

[26] Definition and implementation in *Src/Processes/CMD/Cognition.h* and *.cpp* .

Some efforts were made to avoid the need for manual settings for different debug scenarios. Particularly switching off modules is not required. Normally, for example one would need to switch off the module `ImageProcessor` in order to test the module `BallLocator` (cf. fig. I.1) with ball percepts from a log file. Otherwise, in the `main()` method of the `Cognition` process the `ImageProzessor` would overwrite again the previously received `BallPercept` before of the `BallLocator` is called.

Therefore, there are these Boolean variables in the `Cognition` process: `processSensorData`, `processImage`, and `processPercepts`. All three are set at the end of `Cognition::main()` to `false`.

If new sensor data were received either through `theSensorDataBufferReceiver` or through a message with the ID `idSensorData`, `processSensorData` is set to `true`. And only if `processSensorData` is true, the modules `SensorDataProcessor`, `CollisionDetector`, and `RobotStateDetector` are executed.

Similar in the image processing. If an image is received through the `theImageReceiver` or a `idImage` -message, `processImage` gets true and the modules `ImageProcessor` and `SpecialVision` are executed.

The world modeling modules are only executed when `processPercepts` is true. This happens when either the image processing modules were executed before or when a message with the ID `idPercepts` was received.

If messages with ID `idWorldState` arrive in the process (also from the `Oracle`), none of the three variables gets `true` and only the module `BehaviorControl` is executed.

These mechanisms ensure that only those modules are executed for that new data are existent.

J.5 Graphical User Interface

The graphical user interface *RobotControl* (cf. fig. 5.3) is also strongly modularized: dialog bars and tool bars are programmed in separate files independent from each other. Removing one of them has no side effects on others.

J.5.1 The Main Window

The main window of *RobotControl* is defined in class `CRobotControlMainFrame`[27] and appears after the start of the program. It embeds all dialog bars and tool bars. Furthermore, it has a menu, a status bar, and a single child window.

To be able to embed the tool bars which are not directly based on MFC (cf. sect. J.5.3), `CRobotControlMainFrame` is not derived from the standard MFC class `CMDIFrameWnd`, but from `CMDIFrameWndEx`[28] of the used code library for tool bars. By that it is possible to

[27] Definition and implementation in *Src/RobotControl/RobotControlMainFrame.h* and *.cpp*, instantiation in `CRobotControlApp` (*Src/RobotControl/RobotControl.h*).

[28] Definition and implementation in *Src/RobotControl/MfcTools/IEStyleToolBars/FrameWndEx.h* and *.cpp*.

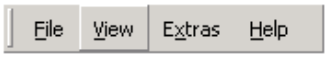


Figure J.7: The menu of *RobotControl* is as all other tool bars movable and size varying.

position the tool bars on a variable number of *bands* on the upper border of the main window. Additionally, the menu `IDR_ROBOTCONTROL` is automatically displayed as dynamically movable tool bar[29] (cf. fig. J.7).

`CRobotControlMainFrame` embeds all dialog bars and tool bars. For the purpose of automatic administration, there are no member variables for them. Instead, instances are stored in maps and arrays. With `dialogBarMap[id]` a dialog bar can be accessed through its ID. The `dialogBarArray[pos]` can be accessed through the position in the array (order of instantiation). Analogous to that there is the `toolBarMap` and the `toolBarArray` in class `CRobotControlMainFrame`.

The macro `CREATE_DIALOG_BAR(..)` creates a dialog bar, embeds it into the main window, and inserts it into the map and the array (cf. sect. K.1.4). Similarly, tool bars are created and embedded with `CREATE_TOOLBAR(..)` (cf. sect. L.1.6).

The Windows API sends messages for each single interaction of the user with windows and controls. For dialog bars, these messages are automatically sent to the corresponding class. But messages for interactions with the menu or with a tool bar arrive in `CRobotControlMainFrame`. Some of them (for instance requests for opening/ closing dialog bar or tool bar) are handled by the class itself. For all other events in `OnCmdMsg(..)` and `OnCommand(..)` first all tool bars and then all dialog bars are queried whether they want to handle the message. Thereto these have to overwrite some virtual functions of their base classes (see section K.2.5 and L.2.3).

Additionally, `CRobotControlMainFrame` is responsible for the update of all controls of *RobotControl*. The Windows-API frequently calls the member function `OnUpdateCmdUI(CCmdUI* pCmdUI)` for all visible controls. It is possible to activate/ deactivate controls, to exchange images on buttons, or to display buttons checked/ unchecked. By default, all controls are enabled in `CRobotControlMainFrame` using `pCmdUI->Enable(true)`. If a programmer wants to deactivate a control dynamically, he overwrites in the class of the dialog bar or tool bar the virtual base class function `updateUI(..)` and calls it for the according control IDs from the `OnUpdateCmdUI(..)` method of the main window (cf. sect. K.2.3 and L.2.2).

The only child window of *RobotControl*, the *Field View* window[30] (see in the center of figure 5.3) fills all the space of the main window that is not occupied by tool bars and dialog bars. It is used by the *Debug Drawing Manager* to display a field and a variety of visualizations on it.

[29]This functionality is provided by class `CMenuBar`. Definition and implementation in *Src/RobotControl/MfcTools/IEStyleToolBars/MenuBar.h* and *.cpp*.

[30]Definition and implementation in *Src/RobotControl/RobotControlFieldView.h* and *.cpp*, instantiation in `CRobotControlApp` (*Src/RobotControl/RobotControl.h*).

J.5.2 Dialog Bars

As the *Microsoft Foundation Classes* do not support resizing and dockable dialogs, the external code library *Resizable Control Bars*[31] by Christie Posea was used. This contains classes for resizing dialogs[32] and mechanisms for embedding these dialogs into dynamic dockable windows[33].

Normally, developers do not get in contact with these classes as all relevant functionality is encapsulated in class `CRobotControlDialogBar`[34]. The classes of all dialog bars derive from it and it has a variety of virtual functions that ensure the automatic integration into the main window (cf. chapter K).

J.5.3 Tool Bars

For the tool bars of *RobotControl*, the code library *IEStyle ToolBars*[35] by Nikolay Denisov was used. It contains class `CToolBarEx`[36], from which derives `CRobotControlToolBar`[37], the base class for all tool bars in *RobotControl*.

`CRobotControlToolBar` extends the functionality of the library by the possibility to add drop-down-lists, edit controls, and sliders on a tool bar. And it also enables the automatic administration of tool bars by providing some virtual functions (cf. chapter L).

J.6 Additional Mechanisms

Despite all the already introduced components, there are some other general mechanisms, which are described in this section.

J.6.1 Central Debug Key Tables

Many dialog bars automatically request via debug keys messages from the robots. As it would be disadvantageous when different dialog bars would send different debug key tables, there are two

[31] The used and adapted version is in directory *Src/RobotControl/MfcTools/DockingControlBars/*, current versions can be found at *http://www.datamekanix.com*. Attention! The version used in *RobotControl* was changed such that it can not be exchanged by a newer version without modifications.

[32] Such dialogs derive from class `CDynamicDialog`, definition and implementation in *Src/RobotControl/MfcTools/DockingControlBars/DynamicDialog.h* and *.cpp*.

[33] The class `CDynamicBar` allows it to dock windows dynamically into the main window, definition and implementation in *Src/RobotControl/MfcTools/DockingControlBars/DynamicBar.h* and *.cpp*. Dialogs are embedded in such dockable windows with the template `CDynamicBarT` (*Src/RobotControl/MfcTools/DockingControlBars/DynamicBar.h*).

[34] Definition and implementation in *Src/RobotControl/RobotControlDialogBar.h* and *.cpp* .

[35] The used adapted version can be found in directory *Src/RobotControl/MfcTools/IEStyleToolBars/*, current versions at *http://www.codeproject.com/docking/sizablerebar.asp* .

[36] Definition and implementation in *Src/RobotControl/MfcTools/IEStyleToolBars/ToolBarEx.h* and *.cpp*.

[37] Definition and implementation in *Src/RobotControl/RobotControlToolBar.h* and *.cpp*.

central instances in in the class `CRobotControlDebugKeyTables`[38], one for all physical robots and one for all simulated.

It can be accessed via the globally defined function `getDebugKeyTables`[39]. With for example

```
getDebugKeyTables().forPhysicalRobots.set(
  DebugKeyTable::sendImage,DebugKey::always)
```

the debug key table for the physical robots can be modified. Analogous, with `getDebugKeyTables().forSimulatedRobots` the debug key table for the simulated robots can be accessed.

With `getDebugKeyTables().sendForPhysicalRobots()` and `getDebugKeyTables().sendForSimulatedRobots()` the changes are sent to the physical and simulated robots.

J.6.2 Configuration Manager

Developers should make sure that settings made by the user are stored in the Windows registry and that they are re-read from there when *RobotControl* starts the next time. Besides that, there is the possibility to store the complete screen layout of *RobotControl* in different *configurations* in the registry and to switch between them at runtime. A configuration contains the visibility and positions of dialog bars and tool bars as well as settings of single dialog bars that are dependent on the window layout.

These configurations are managed by the class `CRobotControlConfigurationManager`[40]. The configurations can be created, saved, deleted, and switched through the menu of *RobotControl*.

For each configuration a separate key is stored in the registry in $...\backslash RobotControl\backslash Configurations$. The respective settings a stored below this key.

In order to save settings of single dialog bars together with configurations, their classes must overwrite the virtual functions `OnConfigurationLoad(CString sectionName)` and `OnConfigurationSave(CString sectionName)`[41].

`OnConfigurationLoad(..)` is called automatically at the start of *RobotControl* and when the configuration changes. The parameter `sectionName` is the name of the registry key from where the settings shall be read. In `OnConfigurationSave(..)` the settings are written to the registry below the passed section name. See also section K.3.2.

[38] Definition and implementation in *Src/RobotControl/RobotControlDebugKeyTables.h* and *.cpp*, instantiation in `CRobotControlApp` (*Src/RobotControl/RobotControl.h*).

[39] Definition in *Src/RobotControl/RobotControlDebugKeyTables.h* .

[40] Definition and implementation in *Src/RobotControl/RobotControlConfigurationManager.h* and *.cpp*, instantiation in `CRobotControlApp` (*Src/RobotControl/RobotControl.h*).

[41] Definition in base class `CRobotControlDialogBar` (*Src/RobotControl/RobotControlDialogBar.h*).

J.7 Main Program

All main components of *RobotControl* are combined in class `CRobotControApp`[42]. This class is responsible for the start of *RobotControl*, the instantiation of the single components, and the exchange of messages between the components. Thanks due to the good modularization this class looks very concise:

```
class CRobotControlApp : public CWinAppEx
{
public:
  DECLARE_SYNC;

  CRobotControlApp();
  ~CRobotControlApp();

  CRobotControlQueues               queues;
  CRobotControlDebugKeyTables       debugKeyTables;
  CRobotControlConfigurationManager configurationManager;
  CRobotControlSimulatedRobots      simulatedRobots;
  CRobotControlPhysicalRobots       physicalRobots;
  CRobotControlMainFrame*           pMainFrame;

  CMDIChildWnd* pChildWnd;

  virtual BOOL InitInstance();
  virtual BOOL OnIdle( LONG lCount );
};

CRobotControlApp& getRobotControlApp();
```

`CRobotControlApp` contains instances of all main components. With `getRobotControlApp()` one can access the instance of this class and by that the instances of the components. But *RobotControl.h* should be included only rarely into other files to keep the dependencies between the sources low.

J.7.1 Start of RobotControl

At the start of *RobotControl*, with `CRobotControlApp theApp;`[43] a static instance of `CRobotControlApp` is created. During that the constructors of the main components are invoked. Note that from these constructors other components must not be accessed, particularly not the message queues.

[42] Definition and implementation in *Src/RobotControl/RobotControl.h* and *.cpp*, instantiation at the start of *RobotControl*, see sect. J.7.1.

[43] At the beginning of *Src/RobotControl/RobotControl.cpp*.

Next, in `CRobotControlApp::InitInstance()` the components are initialized. After a few MFC-specific settings, a splash screen[44] is created and shown (the start of *RobotControl* usually takes a long time). Then with

```
configurationManager.init();
```

the configuration manager is initialized. It reads the list of available configurations from the registry. Thereafter, the main window and the child window for the *Field View* are created:

```
pMainFrame = new CRobotControlMainFrame();
pMainFrame->LoadFrame(..);
..
pChildWnd = pMainFrame->CreateNewChild(..);
```

Both are not displayed yet, as the docking of dialog bars is faster when the main window is not visible. Afterwards the simulated robots are created:

```
simulatedRobots.create();
```

The scene for the simulator is loaded and for each active simulated robot the robot processes are created and connected. This can last a very long time depending on the number of simulated robots. Only in the end with

```
pMainFrame->createDialogBarsAndToolBars();
```

the dialog bars and tool bars are created while *RobotControl* is still invisible. At the end of `createDialogBarsAndToolBars()` the last configuration is loaded, the main window is displayed, and the splash screen is closed.

J.7.2 Synchronisation

The main message queues, the simulator, the graphical user interface, and the WLAN communication run in the thread of the main application. The simulated robots run in independent threads. To avoid crashes, the exchange of data (from message queues) must be synchronized. That means that it must be ensured that critical resources are always accessed by only one thread at the same time.

Objects are declared synchronizable with `DECLARE_SYNC`[45]. The macro `SYNC_WITH(obj)` is used to synchronize with such an object. It is inserted in the code before the critical resource is accessed. The control remains as long in the statement as no other object accesses the requested resource.

For example the simulated robots write messages into the queue `fromSimulatedRobots` only after synchronization with the main thread:

[44] Definition and implementation in *Src/RobotControl/Dialogs/SplashScreenDlg.h* and *.cpp*.

[45] The macros `DECLARE_SYNC`, `SYNC`, and `SYNC_WITH` are defined in *Src/Platform/Win32/Thread.h*.

```
bool OVirtualRobotComm::handleMessage(InMessage& message)
{
  SYNC_WITH(getRobotControlApp());
  message >> getQueues().fromSimulatedRobots;

  return true;
}
```

Into the other direction, `CRobotControlSimulatedRobots::onIdle()` synchronizes with the process `OVirtualRobotComm` .

The function `CRobotControlApp::OnIdle()` is always called when all events for the GUI were processed, all controls were updated, and when there is free processing time. This makes `OnIdle()` to a good function for accessing the shared message queues from the main thread. First, one step of the simulation is executed. Then messages are exchanged with the simulated and physical robots. After that the message handlers are invoked for the main queues.

Appendix K

Adding a Dialog Bar to RobotControl

This chapter describes how to add a dialog bar (cf. sect. J.5.2) to *RobotControl*. Although it is of course possible just to copy and adapt an existing dialog bar, following this instructions might be easier and less error-prone. In single steps it is explained how to create a dialog bar, how to embed it into the main window, and how to program it. The already existing *MofTester* dialog bar[1] serves as an example (cf. fig K.4). The appearance of the dialog bar has changed in meantime – but the general principles and procedures are still the same.

The creation of dialog bars is identical in Visual C++ 6.0 and Visual C++ 2003 .Net. There is no difference in the appearance afterwards and each step is identical in both environments. Only the user interfaces of Visual Studio vary. In this description the figures K.2 and K.1 were created with Visual C++ 2003 .Net. In section K.4 the corresponding screen shots for für Visual C++ 6.0 can be found.

K.1 Creation of a new Dialog Bar

Dialog bars are created similar to normal modal dialogs. The difference is how they are displayed and embedded into the main window.

Before, a good name has to be found. It should be as common as possible but it should also say something about that the dialog bar does.

K.1.1 Creation of a dialog resource

First, a resource for the dialog bar has to be created. Therefore, one opens the “resource view” of Visual Studio, right-clicks the entry “dialog”, and selects “add resource” in the appearing context menu. One chooses “dialog” in the appearing dialog and clicks on “new”.

Then, one right-clicks into the new dialog bar and selects “properties” in the context menu. In the appearing dialog, the *ID* of the dialog bar must be changed from `IDD_DIALOG1` to

[1] Definition and implementation in *Src/RobotControl/Bars/MofTesterDlgBar.h* and *.cpp*, instantiation in `CRobotControlMainFrame` (*Src/RobotControl/RobotControlMainFrame.h*).

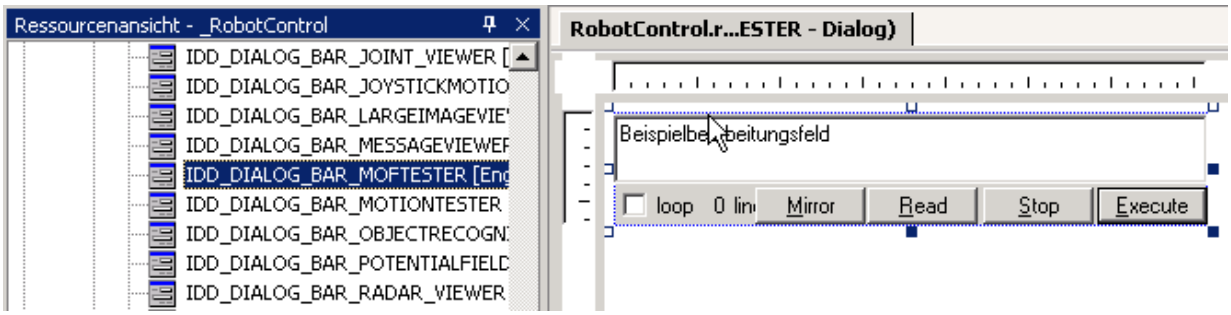


Figure K.1: The *MofTester* dialog bar in the resource view of Visual Studio. New dialog bars should be arranged as compact as this one, as dialog bars are resizable and the user specifies its real size (cf. fig. K.4). This editor looks different in Visual C++ 6.0 (cf. fig. K.5a).

`IDD_DIALOG_BAR_MYDIALOG`, for example to `IDD_DIALOG_BAR_MOFTESTER`. All other settings have to be made such that they completely match those in figure K.2.

After deleting the both existing buttons, new controls can be added. The layout should be as compact and small as possible (cf. fig. K.1), as the size specified in the resource is the minimum size of the resizing dialog bar. The user should choose how big to display it.

For each control an ID has to be specified. As an convention, these should start with `IDC_MYDIALOG_...`, in the example with `IDC_MOFTESTER_...`. A button with the ID `IDC_STOP_BUTTON` can potentially be part of many dialog bars, that's why `IDC_MOFTESTER_STOP_BUTTON` should be used.

K.1.2 Changes in Resource.h

After that, the *Resource.h*[2] must be edited manually. Therefore, one saves and closes all resource windows and opens *Resource.h*. The id for the new dialog bar is changed such that it is between `IDD_DIALOG_BAR_FIRST` and `IDD_DIALOG_BAR_LAST`. This is needed to manage the dialog bars automatically.

In the example, the begin of *Resource.h* should look like this:

```
...
#define IDD_DIALOG_BAR_FIRST            183
...
#define IDD_DIALOG_BAR_MOFTESTER        190
...
#define IDD_DIALOG_BAR_LAST             270
...
```

[2] *Src/RobotControl/resource.h* .

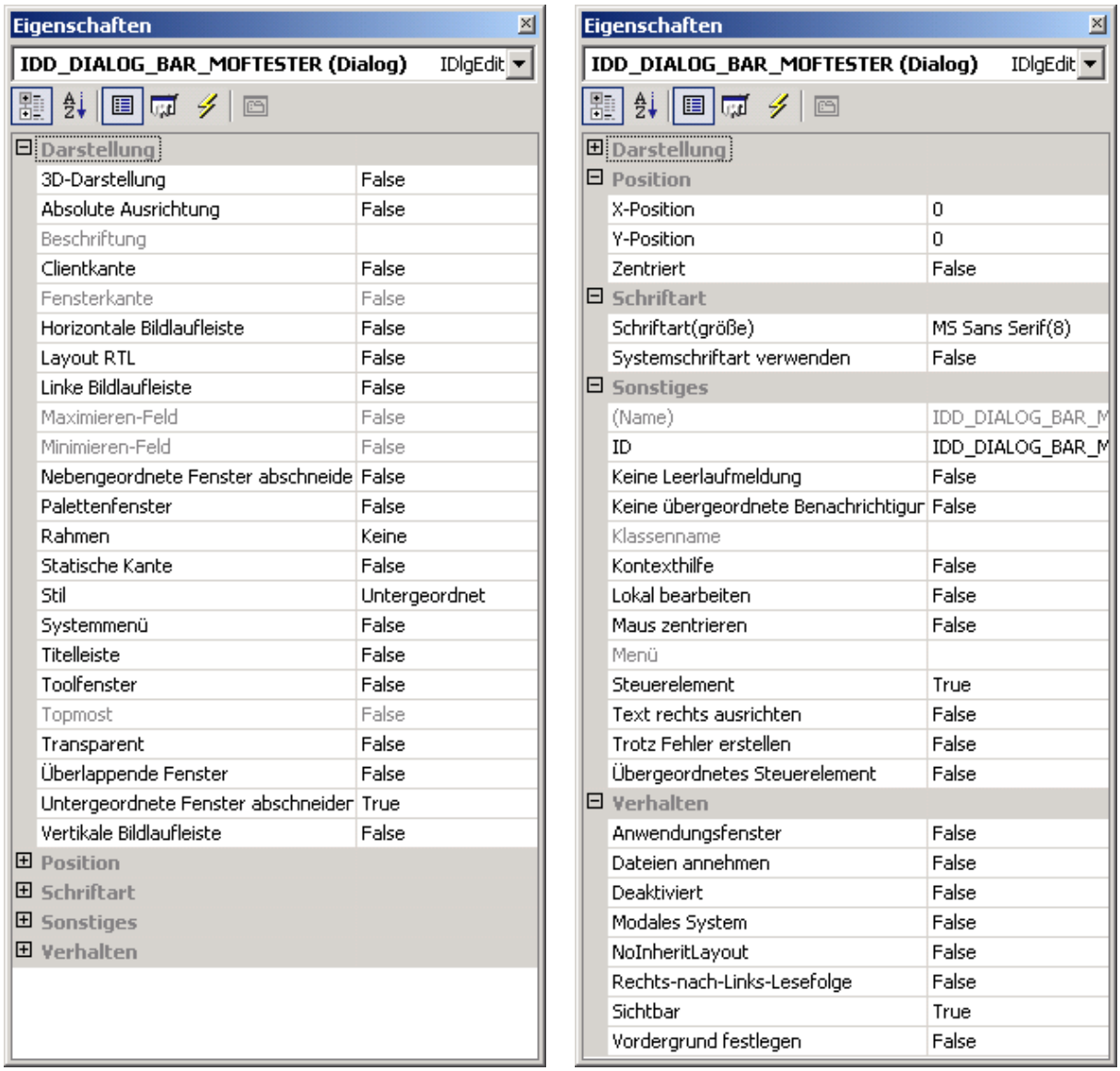


Figure K.2: Exactly this settings have to be done for a new dialog bar. The property dialog looks different in Visual C++ 6.0 (cf. fig. K.5b).

Then, the IDs for the controls are changed. As the IDs of all dialog bars and tool bars are defined together in one single file, they should be grouped. In order to be able to add further controls to existing dialog bars, there should be a gap of at least 10 to the previous group:

```
...
#define IDC_STATUS_BAR_LOGPLAYER_FILENAME 1465
#define IDC_MOFTESTER_LOOP              1475
#define IDC_MOFTESTER_EXECUTE_BUTTON    1476
#define IDC_MOFTESTER_READ_BUTTON       1477
#define IDC_MOFTESTER_STOP_BUTTON       1478
#define IDC_MOFTESTER_MIRROR_BUTTON     1479
#define IDC_MOFTESTER_MOF_EDIT          1480
#define IDC_MOFTESTER_LINES             1481
#define IDC_MOFTESTER_STATIC1           1482
...
```

At the end of *Resource.h*, the value that shall be used for the next added control has to be set (value of the last control + 10):

```
#define _APS_NEXT_CONTROL_VALUE         1495
```

K.1.3 Creating a Class for the Dialog Bar

For each dialog bar, a single class (in the example `CMofTesterDlgBar`) is created in a single *.h* and *.cpp* file[3]. Thereto, one does not use the class assistant but the templates *template-for-new-dialog-bar.h.txt* and *template-for-new-dialog-bar.cpp.txt*[4]. They are copied and renamed to e.g. *MofTesterDlgBar.h* and *.cpp*. The templates contain detailed instructions how to use them (mainly by search and replace).

At last the both files are added to the project.

K.1.4 Embedding a Dialog Bar into the Main Window

Every dialog bar is embedded into the main window[5]. Therefore, in *RobotControlMainFrame.cpp* the header file of the dialog bar has to be included:

```
#include "Bars/MofTesterDlgBar.h"
```

[3] Directory for all dialog bars: *Src/RobotControl/Bars/*.

[4] In the same directory.

[5] Class `CRobotControlMainFrame`, definition and implementation in *Src/RobotControl/RobotControlMainFrame.h* and *.cpp* .

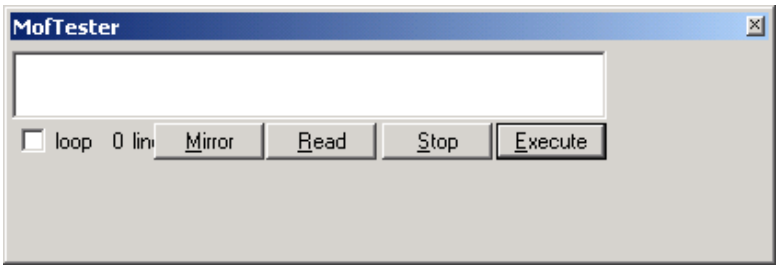


Figure K.3: The *MofTester* dialog bar without resizing mechanisms.

Then, in function `createDialogBarsAndToolBars()` the dialog bar is created and embedded using the macro `CREATE_DIALOG_BAR`[6]:

```
CREATE_DIALOG_BAR(CMofTesterDlgBar,"MofTester",
                  IDD_DIALOG_BAR_MOFTESTER);
```

The first parameter is the class name, the second the window title and the third the ID of the dialog bar.

In order to be able to open the dialog bar, an entry in the "View" menu of *RobotControl* has to be created. Thereto, one opens the menu `IDR_RobotControl` in the resource editor and adds below "View" a new entry for the dialog bar. As ID for the menu entry the ID of the dialog bar is chosen, in the example `IDD_DIALOG_BAR_MOFTESTER`.

Then the "String-Table" is opened in the resource editor. For the ID of the dialog bar a explanatory text is added. This text is shown in the status bar when the mouse moves over the menu entry. Separated by $\backslash n$, a short text for of the menu command is added. For example:

"*Open the Mof tester dialog\nMof Tester*"

At last, the tool bar `IDB_MENU_BUTTONS` is opened in the resource editor. A suitable icon for the menu entry is selected from the existing ones or newly drawn. At the end of function `CRobotControlMainFrame::createDialogBarsAndToolBars()` every menu entry is assigned to an icon. For example

```
mapIDToImage[ IDD_DIALOG_BAR_MOFTESTER ] = 6;
```

assigns the menu entry for the *MofTester* dialog bar to the seventh[7] icon from `IDB_MENU_BUTTONS` .

After all the steps until here it should be possible to display the dialog bar. Provided that the ID of the dialog bar really was defined between `IDD_DIALOG_BAR_FIRST` and

[6] Definition in *Src/RobotControl/RobotControlDialogBar.h* .

[7] Counting starts from 0.

IDD_DIALOG_BAR_LAST, the class CRobotControlMainFrame automatically opens the dialog bar when a user selects the corresponding menu point. The result should look as in fig. K.3.

In some cases one gets a crash when trying to open the newly created dialog bar. This is caused by a resource compilation problem in Visual Studio and normally can be solved by a recompile all.

K.2 Programming a Dialog Bar

With the steps described in the last section, it is possible to create a passive dialog bar. This section deals with the actual programming.

K.2.1 Member Variables for Control

For each control that shall be used by the program logic or that shall automatically adapt to the size of the window, a member variable must be added to the class of the dialog bar.

Visual Studio and MFC experts use the class assistant for this, paying attention not to destroy anything. The safer (and mostly also faster) way is the manual creation of variables. They are declared in the header file of the dialog bar between the lines //AFX_DATA(..) and //AFX_DATA:

```
//{{AFX_DATA(CMofTesterDlgBar)
enum { IDD = IDD_DIALOG_BAR_MOFTESTER };
CStatic  m_lines;
CStatic  m_static1;
CButton  m_stopButton;
CEdit    m_mofEdit;
CButton  m_readButton;
CButton  m_loopCheck;
CButton  m_executeButton;
CButton  m_mirrorButton;
//}}AFX_DATA
```

As an convention, the variables should begin with m_ and end with the ID of the control. In the example, the variable for the control IDC_MOFTESTER_STOP_BUTTON is m_stopButton. The type of the variable depends on the type of the control:

- CButton: for Buttons, checkboxes and radio buttons
- CEdit: for text controls
- CStatic: for not editable text elements
- CSliderCtrl: for sliders

- `CListBox`: for list boxes
- `CComboBox`: for combo boxes
- `CScrollBar`: for scroll bars

These variables are then assigned to the IDs of the corresponding controls in the `.cpp` - file of the dialog bar (function `DoDataExchange(..)`):

```
void CMofTesterDlgBar::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CMofTesterDlgBar)
  DDX_Control(pDX, IDC_MOFTESTER_LINES, m_lines);
  DDX_Control(pDX, IDC_MOFTESTER_STATIC1, m_static1);
  DDX_Control(pDX, IDC_MOFTESTER_STOP_BUTTON,
    m_stopButton);
  DDX_Control(pDX, IDC_MOFTESTER_MOF_EDIT, m_mofEdit);
  DDX_Control(pDX, IDC_MOFTESTER_READ_BUTTON,
    m_readButton);
  DDX_Control(pDX, IDC_MOFTESTER_LOOP, m_loopCheck);
  DDX_Control(pDX, IDC_MOFTESTER_EXECUTE_BUTTON,
    m_executeButton);
  DDX_Control(pDX, IDC_MOFTESTER_MIRROR_BUTTON,
    m_mirrorButton);
  //}}AFX_DATA_MAP
}
```

The first parameter of the macro `DDX_Control` is always `pDX`. The second is the ID and the third the variable of the corresponding control.

After all variables were added, the controls can be accessed from within the class. For example with `m_static1.SetWindowText(..);` one can change the text of the `IDC_MOFTESTER_STATIC1` control.

K.2.2 Dynamic Resizing

For docking the dialog bars into the main window, these must be able to resize dynamically (cf. fig. K.4).

In function `OnInitDialog()`, for each control it is defined how it behaves when the window resizes:

```
BOOL CMofTesterDlgBar::OnInitDialog()
{
  CDynamicBarDlg::OnInitDialog();
```

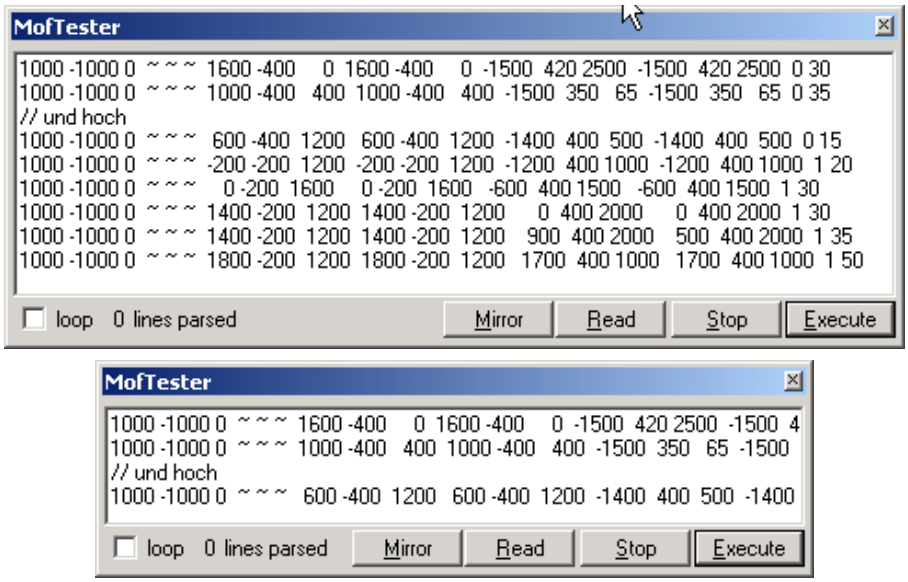


Figure K.4: The *MofTester* dialog bar in *RobotControl* can adapt its size dynamically to the size of the embedding window.

```
  AddSzControl(m_mofEdit,mdResize,mdResize);
  AddSzControl(m_loopCheck,mdNone,mdRepos);
  AddSzControl(m_lines,mdNone,mdRepos);
  AddSzControl(m_static1,mdNone,mdRepos);
  AddSzControl(m_readButton,mdRepos,mdRepos);
  AddSzControl(m_stopButton,mdRepos,mdRepos);
  AddSzControl(m_executeButton,mdRepos,mdRepos);
  AddSzControl(m_mirrorButton,mdRepos,mdRepos);
  return TRUE;
}
```

The first parameter is the variable of the control. Parameter 2 and 3 specify the behavior for horizontal and vertical changes in window size:

- `mdNone`: The control stays at the horizontal/ vertical position that was specified in the resource and keeps its size.

- `mdResize`: The control stays at the horizontal/ vertical position that was specified in the resource and is resized proportionally to the horizontal/ vertical change of window size. (The distance to the left/upper and right/lower border of the window remains as specified in the resource.)

- `mdRepos`: The control keeps its size and is moved horizontal/ vertical relative to the change of window size. (The distance to the left/upper and right/lower border of the window remains as specified in the resource.)

In the example, for the *Stop*-button `mdRepos` and `mdRepos` was set. So the position of the control keeps constant to the right lower corner of the window. For the main text field (`m_mofEdit`), `mdResize` and `mdResize` was specified – it keeps its distance to all borders constant. For the *Loop*-checkbox `mdNone` and `mdRepos` was set – keeps its horizontal position constant and is always at the lower border of the dialog bar.

K.2.3 Activating and Deactivating Controls

It is not possible to activate or deactivate a control from within a dialog bar (the main window is responsible for that , cf. sect. J.5.1). By standard, all controls are activated by the main window. To activate or deactivate a control dependent on some states of a dialog bar, one first overwrites the virtual function `updateUI(..)`[8]:

```
/**
* Enables the controls in the dialog bar.
* This function is called from the main window.
* @param pCmdUI An interface to the control that allows
*        enabling/disabling, checking etc.
*/
virtual void updateUI(CCmdUI* pCmdUI);
```

Then this function is implemented in the `.cpp` file. The ID of the control can be accessed with `pCmdUI->m_nID`. A control is disabled with `pCmdUI->Enable(false)` and enabled with `pCmdUI->Enable(true)`:

```
void CMofTesterDlgBar::updateUI(CCmdUI* pCmdUI)
{
  switch(pCmdUI->m_nID)
  {
  case IDC_MOFTESTER_EXECUTE_BUTTON:
  case IDC_MOFTESTER_LOOP:
    {
      CString input;
      m_mofEdit.GetWindowText(input);
      pCmdUI->Enable(input.GetLength()==0? false: true);
    }
    break;
  default:
```

[8] Defined in the base class `CRobotControlDialogBar` .

```
      pCmdUI->Enable(true);
    }
  }
```

In the example the *Loop*-checkbox and the *Execute*-button are only activated when the main text field contains text.

Finally, `OnUpdateCmdUI(..)` must be called from the `updateUI(..)` function of the main window[9] for the desired controls:

```
void CRobotControlMainFrame::OnUpdateCmdUI(CCmdUI* pCmdUI)
{
   switch(pCmdUI->m_nID)
   {
   ...
   case IDC_MOFTESTER_EXECUTE_BUTTON:
   case IDC_MOFTESTER_LOOP:
     dialogBarMap[IDD_DIALOG_BAR_MOFTESTER]
       ->updateUI(pCmdUI);
     break;
   ...
   }
}
```

K.2.4 Handling Window Messages

For every interaction of the user with a dialog bar the Windows-API sends certain messages that can be handled or not. For example if a button was pressed, Windows sends a `WM_COMMAND` message. There are also messages for releasing a mouse button, double-clicking an control, and so on.

For every control/ event pair that shall be handled, a member function must be added to the class of the dialog. Experienced MFC programmers can do this with the class assistant. But also in this case it is safer and faster to do this manually.

For each control/ event pair in the header file of the dialog bar a member function is declared between the lines `//AFX_MSG(..)` and `//AFX_MSG`:

```
//{{AFX_MSG(CMofTesterDlgBar)
virtual BOOL OnInitDialog();
afx_msg void OnExecuteButton();
afx_msg void OnReadButton();
afx_msg void OnStopButton();
```

[9] In *Src/RobotControl/RobotControlMainFrame.cpp*.

```
afx_msg void OnMirrorButton();
//}}AFX_MSG
```

Each function name should start with `On..` and should be similar to the name of the handled control. “Message map” macros in the `.cpp` file of the dialog bar map the functions to controls and events:

```
BEGIN_MESSAGE_MAP(CMofTesterDlgBar, CDynamicBarDlg)
//{{AFX_MSG_MAP(CMofTesterDlgBar)
ON_BN_CLICKED(IDC_MOFTESTER_EXECUTE_BUTTON,OnExecuteButton)
ON_BN_CLICKED(IDC_MOFTESTER_READ_BUTTON,OnReadButton)
ON_BN_CLICKED(IDC_MOFTESTER_STOP_BUTTON,OnStopButton)
ON_BN_CLICKED(IDC_MOFTESTER_MIRROR_BUTTON,OnMirrorButton)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Finally, the event handling function must be implemented, for example:

```
void CMofTesterDlgBar::OnExecuteButton()
{
  generateJointDataSequence();
  sendSequence();
}
```

This function is always called when the user the clicks on the *Execute*-button.

For the handling of other message types, see the MSDN documentation or existing dialog bars.

K.2.5 Handling External Window Messages

Messages for controls outside the dialog bar are handled differently. This can be messages from the main menu of *RobotControl* or from tool bars.

For this, one first overwrites the virtual function `handleCommand(..)`:

```
/**
* Handles control notifications which
* arrived in the main frame.
* @param command The id of the control,
*        menu, accelerator etc.
* @return If the command was handled.
*/
```

```
virtual bool handleCommand(UINT command);
```

Then this function is implemented in the `.cpp` file. Parameter `command` tells which menu entry or button on a tool bar was pressed:

```
bool CMofTesterDlgBar::handleCommand(UINT command)
{
  if (command == IDC_SEND_MOTION_NET)
  {
    // Behandlung der Nachricht
    ...
    return true;
  }
  else return false;
}
```

If the message given with `command` was handled, `true` must be returned, otherwise `false`.

The main window calls this function automatically. For each message first all tool bars and then all dialog bars are queried whether they want to handle the event. As soon as one of them returns false, this procedure is interrupted. For all dialog bars that do not want to handle external messages, the virtual `handleCommand(..)` function of the base class `CRobotControlDialogBar` automatically returns `false`.

K.3 Integration into the Overall Application

As already discussed in section J.1, all dialog bars only communicate via message queues with the physical and simulated robots as well as other components of *RobotControl*.

K.3.1 Using Message Queues

To receive messages from physical or simulated robots, one overwrites the virtual function `handleMessage(..)`:

```
/**
* Is called for incoming debug messages.
* @param message The message to handle.
* @return If the message was handled.
*/
virtual bool handleMessage(InMessage& message);
```

In the implementation one decides dependent on `message.getMessageID()`, whether the message shall be handled and, if so, reads the message from the queue (cf. sect. H.1.2):

```
bool CMofTesterDlgBar::handleMessage(InMessage& message)
{
  if (message.getMessageID() == idSensorData)
  {
    SensorDataBuffer sensorDataBuffer;
    message.bin >> sensorDataBuffer;
    ...
    return true;
  }
  return false;
}
```

In the `MessageHandlerForQueueToGUI` (cf. sect. J.2.3) one specifies, which message types shall be handled by the dialog bar.

The message queues to the physical and simulated robots can be accessed with the global function `getQueues()`[10] (cf. sect. J.2.1).

With `getDebugKeyTables()`[11], the global debug key tables of *RobotControl* can be accessed, modified, and sent to the robots to request certain messages that are needed by the dialog bar (cf. sect. J.6.1).

K.3.2 Storing Settings in the Registry

Developers are urged on storing settings of a dialog bar in the registry when *RobotControl* closes and to read them from there if it starts again. In `OnInitDialog()` with `AfxGetApp()->GetProfileInt(..)` or `GetProfileString(..)` settings can be read:

```
BOOL CTimeDiagramDlgBar::OnInitDialog()
{
  ..
  averageRange = AfxGetApp()->GetProfileInt("TimeDiagram",
    "average range",10);
  ..
  return TRUE;
}
```

[10] Definition in *Src/RobotControl/RobotControlQueues.h*.

[11] Definition in *Src/RobotControl/RobotControlDebugKeyTables.h*.

In this example the *Time Diagram* dialog bar[12] reads the range, over that values shall be averaged, from the registry. If there was nothing stored (after first start), the default value `10` is read. Normally, the section name in the registry is the name of the dialog, in the example "TimeDiagram".

On every change made by the user, but latest when *RobotControl* closes (in the constructor of the dialog bar), these settings must be stored to the registry:

```
AfxGetApp()->WriteProfileInt("TimeDiagram",
  "average range",averageRange);
```

Additionally, it is possible to associate settings of a dialog bar with "configurations" (cf. sect. J.6.2). This allows the user to store different settings of a dialog bar in different configurations and to switch between them.

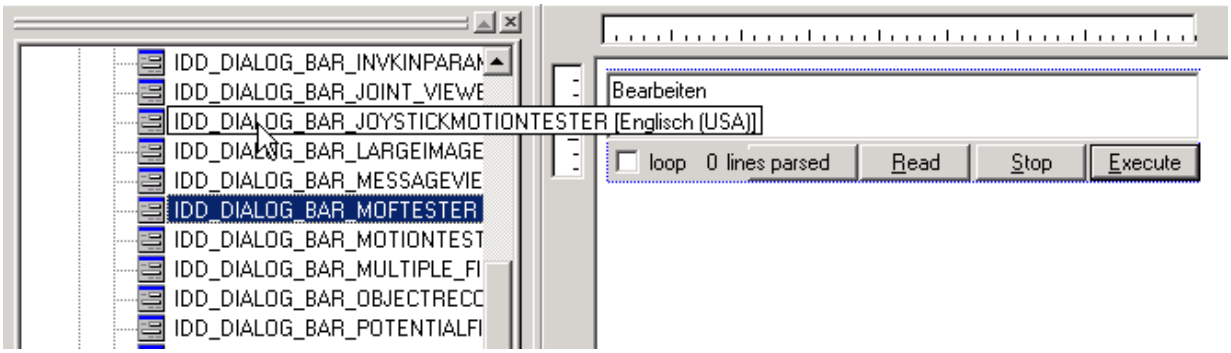
For this, one overwrites the virtual functions `OnConfigurationLoad(..)` and `OnConfigurationSave(..)` and implements them as described in section J.6.2.

K.4 Creating Dialog Bars With Visual C++ 6.0

The procedure for creating and programming dialog bars is independent from whether Visual C++ 6.0 or Visual C++ 2003 .Net is used. Only the resource editors vary (cf. fig. K.5).

[12] Class `CTimeDiagramDlgBar`, Definition and Implementation in *Src/RobotControl/Bars/TimeDiagramDlgBar.h* and *.cpp*.

a)



b)

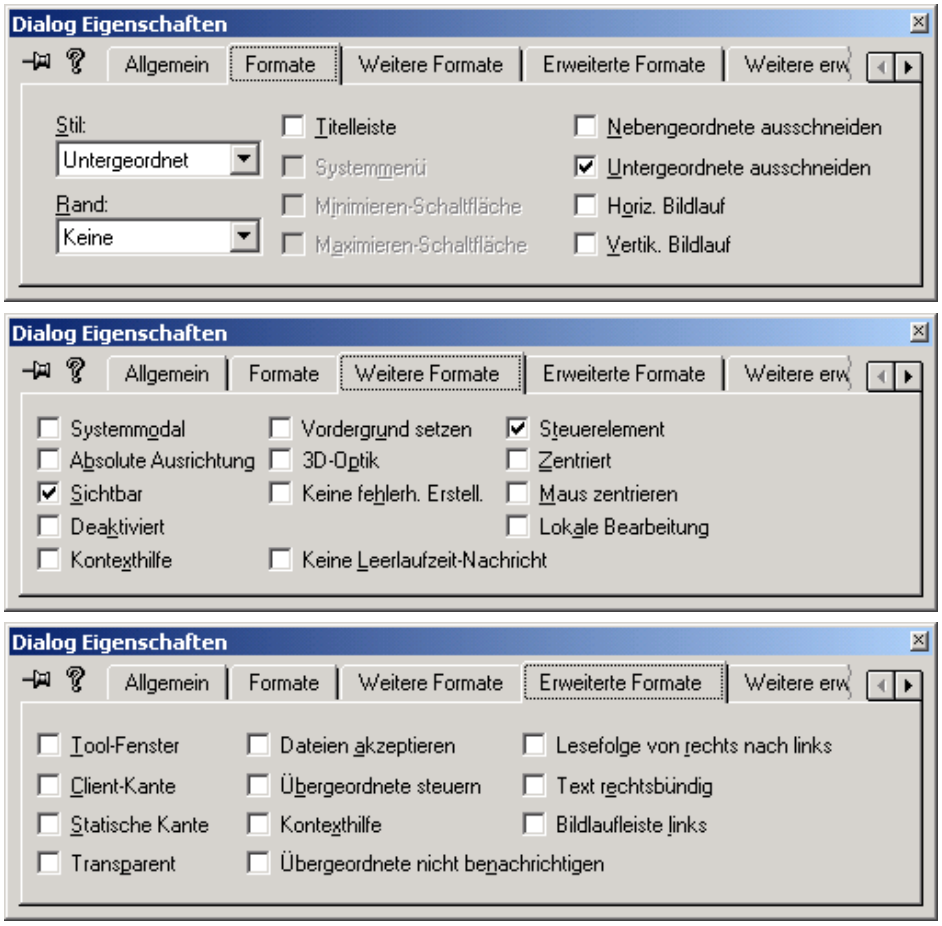


Figure K.5: a) The resource editor from fig. K.1 in Visual C++ 6.0 . b) The dialog from K.2 in Visual C++ 6.0 .

Appendix L

Adding a Tool Bar to RobotControl

This chapter describes how to add a tool bar (cf. sect. J.5.3) to *RobotControl*. There are some similarities to adding dialog bars (see previous chapter). Again, it is possible just to copy an existing tool bar, but following this instructions reduces the potential errors. And despite the different user interfaces of the resource editors, the procedure is the same in Visual C++ 6.0 and Visual C++ 2003 .Net

As an example for all steps serves the existing *Log Player* tool bar[1] (cf. fig. J.2).

L.1 Creating a Tool Bar

After finding an expressive name for the tool bar, the images for the buttons are created and drawn.

L.1.1 Creating images for the buttons

For each button (strictly speaking, for each control) in the tool bar there are two images: one for the normal view (cold view) and one for an activated view (when the mouse moves over the button or when the button is checked). All images are 16×16 pixel wide and have 16 colors. All images for the cold view and all for the active view are stored in a single bitmap-file each (in *Src/RobotControl/RES*). The file names should be similar to the name of the tool bar and the images for the normal view should get the suffix `Cold`. In the example of the *Log Player* tool bar the files should be called `LogPlayerCold.bmp` and `LogPlayer.bmp` (cf. fig. L.1).

The easiest way to create these bitmaps is to copy and rename existing files. After that one imports the bitmaps to the resources by right-clicking the entry “Bitmap” and then “Import”. The imported resources get the IDs `IDB_BITMAP1` and `IDB_BITMAP2`, which should to be renamed according to the file names. In the example it is `IDB_LOGPLAYER_COLD` and `IDB_LOGPLAYER`.

[1] Class `CLogPlayerToolBar`, definition and implementation in *Src/RobotControl/Bars/LogPlayerToolBar.h* and *.cpp*, instantiation in `CRobotControlMainFrame` (*Src/RobotControl/RobotControlMainFrame.h*).

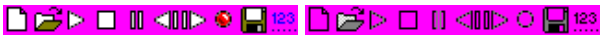


Figure L.1: Button images for the *Log Player* tool bar. Left: `LogPlayer.bmp`, right: `LogPlayerCold.bmp`.

Then the images can be opened. With the “Image” – “Tool Bar Editor” menu they can be edited more comfortable. The background of the images has to be pink (cf. Fig L.1). This color is then automatically replaced by the color that was selected by the user for dialog backgrounds (usually gray). As a convention, the images for the normal (cold) view should only use the colors white, light gray, dark gray, and black. Note that the resource editor of Visual C++ can only handle images with 16 colors. If the bitmaps are edited with another program and if more than 16 colors are used, it is not possible to reopen them in Visual C++.

L.1.2 Creating IDs for Controls

Next, for the tool bar itself and for each control on it an ID has to be defined. For that, the file “Resource.h”[2] Is edited manually. Between `ID_TOOLBAR_FIRST` and `ID_TOOLBAR_LAST` eine ID an ID for the tool bar is defined:

```
#define ID_TOOLBAR_FIRST                104
#define ID_TOOLBAR_LOGPLAYER            104
..
#define ID_TOOLBAR_LAST                 116
```

After that for each control an ID is defined. As convention, these should start with the name of the tool bar, for example `IDC_LOGPLAYER_...` .

```
#define IDC_LOGPLAYER_NEW               1038
#define IDC_LOGPLAYER_OPEN              1039
#define IDC_LOGPLAYER_PLAY              1040
#define IDC_LOGPLAYER_STOP              1041
#define IDC_LOGPLAYER_PAUSE             1042
#define IDC_LOGPLAYER_STEP_FORWARD      1043
#define IDC_LOGPLAYER_STEP_BACKWARD     1044
#define IDC_LOGPLAYER_RECORD            1045
#define IDC_LOGPLAYER_SAVE              1046
#define IDC_LOGPLAYER_PLAY_SPEED        1047
...
```

At the end of *Resource.h*, the value that shall be used for the next added control has to be set (value of the last control + 10):

[2] Src/RobotControl/Resource.h

```
#define _APS_NEXT_CONTROL_VALUE         1460
```

L.1.3 Labels and Help Texts

For each control of a tool bar, an help text and a short description has to be defined. The first appears in the status bar of *RobotControl* when the user moves the mouse over the control. The short text is displayed either beside the button or as a tool tip.

The texts are edited through the "String Table" in the resource editor of Visual Studio. For each control, both texts are added together, separated by a `\n`, for example for `IDC_LOGPLAYER_STEP_FORWARD`:

```
Play the log file one step forward\nStep Forward
```

Attention! *RobotControl* crashes if these both texts are not defined.

L.1.4 Creating a Class for the Tool Bar

For each tool bar a separate class (in the example `CLogPlayerToolBar`) is created in a separate *.h* and *.cpp* file[3]. Thereto, the templates *template-for-new-tool-bar.h.txt* and *template-for-new-tool-bar.cpp.txt*[4] are copied and renamed to e.g. *LogPlayerToolBar.h* and *.cpp*. The templates contain detailed instructions how to use them (mainly by search and replace).

At last, the new files are added to the project.

L.1.5 Arranging Controls on a Tool Bar

Different from dialog bars, the layout of a tool bar is not defined in the resource but manually in the corresponding class of the tool bar. In the `Init()` function (already created by the template described in the previous section) a two-dimensional array `tbButtons` is defined:

```
void CLogPlayerToolBar::Init()
{
  static TBBUTTONEX tbButtons[] =
  {
    {{0, IDC_LOGPLAYER_NEW, TBSTATE_ENABLED,
         TBSTYLE_BUTTON, 0, 0 }, true },
    {{1, IDC_LOGPLAYER_OPEN, TBSTATE_ENABLED,
         TBSTYLE_BUTTON, 0, 0 }, true },
    {{0, 0, 0, TBSTYLE_SEP, 0,0 }, true },
    ...
```

[3] Directory for all tool bars: *Src/RobotControl/Bars/*.

[4] In the same directory.

```
  };
  ...
}
```

The first field of each line is the position of the corresponding image in the bitmap[5]. The second is the ID of the control. Third is the state of the control, which should be always `TBSTATE_ENABLED`[6]. The fourth is the style of the control, which is normally `TBSTYLE_BUTTON` . `TBSTYLE_SEP` creates a vertical separator line. `TBSTYLE_CHECK` creates a check-button [7]. The next three fields are always set to `0`, `0`, and `true` .

After the definition of the control field the button images are set with:

```
SetBitmaps(IDB_LOGPLAYER, IDB_LOGPLAYER_COLD);
```

The first parameter is the ID of the bitmap for the activated view and the second for the normal view (cf. sect. L.1.1).

With

```
SetButtons(sizeof( tbButtons ) / sizeof( tbButtons[ 0 ] ),
  tbButtons);
```

the controls of the tool bar are initialized.

During that, the function `hasButtonText(..)` is called for each control. It has to return, whether the short description text shall be displayed besides the button or not:

```
bool CLogPlayerToolBar::HasButtonText(UINT nID)
{
  switch (nID)
  {
  case IDC_LOGPLAYER_PLAY_SPEED: return true;
  default: return false;
  }
}
```

[5]Counting starts at 0

[6]The state of a button such as active/ inactive or checked/ unchecked is set in the `updateUI` method (cf. sect. L.2.2.).

[7]This should also better be done in the `updateUI()` method.

Figure L.2: The *LogPlayer*-tool. Left at full width, right at restricted width and after clicking on the chevron.

L.1.6 Embedding a Tool Bar into the Main Window

Each tool bar is embedded into the main window[8]. Thereto, the header file of the tool bar is included into *RobotControlMainFrame.cpp*:

```
#include "Bars/LogPlayerToolBar.h"
```

Then, the tool bar is created using the macro `CREATE_TOOLBAR`[9] in the function `CRobotControlMainFrame::createDialogBarsAndToolBars()`:

```
CREATE_TOOLBAR(CLogPlayerToolBar,ID_TOOLBAR_LOGPLAYER,
               "Log Player","&Log Player");
```

The first parameter is the class name of the tool bar, the second the ID. The third parameter defines, which text is displayed at the left border of the tool bar. The fourth specifies the text for the entry in the context menu for opening and closing tool bars. With `&` single letters can be underlined.

After all the steps up to here, the tool bar should be displayed in *RobotControl* (It can be switched on/ off with the context menu in the tool bar and menu area). The *LogPlayer*-tool bar looks after the previous steps as in figure L.2.

L.2 Programming Tool Bars

Programming Tool Bars is very similar to programming dialog bars (see previous chapter). That's why in this section only differences are described.

[8] Class `CRobotControlMainFrame`, definition and implementation in *Src/RobotControl/RobotControlMainFrame.h* and *.cpp* .

[9] Definition in *Src/RobotControl/RobotControlToolBar.h* .

L.2.1 Adding Drop-Down-Lists, Edit Controls and Sliders

The used tool bar library was extended by the possibility to place drop-down-lists, edit controls, and sliders on a tool bar. For all of these three control types, at first a normal button must be added to the tool bar (see previous section).

For example in the *LogPlayer*-tool bar there is a drop-down-list for the playing speed. For this, an image for the control was added to button bitmap (at 10th position in fig. L.1), the image was assigned in `CLogPlayerToolBar::Init()` to the ID `IDC_LOGPLAYER_PLAY_SPEED`, a button was added at the third position of the tool bar, the short text "Speed" was specified, and `hasButtonText()` returned true for the ID of the control (cf. fig. L.2).

After adding a button for the control, a member variable of the type `CComboBox` is added to the tool bar, for example:

```
/** A combo box for the play speed */
CComboBox speedCombo;
```

At the end of the `Init()` function the control is added with `AddCombo(..)` to the tool bar:

```
AddCombo(&speedCombo, IDC_LOGPLAYER_PLAY_SPEED, 60);
```

The first parameter is the address of the member variable of the drop-down-list, the second is the ID of the control. The third defines the width of the control in pixels (Attention! The width should similar to the width of the image + short text[10]). After that the tool bar looks as in figure J.2.

The embedding of edit controls and sliders is very similar. For edit controls, a member variable of the type `CEdit` and one of the type `CComboBox` is created. In the `Init()` method the edit control is added with `AddEdit(..)`. Parameter 1 is the address of the `CEdit`-variable, parameter 2 defines the ID of the control, parameter 3 the width and parameter 4 die address of the `CComboBox`-variable. An example for that can be found in the *DebugKeys*-tool bar[11] (cf. fig. H.1).

For a slider control, a variable of the type `CSliderCtrl` is created. It is embedded in `Init()` with `AddSlider(`address, ID, width`)`. An example for that can be found in the *Game*-tool bar[12].

[10] This can be seen as a hack.

[11] Class `CDebugKeysToolBar`, definition and implementation in *Src/RobotControl/Bars/DebugKeysToolBar.h* and *.cpp* .

[12] Class `CGameToolBar`, definition and implementation in *Src/RobotControl/Bars/GameToolBar.h* and *.cpp* .

L.2.2 Changing the State of Controls

It is not possible to activate/ deactivate or check/ uncheck controls from within the tool bar. The main window is responsible for that (see section J.5.1). By default, all controls of *RobotControl* are activated there. To activate or deactivate a control dependent on some states of a tool bar, one first overwrites the virtual function `updateUI(..)`[13]:

```
/**
* Enables the controls in the tool bar.
* This function is called from the main window.
* @param pCmdUI An interface to the control that allows
*        enabling/disabling, checking etc.
*/
virtual void updateUI(CCmdUI* pCmdUI);
```

Then this function is implemented in the `.cpp` file. The ID of the control can be accessed with `pCmdUI->m_nID`. A control is disabled with `pCmdUI->Enable(false)` and enabled with `pCmdUI->Enable(true)`. With `pCmdUI->SetCheck(true)` a button can be checked.

Thereto, `OnUpdateCmdUI(..)` must be called from the `updateUI(..)` function of the main window[14] for the desired controls:

```
void CRobotControlMainFrame::OnUpdateCmdUI(CCmdUI* pCmdUI)
{
   switch(pCmdUI->m_nID)
   {
   ...
   case IDC_LOGPLAYER_NEW:
   case IDC_LOGPLAYER_OPEN:
   ...
   case IDC_STATUS_BAR_LOGPLAYER_FILENAME:
     toolBarMap[ID_TOOLBAR_LOGPLAYER]->updateUI(pCmdUI);
     break;
     break;
   ...
   }
}
```

The class of the tool bar is accessed with `toolBarMap[id]` (`id` is the ID of the dialog bar).

[13] Defined in the base class `CRobotControlToolBar` .
[14] In *Src/RobotControl/RobotControlMainFrame.cpp*.

L.2.3 Handling Window Messages

Different from dialog bars, messages about the interaction of a user with the tool bars are not handled in the corresponding class but in the main window. This automatically asks all tool bars whether they want to handle the message.

For this, one overwrites the virtual function `handleCommand(..)`:

```
/**
* Handles control notifications which arrived in the main frame
* @param command The id of the control, menu, accelerator etc.
* @return If the command was handled.
*/
virtual bool handleCommand(UINT command);
```

In the implementation of the function `true` is returned when the message was handled, otherwise `false`:

```
bool CLogPlayerToolBar::handleCommand(UINT command)
{
  switch(command)
  {
  case IDC_LOGPLAYER_NEW:
    logPlayer._new();
    fileName="new log file";
    return true;
  ...
  default:
    return false;
  }
}
```

Analogous to that, selection changes in drop-down-lists are handled with `handleSelChange(UINT nID)` and changes in edit controls are handled with `handleEditChange(UINT nID)`:

```
/**
* Handles selection change events for combo boxes.
* That function is automatically called from the
* main frame window for all combo boxes of the toolbar.
* @param nID the command id of the combo box
* @return if the message was handled
*/
virtual bool handleSelChange(UINT nID);
```

```
/**
* Handles change events for edit controls.
* That function is called automatically from the
* main frame window for all edit controls of the toolbar.
* @param nID the command id of the edit control
* @return if the message was handled
*/
virtual bool handleEditChange(UINT nID);
```

L.3 Integration into the Overall Application

These mechanisms are identical to those in dialog bars, see section K.3.

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] 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.

[12] 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.

[13] 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.

[14] James Clark. W3C recommendation: XSL transformations (XSLT) version 1.0, 1999. http://www.w3.org/TR/XSLT.

[15] Sony Corporation. Open-r documentation - open-r internet protocol version 4. Technical report, 2004. Available online: http://openr.aibo.com/openr/eng/index.php4.

[16] I. Dahm, D. Deom, M. Hebbel, M. Hülsbusch, J. Kerdels, T. Kindler, H. Koh, T. Lohmann, M. Neubach, W. Nistico, C. Richter, C. Rink, A. Rossbacher, F. Roßdeutscher, B. Schmidt, C. Schumann, P. Serwe, and Dr. J. Ziegler. Project group 442 - final report. Technical report, 2004.

[17] Ingo Dahm and Jens Ziegler. Adaptive methods to improve self-localization in robot soccer. In *RoboCup Symposium Fukuoka*, 2002.

[18] 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.

[19] Uwe Düffert. Vierbeiniges Laufen - Modellierung und Optimierung von Roboterbewegungen - diploma thesis. http://www.uwe-dueffert.de/publication/dueffert04_diploma.pdf, 2004.

[20] F. Dylla, A. Ferrein, G. Lakemeyer, J. Murray, O. Obst, T. Röfer, F. Stolzenburg, U. Visser, and T. Wagner. Towards a league-independent qualitative soccer theory for robocup. In *8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.

[21] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[22] David C. Fallside. W3C recommendation: XML schema part 0: Primer, 2001. http://www.w3.org/TR/xmlschema-0/.

[23] 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.

[24] 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.

[25] 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.

[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] V. Jagannathan, R. Dodhiawala, and L. Baum. *Blackboard Architectures and Applications*. Academic Press, Inc., 1989.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Tim Laue and Thomas Röfer. A behavior architecture for autonomous mobile robots based on potential fields. In *8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.

[34] 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.

[35] Scott Lenser, James Bruce, and Manuela Veloso. Vision - the lower levels. Carnegie Mellon University Lecture Notes, October 2003. http://www-2.cs.cmu.edu/ robosoccer/cmrobobits/lectures/vision-low-level-lec/vision.pdf.

[36] Martin Lötzsch. DotML Documentation, 2003. http://www.martin-loetzsch.de/DOTML.

[37] Martin Lötzsch. XABSL web site, 2003. http://www.ki.informatik.hu-berlin.de/XABSL.

[38] 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.

[39] 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.

[40] D. MacKenzie, R. Arkin, and J. Cameron. Multiagent mission specification and execution. *Autonomous Robots*, 4(1):29–52, 1997.

[41] Jonathan Marsh and David Orchard. W3C candidate recommendation: XML inclusions (XInclude) version 1.0, 2002. http://www.w3.org/TR/xinclude/.

[42] R. Mohr and B. Triggs. Projective geometry for image analysis, 1996.

[43] M. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society of Industrial and Applied Mathematics*, 5(1):32–38, March 1957.

[44] David Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1996.

[45] 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.

[46] 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.

[47] Thomas Röfer. Quality of ers-7 camera images. http://blechkopf.informatik.uni-bremen.de/pubwiki/bin/view/Bbyters/CameraERS210-ERS7.

[48] Thomas Röfer. Simrobot homepage. http://www.tzi.de/simrobot.

[49] 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.

[50] 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.

[51] L.G. Roberts. Machine perception of three dimensional solids. *Optical and Electro-Optical Information Processing*, pages 159–197, 1968.

[52] S. Russel and P. Norvig. *Artificial Intelligence, a Modern Approach*. Prentice Hall, 1995.

[53] D. Schulz and D. Fox. Bayesian color estimation for adaptive vision-based robot localization. In *Proceedings of IROS*, 2004.

[54] Edwin Hsing-Mean Sha and Kenneth Steiglitz. Maintaining bipartite matchings in the presence of failures. In *International Parallel Processing Symposium*, pages 57–64, 1993.

[55] 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.

[56] 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.

[57] A. H. Timmer and J. A. G. Jess. Exact scheduling strategies based on bipartite graph matching. pages 42–47.

[58] Thomas Wagner and Kai Hübner. An egocentric qualitative spatial knowledge representation based on ordering information for physical robot navigation. In *8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.

[59] Dirk Wilking and Thomas Röfer. Real-time object recognition using decision tree learning. In *8th International Workshop on RoboCup 2004 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer, 2005. to appear.