# *SoccerBot*: Software architecture of a camera-vision equipped robot featuring real-time object recognition.

Georg Klima*, Krystian Szczurek*, Peter Wild*
*Department of Computer Science
University of Salzburg, A-5020 Salzburg, Austria
Email: {gklima, szczurek, pwild}@cosy.sbg.ac.at

***Abstract*— *SoccerBot* is a semi-autonomous robot based on a Lego® Mindstorms$^{TM}$ (RCX) chassis and a web camera as the main sensory system for navigation. Motivated by soccer-playing robots we construct a robot able to recognize and catch a ball distinguishable from the environment by its color and shape. For computation intensive real-time image processing and object recognition, an external computer (notebook/PC) is used, which then transmits its navigation commands via an IR link to the RCX. Control software on notebook/PC is written in Java using Giotto for exact specification of timing behavior.**

**After an architectural overview, we visualize the PC-RCX communication layer and introduce used embedded controller software on RCX. Subsequently, we present the used algorithms for object recognition, i.e. a Split-and-Merge algorithm using a homogenity criteria comparing colors in HSV color space for image segmentation and a best-fit search using CIE lab colors including a shape-test and color-test for object classification. A controller implemented as a finite state automaton for navigation is illustrated and finally, we give an outlook on future developments.**

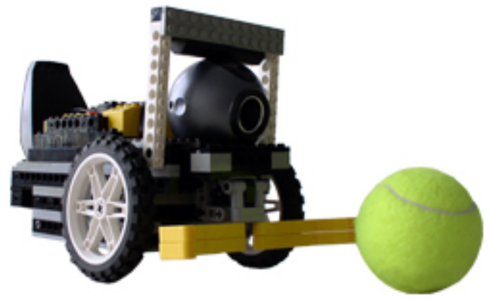
Fig. 1. *SoccerBot*: 2-independent-wheel-drive Lego® Mindstorms$^{TM}$ robot mounted with Logitech® QuickCam® Pro 5000.

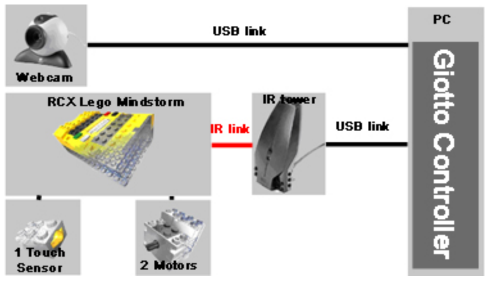

Fig. 2. Original concept consisting of two subsystems (RCX and PC).

## I. Introduction

The construction of autonomous robots being able to play soccer has been interdisciplinary research work for a couple of years in artificial intelligence, image processing, electronics and mechanical engineering. Competitions like RoboCup (`http://www.robocup.org`) exist, to promote development and constitute a platform of challenge for working groups. One of the most important tasks facing these groups is the robust and real-time capable localization of the ball within the playing field.

### A. Problem definition

Typically all objects in a soccer game, i.e. ball, players and goal have different colors and thus seem to be easily distinguishable. However, their unambiguous localization is yet an insufficiently solved problem, as [1] points out, since most approaches require long calibration phases. The goal for our project was to provide an adequate design and implementation solution to equip a robot with software able to locate and catch such a ball within a playing field. While some AI approaches focus on intelligent controllers and use simple object recognition techniques, we provide accurate object classification under variant lighting conditions. Thus, the controller can be implemented as a simple state automaton. Second, there is often less focus on software architecture and the selection of adequate embedded software approaches. We decided to use Giotto for an exact specification of timing behavior.

Timing constraints are set a priori according to predefined requirements. These **real-time requirements** are empirically determinable as follows:

- a *reaction-time* of less than 300 ms, i.e. not more that 300 ms pass from the detection of the ball by the camera sensor until motor actuators react.
- an *image processing rate* of at least 10 fps, as suggested by [1].

Besides, to obtain reliable object recognition the following **quality requirements** exist:

- an *accuracy* of at most 1 false classification in 100 processed frames.
- *robustness* against changes in lighting.

Unfortunately, despite a fast implementation we are not yet

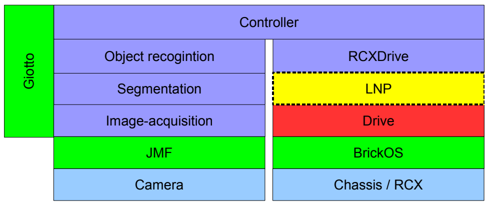

Fig. 3. Two-stacked layered architecture with separated timing code (Giotto).

capable of executing the object recognition task within the processing environment of Lego® Mindstorms$^{TM}$ . For this reason we decided to swap out computation to an external computer and use an architecture illustrated in Figure 2. However, as hardware is becoming smaller and faster, we believe this external computing power could be replaced by an on-board sub-notebook or tablet PC. In the following, we report on software architecture and initial setup.

## B. Architectural overview

The software of *SoccerBot* (see Figure 1) is built based on a modular layer-oriented architecture to support future additions and a strict separation of different functionalities. On PC we completely rely on *Java* code, on RCX the embedded controller thread is written in *C*.

*1) Two-stacked layered architecture:* Figure 3 illustrates two stacks, one for the acquisition and processing of the camera image and another for PC-RCX communication. The camera is controlled via *Java Media Framework* (JMF). An *Image-acquisition* tool extracts a buffered $352 \times 288$ RGB image, which is then cropped to a $256 \times 256$ image and segmented into regions by a *Segmentation* process. These regions provide the base for object classification within the *Object Recognition* layer. On the bottom of the second stack BrickOS running on RCX hosts an embedded controller thread (*Drive*), which controls the motors and reads bumper sensor values. Communication with the according *RCXDrive* layer on the PC is done using *Lego Network Protocol* (LNP) messages. The central component within this architecture is the *Controller* which generates navigation commands.

*2) Separate timing code:* We do also clearly separate functionality from timing code. This is done using the high-level language Giotto for control software development. Giotto (see [2]) offers programming models for time-triggered applications introducing the *embedded software model* as a new programming paradigm and features exact specification of timing behavior, i.e. when sensors are read or actuators are written. Thus, functional code is clearly separated but linked to and supervised by the Giotto *timing program* compiled to E code. By defining real-time interactions of periodically executed *tasks* with the environment, we get the following benefits:

- Giotto semantics imply **fixed logical execution times** (FLET), i.e. sensors are read only at the beginning, actuators are written at the end of a task's period.
- **environment-determinedness**, i.e. value and time of actuator updates are predictable as a consequence of FLET. E code is also verified *time-safe* by the Giotto compiler.
- **composability**, since the compiler maintains compliance with logical semantics.

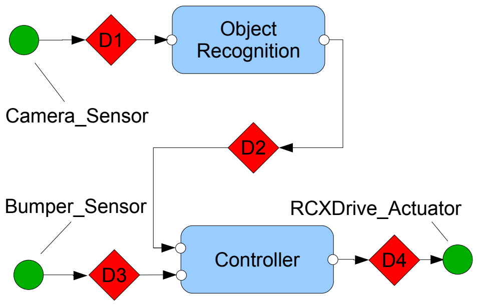

Fig. 4. Giotto model for pursuit mode *main*.

This, we believe, makes our system flexible and deterministic. Next, we present the Giotto timing program in detail.

## C. Giotto model

Consider *SoccerBot* in pursuit mode *main*, illustrated in Figure 4. Having access to both sensors, the *Camera_Sensor* and *Bumper_Sensor*, *SoccerBot* tries to catch the ball. There are two tasks, both given in Java code: the image processing task *Object Recognition*, and the navigation task *Controller*.

While the *Object Recognition* task processes camera images every 100 ms and provides recognized object information to the control task. The latter extracts navigation commands following simple control laws, and writes the result to the actuator *RCXDrive_Actuator*. This actuator is actually a stub, since navigation commands are transmitted through an IR link to the RCX. The control task is executed once within the period of 100 ms. Since it requires output of the *Object Recognition* task it operates on input that is already 100 ms old. Together with a communication delay of another estimated 100 ms, until the embedded controller receives the navigation command message, we fulfill the requirement of 300 ms reaction-time.

Additional modes are *init* and *deinit* for hardware initialization and shutdown, respectively. Here is the Giotto description of timing behavior:

```
mode init() period 500{
    exitfreq 1 do main(ExitInit_Driver);
    taskfreq 1 do Init();
}
mode main() period 100 {
    actfreq 1 do RCXDrive(RCXDrive_Driver);
    exitfreq 1 do deinit(ExitMain_Driver);
    taskfreq 1 do ObjectRecognition(
                      ObjectRecognition_Driver
                  );
    taskfreq 1 do Controller(Controller_Driver);
}
mode deinit() period 500{
    taskfreq 1 do DeInit();
}.
```

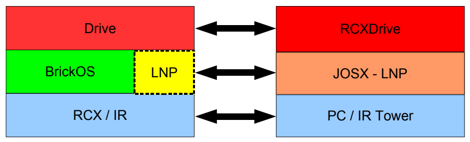

Fig. 5. Communication stack implemented on both PC and RCX.

## II. PC-RCX COMMUNICATION

*RCXDrive*, the communication layer for *SoccerBot* enables bi-directional communication between RCX and PC. It offers a set of communication primitives using the *josx* stack, which is provided as a part of *leJOS*, a Java based replacement firmware for the RCX micro-controller. Using a subset of the available operations, we provide the following functionality:

- **sending of Drive-commands**, which set the direction and speed for the actual movement of *SoccerBot* and
- **receiving Bumper-events**, which are released whenever *SoccerBot* has physical contact with the ball through its front-mounted fork.

The communication stack is implemented on both platforms, one for the RCX and the other for the PC. In order to keep out of compatibility issues we use LNP, supported by both, *BrickOS* and *josx* (see Figure 5).

*RCXDrive* is used to emulate sensors and actuators on the PC. Actually the motors are seen as actuators in terms of Giotto as well as the bumper, which is seen as a sensor.

Since *SoccerBot* is meant to be an embedded real-time system, the following requirements for communication exist:

- **reliability**: we used LNP integrity messages for communication, thus eliminating the problem of message loss or message integrity corruption due to bad connectivity.
- **dependable low latency**: we modified *josx* to achieve an actual communication delay of less than 100 ms.

Modifications of *josx* include:

- **refactoring the *josx* stack** using Java `DataInput` and `DataOutput` interfaces for the transmission of primitive data types: Originally *josx*' top most layer used Java streams to communicate messages, even if the used driver for the infrared tower supports packet assembling over the underlying serial infrared protocol.
- **extending packet size**: Originally the size of packages was limited to 2 bytes RX/1 byte TX. However, the more resource limited RCX can send and receive messages with a length of at least 240 bytes.

We emulate Lego® Mindstorms$^{TM}$ actuators and sensors on the PC. Hardware sensors are usually event-triggered. Thus, the connection between PC and RCX is event-triggered too. However, in Giotto, sensor emulating software components are polled in a timed fashion. Therefore mechanisms exist to queue events. This conversion from event-triggered to time-triggered events is illustrated in Figure 6.

### A. Communication at PC: RCXDrive layer

RCXDrive on the PC side communicates with RCX. Functions provided by the layer-interface are as follows:

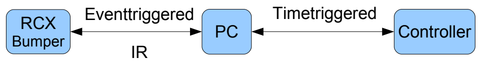

Fig. 6. Event-triggered / Time-triggered communication.

- `setPower(byte power)` sets the speed for driving straight or the strength of a curve or turn.
- `sendCommand(byte command)` sends a command to be executed by RCX.
- `getBumper()` returns a boolean with the state of the bumper since the last query.

Both the RCX and the PC use the following commands:

```
#define CMD_FORWARD        0
#define CMD_LEFT           1
#define CMD_RIGHT          2
#define CMD_TURN_LEFT      3
#define CMD_TURN_RIGHT     4
#define CMD_STOP           5
#define CMD_REVERSE        6
#define CMD_EXIT           7
```

### B. Communication at RCX: Drive layer

The *Drive* layer running on RCX handles communication using operating system routines of *BrickOS*. There is an embedded controller on the RCX to receive and execute commands, as well as to generate bumper-events, whenever *SoccerBot* hits the ball. Independent, parallel threads for motor control, communication and handling/generating bumper-events exist.

Plain straight driving is always a problem with the mechanical configuration we are using, because of the two independent wheels. This, however, is solved by the camera sensor, which supersedes all other inputs. As a further improvement we are planning to use rotational sensors for both wheels. This way, we can measure wheel rotations and correct the discrepancy between the left and right wheel.

## III. OBJECT RECOGNITION TASK

The object recognition task acquires image data from a video stream and performs segmentation and object recognition. It operates on the following ports:

- *Input port*: $352 \times 288$ 24-bit still camera image (`BufferedImage_port`).
- *Output port*: best-matching region classifying the ball, if the ball is visible (`Region_port`).

In order to provide the controller task with necessary information, two steps are carried out: First, a **segmentation** step dividing the image into regions using a *Split-and-Merge* algorithm with homogenity criteria in *HSV* color space. Second, the actual **object recognition** for the localization of the ball. This step is executed using closest color-distance to a threshold in *CIE Lab* color space, a *color-test* and an aspect ratio *shape-test*. While the first part is a rather time consuming step (estimated *worst case execution time* - WCET: 80 ms), object recognition can be carried out fast (estimated WCET: 10 ms).

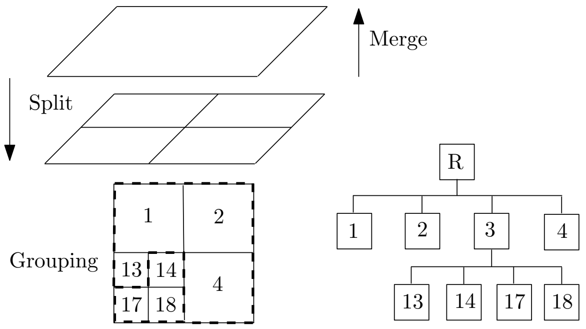

Fig. 7. *Split-and-Merge* algorithmic steps: Split, Merge and Grouping.

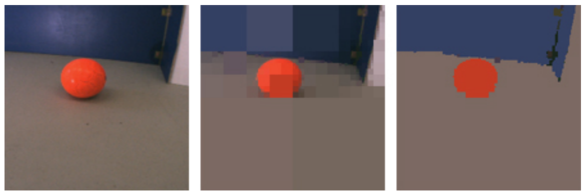

Fig. 8. (a) original $256 \times 256$ 24-bit picture, (b) result after Merge: 748 regions, (c) result after Grouping: 19 regions.

### A. Segmentation using Split-and-Merge

Split-and-Merge is a region-based segmentation algorithm by Horowitz and Pavlidis (see [3]), that divides a square image into regions and merges *homogeneous* regions recursively, using quadtrees. The intrinsic idea is to extract semantic segments, called *regions*, since pixels are too fine grained to carry object information. Regions in the image-domain often correspond to or have a link to objects in the real-world-domain. Each region is expected to be *homogeneous* with respect to a *homogenity criteria*. This criteria is mainly responsible for the quality of segmentation. Our slightly modified version of Split-and-Merge consists of 3 steps which are presented in Figure 7:

- **Split**: Initial segmentation (full subdivision);
- **Merge**: if 4 homogeneous sub-regions may be merged to a region fulfilling the *homogenity criteria* $H(r)$, they are merged;
- **Grouping**: if 2 neighboring regions (even at different levels) may be merged to a region fulfilling the *uniformity criteria* $U(r)$, they are merged.

Sometimes there is no clear distinction between homogenity and uniformity criteria, as many predicates may be used for both.

The result of this segmentation procedure is a list of regions, each containing the following typical information:

- **average color** to perform color-tests for object recognition,
- **central coordinate** for the location of a region within the image,
- **width and height** for aspect ratio shape-tests,
- **size** (given as the number of contained pixels) for disregard of small regions.

A typical example of the Split-and-Merge algorithm using a $256 \times 256$ image as input is illustrated in Figure 8.

#### 1) Homogenity criteria:

Many possible criteria exist for the homogenity of regions. We call a predicate $H(r)$ on the set of regions a *homogenity criteria*. If it evaluates to true, the region is called *homogeneous*.

Some popular examples are:

- *Min-Max-Difference*: a region $r$ is called homogeneous if the difference between maximum $max_r$ and minimum grey value $min_r$ within $r$ lies below a threshold.
$$H(r) = true \;\Leftrightarrow\; max_r - min_r \leq \; thresh$$
- *Color-Variance*: the sum of variances of red, green and blue image channels lies below a threshold.
$$H(r) = true \;\Leftrightarrow\; \sigma_r^2 + \sigma_g^2 + \sigma_b^2 \leq \; thresh$$

However, many problems arise, like the choice of suitable thresholds, noise within images and unsatisfying results for different conditions in lighting. Such predicates often create few large, but many small regions. The criteria used in our algorithm performs a check between all pairs of quadtree sub-regions whether the average colors in *HSV-space* of both regions are *similar* according to matrices by Volker Rehrmann [4]. Therefore another predicate $D$ is induced to check similarity between two colors in HSV space $c_1 = (h_1, s_1, v_1), c_2 = (h_2, s_2, v_2)$:

$$D(c_1, c_2) := \begin{cases} true, & for\ c_1\ similar\ to\ c_2 \\ 0 & otherwise. \end{cases}$$

$$D((h_1, s_1, v_1), (h_2, s_2, v_2)) = true \Leftrightarrow$$

$$|h_1 - h_2| < hue_t \wedge |s_1 - s_2| < sat_t \wedge |v_1 - v_2| < val_t$$

with

$$hue_t = hue\_tab(min(s_1, s_2), max(v_1, v_2)),$$

$$sat_t = sat\_tab(min(s_1, s_2), max(v_1, v_2)),$$

$$val_t = val\_tab(min(s_1, s_2), max(v_1, v_2)).$$

The homogenity criteria operating on quadtree regions is applied bottom-up starting at the maximum quadtree hierarchy level. Let $h$ be the function assigning to each quadtree region $r$ its hierarchy level $0 \leq h(r) \leq n$ and let $c$ be the function mapping regions to their average color values. Then the homogenity criteria is defined as follows:

$$H(r) = true \Leftrightarrow$$

$$\forall r_1, r_2 : (h(r) < h(r_1) = h(r_2) \leq n) \Rightarrow D(c(r_1), c(r_2)).$$

#### 2) Uniformity criteria:

As uniformity criteria we use a different predicate $U(r)$. Let $Q(r)$ denote a predicate indicating, whether a region $r$ is a maximum homogeneous quadtree region, i.e. $r$ is exactly represented by a quadtree node that is homogeneous and can not be merged. Furthermore, let $R_{min} \subset r$ be the biggest (and uppermost, leftmost if it is not unique) quadtree region within $r$. For example, in Figure 7, $R_{min}$ is region 1 in case of the bigger group, and region 14 for the smaller one.

$$U(r) = true \Leftrightarrow$$

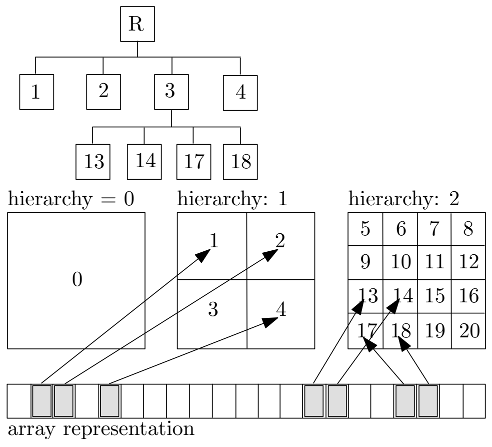

Fig. 9. Array representation of a quadtree.

$$r = \bigcup_i R_i,\ Q(R_i) \wedge D(c(R_i), c(R_{min})).$$

Experimental results [1], [5] have shown that *HSV-space* has better stability against changes in lighting. Additionally, the induced criteria can be calculated fast.

*3) Implementation:* To allow an efficient implementation, we introduce an enumeration scheme for quadtrees. Starting at the root node we assign each possible quadtree node a unique number. The root node is assigned 0, then we enumerate each hierarchy level from top left to bottom right as illustrated in Figure 9. Let $p$ denote the function assigning a linear position to each quadtree. If we identify each quadtree node by the triple $(h, x, y)$ where $h$ is the node's hierarchy level, $x$ is the row number and $y$ is the column number (both starting at $0$) then $p(h, x, y)$ is calculated as follows:

$$p(h, x, y) = \frac{1 - 4^h}{1 - 4} + x + y \cdot 2^h$$

Modelling quadtrees via arrays reduces overhead, as no dynamic memory allocation for tree nodes is needed. Besides, a useful property reducing calculations is the mapping of quadtree nodes with low hierarchy levels to small linear positions.

See [6] for a more detailed observation of color segmentation techniques including Split-and-Merge.

## B. Object recognition

For the localization of the ball, the task of the object recognition is to find the region with the closest *color-distance* to a *threshold* in *CIE Lab* color space, for which the following properties hold:

- its *shape-test* is positive: $0.5 \leq height/width \leq 2.5$,
- its *color-test* is positive: $D(c_{reg}, c_{thres}) = true$.

*CIE Lab* color space is useful, since Euclidian distances between *CIE Lab* colors are near the perceptual measure of color difference.

## IV. Controller task

The controller task is responsible for navigating *SoccerBot* according to information obtained by the object recognition task. Basically, it is a finite-state automaton (see Fig. 10) with the following states:

- `stop`: *SoccerBot* doesn't move,
- `searchLeft`: turns left at a spot,
- `searchRigh`: turns right at a spot,
- `goLeft`: drives forward doing a light left curve,
- `goRight`: driver forward doing a light right curve,
- `goForward`: drives straight forward.

The object recognition task returns a region representing the ball, if it has been found, else it returns `null`. When the ball is not found (`notFound`), then *SoccerBot* initially starts to turn either left or right, depending on the previous state. If on the other hand the ball is found, then the $x$-coordinate of the ball's center is used to decide where to go next. Therefore, the scene is divided into three vertical strips: the left (`foundLeft`), center (`foundCenter`) and right (`foundRight`) strip (see Fig. 11). The left and right strip each take up 30% of the scene, while the center one takes the remaining 40%. If, for instance, the ball's center is found within the right strip, then *SoccerBot* would change into the `goRight` state (we refer to Fig. 10 for details).

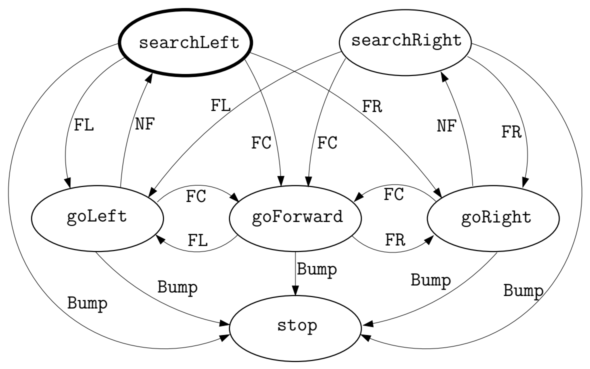

Fig. 10. *Finite state machine*: `NF` = `notFound`, `FL` = `foundLeft`, `FC` = `foundCenter` and `FR` = `foundRight`. The initial state (`searchLeft`) is marked with a thicker border. The `Bump` event occurs, when *SoccerBot* touches the ball.

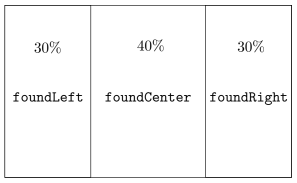

Fig. 11. Scene subdivision into left, center and right strip.

The rather simple task of the current controller is to locate the ball by turning around its axis and heading for it, when found, until the ball is reached. Then *SoccerBot* stops. This controller can easily be replaced with a more sophisticated one, e.g. with one for playing soccer.

## V. Summary and Outlook

We managed to construct a robot - the *SoccerBot* - based on Lego® Mindstorms™, which is able to recognize and catch a ball. All software parts are structured in a modular way, which has the advantage of being able to replace parts without having to redesign the entire robot. Using an improved Split-and-Merge algorithm, we can recognize objects, such as a ball, even under changing lighting conditions. We provide a Giotto timing program with a time-triggered controller, which meets predefined temporal constraints. Further work will incorporate the development of a controller performing more complex tasks. Furthermore, we are planning to improve and extend the object recognition layer. Extensions include a color-threshold adaptation within the Split-and-Merge algorithm to improve its robustness and the integration of Hough Transformation within the object recognition layer in order to be able to recognize objects more accurately.

## A. Hough Transformation

The *Hough Transformation* HT [7], [8] is a widely used method for finding parameterized shapes, especially lines, circles, ellipses etc. within images. Its main advantage is a high robustness against noise, as well as the ability to find the seeked shape, even if parts of it are missing or occluded. In its standard form though, it is not applicable for *SoccerBot*, because of its relatively long execution time as well as high memory requirements (especially for detecting arbitrarily sized circles). Although the algorithm for finding straight lines could be adjusted to meet our temporal constraints, still it only returns a collection of unbounded lines, which would need to be grouped to build more complex objects.

As our main target was to find a ball, we concentrated on the circular Hough Transformation (CHT) to detect circles within an image. The first step of the HT is to convert the input RGB image into gray-scale (we use the standard NTSC color to gray-scale conversion). Then, the grey-scale image is converted into a binary image using the Laplacian edge detector. A separate 2D accumulator is needed for every circle radius, thus the range of queried radii is an important factor for speed and memory requirements. In order to improve speed, we only take radii in steps of 2 pixels and use one 2D accumulator for all radii, accepting the drawback of less accuracy.

With this configuration, a $352 \times 288$ image takes about $250$ ms. Thus, our predefined requirements are not met. But we still have some ideas of how to improve speed and how to incorporate HT into our project.

*1) Improvement of HT:* Instead of using the Laplacian edge detector, the Canny edge detector could be used for the grey-scale to binary conversion, as it also returns direction and magnitude of the found edges. Using this information can reduce the steps within the HT significantly. Work in this field is currently in progress.

## B. Improvements of Split-and-Merge

Currently, we are using static matrices for homogenity and uniformity tests. An improvement would be to adopt the matrices at runtime, depending on the current lighting condition. Such *adaptive matrices* could easily be derived multiplying the original matrices with different empirically determined factors. Another approach is to adopt matrices in case a ball can not be found.

Furthermore, we are planning to replace the manual calibration step by an automated one. In such a case, the initial threshold is calculated iteratively and updated every time we can be absolutely sure, that the recognized region exactly matches the ball. Such a test could be performed using the introduced CHT.

*1) Integration of HT into Split-and-Merge:* One of the main ideas is to find regions likeable to be the ball using Split-and-Merge and pass that part of the image to the CHT-algorithm to verify the *quality* of the found object. A quality of the circle can be obtained as a by-product of the CHT by using the accumulator value: a perfect circle returns a maximum accumulator value. The ratio between the current value and the maximum value returns good quality measurement. As only a sub-image has to be analyzed and the range of radii is also restricted, we suppose that the time consumption will not exceed our requirements.

Another possibility is to take advantage of the fact, that Split-and-Merge creates a segmented image. Out of this image, edges can be extracted easily, which could further be processed by the CHT. The question is, which impact will the loss of original edge information have on the quality of the CHT.

## Acknowledgment

The authors would like to thank the academic supervisor of this project, Prof. Christoph Kirsch, for his support and guidance.

A lot of thanks deserve to go to Prof. Helmut Mayer for supervising Emma2Object, a parallel project of the authors, for the local robolab group. Many ideas for the design of our robot grew out of this project.

## References

[1] T. Erdmann and V. Rehrmann, “Farbbildverarbeitung für fußballspielende Roboter,” in *Workshop Farbbildverarbeitung*, no. 6, 2000, Berlin. [Online]. Available: http://www.uni-koblenz.de/~lb/publications/Erdmann2000FFF.pdf

[2] T. Henzinger, C. Kirsch, and B. Horowitz, “Giotto: A time-triggered language for embedded programming,” *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, January 2003. [Online]. Available: http://www.cs.uni-salzburg.at/~ck/publications/journals/ProcIEEE03-Giotto.pdf

[3] S. Horowitz and T. Pavlidis, “Picture segmentation by a tree traversal algorithm,” *JACM*, vol. 23, no. 2, pp. 368–388, April 1976.

[4] V. Rehrmann and L. Priese, “Fast and Robust Segmentation of Natural Color Scenes,” in *3rd Asian Conference on Computer Vision (ACCV’98)*, ser. LNCS, R. T. Chin and T.-C. Pong, Eds., no. 1351. Springer Verlag, 1998, pp. 598–606. [Online]. Available: http://www.uni-koblenz.de/~lb/publications/Rehrmann1998FAR.pdf

[5] C. Lange, “Untersuchung von Algorithmen zur visuellen Roboterlokalisierung,” Master’s thesis, Universitaet Bielefeld, 2000, Bielefeld. [Online]. Available: http://www.techfak.uni-bielefeld.de/~clange/diplom.pdf

[6] L. Priese, “Vergleich von Farbsegmentierungstechniken,” in *9. Heidelberger Bildverarbeitungsforums, 3. Nov. 1998*. Verband Deutscher Maschinen- und Anlagenbau e.V., Frankfurt, 1998. [Online]. Available: http://www.uni-koblenz.de/~lb/publications/Priese1998VVF.pdf

[7] H. P.V.C., “Method and means for recognizing complex patterns,” USA Patent US Patent #3 069 654, 1962.

[8] R. O. Duda and P. E. Hart, “Use of the hough transformation to detect lines and curves in pictures,” vol. 15, January 1972, pp. 1–15.

[9] M. L. Noga, L. Villa, P. Masetti, S. M. Moraco, *et al.*, “BrickOS Alternative LEGO Mindstorms OS.” [Online]. Available: http://brickos.sourceforge.net/index.html

[10] J. Solorzano *et al.*, “leJOS.” [Online]. Available: http://lejos.sourceforge.net/index.html