Robot Heal Thyself: Precise and Fault-Tolerant Control of Imprecise or Malfunctioning Robots

Alexander Gloye, Fabian Wiesel, Oliver Tenchio,
Mark Simon, and Raúl Rojas

Freie Universität Berlin, Institut für Informatik
Takustraße 9, 14195 Berlin
Germany
{gloye,rojas}@inf.fu-berlin.de

Abstract. This paper shows how an omnidirectional robot can learn to correct inaccuracies when driving, or even learn to use corrective motor commands when a motor fails, whether partially or completely. Driving inaccuracies are unavoidable, since not all wheels have the same grip on the surface, or not all motors can provide exactly the same power. When a robot starts driving, the real system response differs from the ideal behavior assumed by the control software. Also, malfunctioning motors are a fact of life that we have to take into account.
Our approach is to let the control software learn how the robot reacts to instructions sent from the control computer. We use a neural network, or a linear model for learning the robot's response to the commands. The model can be used to predict deviations from the desired path, and take corrective action in advance, thus increasing the driving accuracy of the robot.
The model can also be used to monitor the robot and assess if it is performing according to its learned response function. If it is not, the new response function of the malfunctioning robot can be learned and updated. We show, that even if a robot loses power from a motor, the system can re-learn to drive the robot in a straight path, even if the robot is a black-box and we are not aware of how the commands are applied internally.

1 Motivation: Robots are Imprecise

The FU-Fighters participated in RoboCup 2004 in Lisbon. The robots had a combination of motors and electronics. The motors performed satisfactorily until we started losing some of them. We drive the motors above their specification, as almost all teams do, and it can happen, during a long tournament, that motors get damaged. Our omnidirectional robots have four wheels, so that the remaining three motors could still be used, but the robot oscillated wildly because the PID controller tried to operate with the four motors. Therefore, a natural question to ask is whether the high-level control can observe the problem through the

computer vision and take corrective action. We would like to send the right commands without having to modify the robot, and without changing the PID controller in the robot's electronics.

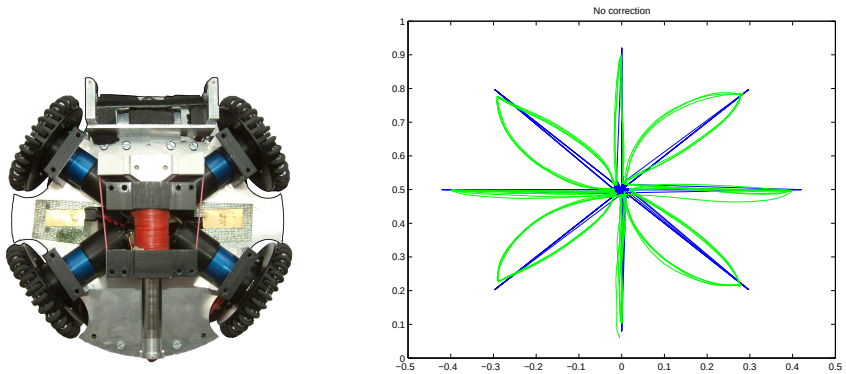

Fig. 1. An omnidirectional robot (left) receives commands for driving in a star-shaped path starting from the origin (right). The robot keeps its orientation towards the north. The robot has difficulties driving straight along the diagonals.

A related problem is that of driving a robot with an imperfect chassis. When the wheels and motors are mounted, it can happen that one wheel finishes having an extra millimeter or two of axis length. More frequently, the motors themselves are different. Some are older motors, some are new. The older motors do not provide exactly the same torque as the newer motors, for the same PWM signal. The PID controller takes care of equalizing the motor speeds, but when the robot starts and before the PID controller can become active, the differences can be high enough to send the robot in a slightly different direction. The difference in the grip of the wheels on the floor can exacerbate the problem. It is difficult to adjust the PID controller to cope with all eventualities, also because such differences are dynamic. Fig. 1 shows how a robot drives when trying to move very fast along a star-shaped path. The inaccuracies are larger along the diagonals, but are present in all eight directions.

The first solution which comes to mind to the problems mentioned above is that of having a perfect physical model of the robot, which can be used to predict the robot's behavior whenever something happens (for example, a motor delivers only half of the torque, or an axis is 3 mm longer than the others, etc.). However, it is difficult to derive a good analytic approximation of the real robot. In addition, since the robots are different we would need one model for each robot.

The solution we propose here is to let the vision system learn the behavior of the robot "on the fly", from observations collected by the cameras. The learned robot response is as good as an analytical model if it can be used to predict

the behavior of the robot to the next command. We can then anticipate if the robot will perform according to our wishes or not, and take corrective action in advance. Think of a soldier who mixes up right with left. You give the command "turn right", and the soldier turns left. After two or three such experiences, you just order the soldier to turn left, when it should go to the right, and vice versa – the soldier behaves now as we would like.

Moreover, we propose to dynamically retrain the robot's behavior predictor, so that whenever the system observes that the current predictor is not accurate enough, a new predictor is computed. Using the new predictor, we can again anticipate what a malfunctioning robot will do with the next command, and then take corrective action. The work described here takes corrective actions purely by software. The robot learns to heal itself.

Related work has been done on methods to cope with differences in the hardware platform, as described by Kleiner, but at a higher behavioral level [9]. The authors teach slightly different robots to optimally shoot a ball. Reinforcement learning is used to learn the correct activation of the behavior needed. In a previous paper, we described how to apply learning algorithms to optimize the PID controller needed for driving an omnidirectional robot [7]. This is learning applied to the lowest hardware level.

There is a general interest in the issue of fault tolerant architectures for robotic control. Parallel control architectures can be used [8]. Self-repairing strategies for 3D motion planning have been investigated [5], and also for production systems [10]. Eventually, self-repairing robots will be built.

2 Learning the robots behavior

We started applying predictors for robot behavior when our robots became too fast for the existing system delay. A robot driving at 2 m/s can move 20 cm in 100 ms. This could be the difference between stopping just in front of another robot or colliding with it. Another advantage of learning the behavior of robots in response to commands, is that the predictor can be used in a simulator.

In our control system loop the only external physical sensors used for behavior control are two video cameras (the motors on the robots have pulse counters, but this information is not available to the off-the-field controlling computer). The global computer vision system analyzes the video images and produces as output the positions and orientations of the robots and ball. Our adaptive vision is described in detail in [11, 4].

The data available for behavior control comes from the past. The frame captured at time t takes a certain time to get to the computer and be processed. Once the video image has been processed, it takes some time for the wireless command to reach the robot, to be decoded, and executed. All of this elapsed time is the system delay, which in our small-size platform varies between 100 to 150 ms.

To cope with the system delay, we let a neural network or linear associator learn the correspondence between past positions and commands sent to the robot

(for example, during the last previously seen six frames) and the future positions (from one to four frames in advance). This data is collected just by driving the robot around the field and logging all positions and commands.

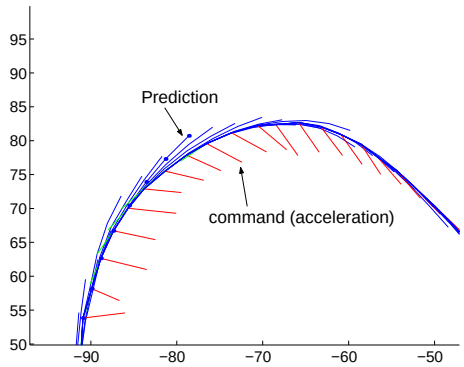

Fig. 2. The robot moves in a curved path. The line segments show the predicted movement for the succesive acceleration commands. The prediction error is larger at the sharp curve.

Once the predictor has been trained, we control the robot according to the predicted position when the commands arrive, and not the past position. Typically, we predict four frames in advance (see [2]). Fig. 2 shows a robot driving and its predicted position four frames in advance, at every frame. The behavior control module send the commands to the robots: desired rotational velocity, driving speed and direction, as well as the activation of the kicking device. The hierarchical reactive behavior control system of the FU-Fighters team is described in [1].

The execution of commands in the robot is overseen by a microcontroller running a PID controller for omnidirectional wheels, and which uses the feedback from the motor pulse counters (see [3, 7]).

For more information about measuring the delay, the design of the neural and linear prediction system, and the different learning algorithms, please refer to our previous work described in [2, 6].

3 Correcting a Robot through Prediction

In what follows we assume that the vision system has tracked the robot for some time (several seconds) and has trained a neural network or linear associator to predict the position of the robot several frames in advance. We then apply this predictor in different ways, as described next.

3.1 Startup Correction

As Fig. 1 shows, when a robot starts driving fast, the intended direction is different from the actual direction taken by the robot. We first show how to correct this simple case. We assume that the history of the robot does not influence its motion in the future, and we apply the standard prediction module in the experiments.

The inputs to our predictor are usually the positions and commands in the last frames. We set these inputs to zero, and we leave as only non-zero input the command we want to test. The output of the predictor is the predicted "instantaneous" response of the system. If the predicted behavior (the direction (x, y) in which the robot moves) differs from the desired behavior, we make a search for a better command for the Euclidean velocities along the x and y axis, i.e. (v_x, v_y), varying the relative magnitude of v_x and v_y. We then just select the command for the most accurate predicted motion (in the desired direction (x, y)).

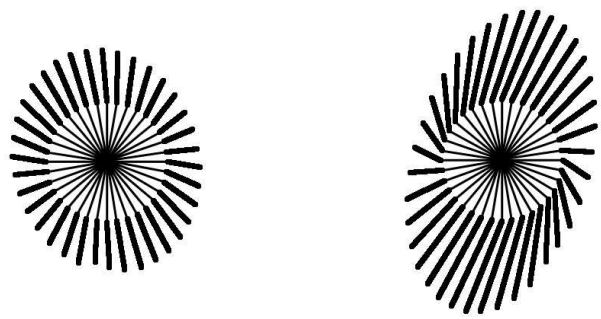

Fig. 3. The robot command correction for two types of robots with the startup method. A normal robot, and a robot with a 30° rotated cover, (from left to right). The thin radial lines represent the ordered driving direction. The thick line is the real direction driven by the robot after four frames.

Fig. 3 illustrates the idea of the startup correction. Each picture shows the desired driving direction for a command (v_x, v_y) as a line from the center of a circle towards the periphery. At the periphery of the circle we continue the line with the real direction in which the robot drives when it receives this command. For a perfect robot, each radial line continues the other. A broken line shows that the robot does not drive as precisely as it should for a given command. The first image in Fig. 3 shows the behavior of a conventional robot, the second image to the right shows the behavior of a robot where the cover has been rotated 30°. As can be seen, from this information it is easy to select the command which drives the robot not in the ideal direction, but in the real direction we want to obtain.

An extreme example that this simple method works, is the case of a robot where the cover has been rotated 180°. The robot is uncontrollable and drives widly on the field. After the startup correction is applied, the robot can be controlled and follows the star-shaped path (see Fig. 4).

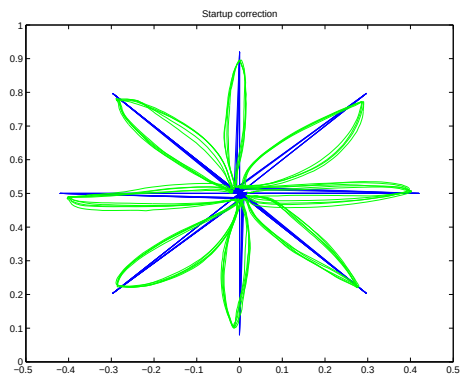

Fig. 4. The path of a robot moving along a star-shaped path, when the cover has been rotated 180° and the startup correction is applied. Without the correction the robot is uncontrollable.

In the second method, considered below, we include the history of the robot in our computation, further improving the results.

3.2 Online Correction

We can apply the same basic idea discussed above in a dynamic world. We include in the prediction computation the history of the robot and the commands sent in the past. We predict the position of the robot four frames in advance from the last seen position, and five frames in advance (adding one fourth of the displacement and rotation for the four-frame prediction).

Now, we can experiment with different robot commands to match the desired motion with the predicted one. The predicted motion of the robot is the difference between its position in the 4th frame in the future, and the predicted position in the fifth frame in the future (remember that commands are only effective in the future due to the system delay). We now minimize the difference between the predicted motion and the desired motion.

We can easily modify our predictor to compute directly the difference we need. We trained a linear predictor, using as input the last six positions and orientations of the robot (relative to the current position and orientation), and also the last six and current motion commands. The output of the new predictor is the relative direction between the fourth and the fifth frame in the future. Varying the current robot command results in different outputs, which are compared

to the desired direction. Thus, we can make a search for the robot command, which minimizes the error between the desired and the real (predicted) direction. The search is very fast.

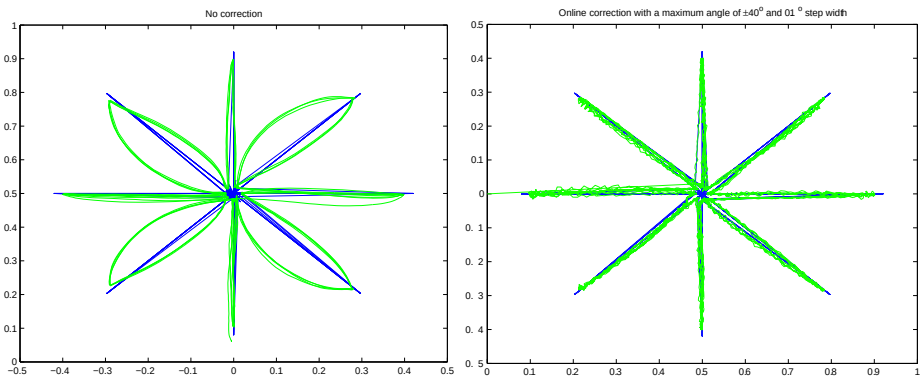

Fig. 5. The path of a robot following a star-shaped path, with no correction of the commands (left) and with online correction (right). The blue line shows the optimal path and the green line shows the actual motion.

Fig. 5 shows a test with a real robot. A standard robot drives in a star-shaped path without correction of the motion commands (on the left side of the figure) and with online correction (on the right). As can be clearly seen, the actual driving behavior of the robot differs substantially from the actual driving path. By applying the inverse prediction system online a more accurate path is achieved. The small deviations in the horizontal and vertical axes have been nearly eliminated and the deviations in the diagonal have been strongly reduced.

4 Damaged robots and improved cost function

The same techniques applied in the previous section to robots which are not accurate enough can now be applied to solve the problem of a damaged motor.

As a first example, we can make the driving behavior of a robot substantially worse just rotating its cover. This is an "artificial damage" because the robot cannot drive straight and oscillates. Fig. 6 shows the case of a robot with a cover which has been rotated 30°. The left side of the figure shows the driving behavior of the robot without correction. The right graph shows the path with online correction.

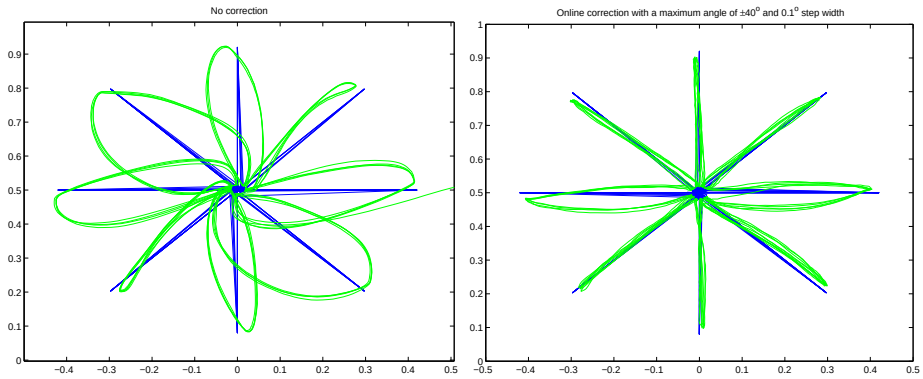

Fig. 6. The path of a robot following a star-shaped path, when the cover has been rotated 30°. Left, with no correction of the commands, right with online correction.

4.1 Motor damage

Our robots have four omnidirectional wheels, when one motor is damaged, the robot has enough redundancy to still drive omnidirectionally but the PID controller on the robot tries to control four motors. We could of course have different PID controllers in the robot, and when a motor fails, we could switch from a four wheel to a three wheel controller. However, if the motor just partially fails (it starts to deliver less power, if for example the motor has become very hot) it would be desirable to have a way of adapting the commands sent by the high-level control. Also, the robot electronics could be a black-box which we do not want or cannot modify.

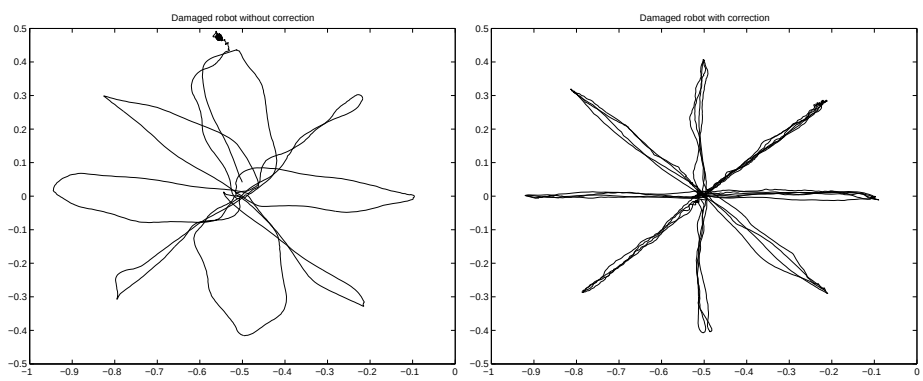

Fig. 7. Movement of a robot with a damaged motor, when moving along a star-shaped path. Left, without correction; right, with the correction of a specially trained predictor.

In our experiments, we took a robot with four motors and disconnected one of them. The vision system tracks the robot for some time and learns to predict its response function to commands, as discussed above. We then apply the online correction to the damaged robot with great success. As can be seen in Fig. 7 the driving behavior of the damaged robot is similar to that of a fault-free robot. The robot is somewhat slower, but it can drive accurately again.

As this simple experiment shows, it is then feasible to make these types of corrections during RoboCup games. If a motor completely fails, or loses power, the high-level control can let the robot drive for some time, relearn its driving behavior, and apply the online correction. The result is a robot that heals after a few seconds because the coach (the central computer) knows which commands to send.

4.2 Improved cost function

The angle between the desired direction and the predicted direction of the tested robot command can be used as a cost function for analyzing commands. The command with the smallest cost is the best command. The example in Fig. 8 shows, however, that this is not always the case. If the robot is about to change abruptly its direction, the best command should allow this direction change as fast as possible. As can be seen in Fig. 8 (left), the smallest angle is produced by a robot command which drives the robot far away from its desired direction. We need to modify the original approach

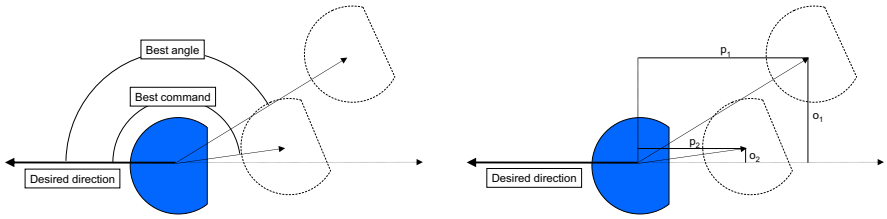

Fig. 8. An example of a drastic change of direction. In the left diagram, we want the robot to reverse direction by 180 degrees. The "best" angle is calculated by finding the smallest angle between the desired and all tested predicted movements. The smallest angle is not the best choice. The larger angle is better, because the command slows down the robot faster, although the robot still moves forward due to its inertia. In the second method on the right side, the predicted direction is decomposed into a parallel p and an orthogonal o component relative to the desired direction.

We can improve the cost function to handle the case mentioned above. A robust method for calculating a convenient cost function and thus finding the best command is shown on the right side of Fig. 8. The predicted direction is decomposed into two parts with respect to the desired direction: a parallel component p, and an orthogonal component o. The best command can then be found calculating the maximum of the sum

$$\text{argmax}_{v_i \in V}(p_i - \lambda|o_i|) \tag{1}$$

where V is the set of all predicted directions v_i. We tested several values for the constant λ. The results presented here were computed with $\lambda = 2$. Tests with $\lambda = 3$, $\lambda = 4$, and $\lambda = 5$ produced similar results. The set of tested robot commands contains all commands between $d - 40^o$ and $d + 40^o$, where d is the desired direction, and with a step size of 0.1^o. The improvement of this decomposition method compared to the previous straightforward comparison of directions is shown in Fig. 9.

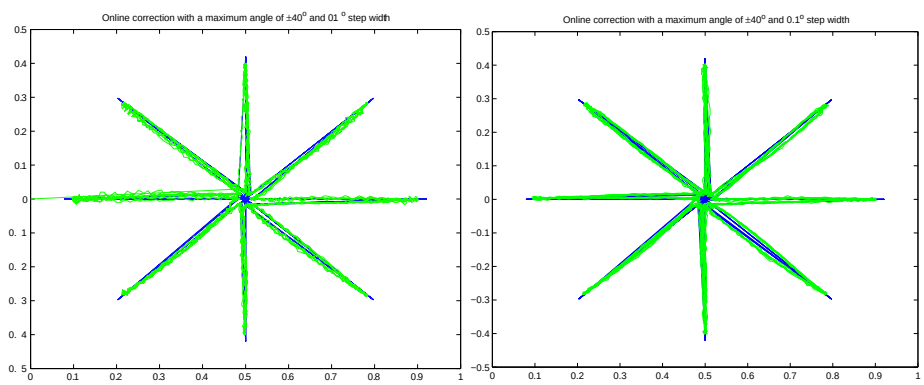

Fig. 9. A robot driving in a star-shaped path, using the angle based cost function for the correction (left), and using the decomposition based cost function (right). The robot drives more smoothly with the second approach.

5 Computational cost

The computational cost of the online correction method described in this paper is low. Since we use neural networks or linear predictors, and since the history of the robot remains constant while we test different commands, we only need to recompute those parts of the network or linear model where the input has changed. Fig. 10 gives an idea of the computations involved. The dotted lines represent variable inputs. Only the command (b_x, b_y, r) (x velocity, y velocity, rotational velocity) is modified, the rest of the inputs remain constant. It is thus

possible to keep track of the changes, and the computational cost for a new prediction is greatly reduced.

This is specially true for a linear model, which is a special kind of neural network with summation as activation function. In a linear model, a few additional multiplications and additions allow us to test a command to obtain the predicted movement of the robot.

One possibility we are examining, is training another network to learn the inverse of the prediction network. This is more difficult than it seems, because the commands are constrained by the mechanical characteristics of the robots. It is easier to test feasible commands, as we do, than invert the function and preserve all mechanical constraints, so that only feasible commands are produced by the inverted network.

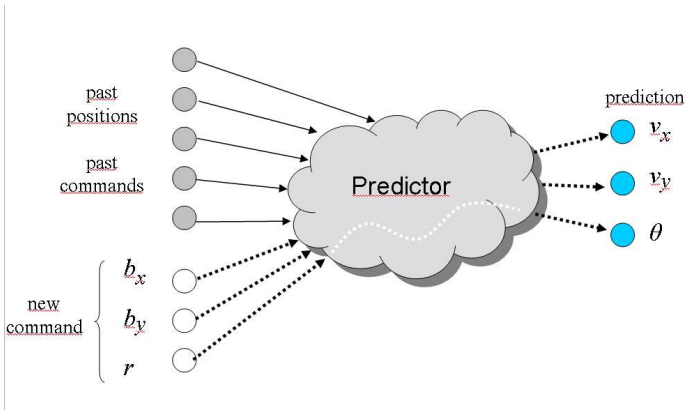

Fig. 10. The neural net used for the prediction and the varying inputs.

6 Conclusion and Outlook

In this paper we discussed how to find the correct command to be sent to a robot which is moving inaccurately due to (a) mechanical imperfections, (b) a rotated cover, (c) a malfunctioning motor.

Our robots are mechanically rather inaccurate when they start to move. Once a robot is moving forward, for example, it is easy to keep the direction of motion. But it is difficult to start moving fast exactly in the desired direction due to many mechanical reasons. The online correction method described here has been incorporated in the control module of our robots which can now drive much more accurately, even when they start moving at great speed.

The experiments with malfunctioning robots were performed individually, not during a game. However, it would be easy to extend our control architecture

to handle malfunctioning robots while the game is in progress. In another paper we describe the work we are doing on an coach for the small-size league, which would have this as one of its duties.

One possible improvement to the methods presented here is to integrate in the computations any incorrect rotation of the robot caused by its motion. We would have to define a function assigning a weight to incorrect motion and another to incorrect orientation. It is easier to drive straight when the orientation is not important and vice versa. The high-level control must then decide, according to the play situation, what is more important, accurate linear movement or accurate orientation.

Our work is a further step towards fully adaptive autonomous robots, which learn to adapt to new conditions even in the case of mechanical failure. Two, or even three damaged motors (out of four) can be compensated, but then whole sequences of commands, not single commands, must be learned. We will report on that work when it is finished.

References

1. Behnke, S. and Rojas, R., "A Hierarchy of Reactive Behaviors handles Complexity", *Balancing Reactivity and Social Deliberation in Multi-Agent Systems*, Workshop at ECAI 2000, 14th European Conference on Artificial Intelligence, Berlin, 2000.
2. Behnke, S., Egorova, A., Gloye, A., Rojas, R. and Simon, M., "Predicting away the Delay,", in *RoboCup-2003: Robot Soccer World Cup VII*, Springer-Verlag, 2004.
3. Egorova, A., Gloye, A., Göktekin, C., Liers, A., Luft, M., Rojas, R., Simon, M., Tenchio, O., Wiesel, F., "FU Fighters Small Size Team 2004", *RoboCup-2004: Robot Soccer World Cup VIII*, Springer-Verlag, 2005.
4. Egorova, A., Simon, M., Wiesel, F., Gloye, A., and Rojas, R., "Plug & Play: Automatic Geometry and Color Calibration for Cameras Tracking Robots", *RoboCup-2004: Robot Soccer World Cup VIII*, Springer-Verlag, 2005.
5. Fitch, R. Butler, Z. Rus, D. . "3D rectilinear motion planning with minimum bend paths", *Intelligent Robots and Systems 2001*, Proceedings, pp. 1491–1498.
6. Gloye, A., Simon, M., Egorova, A., Wiesel, F., Tenchio, O., Schreiber, M., Behnke, S., and Rojas, R., "Predicting away Robot Control Latency," Technical Report B-08-03, FU-Berlin, Juni 2003.
7. Gloye, A., Göktekin, C., Egorova, A., Tenchio, O., and Rojas, R., "Learning to Drive and Simulate Autonomous Mobile Robots", *RoboCup-2004: Robot Soccer World Cup VIII*, Springer-Verlag, 2005.
8. Jackson, A.H. Canham, R. Tyrrell, A.M., "Robot fault-tolerance using an embryonic array",: Evolvable Hardware, 2003, Proceedings, pp. 91–100.
9. Kleiner, A., Dietl, M., Nebel, B., "Towards a Life-Long Learning Soccer Agent", *Proc. Int. RoboCup Symposium 2002*, Fukuoka, Japan, 2002.
10. Praehofer, I. Jahn, G. Jacak, W. Haider, G., "Supervising manufacturing system operation by DEVS-based intelligent control", Fifth Annual Conference on Distributed Interactive Simulation Environments, 1994, pp. 221–226.
11. Simon, M., Behnke, S., and Rojas, R., "Robust Real Time Color Tracking,", *Proceedings of: The Fourth International Workshop on RoboCup*, pp. 62-71, Melbourne, Australia, 2000.