TUMsBendingUnits from TU Munich: RoboCup 2012 Logistics League Champion

Sören Jentzsch, Sebastian Riedel,
Sebastian Denz, and Sebastian Brunner

Robotics and Embedded Systems, Department of Informatics,
Technische Universität München, Munich, Germany
{jentzsch,riedels,denz,brunnese}@in.tum.de
http://www6.in.tum.de/

Abstract. The new RoboCup Logistics League sponsored by Festo offers a competition within a simulated industrial environment. In order to solve the logistical tasks, all three Robotinos not only have to operate autonomously in a flexible, effective and robust way on their own, they should also collaborate efficiently in order to maximize the overall outcome. In this paper, the first world champion of the Logistics League, TUMsBendingUnits from the Technical University of Munich (TUM), presents their logistical system with focus on approaches concerning robot hardware modifications, software architecture, task planning and execution, multi-robot collaboration, visual perception, motion planning and execution.

1 Introduction

The RoboCup Logistics League sponsored by Festo is a new industrially inspired competition. After two years of demonstration the Logistics League has been added to the approved RoboCup portfolio in 2012 [6], offering a competition with focus on achieving a flexible and efficient material and information flow inside a factory area.

A team consists of three robots based on the robot platform Robotino from Festo [3] which have to operate autonomously without any kind of external computing power, control station or human interference [5]. Each team competes for the highest score in 15 minutes on its own separate competition area with a size of 5.6m × 5.6m, as shown in Figure 1. The main task is to continuously execute a 3-staged production cycle and to transport the final product to the currently active delivery gate. Whereas the machine positions are known in advance, their types are not and have to be explored by the robots. Each product is represented by a data carrying RFID tag mounted on a red hockey puck and each machine consists of an RFID read/write device and a signal unit in order to show the robot the actual status of this very machine. Alongside this main task more logistical challenges have to be solved, for example dealing with out-of-order machines, handling express goods within a certain timeframe, reacting

X. Chen et al. (Eds.): RoboCup 2012, LNAI 7500, pp. 48–58, 2013.

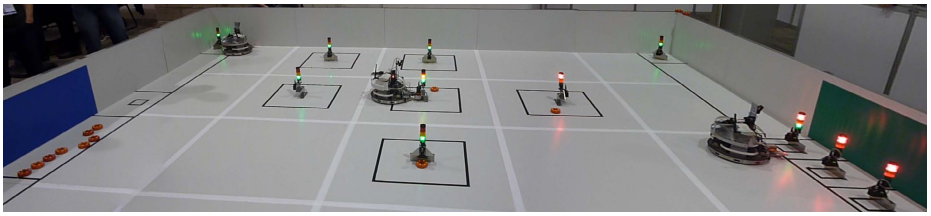

Fig. 1. Factory area of the Logistics League including the raw material zone (left), the production machines (middle), the delivery gates (right) and three autonomously operating Robotinos of the team TUMsBendingUnits

to changing delivery gates and recycling consumed goods in order to obtain new raw materials.

In this paper, the winning team TUMsBendingUnits of the RoboCup Logistics League 2012 championship in Mexico City presents their hard- and software approaches which did master these logistical tasks.

2 TUMsBendingUnits: System Overview

Following the championship of the demonstrational Festo Logistics Competition at RoboCup 2011 in Istanbul, team TUMsBendingUnits of the Technical University of Munich (TUM) could successfully defend their title at the RoboCup Logistics League 2012. After convincingly winning all games at the semi-finals group stage, TUMsBendingUnits won the finals against team Leuphana with a new score record of 160:49.

This section describes the hard- and software of TUMsBendingUnits, which includes the robot sensor equipment, software architecture, task planning and execution, logistical collaboration between the robots, visual perception, as well as motion planning and execution.

2.1 Robot Hardware

The Logistics League is based on the robot platform Robotino, designed and manufactured by Festo [3]. Robotino is actuated by a three-wheeled holonomic drive system. It provides an embedded PC/104 (AMD LX800 processor with 500 MHz) combined with an I/O control board and a LPC2377 32-bit microcontroller to access actuators and sensors. Alongside basic equipment like a WLAN module and the passive puck-pushing device, the teams are free to change or add any kind of sensors. For our Robotinos, we use the default front IR sensor for monitoring the puck possession, two pre-calibrated binary optical sensors to detect and align with black lines on the ground, and two more optical sensors mounted along the customized pushing device in order to align with production machines. A Logitech Webcam C905 is used for visual perception and wheel odometry is assisted by a CruizCore XG1000 gyroscope. Figure 2 (left) shows our final version of the Robotinos.

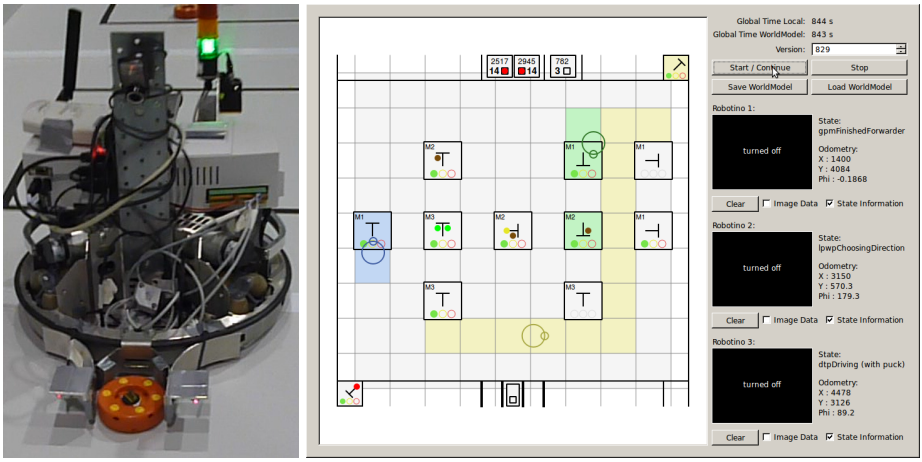

Fig. 2. Robotino of TUMsBendingUnits (left) and their graphical visualization, debugging and controlling tool (right) for the competition

2.2 Software Architecture

The overall software architecture of the Robotino consists of several layers which are shown in Figure 3. In order to control the wheels and to manage analog and digital inputs and outputs, the realtime-based control daemon of the PC/104's Ubuntu uses serial communication to access the I/O board. On top of this architecture we have our application (TBU_ACAPS), programmed in C++ using several external libraries and an internal one (TBU_BASE). Furthermore, we established direct access to the control daemon's shared memory segment, thus effectively bypassing the Robotino-API ComServer and boosting the sensor- and actuator-based communication frequency significantly.

Now let us take a closer look at the software architecture of our application, as visualized in Figure 4.

Starting with the sensorial part, the sensor server has direct access to the shared memory with all the raw sensor and odometry values while retrieving visual perception data from the camera module depending on the currently desired detection mode. Except for the sensor server, each module consists of or operates within own threads, thus resulting in a highly multithreaded architecture. The sensor event generator module constantly polls the sensor values through the server and converts the raw values to discrete events like EvCameraGreenLightDetected or EvHasPuck, depending on certain changes in the data, triggers or thresholds.

The heart of our software architecture is the state machine, which asynchronously processes events from the event queue and executes basic, generalized routines, like grabbing a puck, driving to a certain machine or delivering a puck. These behaviors are realized as own sub-state machines and invoked by the job handler as the robot proceeds in the execution of its current job. Once a job is completed, the job planner decides for a new job, taking the current state of the plant as well as the robot's teammates into account. This information is

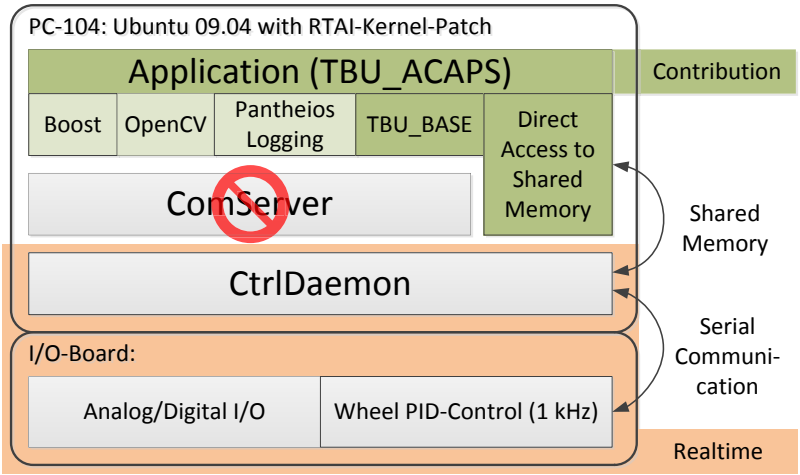

Fig. 3. Overall Robotino software layers with our software code on top of the realtime components of the standardized Robotino platform

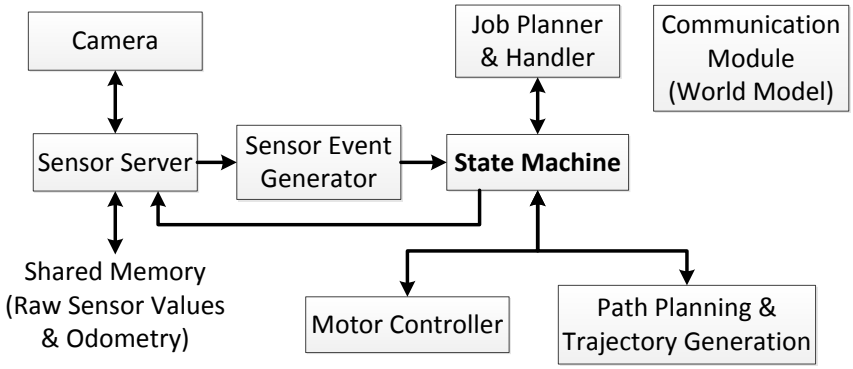

Fig. 4. Abstract block diagram of the software architecture, where each block represents a separate software module with the arrows indicating the main information flow and function calls, respectively

subsumed in a world model structure, which is shared and synchronized between the robots via the communication module.

The state machine itself has access to other modules, for example in order to control the camera via the sensor server, ask for the next job, execute a motion in the motion controller module or request the path planning module to plan a certain path. With completion of a task, these modules again utilize event-based communicatation with the state machine.

Finally, a graphical user interface running on an external computer was developed for debugging and basic interaction purposes, like starting or pausing the robots (Figure 2, right). It visualizes the current world model and additional information about each robot, like the current state-machine state, the camera image and odometry values.

2.3 Task Planning and Execution

The Logistics League does not allow teams to utilize an external central controlling system and thus each robot has to be able to plan its next task on its own. This chapter introduces our solution to decentralized planning and execution in a multi-robot team.

Job Planning. The job planning module of each robot takes the current state of the world model and the last executed job as input for determining the next job. We distinguish two different types of jobs: the DeliverPuckJob and the CheckInsertionAreaJob.

The DeliverPuckJob is the basic job type that covers all of the actions which are required to solve the basic production cycle. It always takes a start node, such as the input zone or a machine, a certain puck at that node, and a destination to which the good has to be delivered to.

In order to meet the express good challenge within the official time span of 120 seconds, the CheckInsertionAreaJob periodically checks for an express good within the insertion area. Note that, as all of our robots run the same software code, we avoid delegating a robot only for the express good challenge, thus focusing on the main production cycle.

The job planner creates a list of all possible jobs and then chooses the next task based on the highest priority value for each job. These priority values are assigned for each job as follows:

$$\begin{aligned} priority(dJob) = available(dJob) \cdot (\, & basicPriority(dJob) \\ & - distPosStart(dJob) - distStartTarget(dJob) \,) \end{aligned} \quad (1)$$

$$priority(cJob) = available(cJob) \cdot (timeElapsed() - distPosIA()) \quad (2)$$

Equation 1 calculates the priority for the DeliverPuckJob (*dJob*) as follows: At first, the function *available* checks for availability of the job and returns 1 only if the following conditions are met (otherwise 0): the picked up good is available at the starting machine, the target machine has the appropriate machine type, requires this good, has the status of being ready, has no blocking good underneath and both the starting machine and the target machine are not occupied by another robot. Then, we assign a certain basic priority to each job type (*basicPriority*). For example, the basic priority for delivering the final product to the delivery gate is much higher than exploring an unknown machine with a raw material. These basic priorities were tuned manually in order to prioritize certain job types and to boost the overall production of a final product

effectively. However, we also consider the distance which the robot has to drive from its current position to the starting machine (*distPosStart*) and from the starting machine to the target machine (*distStartTarget*), including a penalty if the robot's last job did not end on this very starting machine. Thus, in the end, also jobs with a low basic priority but sufficiently short travel distances can be prioritized over jobs with a higher basic priority.

Equation 2 outlines how the priority for the CheckInsertionAreaJob *cJob* is calculated. First of all, the job is only available if at least 25 seconds elapsed since the last check, we already identified at least one suitable target machine, the maximum number of express goods were not reached yet and the express good insertion area is not occupied by another robot (*available*). Then the priority value depends linearly on the time elapsed since the last check (*timeElapsed*) minus a penalty depending on the distance between the robot's current position and the insertion area of the express good (*distPosIA*).

In the end, the job planning module selects the job with the highest priority value and passes it to the job handling module. If there is no job currently available, the robot moves to a random neighboring grid node unblocking the current node in order to avoid deadlock situations, and starts over again after a certain amount of time.

Job Handling. The job handler takes over the selected job and sequentially triggers the execution of appropriate sub-state machines. At first the robot needs to get to the starting machine. This may or may not include leaving the current machine (the current machine could also be the starting machine). If so, the job handler calls the corresponding sub-state machine and passes over all relevant job information, for example the exact starting and target machine. This allows the sub-state machine to leave the current machine in the most suitable and efficient direction. In similar manner sub-state machines are triggered for: actual driving to the starting machine, picking up a puck, leaving the starting machine, driving to the target machine and placing the puck underneath the RFID-device. After completion of each sub-state machine, the world model is updated accordingly and synchronized to the other robots. In case of the target machine showing red light and thus signaling the out-of-order state, the job handler triggers the job planning module in order to find a new target for the carried good. Finally, after successfully executing the job, the next one is planned.

Logistical Collaboration. The logistical collaboration basically consists of breaking the core production cycle of producing and delivering the final product into atomic operations. These are exactly the DeliverPuckJobs, where we deliver a puck from one machine to another, as there is no efficient way to hand over a good from one robot to another. By operating on a synchronized world model, the robots then automatically collaborate and avoid collisions.

Updating the world model has to be reliable. Every time a robot changes the state of world model objects, such as grid nodes and machines and all their properties, it synchronizes changes with the other robots. To prevent simultaneous write operations, every robot is only allowed to make changes when it holds

active TCP connections to all robots (including itself). As every robot only allows for one incoming TCP connection and connections are established in a fixed order, this effectively prevents conflicting write operations. A world model version number further prevents from overriding a newer with an older world model in case of rebooting a robot or network failures.

2.4 Visual Perception

In order to safely and precisely grab a puck or navigate to a certain key location (e.g. to a production machine or a delivery gate), the robot's motion must be based on sensor data, instead of relying only on the internal odometry values. We utilize visual perception with our color camera to detect multiple pucks (Figure 5), to perceive the light states of the production machines, and to locate the currently active delivery gate from distance (Figure 6). For each task, we dynamically switch certain camera parameters (e.g. brightness, contrast, saturation, white balance temperature, gain, exposure, no backlight compensation) in order to maximize the visual perception of task-specific key differences in the image. For example, for the light state detection, we heavily increased the exposure value and set the contrast to zero, which results in images shown in Figure 6, where only the lights turned on are perceived with their corresponding color values, as opposed to lights visualized in Figure 5 (right), where basically each light turned on consists of a white core.

Given these color images and the fact that memory and computation power of our robot is very limited, we apply a sliding window technique on color-thresholded HSV images in order to detect pucks and lights. Due to the perspective distortion of the camera, our window size for detecting pucks depends on the current height in the image. To efficiently calculate the matching pixel sum for each window (up to 19,200 windows for the whole 160x120 image), we construct the integral image (or summed area table) for our binary image in advance, as detailed in [7]. For the light detection this entire procedure is done separately for every light color.

In case of detecting and navigating to pucks and the currently active delivery gate, we have to transform pixel coordinates (center points of the matching windows) to metric units in the robot's local coordinate frame. For this task we collected a sufficiently large amount of training data and used the machine learning tool Eureqa [2] to identify an appropriate conversion function.

In order to ensure a robust light state detection including the occurrence of yellow flashing light, we utilize a light state buffer as follows: We process our images as usual, but only after at least a certain amount of time elapsed and a certain amount of images were processed, we make our final decision based on the recognized light states so far. Then, we clear the buffer and start over again.

2.5 Motion Planning and Execution

The regularly oriented and positioned machines allow for significant reduction of the motion planning problem. Paths between points of interest (machines,

Fig. 5. Puck detection at the raw material zone (left) with the corresponding binary image (middle) and near the production machines (right)

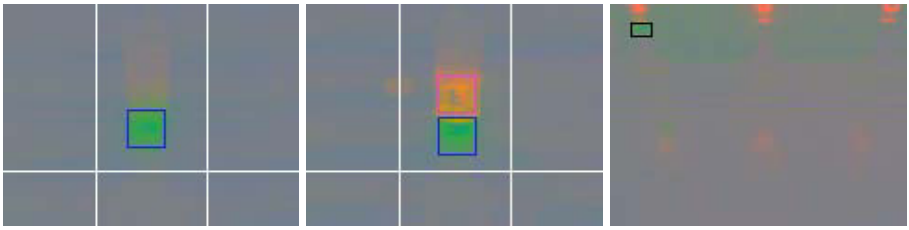

Fig. 6. Light state detection of production machines using modified camera parameters showing green (left), yellow-green (middle), and detecting the currently active delivery gate, that is the one with the green light turned on (right)

raw material zone, delivery gates, etc.) are planned on a coarse 9×9 grid, as visualized in Figure 2 on the right. Each (x, y) grid node expands into four oriented nodes $(x, y, 0°/\pm 90°/180°)$ therefore leading to a shortest path problem in a graph of only 324 nodes. Allowing only four orientations and 81 positions on a rectangular grid, we restrict the motion to right-angled pathways through the factory area. The established coarse "motion corridors" have a slightly larger size than Robotino's diameter. For a smooth trajectory generation along the calculated shortest path, the path is reduced to a list of n via poses $V \in (x, y, \phi)^n$.

Based on the via poses and constraints for maximum velocity and acceleration, a trajectory is calculated following the Linear Segments Parabolic Blends method (LSPB) detailed in [1]. Our trajectories are planned in the world frame, treating x, y and ϕ as independent, but time-synchronized degrees of freedom (Robotino has a holonomic drive system). Generally the LSPB approach is based on linear interpolation between two via points (the "linear segments"). To avoid discontinuous velocity trajectories, parabolic blends are added at each via point. In other words, the robot is allowed to (de)accelerate only near via points, resulting in trapezoidal, continuous velocity trajectories. Finding a trajectory via the LSPB method eventually means to find the minimum linear segment time for each of the $(n-1)$ path segments and the minimum acceleration time for each via point $v \in V$ under the given dynamic constraints. As you cannot solve for these time intervals in a closed form, Craig suggests a heuristic [1]. We decided for an iterative approach: Starting with the first path segment and zero segment time, we increase the segment time until all constraints are met and the

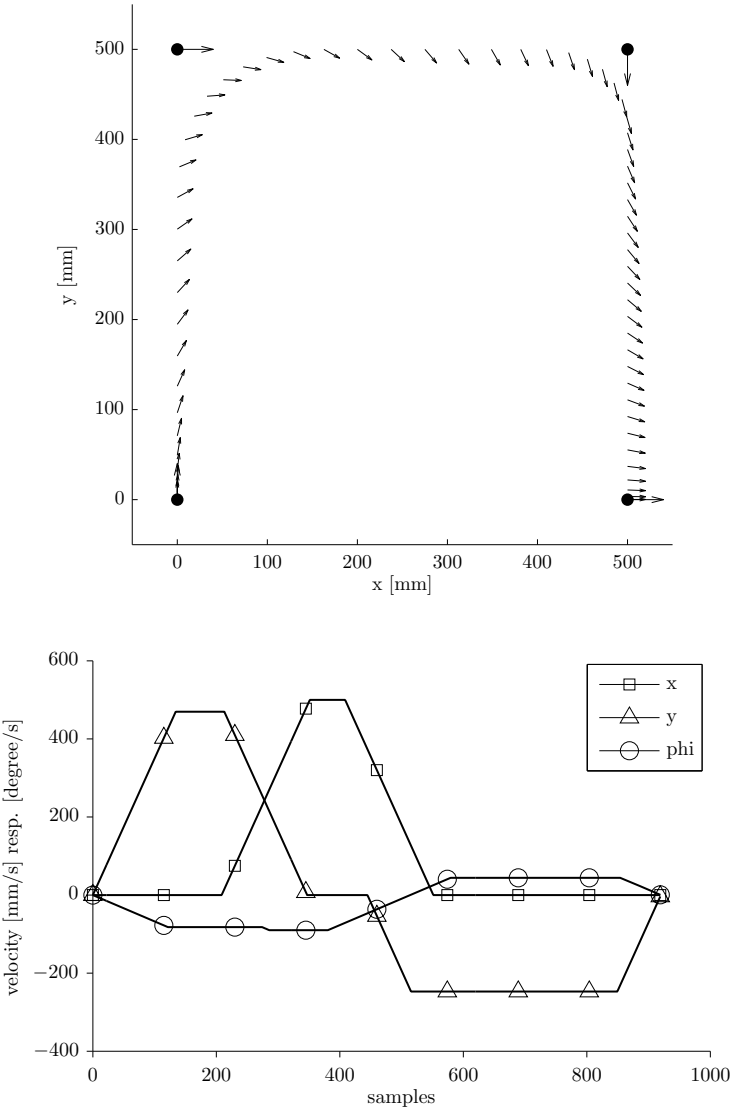

Fig. 7. Generated trajectory sampled at 200 Hz with two intermediate via poses in addition to start and end pose. The quiver plot on top visualizes position and orientation of the sampled trajectory (small arrows at each fifteenth pose), the original via poses are marked by filled dots. Notice how only the start and end pose are actually reached, the intermediate via poses are only approximated. This plays nicely with our coarse grid motion planning. The resulting movement is very smooth, yet still preserves straight line motion on the corridors. The line plot at the bottom represents the corresponding velocity trajectories.

necessary acceleration times are less than half of the segment time. We then move on to the next segment to start with zero segment time again. After defining all time intervals, the trajectory is calculated by sampling it into a look-up table with a sample frequency equal to our desired control loop frequency (an example can be seen in Figure 7).

Trajectory control is then achieved by a position PD-controller targeted at a control loop frequency of 200 Hz. The direct communication with Robotino's real-time subsystem via shared memory allowed us to achieve these short control cycles on Robotino, as discussed in Chapter 2.2 (Figure 3).

3 Conclusion

This paper introduced the RoboCup Logistics League and the 2012 champion TUMsBendingUnits[1].

As the new Logistics League crowned the first world champion this year, our main focus was on establishing a solid and stable overall logistical system. Avoiding cost-intensive sensor equipment like laser scanners, we developed advanced software routines within our event-based state machine, including for example odometry calibration at certain locations or robust steering towards the machines. The overall software architecture allowed us to decouple modules quite easily while the external graphical interface helped us to debug our approaches. Whereas the visual perception algorithms were kept simple, the more sophisticated motion execution, especially the trajectory following, ensured efficient path following, resulting in significant time savings when executing a job. Overall our system allowed all three robots to keep up a stable and coordinated material flow while dealing with the logistical challenges of the competition, including express goods, out-of-order machines, changing delivery gates, and recycling of consumed goods. During the final game no human intervention was necessary and all robots operated autonomously for the whole 15 minutes.

Compared to the other teams of the Logistics League, the communication and collaboration between the robots and the motion routines set us apart the most. Moreover, our software architecture allowed us to easily decouple, debug and visualize our software components, which is of great value in order to achieve the most stable and robust solution. For the next years, we plan to further boost the dynamic and flexible behaviors concerning job handling, collaboration and time-based path planning. Right now, tasks cannot be interrupted and the whole path is reserved when driving to a certain grid location.

With increasing knowledge and performance capabilities of the Logistics League teams, the competition itself will be further improved and refined in the future. Following the roadmap of the Logistics League [4], alongside the core

[1] More information about the team TUMsBendingUnits and video highlights from the competition can be found on the following websites:
`http://www.tumsbendingunits.de/` (under construction)
`http://www.youtube.com/TUMsBendingUnits`

production cycle and the express good challenge, various new production strategies will be introduced, for example Just-in-Time (JIT) and Just-in-Sequence (JIS). Moreover, future dynamic adversarial obstacles will require a much more sophisticated, reactive and flexible behavior of the robots and their collaboration, especially concerning the path planning. These future changes will keep the Logistics League challenging and contribute towards the goal of approaching an industrial application.

Acknowledgments. The authors would like to thank the additional contributing member Peter Gschirr of TUMsBendingUnits, Andre Gaschler from the fortiss GmbH, Dr.-Ing. Gerhard Schrott and Prof. Dr.-Ing. Alois Knoll of the Chair Robotics and Embedded Systems, Department of Informatics, Technische Universität München, for their huge support and commitment.

References

1. Craig, J.J.: Introduction to Robotics: Mechanics and Control, 3rd edn. Prentice Hall (2004)
2. Eureqa, Cornell Creative Machines Lab, `http://creativemachines.cornell.edu/eureqa`
3. Festo Didactic, Education and Research Robots: Robotino, `http://www.festo-didactic.com/int-en/learning-systems/education-and-research-robots-robotino/`
4. Logistics League Roadmap, `http://www.robocup2012.org/pdf/LL_2012_Roadmap.pdf`
5. Logistics League Rulebook, `http://wiki.openrobotino.org/index.php?title=Logistics_League`
6. RoboCup (2012), Sponsored Leagues: Logistics League by FESTO, `http://www.robocup2012.org/comp_SponsoredFesto.php`
7. Richard, S.: Computer Vision: Algorithms and Applications. Springer, London (2010)