An Architecture for a National RoboCup Team

Thomas Röfer

Center for Computing Technology (TZI), FB3, Universität Bremen
Postfach 330440, 28334 Bremen, Germany
roefer@tzi.de

Abstract. This paper describes the architecture used by the GermanTeam 2002 in the Sony Legged Robot League. It focuses on 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 addition, the tools developed by the GermanTeam will be presented, e.g. the first 3-D simulation used in the Sony Legged Robot League.

1 Introduction

All RoboCup leagues share the problem that there are more teams that want to participate in the world championship than a normal contest schedule can integrate. Therefore, each league has its own approach to limit the number of participants to a certain amount. For instance, teams in the simulation league have to qualify by submitting a team description and two log files, one of a game against a strong opponent, the other of a game against an average one. Based on this material, a committee selects the teams for the championship. In contrast, participants in the Sony Legged Robot League only qualify by submitting a description of their scientific goals—before they have even worked with the robots. Each year, the Sony Legged League grows a little bit, e.g. from 16 to 19 teams from 2001 to 2002. So far, teams that were once accepted in the league were allowed to stay during the following years without a further qualification, amongst other reasons because of their investment in the robots. Thus, only a few new teams have the chance to join the league.

Currently, it is discussed how to provide to chance to participate in the league to more research groups. One solution would be to set up *national teams*, as the GermanTeam that was founded in 2001 [1]. The GermanTeam currently consists of students and researchers at five universities: the Humboldt-Universität zu Berlin, the Universität Bremen, the Technische Universität Darmstadt, the Universität Dortmund, and the Freie Universität Berlin. The members of the GermanTeam are allowed to participate as separate teams in national contests, but will jointly line up at the international RoboCup championship as a single team.

The other solution would be to install national leagues, of which the winning team will get the ticket to participate in the international contest. On the one hand, this approach would enforce the competition, but on the other hand, the goal of the RoboCup initiative is to promote research, and competing teams do not work together very well.

Therefore, it may be a good compromise to support collaboration on the lower, national level, while stressing the element of competition on the international level.

The GermanTeam is an example of a national team. The members will participate as separate teams in the German Open 2002, but will form a single team at Fukuoka. Obviously, the results of the team would not be very good if the members will develop separately until the middle of April, and then try to integrate their code to a single team in only two months. Therefore, an architecture for robot control programs was developed that allows to implement 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 varying number of concurrent processes. The approach will be described in section 3. Before that, section 2 will motivate why the robot control programs are implemented in a platform-independent way, and how this is achieved. Finally, in section 4, the tools that were implemented to support the development of the robot control programs are presented.

2 Platform-Independence

One of the basic goals of the architecture of the GermanTeam 2002 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 Motivation

There are several reasons to enforce this approach:

The Use of 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 testing a program 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 image processing methods between the Sony Legged Robot league and the F180 league.

Non Disclosure Agreement. The participants in the Sony Legged Robot League get access to internal information about the software running on the Sony Aibo robot. Therefore, the universities of all members of the league had 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 a championship is usually only made available to the other teams in the league, but not to the public. However, as it is the main goal of the RoboCup initiative to promote research, the presentation of the code to a wider audience is desirable. By encapsulating the NDA-relevant code and by the means of a simulator, the GermanTeam will be able to publish a version of the system that will not violate the NDA between the universities and Sony.

2.2 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 used by the GermanTeam 2002 is illustrated in section 3.3.

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 German-Team 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 German-Team 2002, 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.

2.3 Supported Platforms

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

Sony Aibo Robots. The Sony Aibo (cf. Fig. 1) is four-legged robot with 20 degrees of freedom, a color camera, and more than 30 further sensors. The specialty of the Sony Legged Robot League is that, on the one hand, the movements of the robots are

Fig. 1. Two Sony Aibo robots

the most complex in RoboCup so far, and a camera is the most central sensor, on the other hand, teams need not to build their own robots (it fact, it is not allowed to change them). Thus, it is possible that one team can run the code of another team, similar to the simulation league. However, to be able to also share the source code in the GermanTeam, the architecture described above was implemented on the robots. The implementation is based on the techniques provided by the operation system that natively runs on the robots, i.e. Aperios.

Microsoft Windows. The platform independent environment was also implemented on Microsoft Windows as a part of a special controller in *SimRobot* (cf. section 4.1) and, sharing the same code, in the general development support tool *RobotControl* (cf. section 4.2).

3 Multi-Team Support

The major goal of the architecture presented in this paper is ability is 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 that could be identified 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 3.2. 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. they can group together modules to processes that are running concurrently (cf. section 3.2).

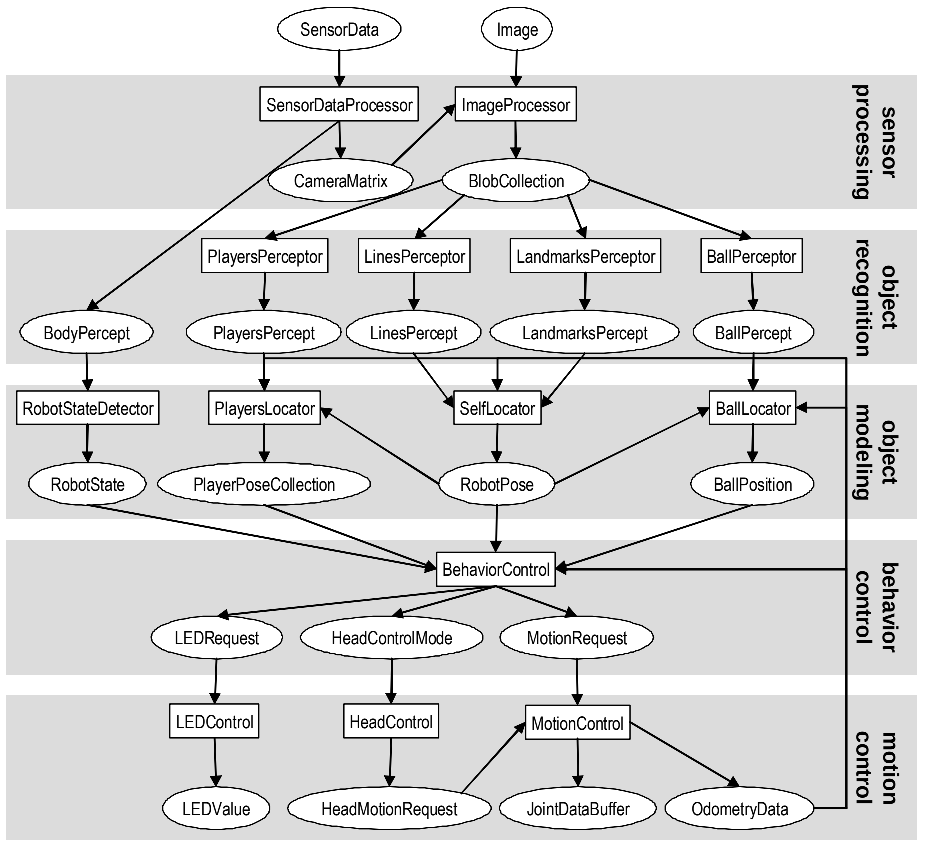


Fig. 2. The modules implemented by the GermanTeam 2002

3.1 Tasks

Figure 2 depicts the tasks that were identified by the GermanTeam for playing soccer in the Sony Legged Robot League. They can be structured into five levels:

Sensor Data Processing. On this level, the data received from the sensors is preprocessed. For instance, the image delivered by the camera is segmented, and then it is converted into a set of blobs, i.e. image regions of the same color class. The current states of the joints are analyzed to determine the direction the camera is looking at. In addition, further sensors can be employed to determine whether the robot has been picked up, or whether it felt down.

Object Recognition. On this level, the information provided by the previous level is searched to find objects that are known to exist on the field, i.e. landmarks (goals and flags), field lines, other players, and the ball. The sensor readings that were associated to objects are called *percepts*.

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 modeled, i.e. the location of the robot itself, the poses of the other robots, 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 fourth 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 fifth 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 many other modules.

3.2 Debugging Support

One of the basic ideas of the architecture is that multiple solutions exist for a single task, and that the developer can switch between them at runtime. In addition, it is also possible to include additional switches into the code that can also be triggered at runtime. The realization is an extension of the debugging techniques already implemented in the code of the GermanTeam 2001 [2]: *debug requests* and *solution requests*. The system manages two sets of information, the current state of all *debug keys*, and the currently active solutions. Debug keys work similar to C++ preprocessor symbols, but they can be toggled at runtime. A special infrastructure called *debug queues* 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 debug queues are also used to transmit other kinds of data between the robot(s) and the debugging tool on the PC (cf. section 4). For example, motion requests can directly be sent to the robot, images, text messages, and even drawings can be sent to the PC. This allows to effectively visualize the state of a certain module, textually and even graphically. These techniques work both on the real robots and on the simulated ones (cf. section 4.1).

3.3 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:

Communication between Processes. In the robot control program developed by the GermanTeam 2001 for the championship in Seattle, the different processes exchanged their data through a shared memory [2], i.e., a blackboard architecture [3] 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 the other. 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 new ability of the Sony Aibo robots to exchange data between processes via a wireless network.

The communication scheme introduced in GT2002 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. A single line of code is sufficient to establish a communication link. A predefined scheme separates the processing time into a communication phase 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 directed 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 instantaneously after they have been asked for, and afterwards it pre-calculates to 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

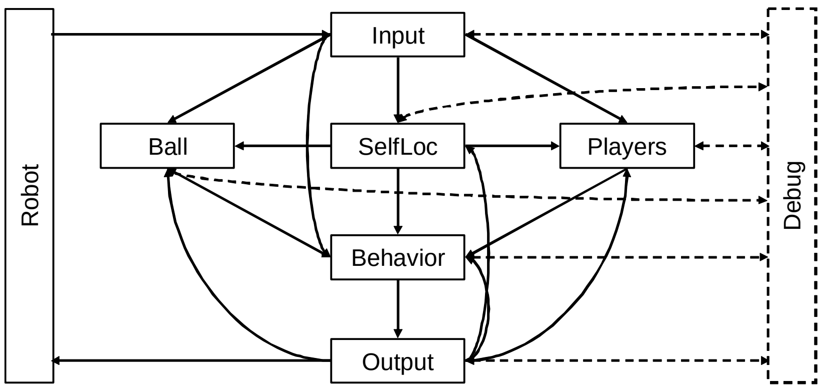


Fig. 3. Process layout of the Bremen Byters. The broken lines indicate the debugging part.

package is the only one that has to be called explicitly. This significantly eases the implementation of new processes.

Different Layouts. The figures 3 and 4 show two different process layouts. Both contain a debug process that is connected to all other processes via debug queues. Note that debug queues are transmitted as normal packages, i.e. a package contains a whole queue. Comparing the two process layouts, it can be recognized that on the one hand, the Bremen Byters try to parallelize as much as possible; on the other hand, the Humboldt Heroes focus on using only a few processes, i.e. the first four levels (cf. Fig. 2) are all integrated into the process *Cognition*. In the layout of the Bremen Byters, one process is used for each of the levels one, four, and five, and three processes implement parts of the levels two and three, i.e. the recognition and the modeling of individual aspects of the world state are grouped together. Odometry is used to decompose information that is dependent: although both the *players* process and the *ball* process require the current pose of the robot, they can run in parallel to the self-localization process, because the odometry can be used to estimate the spatial offset since the last absolute localization. This allows running the ball modeling with a high priority, resulting in a fast update rate, while the self-localization can run as a background process to perform a computationally expensive probabilistic method as, e.g., the one described in [4] or the method used by the GermanTeam 2001 [2].

Currently, it is not known which process layout will be the more successful one. The Darmstadt Dribbling Dackels are using a third approach that is a compromise between the two discussed here, and all three will compete against each other at the German Open. So the best can be used for the world championship.

4 Development Tools on the PC

Two tools were implemented on the PC to ease the development of the robot control programs. The first is a 3-D simulator, and the second is a general tool that provides nearly any support imaginable, even the simulator is integrated.

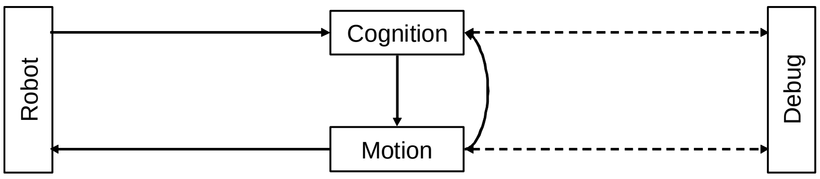


Fig. 4. Process layout of the Humboldt Heroes. The broken lines indicate the debugging part.

4.1 SimRobot

SimRobot is a kinematic robotics simulator that was developed in the author's group [1]. It is written in C++ and is distributed in public domain [6]. It consists of a portable simulation kernel and platform specific graphical user interfaces. Implementations exist for the *X Window System*, *Microsoft Windows 3.1/95/98/ME/NT/2000/XP*, and *IBM OS/2*. Currently, only the development for the 32 bit versions of Microsoft Windows is continued.

A simulation in SimRobot consists of three parts: the *simulation kernel*, the *graphical user interface*, and a *controller* that is provided by the user. The GermanTeam 2002 has implemented the whole simulation of up to eight robots including the inter-process communication described in section 3.3 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 implementators of such modules to compare their results with the correct ones.

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

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

There are only two types of emitters in SimRobot: *radial emitters* send their radiation to all directions, whereas *spot emitters* have a certain opening cone. In addition, an ambient intensity can be specified for each radiation class that defines the base intensity for all surfaces in a simulation scene. Hence, the surfaces that are not reached by any of the emitters in a scene still have a sensible radiation signature. In the RoboCup simulation, a single radial emitter simulates a lamp shining above the field, and a high degree of ambient light imitates any other illumination.

Bodies. Currently, bodies can only be modeled as a collection of polygons. Each polygon has a radiation vector that defines its appearance—together with the radiation of the emitters that reaches the surface.

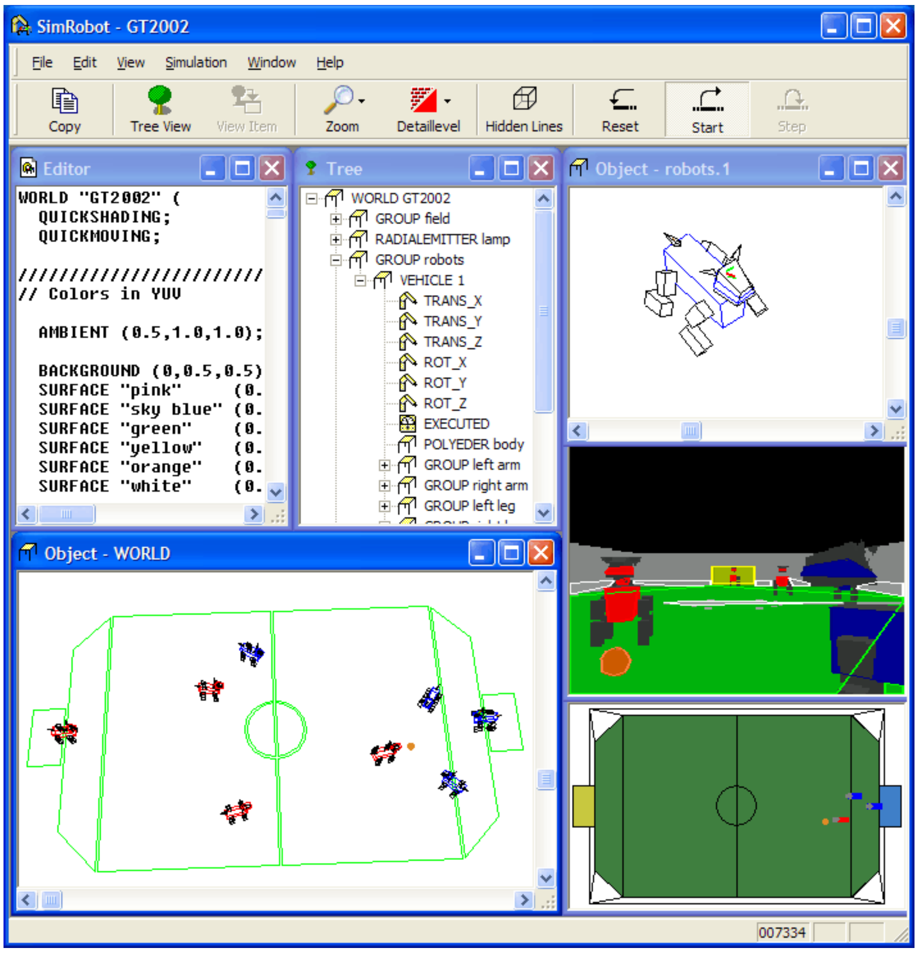
Actuators allow the user or the *controller* to actively influence the simulation. They can be used, e.g., to move a robot or to open doors. Each actuator can contain other objects, i.e. the objects that it moves. SimRobot provides four types of actuators: rotational joints, translational joints, objects moving in space in six degrees of freedom, and vehicles with typical car kinematics, i.e. with a driving axle and a steering axle. SimRobot is only a kinematic simulator; thus it cannot directly simulate walking machines. Therefore, the motion of the simulated Aibos is generated by a trick: the GermanTeam 2002 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.

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

1. Intensities of Radiation. There are two types of cameras that allow to measure two-dimensional arrays of intensities of radiation. The *camera* object imitates normal pinhole cameras, and is used to simulate Aibo's color camera. The *facette* simulates cameras with a spherical geometry, i.e. the angle between all pixels is constant. The sensor readings can be calculated using *flat shading*, i.e. each surface has a single combination of intensities, or with different intensity signatures for every pixel. In addition, it is possible to determine shadows.
2. Distances. There are several sensors that measure distances. A *whisker* can imitate the behavior of an infrared sensor. On the one hand, it is used to simulate the PSD sensor in Aibo's head. On the other hand, whiskers are employed to implement the ground contact sensors in the feet of the robots.
3. Collision detection. For every actuator, it can be detected whether a collision-free execution of the last command was possible. This information is not available in reality, but it is required for the simulator to suppress motions that result in collisions. However, as each robot has 20 degrees of freedom, this costly calculation is even too slow for a single robot, but it surely is for eight. Therefore, a simpler and less realistic 2-D method is employed to detect collisions between robots in the RoboCup simulation.

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

User Interface. The user interface of SimRobot includes an editor for writing the required scene definition files (cf. Fig. 5, upper left window). 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, upper middle window). This tree is the starting point for opening fur-



```
WORLD "GT2002" (
  QUICKSHADING;
  QUICKMOVING;

////////////////////////
// Colors in YUV

  AMBIENT (0.5,1.0,1.0);

  BACKGROUND (0,0.5,0.5)
  SURFACE "pink"      (0.
  SURFACE "sky blue"  (0.
  SURFACE "green"     (0.
  SURFACE "yellow"    (0.
  SURFACE "orange"    (0.
  SURFACE "white"     (0.
```

```
WORLD GT2002
  GROUP field
  RADIALEMITTER lamp
  GROUP robots
    VEHICLE 1
      TRANS_X
      TRANS_Y
      TRANS_Z
      ROT_X
      ROT_Y
      ROT_Z
      EXECUTED
      POLYEDER body
      GROUP left arm
      GROUP right arm
      GROUP left leg
```

Fig. 5. SimRobot simulating the GermanTeam 2002

ther views. SimRobot can visualize any object and the readings of any sensor that are defined in a scene. Objects are displayed as wire-frames with or without hidden line removal (cf. Fig. 5, lower left and upper right window) and parts of the scene can be hidden. In case of the lower left window, the flags, the goals, and the field border are not displayed.

Sensor data can be depicted as line graphs, column graphs, monochrome images, and color images (cf. Fig. 5, middle right window). In addition, depth images can be visualized as single image random dots stereograms. Any of these views and a numerical representation of the sensory data can be copied to the system's clipboard for further processing, e.g., in a spreadsheet application or a word processor. The whole window layout is stored when a scene is closed and restored when SimRobot is started again with the same scene.

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

A very powerful function is the ability to insert *views* into the scene. These are similar to sensors but in contrast to them, their value is not determined by the simulation but instead by the controller. This allows the controller to visualize, e.g., intermediate data. In fact, the middle right window in figure 5 is a view that contains a camera image overlaid by the so-called blobs collection, i.e., colored octagonal areas detected by the robot control program's image processor. The lower right window is completely drawn by the controller: a field with the visualization of the estimations of a robot's own pose (in this case the right goalie), the locations of some other robots, and the position of the ball.

4.2 RobotControl

RobotControl is the successor of *DogControl*, the debugging tool used by the GermanTeam in 2001 [2]. Its purpose is to integrate all functionality that is required during the development of the control programs for the Sony Aibo robots.

Running Robot Controllers. First of all, RobotControl has the ability to run the process-layouts that make up the robot control programs. The simulation kernel of SimRobot is integrated into RobotControl, but in contrast to SimRobot, the robot controller cannot only be provided with simulated inputs. It is also possible to connect it to real robots via the wireless network, and, as a third possibility, the inputs can be generated by replaying log files.

Log Files can either be recorded with a robot, storing them on a memory stick, or the data can be transferred from the robot via the wireless network to RobotControl, and then it will be recorded to a file on the harddisk of the PC. As the same robot controller code runs in all environments, even a simulated robot can produce log files. Log files can contain sensor data and intermediate data as, e.g., blob collections. RobotControl is able to replay log files in real-time or step by step. As RobotControl can run under a debugger, all normal debugging features are available (setting breakpoints, inspecting variables, etc.).

Extensibility. The main purpose of RobotControl is to function as the user interface of the Aibo robots. Therefore, it provides the infrastructure to easily add new toolbars and dialogs. The window layout of RobotControl is always stored and restored on restart. Figure 6 shows a screenshot of RobotControl, in which several toolbars and dialogs can be seen. Toolbars control the replay of log files, they control running the simulation, they allow sending debug keys to the real or the simulated robots, provide

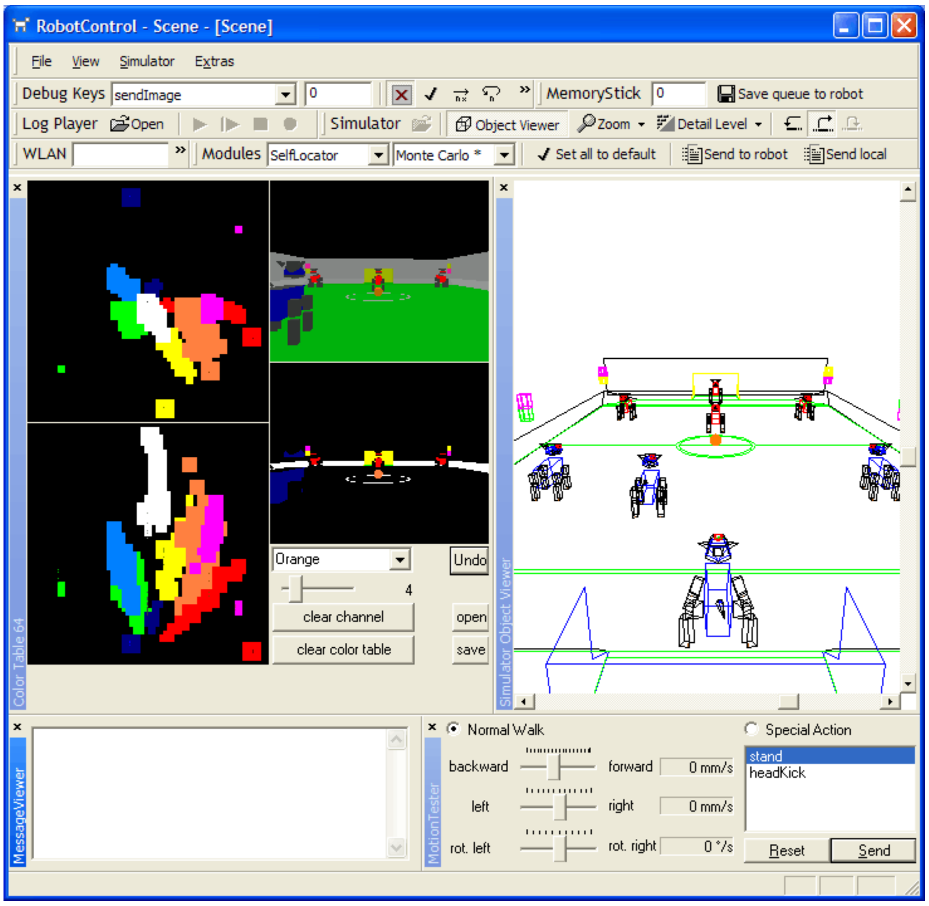


Fig. 6. RobotControl: color tool, simulation, message viewer, and motion tester

the ability of switching solutions and configuring the wireless network, etc. Dialogs allow generating color tables (for image segmentation, shown in Fig. 6 left), display the simulator scene (Fig. 6 right), control the motions of the robot (Fig. 6 lower right pane), and display debug messages (Fig. 6 lower left pane) and debug drawings (Fig. 6 center). As a result, RobotControl is a very powerful and flexible tool.

5 Conclusion and Future Work

The paper has presented the architecture used by the GermanTeam 2002 in the Sony Legged Robot League. The architecture has been designed for a national team, i.e. a team from different universities that compete against each other in national contests, but that will form a single team at the international RoboCup world championship. The architecture is platform-independent, and is currently implemented on two different systems, i.e. the Sony Aibo robots and on Microsoft Windows—integrated into the simulator SimRobot and the control software RobotControl. SimRobot is the first 3-D simulator used in the Sony Legged League, and has also been integrated into RobotControl, a universal tool to support the development of the robot soccer team.

When this paper was written, the implementation of the architecture was finished, but many modules had still to be implemented to be able to successfully participate at the German Open 2002 in Paderborn and the RoboCup world championship in Fukuoka. Thereby, the Bremen Byters will focus on probabilistic modeling of the world state.

Acknowledgements

Although the author contributed to the architecture presented in this paper, namely by realizing the communication scheme and the simulation, the architecture itself is the result of the work of many people in the GermanTeam. From the author's point of view, the main "architects" of the team are Mattias Jüngel, Martin Loetzsch (both Humboldt Universität zu Berlin), Ronnie Brunn, and Max Riesler (both Technische Universität Darmstadt). The author wants to thank them and also all the other members of the team, who are filling the architecture with life.

The author also thanks the Sony Corporation for their professional support, the Deutsche Forschungsgemeinschaft (DFG) for funding parts of the project through the priority program "Cooperating teams of mobile robots in dynamic environments".

References

1. Burkhard, H.-D., Düffert, U., Jüngel, M., Lötzsch, M., Koschmieder, N., Laue, T., Röfer, T., Spiess, K., Sztybryc, A., Brunn, R., Risler, M., v. Stryk, O.: GermanTeam 2001. Technical report. Only available online: http://www.tzi.de/kogrob/papers/GermanTeam2001report.pdf (2001).
2. Brunn, R., Düffert, U., Jüngel, M., Laue, T., Lötzsch, M., Petters, S., Risler, M., Röfer, T., Spiess, K., Sztybryc, A.: GermanTeam 2001. In *RoboCup 2001*. Lecture Notes in Artificial Intelligence. Springer (2001), to appear.
3. Jagannathan, V., Dodhiawala, R., Baum, L.: *Blackboard Architectures and Applications*. Academic Press, Inc. (1989).
4. Lenser, S., Veloso, M.: Sensor resetting localization for poorly modeled mobile robots. In *Proc. of the IEEE International Conference on Robotics and Automation* (2000).
5. Röfer, T.: Strategies for Using a Simulation in the Development of the Bremen Autonomous Wheelchair. In Zobel, R., Moeller, D. (Eds.): *Simulation-Past, Present and Future*. Society for Computer Simulation International (1998) 460-464.
6. SimRobot homepage. http://www.tzi.de/simrobot.