Small Size Holland
Team Description Paper
RoboCup 2016

Joost Overeem, Rimon Oz, Nanko Schrijver, Jeroen de Jong,
Thomas Hakkers, Ryan Meulenkamp, Jelle Meijerink,
Nick van Deth, Michel Teterissa, Sven Dickers,
Emre Elmas, Mark Lefering, and Rob van den Berg

Life Science Engenineering and Design, Saxion University of Applied Sciences,
M.H. Tromplaan 28, 7513 AB Enschede, The Netherlands
robocup.saxion@gmail.com
http://www.SmallSizeHolland.com/

Abstract. This paper outlines the major design decisions, and implementation and test results by Team Small Size Holland since participating in Robocup 2015 in Hefei, China. Progress was made on the development of a new team of modular robots and significant steps were taken in redesigning the software with the goal of going open source right after Robocup 2016 in Leipzig.

Keywords: Robotics, Easy Disassembly, Omni Wheels, Dribbler, CPU, Dual Core Combined with PFGA, Energy Based Strategy

1 Introduction

Small Size Holland (Team SSH) from Saxion University, the Netherlands, builds upon the ongoing research of previous years. This paper outlines the work since the the team's previously published paper [1]. In July of 2015 we started working on redesigning both software and hardware.

At the tournament in Hefei we played with our first generation of match-ready robots. With the insights gained from the analysis of video material of the matches, we went straight back to updating the hardware of the robot and working on a rewrite of both the server- and robot-software. The new team of robots is expected to be fully assembled and ready to go by May.

This paper discusses the major changes in hardware and the software, including an overview of the strategy. Chapter two discusses the mechanical changes. The third chapter presents the new CPU and other electronical improvements. Chapter four describes the new software and a novel approach to SSL-strategy.

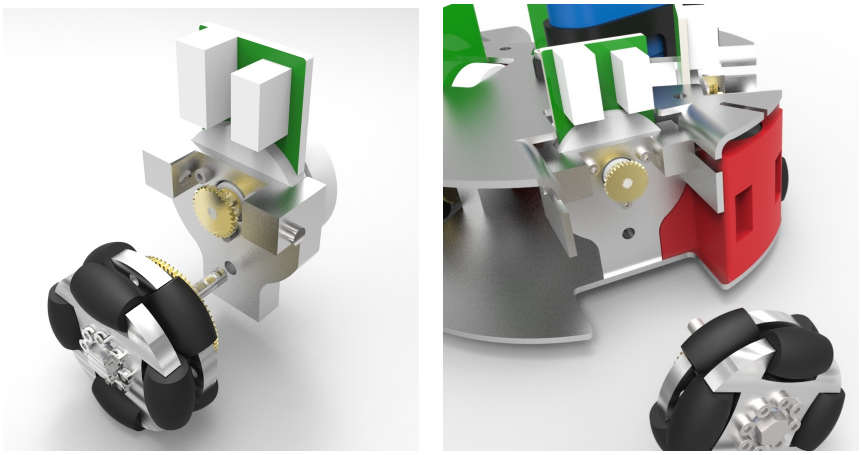
Besides the ongoing development of the robots, we have decided to open source all of the code and are in the process of getting the documentation in order. We expect this process to be done by July.

2 Mechanical Design

During the matches in Hefei, we realised that the previous generation of robots had issues with ease of maintenance. During the games there were quite a few broken motor-controllers and malfunctioning kicking mechanisms. Because of the way the robot was built these repairs were both time-consuming and expensive.

To solve both these problems we decided to modularize the robot; every part has to be easily replaceable in order to drive down expenses and hours of maintenance. To accomplish this, we decided to subdivide the robot into several areas: wheel modules, a chargeboard, a motherboard, a dribbler, a chip-and-kick module, the hull, the battery and its socket, and wheel arches. All these parts can be replaced or removed individually. This will accomodate further research, since prototypes for new modules can be hotswapped without (partially) redesigning the robot.

2.1 Wheels



(a) Seperate Wheel Module (b) Module Connected into the Robot

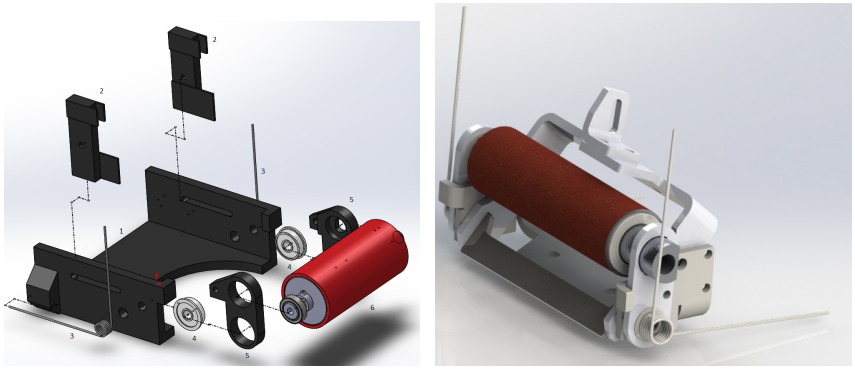
Fig. 1: Wheel Module Assembly

We have decided to move from a 3-wheel design to a 4-wheel design. Main reasons for this were the ease of controlling a 4-wheel robot and a better distribution of leverage for each wheel in all driving directions. This will prevent the robot from undulating and also result in a higher speed in desirable driving directions.

Wheels should have as much grip as possible to accelerate faster. The previous design used omni-wheels with cassette tape wheels as gear teeth to grip the grass.

This approach did not work and as a side-effect the robot bounced up and down while driving. Our proposed solution is to use omni wheels that have a rounded rubber surface as shown in Fig. 1. This idea was based Robert L. et al. findings on grip [2] and suitable wheels we found on the internet.

To make it easy to disassemble the robot or change eg. the gear ratio, we made the wheel module a replaceable component like shown in Fig. 1a. The entire module can be replaced by removing two screws.



(a) Test Set-up (b) Depressing Dribbler Module

Fig. 2: Dribbler

2.2 Dribbler

The first team of robots we played with had fixed dribblers covered with an inner tube from a bikewheel. The dribbler is driven by a 944D Motor with a nominal rotational speed of 2000 *rpm*. During tests, and also the matches played, we found that the robots were not able to receive a pass properly.

We decided to replace the inner tube with a thicker layer of rubber which is both softer and has more grip. The rubber we used is Linatex[1]. Testing of this setup showed that the rubber did not increase the shock absorption enough, but increased the grip enormously.

To increase the likelihood of completing a pass, we designed a construction in which the dribbler depresses as the robot receives the ball. Our test set-up, shown in Fig. 2a, was able to receive a pass at a speed of $6\frac{m}{s}$. The design of the dribbler assembly, which is based on the test set-up, is shown in Fig. 2b. The current construction makes use of 90° torsion springs.

[1] *An industrial rubber from The Weir Group PLC that was specially treated by Egberts Rubber B.V.

Specification	EC16 Motor	994D Motor
Voltage (DC)	12 V	6 V
Nominal Rotational Speed	39300 rpm	2000 rpm
Torque	7.85 mNm	1.76 mNm

Table 1: Dribbler Specifications

We also wanted to be able to control the rotational speed of the dribbler for performing moves such as a back heel, for gaining possession while turning around, etc. For gaining possession we need a better combination of grip and rotational speed than the opposition. Our solution for this is a new dribbler motor: an induction motor with a nominal speed of 39300 rpm. The specs of the old and new motor are shown in Table 1.

3 Electrical Design

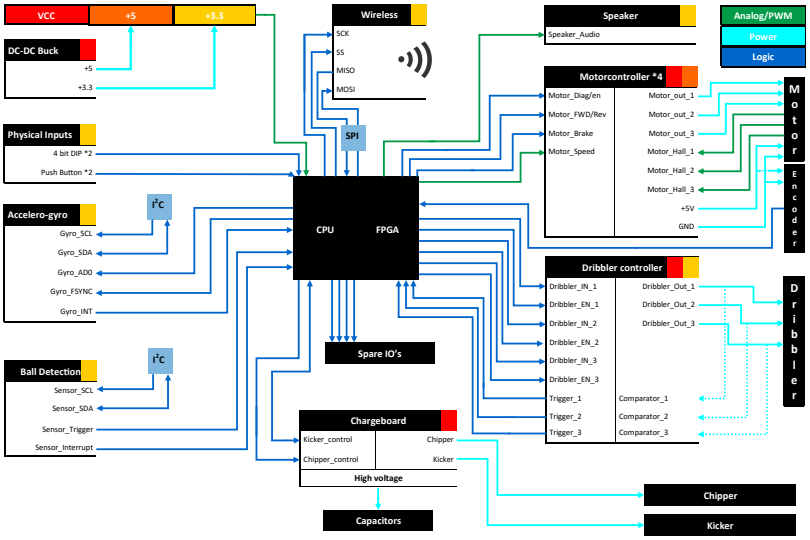


Fig. 3: System Diagram Hardware Motherboard

Our previous charge board had a few issues, and with the redesign of the frame, we decided to do a redesign of the circuit. The charge board is now combined with the switch board (the board in our previous design containing the MOSFETs). The board is better fused and is now situated separately from

other electronics. The charge board is placed in the center of the robot in order to increase the safety for people handling the robot.

The motherboard also contained a few mistakes, so we made a redesign of the motherboard as well. Most of the proposed functionality would be covered by a new and powerful CPU, described in Paragraph 3.1. Furthermore, there were two new motor-controllers which had to be added: one for the extra wheel and one for the new dribbler motor. Lastly, we decided to integrate a few new sensors. All these subsystems are on the motherboard and their connections to external components are shown in Fig. 3.

3.1 CPU

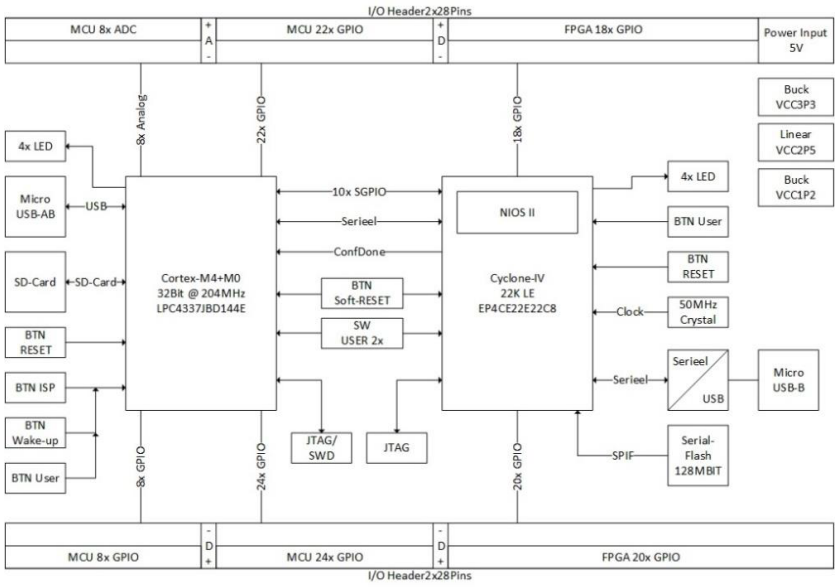


Fig. 4: System Diagram Hardware CPU

The previous generation of robots have an ARM Cortex-M3 on an MBED. This is a single core processor which makes it impossible to send and receive simultaneously. In order to support parallel processing of data the robots need a secondary processor. Increasing the communication capabilities of the robot allow us to send more data. The server will utilize this in collecting a variety of metrics from the robots such as their charge level of both batteries and capacitors, ball possession and debug information during (and outside of) the game.

Because of the latency between the movement of robots and the robots receiving commands to alter their speed and/or direction, the robots should be able to do small movements autonomously. For guidance of small autonomous

movements we have built an accelerometer, a gyro sensor, and encoders into each robot. However, processing the sensor data to a precise position of the robot takes more processing power than is possible with the current MBED.

In order to keep the robot as modular as possible, we decided to make the CPU a pluggable unit, similar to the MBED in the previous generation of robots. To be able to send and receive wireless message simultaneously, we chose a dual core processor. For the real time regulation of the motor controllers, and processing of sensor data, we decided to integrate an FPGA into the system. The result is a hybrid processor consisting of a LPC4337JBD144 microcontroller and a Cyclone IV EP4CE22E22C8 FPGA. The microcontroller consists of a Cortex-M4 and Cortex-M0. We made 10 user defined data lines between the FPGA and the microcontroller in order to have a fast communication, as shown in system diagram in Fig. 4.

3.2 Robot communication

In our first design the most obvious method of data transmission was chosen based on research done previously in SSL, but there was no in-house research done on alternatives. This year we did research on what method of communication is the best. For the method of data transmission and a suitable chip, the most important requirements of the communication were: support for protobuf, reliable data transmission, and low latency. Minor requirements we had were the use of standard equipment and the possibility of updating the robot software via the wireless communication.

Wi-Fi doesnt need a separate base station, such as the one needed for the NRF, because the server has a Wi-Fi card. Also, Wi-Fi makes it possible to easily flash new software on the robots OTA. Based on these advantages, we decided to switch Wi-Fi given it could provide enough performance.

To measure the performance tests were performed with two different Wi-Fi chips. The tests were done as follows:

1. Microcontroller broadcasts a number of packets with timer data.
2. Wi-Fi chip receives packets and sends them via UART to the microcontroller.
3. Microcontroller subtracts the timer data in the packet from the current time.
4. Microcontroller calculates the packet loss by counting the difference in sent and received packets.

With the Espressif ESP8266 we measured an average latency of 6 ms. This was good enough to make the switch to Wi-Fi. We had two chips which met the requirements: the Espressif ESP8266 and the Microchip RN1723. We chose the Espressif ESP8266 since it supports UART as well as SPI and it is easy to program.

4 Software

In an effort to modernize the software architecture, we decided to re-evaluate our application stack. The previous iteration of our software consisted at the front

of a monolithic Swing GUI with controls and a 2D representation of the field. The backend consistend of an event-based system for assigning roles to robots, a Dijkstra path planner for robot movement, several hard-coded strategies and a highly deterministic strategy selector.

Since then we have updated from Java 7 to Java 8, and have replaced the old Swing GUI with a custom-made JavaFX framework with a 3D representation of the field in the front-end. Our back-end has been completely redesigned and replaced with a responsive energy based model [4] managed by our own asynchronous stream-processing framework. Furthermore, we built our own intelligent key-value store which serializes models to human-readable JSON.

The main goal in a rewrite of the software was to build a platform with which (future) team members, and of course other teams, can easily create and test functionality. With this goal in mind we have decided to take the necessary steps to open source our software and documentation. We will release a new version of the software every half year and will have a public Git repo tracking our progress on the software. The first version of our new system, named Leo Regulus, is scheduled to be released at the beginning of July.

4.1 GUI

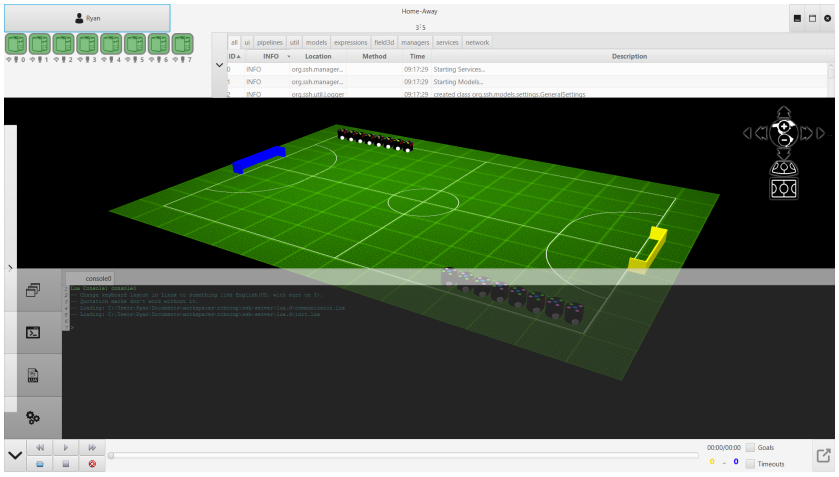


Fig. 5: The new GUI

The GUI has been rewritten in JavaFX, using FXML for markup. In the new iteration of the software we have a composable interface, with widgets capable of representing any values in the model either graphically or by text. This enables our engineers working on the robots to receive instant feedback. The composable views are fully configurable and can be saved in a profile. We integrated profiles

to enable different engineers to work with different workspaces using the same program.

3D representation of the field The GUI has a 3D component to represent the field. Since other teams in SSL soccer are starting to master the use of the chipper, we are trying to anticipate this by using a 3D representation of the field which can visually represent the vertical position of the ball. The 3D environment has controls for several pre-set views of the field and a camera which can be set at any angle and moved to any position of the field. We make use of our own layer built on top of JavaFX which binds all data in the model to the 3D environment. This gives us instantaneous support for arbitrary field sizes, multiple goals, multiple balls, non-standard team-sizes, and other Technical Challenge material.

Game log player In order to review game logs and support replay of game data we have built a game log player into the system. The application allows engineers working on the strategy to test certain moves which have been logged before and review their usability. We are currently in the process of integrating the game log player with the robot control unit in order to play back game logs with physical robots.

4.2 Strategy

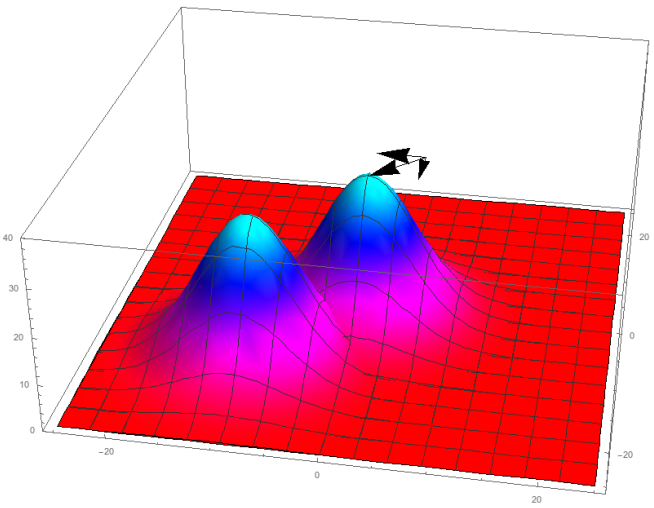


Fig. 6: Scalar energy representation of two robots

The strategy system received a complete overhaul. We decided to use a simple energy-based model [4] in which the information gathered about a robot, such as position, velocity, and orientation, is mapped to scalar quantities in a two-dimensional grid. This way we can visually represent the state of the game in a sort of heatmap (see Figure 6) and translate strategic problems, such as determining the optimal destination for a robot or determining success rate of a shot, into problems of vector calculus, such as finding local minima/maxima or performing a line integral. The model we have developed so far draws on both energy-based models [4] and ideas from anti-gravity movement [5]. As our research in this area appears to be novel, we are in the progress of writing an additional paper on this topic and would welcome any questions and/or suggestions.

4.3 Pipelines

To manage the incoming stream of data from $ssl-vision$ and $ssl-referee$ and the outgoing streams of data to the robots we have built our own asynchronous stream processing framework. The framework is multithreaded and processes packets on a first-come first-served basis. Packets of data are classified and processed by the corresponding data pipeline. An I/O management system is used to read network packets and transform protobuf data to a packet suitable for the pipeline before processing.

The architecture of a data pipeline in this system consists of producers, couplers, and consumers. Producers generate data and are primarily used in the absence of real $ssl-vision$ data to simulate games. Couplers are used to transform data and are used in computing data corrections and transformations, such as the unscented Kalman filter we use, calculation of strategy-related metrics, or eg. a ghosting filter. Lastly, consumers catch packets at the end of a pipeline and are used to update a model, the console, or a file.

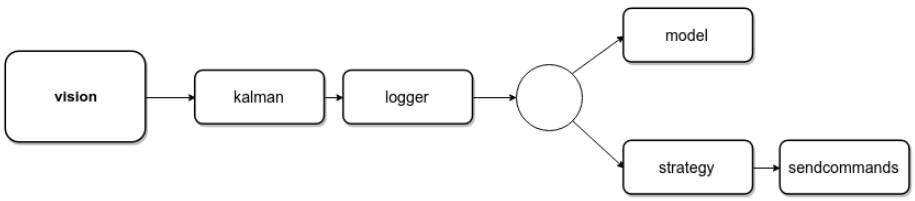


Fig. 7: Pipeline route

Managing dataflow To be able to express the flow of data in a human-readable format we created a regular language to denote pipeline-routes. This language,

named PEPE (short for PEPE: Evaluates Pipeline Expressions), can handle parallel and series processing of data and has support for functions. A PEPE-expression could look like the following:

```
vision > kalman > logger > (model | (strategy > sendcommands))
```

which would result in the pipeline route being created as shown in Figure 7.

4.4 LUA

Besides having a language to express the flow of data, we wanted to have scripts which could be used to alter the flow of data based on the state space configuration. In choosing a language for these scripts we decided to go for a well-known high-level language: Lua. Lua has been used by at least one other team in the league [3] and considering its simplicity and easy-to-read syntax we believe the use of a high-level language for expressing global strategies will enable more people to work on such strategies.

A console was added to the GUI which functions as a Lua REPL (Read-Eval-Print-Loop). Also, an editor with syntax highlighting was created to write, save, and execute Lua scripts. Internally, we expose Java objects to the Lua engine by adding an annotation (@*AvailableInLua*) to the class.

References

1. Emmerink, T., Berg, R. van den, Overeem, J.W., Hakkers, T., Jong, J. de, Van Ommeren, J., Meulenkamp, R.:SSH Team Description Paper for RoboCup 2015. Enschede (2015)
2. Williams II, R.L., Carter, B.E., Gallina, P., Rosati, G.: Dynamic Model with Slip for Wheeled Omni-Directional Robots. Final Manuscript, IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION, (2002)
3. Zhao, Y., Xiong, R., Tong, H., Li, C., Fang, L.: ZJUNlict Team Description Paper for RoboCup 2014. Zhejiang (2014)
4. LeCun, Y., Chopra, S., Hadsell, R., Ranzato, M., & Huang, F. (2006). A tutorial on energy-based learning. Predicting structured data.
5. Anti-Gravity Movement. (n.d.). Retrieved December 5, 2015, from http://robowiki.net/wiki/Anti-Gravity_Movement