Technical University of Crete, Greece
Department of Electronic and Computer Engineering

KMonitor

Global and Local State Visualization and Monitoring for the Robocup SPL League

Maria Karamitrou

Thesis Committee

Assistant Professor Michail G. Lagoudakis (ECE)

Assistant Professor Georgios Chalkiadakis (ECE)

Assistant Professor Antonios Deligiannakis (ECE)

Chania, July 2012

Πολυτεχνειο Κρητης

Τμημα Ηλεκτρονικων Μηχανικων και Μηχανικων Υπολογιστων

KMonitor

Ολική και Τοπική Οπτικοποίηση και Παρακολούθηση Κατάστασης για το Πρωτάθλημα SPL του Robocup

Μαρία Καραμήτρου

Εξεταστική Επιτροπή

Επίκουρος Καθηγητής Μιχαήλ Γ. Λαγουδάκης (ΗΜΜΥ)

Επίκουρος Καθηγητής Γεώργιος Χαλκιαδάκης (ΗΜΜΥ)

Επίκουρος Καθηγητής Αντώνιος Δεληγιαννάκης (ΗΜΜΥ)

Χανιά, Ιούλιος 2012

Abstract

Software debugging is an essential process in any software development project. However, debugging robotic software can be challenging because of the continuous real-time interaction between the robot and the environment. This is particularly true in the RoboCup competition, where teams of autonomous robots compete against each other in various leagues of soccer games. The Standard Platform League (SPL) focuses mostly on robot software development, since all teams use identical robots, namely the Aldebaran Nao humanoid robots. Software development for autonomous robots requires both individual inspection of each module of the code, so as to test the performance of isolated functionality, and joint inspection of the interaction between modules, so as to test the overall performance of the integrated system. In the RoboCup domain, debugging can take the form of monitoring the internal state of each robot (perception of objects, self-location, obstacles, etc.) as well as the global state of the entire team (shared perception, formations, roles, etc.). This thesis presents KMonitor, an integrated monitoring software application for Nao robots, which allows the developer to inspect effectively each one of the basic modules of the code of our RoboCup SPL team Kouretes. KMonitor offers real-time visualization of data on a remote desktop computer using a User Datagram Protocol (UDP) multicast network. The graphical user interface is composed by various tabs offering different monitoring views for individual inspection of the outcomes of vision, localization, obstacle avoidance, and decision making and joint inspection of the interaction between software modules. The user can select one or more of the detected active robots to monitor and check one or more desired graphical features to visualize. KMonitor has been implemented using the Qt application framework over the underlying Monas software architecture and Narukom communication framework of the Kouretes code. KMonitor integrates the required functionality and views under a single intuitive graphical user interface, facilitating the user in switching quickly between different views in order to monitor different aspects of the robot software. Finally, KMonitor is easily extensible and fully parameterizable to accommodate future needs and RoboCup SPL rules modifications.

Περίληψη

Η αποσφαλμάτωση λογισμικού αποτελεί μια ουσιαστική διαδικασία σε κάθε έργο ανάπτυξης λογισμικού. Ωστόσο, η αποσφαλμάτωση λογισμικού για ρομπότ εμπεριέχει μεγαλύτερη δυσκολία λόγω της συνεχούς και πραγματικού χρόνου αλληλεπίδρασης μεταξύ ρομπότ και περιβάλλοντος. Αυτό είναι ιδιαίτερα αληθές στο διαγωνισμό RoboCup, όπου ομάδες αυτόνομων ρομπότ ανταγωνίζονται μεταξύ τους σε διάφορα πρωταθλήματα αγώνων ποδοσφαίρου. Το πρωτάθλημα Standard Platform League (SPL) εστιάζει κυρίως στην ανάπτυξη ρομποτικού λογισμικού, δεδομένου ότι όλες οι ομάδες χρησιμοποιούν πανομοιότυπα ρομπότ, συγκεκριμένα τα ανθρωποειδή ρομπότ Aldebaran Nao. Η ανάπτυξη λογισμικού για αυτόνομα ρομπότ απαιτεί τόσο ατομικό έλεγχο κάθε οντότητας του κώδικα, ώστε να δοκιμαστεί η απόδοση κάθε μεμονωμένης λειτουργικότητας, όσο και από κοινού έλεγχο της αλληλεπίδρασης μεταξύ των οντοτήτων, προκειμένου να δοκιμαστεί η συνολική απόδοση του ολοκληρωμένου συστήματος. Στο πεδίο του RoboCup, η αποσφαλμάτωση μπορεί να λάβει τη μορφή παρακολούθησης της εσωτερικής κατάστασης του κάθε ρομπότ (αντίληψη αντικειμένων, εκτίμηση θέσης, αναγνώριση εμποδίων, κτλ.) καθώς και της καθολικής κατάστασης ολόκληρης της ομάδας (κοινή αντίληψη, σχηματισμοί, ρόλοι, κτλ.). Η παρούσα διπλωματική εργασία παρουσιάζει το KMonitor, μια ολοκληρωμένη εφαρμογή λογισμικού για την παρακολούθηση ρομπότ Nao, η οποία επιτρέπει στον προγραμματιστή να ελέγχει αποτελεσματικά κάθε μία από τις βασικές οντότητες του κώδικα των Κουρητών, της ομάδα μας στο RoboCup SPL. Το KMonitor προσφέρει πραγματικού χρόνου οπτικοποίηση των δεδομένων σε απομακρυσμένο υπολογιστή χρησιμοποιώντας ένα User Datagram Protocol (UDP) multicast δίκτυο. Η γραφική διεπαφή του χρήστη αποτελείται από διάφορες καρτέλες που προσφέρουν διαφορετικές γραφικές όψεις παρακολούθησης για μεμονωμένο έλεγχο των εξαγομένων της μηχανικής όρασης, του εντοπισμού θέσης, της αποφυγής εμποδίων και της λήψης αποφάσεων και από κοινού έλεγχο της αλληλεπίδρασης μεταξύ των οντοτήτων του λογισμικού. Ο χρήστης μπορεί να επιλέξει να παρακολουθήσει ένα ή περισσότερα ενεργά ρομπότ και να οπτικοποιήσει ένα ή περισσότερα επιθυμητά γραφικά χαρακτηριστικά. Το KMonitor έχει υλοποιηθεί με χρήση του πλαισίου εφαρμογών Qt και υποστηρίζεται από την αρχιτεκτονική λογισμικού Monas και το πλαίσιο επικοινωνίας Narukom του κώδικα των Κουρητών. Το KMonitor ενσωματώνει την απαιτούμενη λειτουργικότητα και τις όψεις υπό μία ενιαία διαισθητική γραφική διεπαφή χρήστη, διευκολύνοντας τη γρήγορη μετάβαση μεταξύ διαφορετικών γραφικών όψεων προκειμένου να επιτευχθεί η παρακολούθηση διαφορετικών πτυχών του ρομποτικού λογισμικού. Τέλος, το KMonitor είναι εύκολα επεκτάσιμο και πλήρως

παραμετροποιήσιμο, ώστε να ικανοποιήσει μελλοντικές ανάγκες και τροποποιήσεις στους κανόνες του RoboCup SPL.

Acknowledgements

Τυπικά θα έπρεπε να γράφω το συγκεκριμένο τμήμα του κειμένου της διπλωματικής μου εργασίας στα αγγλικά, για λόγους ομοιμορφίας. Επιλέγω όμως την ελληνική, αφού θεωρώ πως δεν μπορώ να εκφράσω ακριβώς τις σκέψεις μου σε καμία άλλη γλώσσα.

Καταρχήν θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Λαγουδάκη για τη δημιουργία της ομάδας Κουρήτες. Στους Κουρήτες, η εκπόνηση της διπλωματικής εργασίας μετατρέπεται από ατομική άσκηση σε ομαδική, συνεργατική άσκηση επαγγελματικής εκπαίδευσης. Τον ευχαριστώ για την καθοδήγηση και την βοήθειά του καθ΄ όλη την διάρκεια της συγκεκριμένης εργασίας.

Δήμητρα, Αγγελική, Νίκο (Παυλάκη), Ίριδα, Βαγγέλη και Λευτέρη σας ευχαριστώ για την αγάπη και το ενδιαφέρον σας για τους Κουρήτες. Μανώλη, αν δεν ήσουν εσύ στους Κουρήτες, πιθανότατα εγώ δεν θα είχα ξεκινήσει την συγκεκριμένη διπλωματική εργασία. Νίκο (Κοφινά), αν δεν ήσουν εσύ στους Κουρήτες, πιθανότατα εγώ δεν θα είχα ξεκινήσει ψυχοθεραπεία. «Κ-ουρητ-οπέλια», σας ευχαριστώ όλους για την συνεργασία, τις συμβουλές, τις ιδέες και τις μοναδικές στιγμές που περάσαμε στην ομάδα.

Γεωργία, Θεοδώρα και Σοφία σας ευχαριστώ για τις υπέροχες φοιτητικές στιγμές που ζήσαμε και που μου αποδεικνύετε καθημερινά πόσο σημαντική είναι η έννοια της λέξης φιλία. Χρήστο μου, όσα και να είναι τα εμπόδια υπάρχει τουλάχιστον ένα ελεύθερο μονοπάτι, σωστά; Τίποτα δεν είναι αδύνατο, είχες δίκιο τελικά.

Τέλος, ένα μεγάλο ευχαριστώ στην οικογένειά μου, που παρόλα τα λάθη χρονισμού μου, ήταν πάντα δίπλα μου, με στήριξε και με στηρίζει με κάθε δυνατό τρόπο.

Contents

List of Figures

Chapter 1

Introduction

The RoboCup Competition is an international annual aggregation of robotic competitions which intends to promote Robotics and Artificial Intelligence (AI) research. RoboCup Soccer constitutes one of the four RoboCup competitions and focuses mainly on the game of soccer. The research goals concern cooperative multi-robot and multi-agent systems in dynamic adversarial environments and all the participating teams have to find real-time solutions to some of the most difficult problems in robotics, such as perception, cognition, action, and coordination. In the Standard Platform League (SPL) all teams use identical (standard) robots. Currently, the chosen hardware platform is the Aldebaran NAO humanoid robot, therefore the teams concentrate on software development only.

Software development for robots competing in the RoboCup SPL essentially aims at developing autonomous agents. An autonomous robotic agent is a system that continuously perceives its environment through the robotic sensors, analyzes the percept sequence using various AI techniques, and takes actions through the robotic actuators with the goal of maximizing a utility function. The central problems of an autonomous robotic agent include environment perception, robot localization, robotic mapping, path planning, decision making under uncertainty, multi-agent planning and learning, and robot coordination. Therefore, the creation of the desired functionality for an autonomous robot is a complicated procedure, which lies on addressing each one of the basic AI problems and integrating these approaches into a single entity.

The process of debugging is an essential step in any software development project. It contributes to the detection and elimination of flaws from the written code in order to deliver the desired functionality reliably. However, debugging robotic software can

be challenging because of the continuous interaction between the agent and the environment and the real-time aspect of this interaction. When developing robotic agents, several debugging methodologies may have to be followed so as to achieve the expected behavior. In particular, the implementation of an autonomous agent requires both individual inspection of each module of the code, so as to test the performance of isolated functionality, and joint inspection of the interaction between modules, so as to test the overall performance of the integrated system. In the RoboCup domain, debugging can take the form of monitoring the internal state of each robot (perception of objects, self-location, obstacles, etc.) as well as the global state of the entire team (shared perception, formations, roles, etc.).

1.1 Thesis Contribution

This thesis contributes KMonitor, an integrated monitoring software application for RoboCup SPL robotic agents, which allows the developer to inspect effectively each one of the basic modules of the existing code. The proposed technique to accomplish this goal is real-time visualization of the data each module operates on and the data it exports on a remote desktop computer using a User Datagram Protocol (UDP) multicast network. Thus, the values of sensors and actuators, as well as the outcomes of the vision, localization, and obstacle avoidance activities are designed graphically in a user-friendly intuitive way under different monitoring views. Furthermore, the application provides joint inspection of the interaction between basic modules. The user interface of KMonitor is composed by various tabs each one of them offering a different monitoring view. The user can select one or more of the detected active robots to monitor and check one or more desired graphical features to visualize, according to the functionality of the current tab. KMonitor has been implemented using the Qt application framework over the underlying Monas software architecture and Narukom communication framework, also used by the robots. Finally, KMonitor eases any future extension, so as to adopt possible module enhancements and new graphical elements. Additionally, it provides easy XML-based configuration to be consistent at all times with the current parameterization of the code and to catch up with the Robocup SPL rules which are frequently modified.

1.2 Thesis Overview

Chapter 2 describes the RoboCup competition, the Standard Platform League (SPL), the Aldebaran NAO humanoid robot, our SPL team Kouretes, our software architecture Monas, and our communication framework Narukom. Furthermore, it provides basic background information about Qt, the cross-platform application framework on which KMonitor is based, the signal and slot mechanism of Qt, and the Qt Designer GUI layout and forms builder, features of Qt we extensively utilized. In Chapter 3 we discuss the significance of the debugging process for robotic software development and we state the requirements for our monitoring application, while in Chapter 4 we briefly refer to the related work of other SPL teams. In Chapter 5 we describe the design of KMonitor's architecture and describe extensively all the available features and functionalities it provides. In Chapter 6 we present KMonitor's implementation from a technical point of view. In Chapter 7 we present a number of scenarios demonstrating the effectiveness and real-time performance of KMonitor. Finally, in Chapter 8 we discuss the results of this thesis and we suggest some possible future research enhancements and directions.

Chapter 2

Background

2.1 RoboCup

RoboCup, an abbreviation of "Robot Soccer World Cup", is an international annual competition which intends to promote robotics and artificial intelligence research. The founding father of RoboCup, Professor Alan Mackworth, inspired the idea of building a robot to play a soccer game autonomously in 1992. One year later, Hiroaki Kitano [1] and his research group decided to launch a novel robotic competition. Finally, in 1997 the actual establishment of the International RoboCup Federation occurred. The ambitious goal of the RoboCup Initiative is stated as follows:

> "By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win the soccer game, comply with the official rule of the FIFA, against the winner of the most recent World Cup."

All the teams participating in RoboCup have to find real-time solutions to some of the most difficult problems in robotics (perception, cognition, action, coordination) and apply their approaches on the various leagues of the four RoboCup divisions (RoboCup Soccer, RoboCup Rescue, RoboCup@Home, Robocup Junior). Until today, noteworthy progress has been made in advancing the state-of-the-art technology, while the number of the participating researchers who aim to fulfill the initial challenge is constantly growing.

2.1.1 Standard Platform League

RoboCup Soccer constitutes one of the four RoboCup divisions and focuses mainly on the game of soccer, where the research goals concern cooperative multi-robot and multi-agent systems in dynamic adversarial environments. All robots in this division are fully autonomous. RoboCup Soccer consists of five different leagues (Humanoid, Middle Size, Simulation, Small Size, and Standard Platform). In the Standard Platform League (SPL) all teams use identical (standard) robots. Currently, the chosen hardware platform is the Aldebaran Nao humanoid robot, therefore the teams concentrate only on software development. The participating teams are prohibited to make any changes to the hardware of the robot, meaning that off-board sensing or processing systems are not allowed. The use of directional, as opposed to omnidirectional, vision forces a trade off of vision resources between self-localization, ball localization, player identification, and obstacle detection. The robots are completely autonomous and no human intervention from team members is allowed during the games. The only interaction of the robots with the "outer human world" is the reception of data from the Game Controller, a computer that broadcasts information about the state of the game (score, time, penalties, etc.).

The SPL games are conducted on a $4m \times 6m$ soccer field which consists of a green carpet marked with white lines and two yellow goals (Figure 2.1). The ball is an orange street hockey ball. Each team consists of four robots, one goal keeper, and three field players. The robot players are distinguished by colored belts (the so-called waist bands), blue for one team and red for the other. The total game time is 20 minutes and is broken in two halves; each half lasts 10 minutes. During the 10-minutes half-time break, teams have to switch field sides and waist bands and only during this time it is permitted to change robots, change programs, etc. The detailed rules of the SPL games are stated in detail in the RoboCup Standard Platform League (Nao) Rule Book [2], which is annually updated with enhancements and additional challenging requirements that propel the general progress of the league.

2.1.2 Aldebaran Nao Humanoid Robot

The current hardware platform which all SPL teams are obliged to work with is Nao, an integrated, programmable, medium-sized humanoid robot developed by Aldebaran Robotics in Paris, France. Project Nao [3] started in 2004. In August 2007 Nao officially

Figure 2.1: Standard Platform League at RoboCup 2012

replaced Sony's AIBO quadruped robot in the RoboCup SPL. In the past few years Nao has evolved over several designs and several versions.

Nao (version V3.3) [4] is a 58cm, 5kg humanoid robot (Figure 2.2). The Nao robot carries a fully capable computer on-board with an x86 AMD Geode processor at 500 MHz, 256 MB SDRAM, and 2 GB flash memory running an Embedded Linux distribution. It is powered by a 6-cell Lithium-Ion battery which provides about 30 minutes of continuous operation and communicates with remote computers via an IEEE 802.11g wireless or a wired ethernet link.

Nao RoboCup edition has 21 degrees of freedom; 2 in the head, 4 in each arm, 5 in each leg, and 1 in the pelvis (there are two pelvis joints which are coupled together on one servo and cannot move independently). Nao, also, features a variety of sensors and transmitters. Two cameras are mounted on the head in vertical alignment providing non-overlapping views of the lower and distant frontal areas, but only one is active each time and the view can be switched from one to the other almost instantaneously. Each camera is a 640 x 480 VGA device operating at 30fps. The native colorspace provided by the cameras is the YUV422. Four sonars (two emitters and two receivers) on the chest allow Nao to sense obstacles in front of it. In addition, the Nao has a rich inertial unit, with one 2-axis gyroscope and one 3-axis accelerometer, in the torso that provides real-time information about its instantaneous body movements. Two bumpers located at the tip

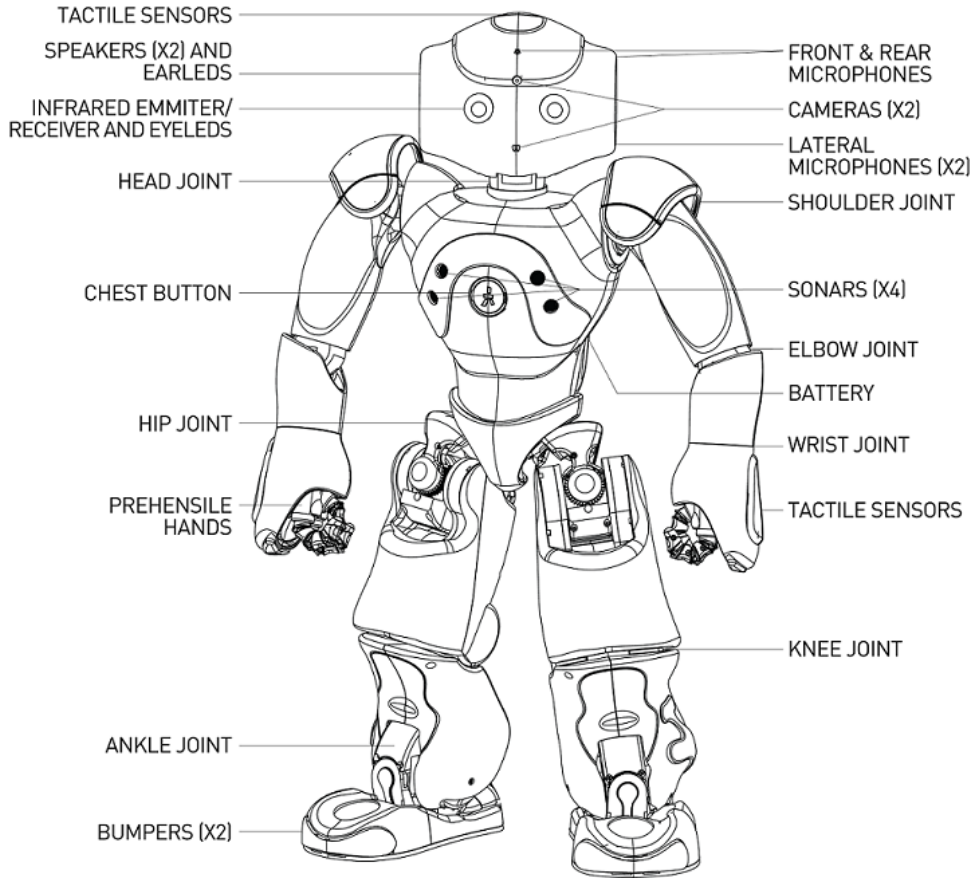

Figure 2.2: Aldebaran Nao robot (v3.3, academic edition) and its components

of each foot are simple ON/OFF switches and can provide information on collisions of the feet with obstacles. Finally, an array of force sensitive resistors on each foot delivers feedback of the forces applied to the feet, while encoders on all servos record the actual values of all joints at each time.

Aldebaran Robotics has equipped Nao with both embedded and desktop software to be used as a base for further development (Figure 2.3). The embedded software, running on the motherboard located in the head of the robot, that the company provides includes an embedded GNU/Linux distribution and NAOqi, the main proprietary software that runs on the robot and controls it. Nao's desktop software includes Choregraphe, a visual programming application which allows the creation and the simulation of animations and behaviors for the robot before the final upload to the real Nao, and Telepathe which provides elementary feedback about the robot's hardware and a simple interface to accessing

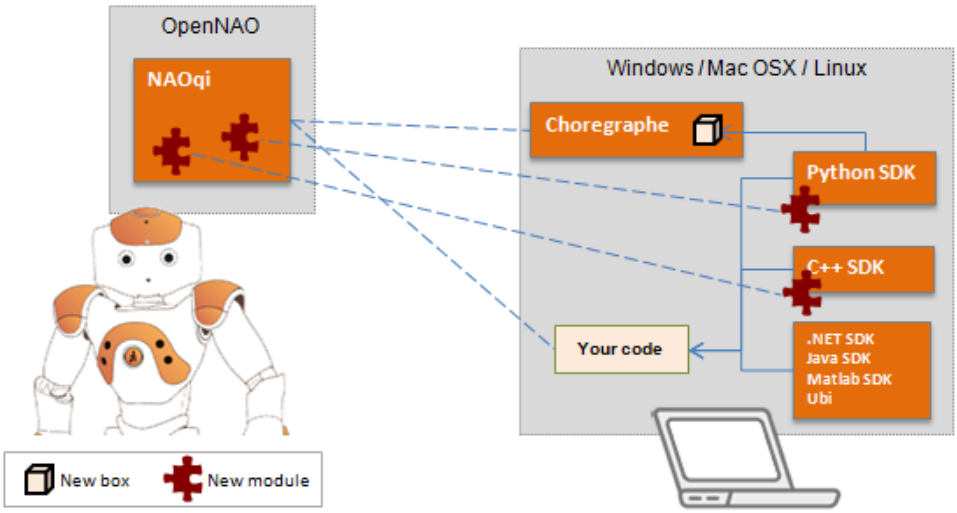

Figure 2.3: Embedded and desktop software for the Nao robot

its camera settings. As far as the NAOqi framework is concerned, it is cross-platform, cross-language, and provides introspection which means that the framework knows which functions are available in the different modules and where. It provides parallelism, resources, synchronization, and events. NAOqi, also, allows homogeneous communication between different modules (motion, audio, video), homogeneous programming, and homogeneous information sharing. Software can be developed in C++, Python, and Urbi. The programmer can state which libraries have to be loaded when NAOqi starts via a preference file called `autoload.ini`. The available libraries contain one or more modules, which are typically classes within the library and each module consists of multiple methods (Figure 2.4).

2.2 RoboCup SPL Team Kouretes

Team Kouretes is the first RoboCup SPL team founded in Greece, hosted in the Intelligent Systems Laboratory at the Department of Electronic and Computer Engineering of the Technical University of Crete. Kouretes started developing their own robotic software framework in 2008 and the code is constantly developed and maintained ever since. The team's publicly-available code repository includes a custom software architecture, a custom communication framework, a graphical application for behavior specification,

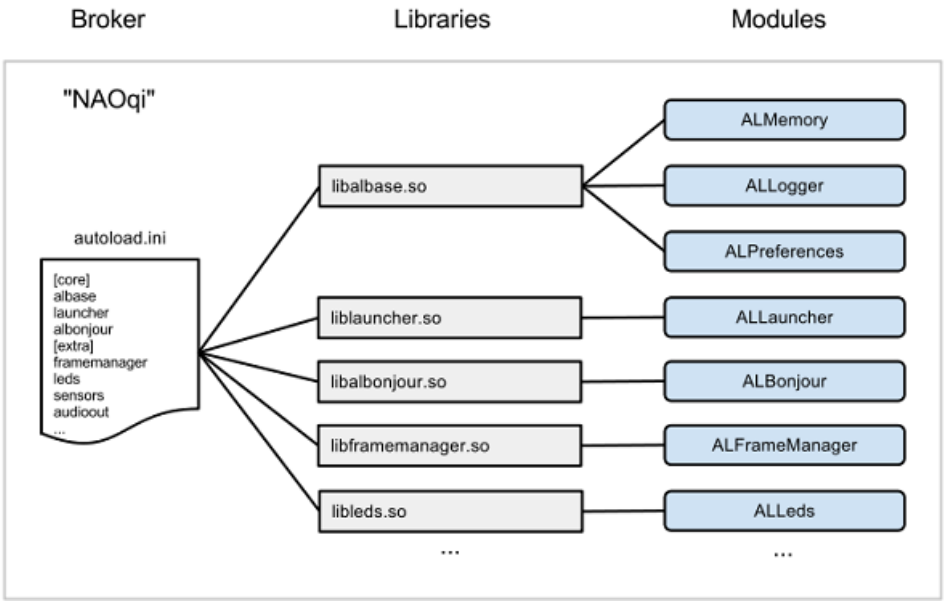

Figure 2.4: The NAOqi process

and modules for object recognition, state estimation, localization, obstacle avoidance, behavior execution, and team coordination, which are briefly described below.

The team participates in the main RoboCup competition since 2006 in various soccer leagues (Four-Legged, Standard Platform, MSRS, Webots), as well as in various local RoboCup events (German Open, Mediterranean Open, Iran Open, RC4EW, RomeCup) and RoboCup exhibitions (Athens Digital Week, Micropolis, Schoolfest). Distinctions of the team include: 2nd place in MSRS at RoboCup 2007; 3rd place in SPL-Nao, 1st place in SPL-MSRS, among the top 8 teams in SPL-Webots at RoboCup 2008; 1st place in RomeCup 2009; 6th place in SPL-Webots at RoboCup 2009; 2nd place in SPL at RC4EW 2010; and 2nd place in SPL Open Challenge Competition at RoboCup 2011 (joint team Noxious-Kouretes). Recently, the team participated in the RoboCup German Open 2012 competition in Magdeburg, in RoboCup Iran Open 2012 in Tehran, and in RoboCup 2012 in Mexico City (Figure 2.5). In the most recent RoboCup 2012 competition, the team succeeded to proceed to the second round-robin round and rank among the top-16 SPL teams in the world.

Figure 2.5: Team Kouretes at RoboCup 2012 in Mexico City

2.2.1 Monas Software Architecture

Monas [5] is a flexible software architecture which provides an abstraction layer from the hardware platform and allows the synthesis of complex robot software as XML-specified Monas modules, Provider modules, and/or Statechart modules. Monas modules, the so-called agents, focus on specific functionalities and each one of them is executed independently at any desired frequency completing a series of activities at each execution. The base activities, that an agent may consist of, are described briefly below:

- `Vision` [6] is a light-weight image processing method for humanoid robots, via which Kouretes team has accomplished visual object recognition. The vision module determines the exact camera position in the 3-dimensional space and subsequently the view horizon and the sampling grid, so that scanning is approximately uniformly projected over the ground (field). The identification of regions of interest on the pixels of the sampling grid follows next utilizing an auto-calibrated color recognition scheme. Finally, detailed analysis of the identified regions of interest seeks potential matches for corresponding target objects. These matches are evaluated and filtered by several heuristics, so that the best match (if any) in terms of color, shape, and

size for a target object is finally extracted. Then, the corresponding objects are returned as perceived, along with an estimate of their current distance and bearing.

- `LocalWorldState` [7] is the activity which realizes Monte Carlo localization. The belief of the robot is a probability distribution over the 3-dimensional space of coordinates and orientation (x, y, θ) represented approximately using a population of particles. Belief update is performed using an auxiliary particle filter with an odometry motion model for omnidirectional locomotion and a landmark sensor model for the goalposts (landmarks). The robot's pose is estimated as the pose of the particle with the highest weight.

- `ObstacleAvoidance` [8] is the activity which accomplishes obstacle avoidance by first building a local obstacle occupancy map, which is updated constantly with real-time sonar information, taking into consideration the robot's locomotion. Afterwards, an A* search algorithm is used for path planning, the outcome of which suggests an obstacle-free path for guiding the robot to a desired destination.

- `Behavior` is the activity which implements the desired robotic behavior. It operates on the outcomes of the `Vision`, `LocalWorldState`, and `ObstacleAvoidance` activities and decides which one is the most appropriate action to be executed next.

- `HeadBehavior` manages the movements of the robot head (camera).

- `MotionController` [9] is used for managing and executing robot locomotion commands and special actions.

- `RobotController` handles external signals on the game state.

- `LedHandler` controls the robot LEDs (eyes, ears, chest button, feet).

Provider modules accomplish the complete decoupling of the robotic hardware by collecting and filtering measurements from the robot sensors and cameras and forming them as messages in order to be utilized as input data by any interested Monas agents. Each provider module can be executed independently and at any desired frequency. Statechart modules [10] are executed using a generic multi-threaded statechart engine, which provides the required concurrency and meets the real-time requirements of the activities on each robot.

2.2.2 Narukom Communication Framework

Narukom [11] is the communication framework developed for the needs of the team's code and it is based on the publish/subscribe messaging pattern. Narukom supports multiple ways of communication, including local communication among the Monas modules, the Providers modules, and the Statechart modules that constitute the robot software, and remote communication via multicast connection among multiple robot nodes and among robot and external computer nodes. The information that needs to be communicated between nodes is formed as messages which are tagged with appropriate topics and host IDs. Three types of messages are supported:

- `state`, which remain in the blackboard until replaced by a newer message,
- `signal`, which are consumed at the first read, and
- `data`, which are time-stamped to indicate the time their values were acquired.

To facilitate the serialization of data and the structural definition of the messages, Google Protocol Buffers were utilized. The user defines the data structure once and then uses the generated source code to write and read the defined structures to and from a variety of data streams using a variety of programming languages. Another great advantage of protocol buffers is that data structures can be enhanced without breaking the already deployed programs, which are capable of handling the old format of the structures. To use protocol buffers one must describe the information for serialization by defining protocol buffer messages in `.proto` files. A protocol buffer message is a small record of information, containing name-value pairs. The protocol buffer message format is simple and flexible. Each message type has at least one numbered field. Each field has a name and a value type. The supported types are integer, floating-point, boolean, string, raw bytes, or other complex protocol buffer message types, thus hierarchical structure of data is possible. Additionally, the user can specify rules, if a field is mandatory, optional, or repeated. These rules enforce both the existence and multiplicity of each field inside the message. As a next step, the user generates code for the desired language by running the protocol buffer compiler. The compiler produces data access classes and provides accessors and mutators for each field, as well as serialization/unserialization methods to/from raw bytes. Officially, Google supports C++, Java, and Python for code generation, but there are several other unofficially supported languages.

Additionally, the blackboard paradigm is utilized to provide efficient access to shared information stored locally at each node and is extended to support history queries and a mechanism that controls the information updates. Finally, to meet the delivery requirements among the remote and/or the local nodes, messages are relayed though a message queue. The message queue is responsible for collecting the published messages and allocating them to the interested subscribers through multiple buffers. Messages that have to be delivered to remote nodes are committed to the KNetwork module, which implements the multicast connection.

2.3 Qt – A Cross-Platform Application Framework

Qt (http://qt.nokia.com/products) is a cross-platform application and UI framework with APIs for C++ programming and rapid UI creation. It is widely used for developing application software with a graphical user interface (GUI) and non-GUI programs, such as command-line tools and consoles for servers. Qt uses standard C++, but makes extensive use of a special code generator (the Meta Object Compiler) together with several macros to enrich the language. It runs on the major desktop platforms and some of the mobile platforms and has extensive documentation support. Non-GUI features include SQL database access, XML parsing, thread management, network support, and a unified cross-platform application programming interface (API) for file handling. Distributed under the terms of the GNU Lesser General Public License (among others), Qt is free and open source software. All editions support many compilers, including the `gcc` C++ compiler, the Visual Studio, and the Eclipse suite.

2.3.1 The Qt Signal/Slot Mechanism

The signals and slots mechanism is a central feature of Qt and is used for communication between objects. In GUI programming, it is preferable objects of any kind to be able to communicate with one another. Older toolkits achieve this kind of communication using callbacks. A callback is a pointer to a function, so, if we want a processing function to deliver a notification about some event, we must pass a pointer to another function (the callback) to the processing function. The processing function then calls the callback,

when appropriate. Callbacks have two fundamental flaws. Firstly, they are not type-safe. It is never certain that the processing function will call the callback with the correct arguments. Secondly, the callback is strongly coupled to the processing function, since the processing function must know which callback to call.

In Qt, there is an alternative to the callback technique, the signals and slots mechanism. A signal is emitted when a particular event occurs. Qt's widgets have many predefined signals, but the developer can always subclass widgets to add his/her own signals to them. A slot is a function that is called in response to a particular signal. Qt's widgets have many pre-defined slots, but it is common practice to subclass widgets and add new slots, so that the developer can handle the signals that he/she is interested in. The signals and slots mechanism is type-safe. The signature of a signal must match the signature of the receiving slot. Signals and slots are loosely coupled. A class which emits a signal neither knows nor cares which slots receive the signal. Qt's signals and slots mechanism ensures that if the developer connects a signal to a slot, the slot will be called with the signal's parameters at the right time. Signals and slots can take any number of arguments of any type; they are completely type-safe. All classes that inherit from `QObject` or one of its subclasses (`QWidget`, etc.) can contain signals and slots. Signals are emitted by objects, when they change their state in a way that may be interesting to other objects. This is all the object does to communicate. It does not know or care whether anything is receiving the signals it emits. This is true information encapsulation and ensures that the object can be used as a software component. Slots can be used for receiving signals, but they are also normal member functions. Just as an object does not know if anything receives its signals, a slot does not know if it has any signals connected to it. This ensures that truly independent components can be created with Qt.

2.3.2 The Qt Designer

The Qt Designer is a standalone application provided by the Qt4 UI framework, which includes a number of components that work together to provide a flexible GUI design tool. This design tool can be considered as a collection of interchangeable components that include the form editor, widget box, and other useful tools for creating graphical user interfaces with Qt. Widgets and dialog windows can be composed using a form-based interface that fully supports drag and drop, clipboard operations, and an undo/redo

stack. The version of Qt Designer, which we chose to develop our KMonitor graphical user interface, introduces a number of editing modes to make different types of editing natural and intuitive. Each editing mode displays the form in an appropriate way for that mode, and provides a specialized user interface for manipulating its contents. Examples of editing modes include Form Editing mode, Signals and Slots Editing mode, Buddy Editing mode, and Tab Order Editing mode. For the requirements of our application design, we utilized the Widget Box and the Property Editor features of the Qt Designer's UI, the Form Templates, the Form Editing Mode and the Resource Editor from the available editing features of the tool, and the Custom Widgets to integrate our customized widgets via the tool'As plugin support. A brief description of the used Qt Designer'As features follows.

The Widget Box displays a categorized list of widgets and other objects that can be placed on a form using drag and drop. When Qt Designer is in Top Level mode, the window containing the widget box also holds the main menu and tool bar. When in Workspace mode, the Widget Box becomes an independent window within the Qt Designer workspace. The contents of the Widget Box are defined in an XML file that holds a collection of `.ui` documents for standard Qt widgets. This file can be extended, making it possible to add custom widgets to the Widget Box.

The Property Editor allows designers to edit most properties of widgets and layout objects. The property names and values are presented in an editable tree view that shows the properties of the currently selected object. Certain resources, such as icons, can be configured in the Property Editor. Resources can be taken from any currently installed resource files, making it easier to design self-contained components.

Form Templates provide ready-to-use forms for various types of widgets, such as `QWidget`, `QDialog`, and `QMainWindow`. Custom templates based on these widgets can also be created. Templates can contain child widgets and layouts. Designers can save time by creating templates for the most common user interface features for repeated use. The Form Editor allows widgets to be dropped into existing layouts on the form. Qt Designer supports direct manipulation of widgets, such as cloning a widget by dragging it with the `CTRL` key held down, and it is even possible to drag widgets between forms. In-place widget editors provide specialized editing facilities for the most-used widget properties. Resources can be associated with a given form and these can be modified and extended using a file browser style interface. The Resource Editor uses files that are

processed by various components of the Qt Resource System to ensure that all required resources are embedded in the application.

Plugins can be used to add new custom widgets, special editors, and support for widgets from the Qt3 support library. Support for custom widget plugins allows user interface designers to use application-specific widgets in their designs as early as possible in the development process. Qt Designer handles custom widgets in the same way as standard Qt widgets and allows custom signals and slots to be connected to other objects from within the Signals and Slots Editing mode.

Chapter 3

Problem Statement

3.1 General Purpose Software Debugging

The process of debugging is an essential step in any software development project. It contributes to the detection and elimination of flaws from the written code in order to deliver the desired functionality reliably. Several debugging methods can be followed, so as to achieve the expected behavior, based on numerous aspects. The most significant criterion is the coupling of the software modules of the application, since the greater the interaction the more complicated the debugging process tends to be. The most common technique is the so called tracing debugging. Tracing debugging is the technique of watching live or recorded trace statements, or print statements, that indicate the flow of execution of a process. This is sometimes called `printf` debugging, due to the use of the `printf` statement of C. Another technique of general purpose debugging is based on efficient manipulation of breakpoints. A breakpoint is an intentional stopping or pausing place in a program and consists of a specific code line and optionally one or more conditions that must be met for execution to pause at that place. While at a breakpoint, the software developer can inspect the values of selected variables and expressions, exploiting the ability to interrupt the code execution whenever needed.

3.2 Autonomous Robot Software Debugging

Autonomous robot software aims at developing intelligent mechanical devices (autonomous agents). An autonomous agent is a system that continuously perceives its environment through the robotic sensors, analyzes the percept sequence using various AI techniques and takes actions through the robotic actuators with the goal of maximizing a utility function. The central problems of an autonomous robotic agent include environment perception, robot localization, robotic mapping, path planning, decision making under uncertainty, multi-agent planning and learning, and robot coordination. Therefore, the creation of the desired functionality for an autonomous robot is a complicated procedure, which lies on addressing each one of the basic AI problems and integrating these approaches into a single entity. Consequently, debugging of the code of a robotic software framework that implements autonomous rational agents has two key requirements :

- *individual inspection* of each module of the code, so as to test the performance of isolated functionality, and
- *joint inspection* of the interaction between modules, so as to test the overall performance of the integrated system.

Obviously, the general purpose debugging techniques are inadequate in autonomous robot software application development. The tracing debugging is prohibitively time consuming and the breakpoint technique is most likely to lead to robotic hardware damages, due to the real-time aspect of code execution on the embedded system. Several techniques have been developed to facilitate the debugging in autonomous robot software. The three most preferred are the simulation, the online, and the offline monitoring and are briefly presented below.

The method of simulation lies on imitating the execution of the code of the robotic software framework over time without depending physically on the actual hardware platform. The robots and their environment are modeled in 2D and/or 3D graphical views and scenes are rendered in the simulator application. Virtual robots are theoretically capable of emulating the behavior of their real counterparts, since the simulator is developed based on a model which represents the behavior of the selected physical system as accurately as possible. The major advantage of simulated execution is the disengagement from the robot hardware. Since the developer is able to write, test, and debug

code within a simulated environment, a significant part of time is gained and hardware damages are avoided. The considerable disadvantage is that simulation uses simplifying approximations and assumptions about the real world model, which may not hold in the real environment. It is very difficult to model all the physical features and constraints of the real world, since robotic environments are typically partially observable, stochastic, and dynamic. Furthermore, simulation is insufficient in fully modeling the real-time constraints imposed by the hardware platform.

The technique of online monitoring is based on the continuous, real-time inspection of the robotic behavior via a communication network connection, while the code is executed on the actual hardware platform. The data from the most significant sensors, the outputs of the intermediate modules, and the final outcomes of decision making are visualized properly on a graphical user interface to facilitate the developer in debugging the code of the robotic software framework. The basic advantage of online monitoring is the real-time execution of the code on the robot and consequently the reliability of the monitored data, which would be impossible to reproduce accurately on a simulator. Its drawback is that this kind of debugging burdens the normal execution time with additional execution time.

The offline monitoring is a combination of the simulation and the online monitoring. The most-important information is stored in log files during the code execution on the robot and afterwards the behavior of the agent is replayed on the virtual environment using the data logged from the actual robotic system. The advantage of the offline monitoring is the disengagement from the hardware platform and the reliability of the logged data. However, the synchronization of the logged data is a complicated task and the storage of lengthy executions can introduce important space constraints.

3.3 Kouretes Software Debugging Requirements

As mentioned in Section 2.2.1, the robotic software framework of the Kouretes team consists of a collection of XML-specified Monas modules, Provider modules, and/or Statechart modules that synthesize the complex robot software. Even though the software architecture allows for a nice decomposition of the autonomous robot software development problem into separate perception, cognition, localization, action, and coordination software modules, still the interaction of all the individual modules is crucial for the

overall robotic behavior. Thus, the need for individual inspection of each one of them arises, so as to test their functionality in isolation, but also the need for joint inspection, so as to test the functionality of the integrated system.

An autonomous robot soccer player maintains a set of internal beliefs which are continuously updated by applying various AI algorithms on the percept sequence of the robot. The aforementioned set concerns the robot's belief about the presence or absence of objects of interest in the camera view, the belief about the position of the ball in the field, the belief about the type and position of the field landmarks, its current location in the field, as well as the ones of its teammates and its opponents, the belief about the kinematic and dynamic state of its body, and the belief about the presence or absence of any static or dynamic field obstacles. These beliefs are of great importance, since they contribute to the overall behavior of the robot player. However, during deployment the developer can only partially infer something about these beliefs simply by knowing how the robot makes decisions and observing the robot's actions at real-time. Apparently, such an approach is limited, especially when the update of beliefs has not been tested thoroughly and the developer is trying to debug it. An alternative, which facilitates this testing and debugging process without sacrificing the real-time aspect of deployment in the real world, is to extract the internal beliefs of the robot and visualize them in some intuitive way in real-time on an external computer, so that the developer can monitor closely the belief update process as well as the related decisions.

The belief about visible objects is of crucial importance, because the camera is the main source of information about the environment. Object recognition relies on correct identification based on color, shape, and size and, apart from the type of the recognized object, it returns estimates of distance and bearing. Given the diversity of data involved in this process, different visualization views may be required to assist the developer in testing and debugging the vision module. For example, sometimes it may be useful to monitor the raw camera image or the color-segmented image in order to calibrate the camera and tune the color recognition process, whereas other times it may be useful to monitor the span of the camera view and the observations extracted from the camera image (object type, distance, and bearing) and relate them directly to the current location of the robot in the field.

The filtering process for maintaining a reliable belief about the state of the world is also of crucial importance for any autonomous robot. The robot's belief about its

current location (position and orientation) in the field is a key input to other robot software modules related to behavior and coordination. This is also true for the belief about the position of the ball in the field, which is based on sporadic visual observations. Consequently, observation filtering errors may result also in further errors with potentially fatal consequences. Therefore, the need for visualizing and monitoring ball and landmark observations, the filter elements (e.g. particles, Gaussian distributions, etc.), the beliefs themselves, the estimated locations, and the trace of estimated locations for each robot separately is obvious.

The belief about the presence or absence of obstacles is important in avoiding collisions and planning appropriate paths to any desired destination. The monitoring of the range data, the probabilistic map, and the path planning outcome impose new visualization constraints which can hardly be integrated with the ones of the filtering process in a clear and intuitive way. Therefore, a separate view that highlights all the aspects of obstacle avoidance in detail may be much more informative for the developer.

The measurements of the robot sensors, such as the joint encoders, the accelerometer, the gyroscope, and the force sensitive resistors, carry significant information about the kinematic and dynamic state of body of the robot. The monitoring of these measurements conveys useful information to the developer for a number of useful tasks, such as inspecting the performance of a predefined special action, verifying the robot kinematics, testing a dynamic robotic walk, testing the detection of a fall, etc.

Last, but not least, the robot maintains identification information (host name, player number, team number), as well as information about the game state broadcast by the game controller or delivered by the button interface (team color, game and player state). In the process of debugging, the developer needs a way to identify robots uniquely and a way to monitor their belief about the game state. The overall behavior of a robot is significantly affected by this belief, which may be different than expected due to frequent broken connections to the game controller or mistaken pushes on the robot buttons. Monitoring closely this information is required to interpret correctly and understand the behavior of the robot.

The discussion above focused mainly on visualizing and monitoring the beliefs of a single robot. However, in a RoboCup game there are several robots in each team and a key problem studied in such a multi-agent settings is that of coordination. Therefore, it is important to be able to assist the developer in testing and debugging the software

modules related to coordination by providing a different view with the most relevant pieces of information. This view may include the beliefs about self-location and ball location of any subset of robots, the shared belief about the location of the ball, the layout of an agreed formation in the field, etc. This way the developer will be able to study, test, and debug the global team behavior at a high-level.

A last requirement towards a functional visualization and monitoring application would be the integration of the functionality and views mentioned above under a single intuitive graphical user interface. Under this design principle the user will be able to switch quickly between different views to monitor different aspects of the robot software. A fully integrated visualization and monitoring application will facilitate the debugging of the entire robot code and will allow the developer to not only assess each module individually, but also jointly as components of a larger integrated software system.

Chapter 4

Related Work

4.1 B-Human

The B-Human team [12], the joint RoboCup team of the University of Bremen and the German Research Center for Artificial Intelligence (DFKI), has developed the SimRobot simulator to deal with the needs of monitoring and debugging. SimRobot is a physical robotics simulator which visualizes 3D simulated SPL games in which both the environment view and the robot views are modeled in the 3D space. It also accomplishes offline monitoring based on log files which can be replayed on the corresponding scene view of the tool and also provides online monitoring, since the developer can be connected directly on the actual robot via LAN or WLAN to inspect the interesting features.

As shown in Figure 4.1, the main application window consists of the scene graph on the left pane, the central 3D scene view, the information views on the right pane, and the command window at the bottom pane. There are two types of scene description files which can be used to define the central scene view: the ones required to simulate one or more robots, such as `BH2011`, `BikeScene`, `Game2011`, `MultiplePlayers2011`, `OpenChallenge2011` and the ones that are sufficient to connect to a real robot or to replay a log file, such as `BikeSceneRemoteWithPuppet`, `RemoteRobot`, `ReplayRobot`, `ReplayVideo`, `ScriptRemoteRobot`, `TeamComm3D`, and `TeamComm`. There are 10 kinds of information views: image views, which display the raw camera image and additional information in the coordinate system of the camera; field views, which display information in the global coordinate system of the soccer field; plot views, which display debug drawings or plots received from the robot; color space views, which visualize certain color

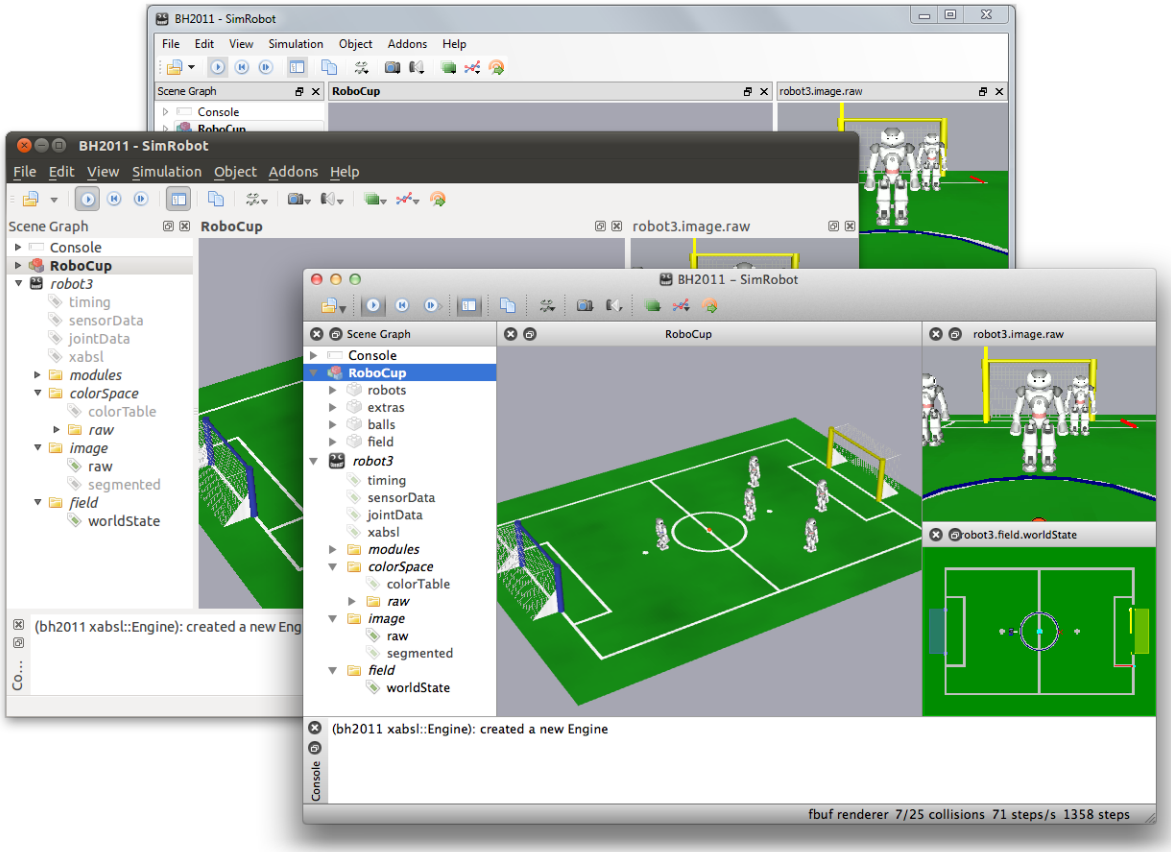

Figure 4.1: SimRobot simulator

channels or the current color table; the Xabsl view, which visualizes specific information about the current state of the robot's behavior; the sensor data view, which represents the robot's sensor readings; the joint data view, which presents the values of the joints; the timing view, which visualizes the timing of the modules it executes; the module view, which presents the module configuration; the kick view, which is actually a kick editor. The user is able to select his/her preferred information view from the scene graph and define which debug drawings to plot on the selected view via various console commands.

Finally, SimRobot also provides logging functionality, which is accomplished in two ways. On one hand, the user can record a log connecting SimRobot to a Nao, define the requested representations via console commands, and then the simulator stores the received data to a specific file and utilizes it for the monitoring process. On the other hand, the user can save log data without the need of a connection to the robot. The

requested debug messages are stored in a special `MessageQueue` called `LogQueue`, which saves its data to a file, when a given threshold is exceeded, in order to prevent the loss of data, because of a full queue.

4.2 Nao Devils Dortmund

The Nao Devils Dortmund [13] is a RoboCup team hosted at the Robotics Research Institute of TU Dortmund University (part of the ex joint team BreDoBrothers, the cooperation of the University of Bremen and the TU Dortmund University). Nao Devils utilize SimRobot in order to simulate their developed robotic software.

Furthermore, they have developed a tool which accomplishes offline monitoring by logging each behavior decision, the resulting information about the world model, and the corresponding sensor information which led to the specific decision during the execution. In order to compare the logged data with reality, a video camera is used to record the real game in a movie file. Thus, the debugging process of Nao Devils proceeds as follows: the robot logs its internal state machine, the required symbols, and hardware commands once per frame; a graphical user interface allows for an easy reading of the logfile and shows the input/ output symbols and the current XABSL state tree; a 2D field view shows the modeled robot position, direction, and velocity, the ball position, and the team mate positions; a video can be loaded and displayed and the time is synchronized manually by matching known state information to the video file. Finally, the behavior log can be played back, stopped, stepped through frame by frame, and rewound to interesting positions. A snapshot of the debugging tool is shown in Figure 4.2.

4.3 TT-UT Austin Villa

TT-UT Austin Villa [14] is a joint RoboCup team from the Department of Computer Science at The University of Texas at Austin and the Department of Computer Science at Texas Tech University. They utilize Webots [15] as the basic simulator for the visualization of their code system and additionally they have developed their own debug tools . Even though the documentation of their debug tools is somewhat unclear, they claim that their debugging process proceeds as follows: at each time step only the contents of the current memory are required to make the logical decisions, so a “snapshot” of the current

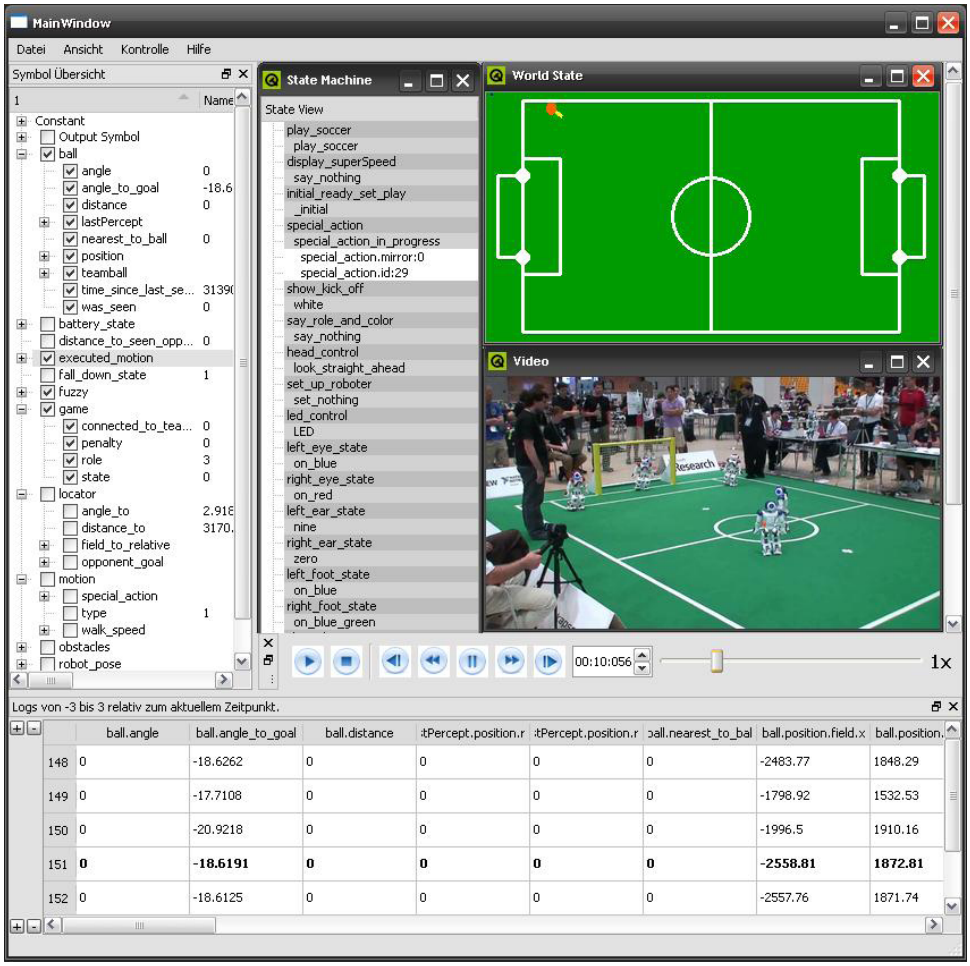

Figure 4.2: Nao Devils debugging tool

memory is saved to a log file (or is sent over the network); the recorded log is examined within their debug tools to discover any problems. Each tool is able to read and display logs, but can also take logs and process them through the logic modules. Their suite of debug tools includes the vision tool (Figure 4.3, left), the localization tool (Figure 4.3, right), the behavior tool (Figure 4.4, left), and the kick region tool (Figure 4.4, right).

4.4 Nao-Team HTWK

Nao-Team HTWK [16] is another German RoboCup team from Leipzig University of Applied Sciences. As far as their code debugging requirements are concerned, they have

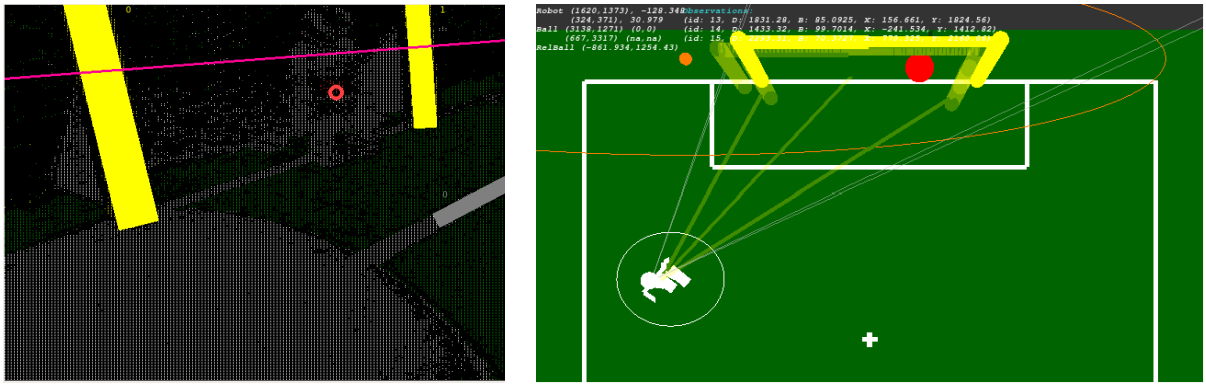

Figure 4.3: Austin Villa vision (left) and localization (right) debugging tools

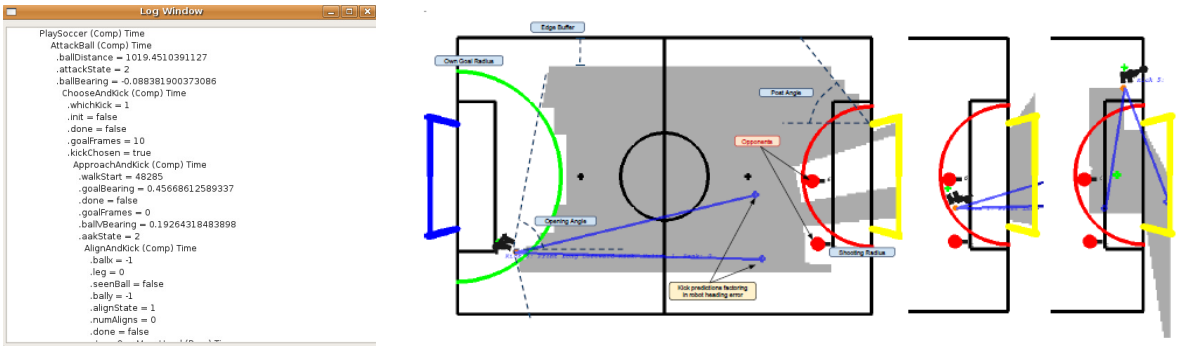

Figure 4.4: Austin Villa behavior (left) and kick region (right) debugging tool

developed NaoControl, a monitoring application which provides a virtual parameterized soccer field, a visualization of the robot's and the estimated ball's position, and the robot's field of view (Figure 4.5). In addition to these features, the raw images from the two cameras of the robot, as well as the segmented ones, can be visualized. Finally, NaoControl supports sending commands to the real robots for testing purposes. According to the team, NaoControl is ongoing work still in progress and in the near future it will be enhanced with simulation tasks.

4.5 Dutch Nao Team

The Dutch Nao Team [17] consists of undergraduate students focusing on Artificial Intelligence from the University of Amsterdam in the Netherlands. The team has developed a localization and a landmark detection monitoring script using Pygame, a cross-platform

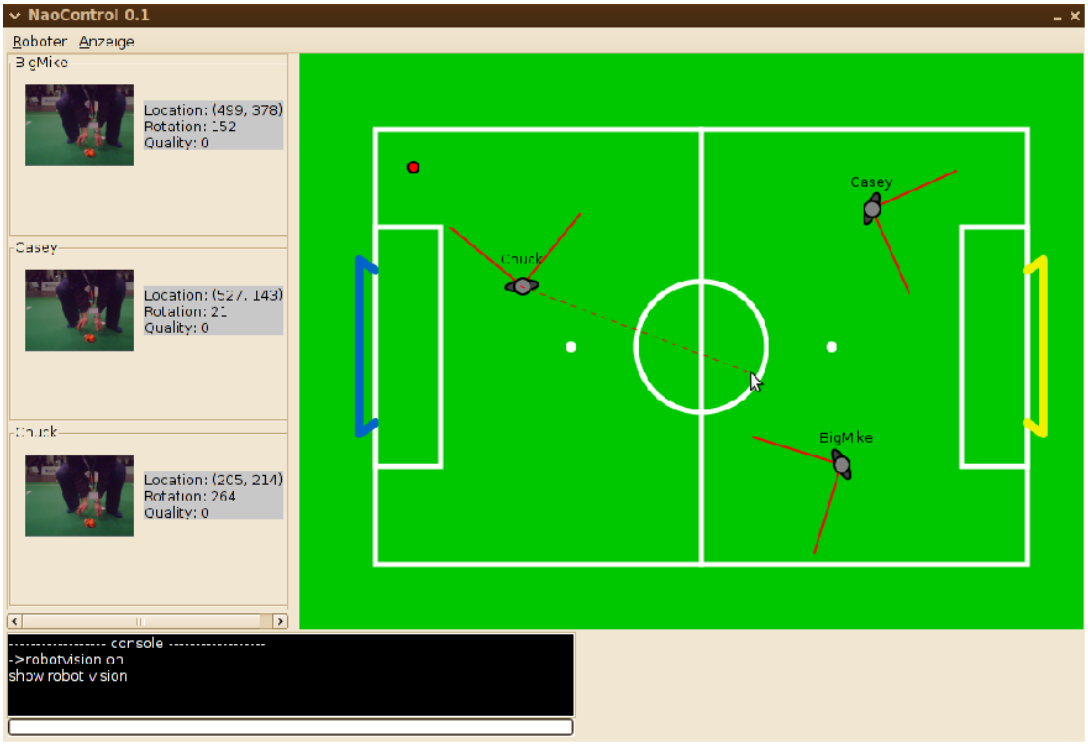

Figure 4.5: HTWK NaoControl

set of Python modules for writing video games. This script offers a visual overview of the belief created by their Dynamic Tree Localization method (Figure 4.6, left) and the goalpost and beacon observations of the robot (Figure 4.6, right). In addition, they utilize USARsim, a simulation environment that uses the local installation of NaoQiSDK to process commands to the Nao robot, which means that the same Python code can be used for the real robot and the simulated robot. The simulation environment consist of two parts: `USARsim`, which represents the SoccerField where one or more robots can be spawned, and `USARNaoQi`, which represents the robot (and translates NaoQi commands into USARsim commands). The graphical interface is depicted in Figure 4.7.

4.6 UPennalizers

UPennalizers [18] is the SPL team of the University of Pennsylvania in the USA. Their actual debugging software focuses on monitoring their vision code. In particular, they receive image packets from an active robot and display them on an external computer.

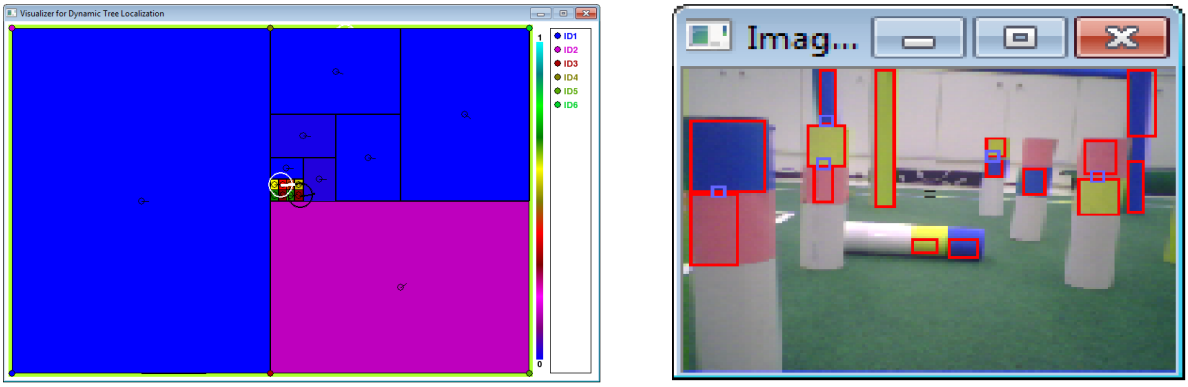

Figure 4.6: Dutch Nao Team monitoring scripts

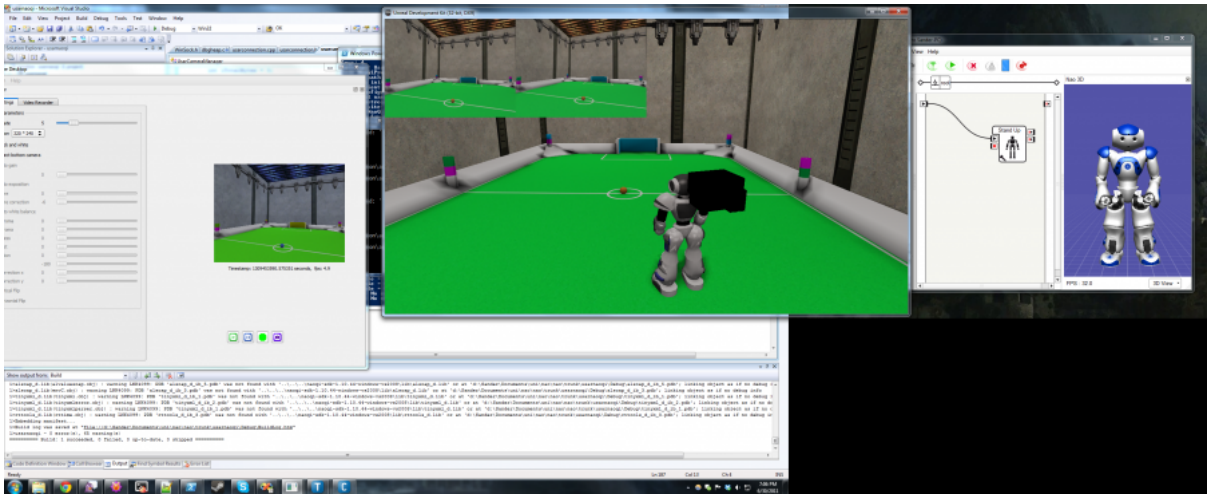

Figure 4.7: Dutch Nao Team USARsim simulation environment

The broadcast images may be either raw or color-segmented images from each camera. They display the received data via a Matlab interface and overlay visual indicators of state, such as ball and goal locations. A snapshot of the UPennalizers debugging tool is shown in Figure 4.8.

4.7 NAO-Team Humboldt

The NAO-Team Humboldt [19] is the SPL team that consists of students and researchers working at the Humboldt University of Berlin. In order to simulate their robot software

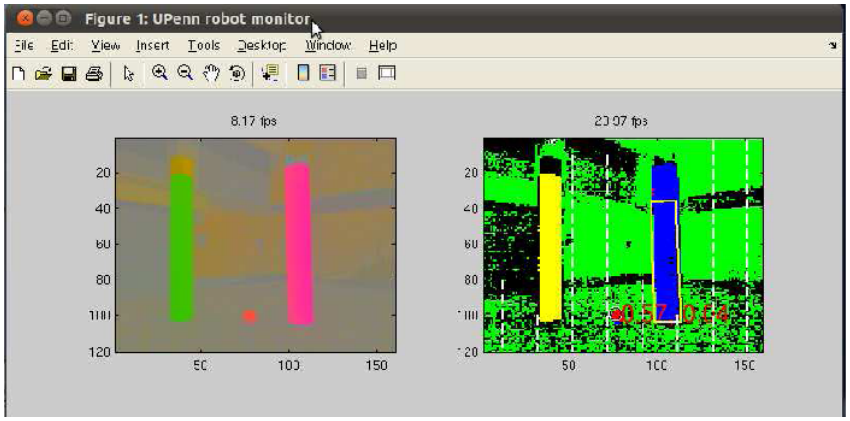

Figure 4.8: UPennalizers monitoring vision tool

framework, they use Webots and SimSpark (the official simulator of the RoboCup 3D Simulation league). They furthermore have developed RobotControl, a robot data monitoring and debugging tool implemented in Java, which visualizes the debugging results that are transferred over the network (Figure 4.9).

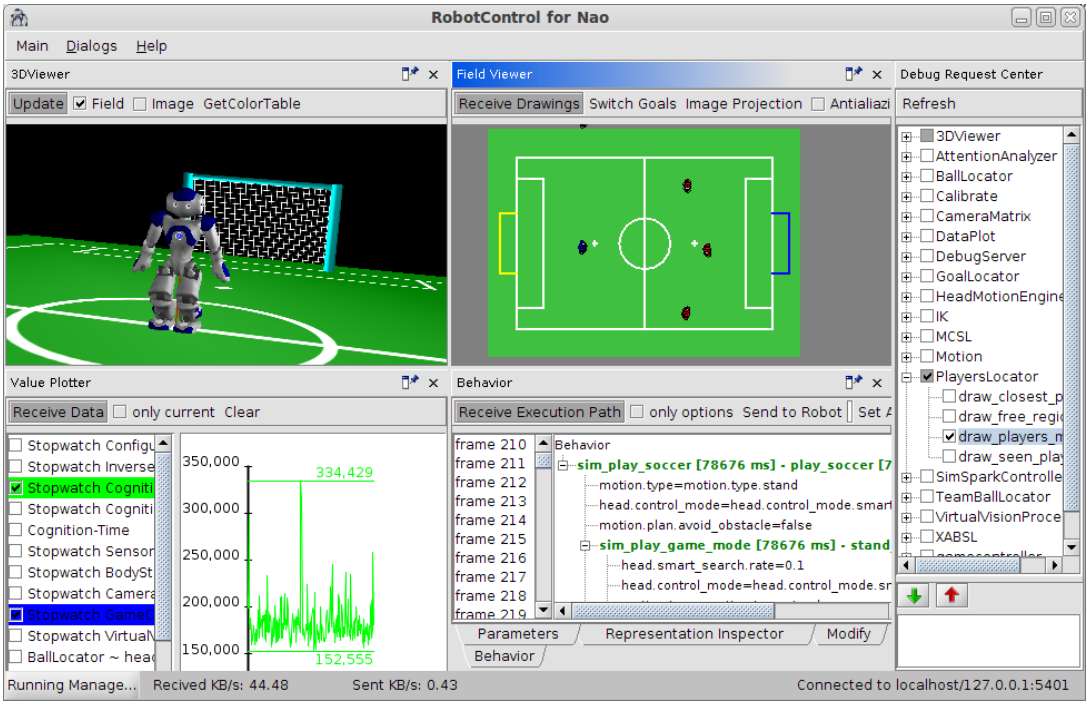

Figure 4.9: NAO-Team Humboldt RobotControl monitoring tool

Chapter 5

Our Approach

The development of an integrated and adequately functional online monitoring tool, which meets the requirements stated in Section 3.3, was always high in the task list of the Kouretes team. To address these visualization and monitoring needs, we have developed a graphical software application, named KMonitor (Kouretes Monitor), which is the subject of this thesis. KMonitor allows the developer to inspect effectively each one of the basic modules of the existing code. The proposed technique to accomplish this goal is real-time visualization of the data each module operates on and the data it exports on a remote desktop computer using a User Datagram Protocol (UDP) multicast network in which robots and computers participate as nodes. Each node in the network is identified as a host with an automatically-generated unique ID and a user-specified host name. Based on this protocol, the significant debugging information, which is structurally defined and serialized using Google Protocol Buffers, is exchanged via messages (datagrams) without creating data paths or other special transmission channels. Originating nodes do not have prior information about the existence or the identity of the receiving nodes. Multicast utilizes the network infrastructure by requiring the source to send a packet only once, even if there are multiple recipients, and then the network nodes are responsible for relaying the messages, if necessary, to all recipients. Thus, and according to the user's requests, the values of sensors and actuators, as well as the outcomes of the vision, localization, and obstacle avoidance activities are designed graphically in a user-friendly intuitive way under different monitoring views. Furthermore, the application provides joint inspection of the interaction between basic software modules on the same or on different robots. Additionally, KMonitor eases any future extension, so as to adopt

possible module enhancements and new graphical elements. Finally, it provides easy XML-based configuration to be consistent at all times with the current parameterization of the code and to catch up with the Robocup SPL rules which are frequently modified.

5.1 KMonitor Architecture

The architecture of KMonitor is represented graphically in Figure 5.1. The debugging utilities of the robotic software framework have been split into six categories, according to the debugging relevance of the existing activities and the interaction among them. Thus, KMonitor is composed of six distinct tabs, each one of them providing the required debugging functionality for the corresponding category, as described in detail in the Sections below. Each one of the six tabs consists of the modules which implement the actual Graphical User Interface and the `View-Controller` module which manages the flow of incoming data. The `Remote Hosts` and the `Available Elements` modules of each tab provide interactivity between the user and KMonitor, since the user is able to select one or more available hosts (robots) to monitor and several of the available elements to visualize, according to the functionality of the current tab. On the other hand, the `2D Scenes` are passive modules, meaning that they exist as base modules that visualize the dynamically-requested `Graphical Robot Elements` and the user cannot interact with them, but can only monitor the elements that he/she has requested. The `View-Controller` module is responsible for controlling the user requests and making the corresponding graphical elements on the `2D Scene` of the tab visible, using the latest data received from the `Message Allocator` module. The `Message Allocator` module is responsible for starting the multicast thread, checking periodically its buffer for incoming messages, and allocating them to the corresponding `View-Controller` modules. The filtering of the received messages is based on both the user's element choices and the current active tab.

5.1.1 The Message Allocator Module

The debugging data are transmitted over the UDP network using protocol buffer messages (Section 2.2.2). The main functionality of the `Message Allocator` module is to manipulate the flow of the data received by the multicast network, by subscribing to

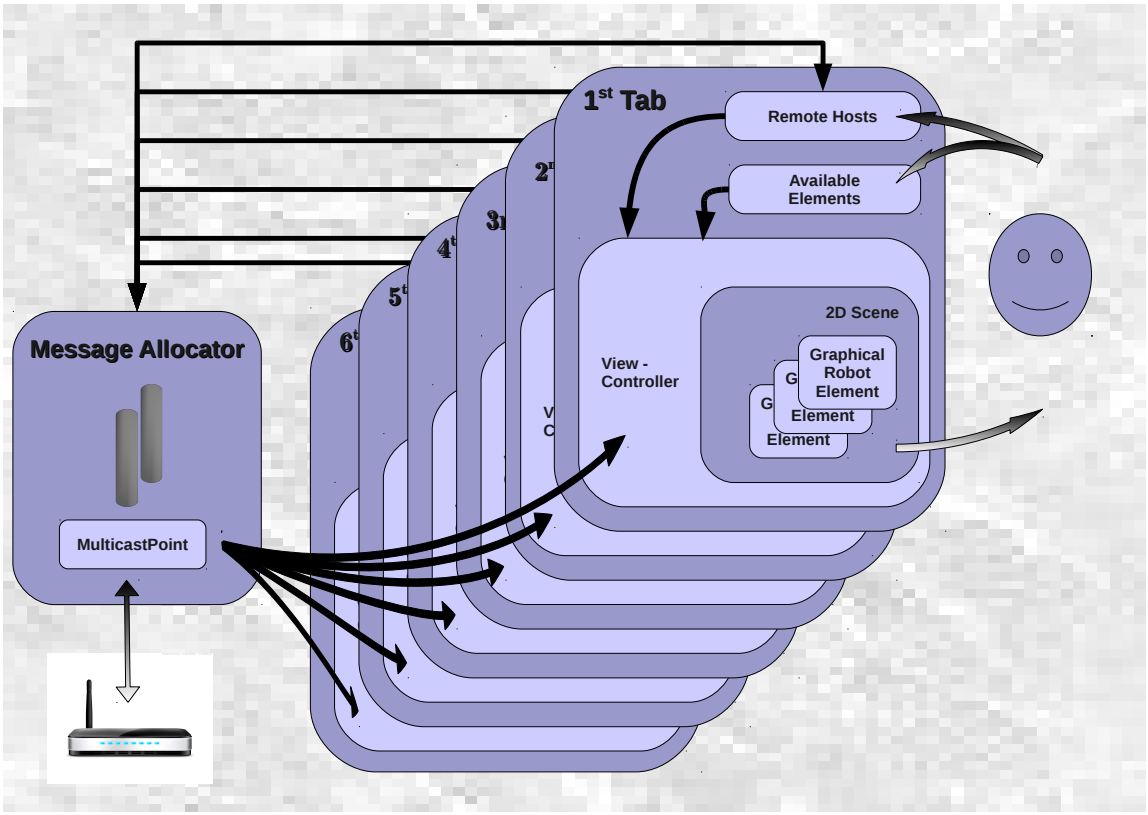

Figure 5.1: KMonitor's architecture

and unsubscribing from certain topics and allocating the received data streams to the corresponding graphical interfaces using the Qt's Signal/Slot mechanism.

The selection of topics to subscribe to or unsubscribe from is based on both the user requests and the current active tab of the Graphical User Interface. Each tab is associated with specific topics based on their relevance to the tab's functionality; whenever a tab becomes active, KMonitor subscribes to those relevant topics and unsubscribes from all other topics. Therefore, the monitoring process becomes faster, since the data flow is limited to the minimum required. The actual (un)subscription and reception is accomplished via the message buffers of the multicast thread of the `Message Allocator` module. The module realizes the required (un)subscriptions through the read buffer of the multicast thread and checks periodically its write buffer for incoming messages. For each received message (if any), it checks the name type and the publisher host and emits the corresponding signal.

5.1.2 The Graphical User Interface

KMonitor is composed of six tabs, each one of which provides the user with various graphical elements to monitor and visualize. These tabs are the following: Global World State, Local World State, Local Polar Map, Local Robot View, Local Sensors Data, KCC Beta. The lay out and the ordering of the tabs was dictated by the frequency of use by the team members aiming to make the interface convenient for the users. The basic element which all tabs contain is the detected hosts from the multicast network. The user can select one or more desired available robots to monitor and check one or more desired graphical elements to visualize, according to the functionality of the current tab.

The multicast module of KNetwork (Section 2.2.2) provides KMonitor periodically with a list of known hosts, that is other robots or computers which listen to this network. The visualization of detected hosts cannot be confined to simple presentation of the hosts as graphical elements, since the user may have already made selections on the GUI which should not be overwritten. In addition, the list may have been updated due to new connections and disconnections. To address this issue, the list of known hosts is updated as follows:

1. *Removal of old disconnected hosts.* The corresponding function iterates through the list which stores the known hosts and for each host encountered it checks if it still exists in the received updated list of hosts. If it exists, it does nothing (to preserve the user selections), whereas if it does not exist, it removes it from the list (along with any associated user selections and graphical elements).

2. *Addition of new connected hosts.* Creation of new graphical element for newly connected hosts. The corresponding function iterates through the received updated list of hosts and for each host encountered it checks if it already exists in the stored list of hosts. If it exists, it does nothing (to preserve the user selections), whereas if it does not exist, it adds it to the list.

For visualization purposes, each host in the list of known hosts is presented in two different ways in KMonitor according to the functionality each tab provides. In tabs where elements from multiple robots can be visualized, the known hosts are presented as a graphical tree widget (Figure 5.2, left), whereas in tabs where elements of a single robot can be visualized, the known hosts are presented as a combo box item (Figure 5.2, right).

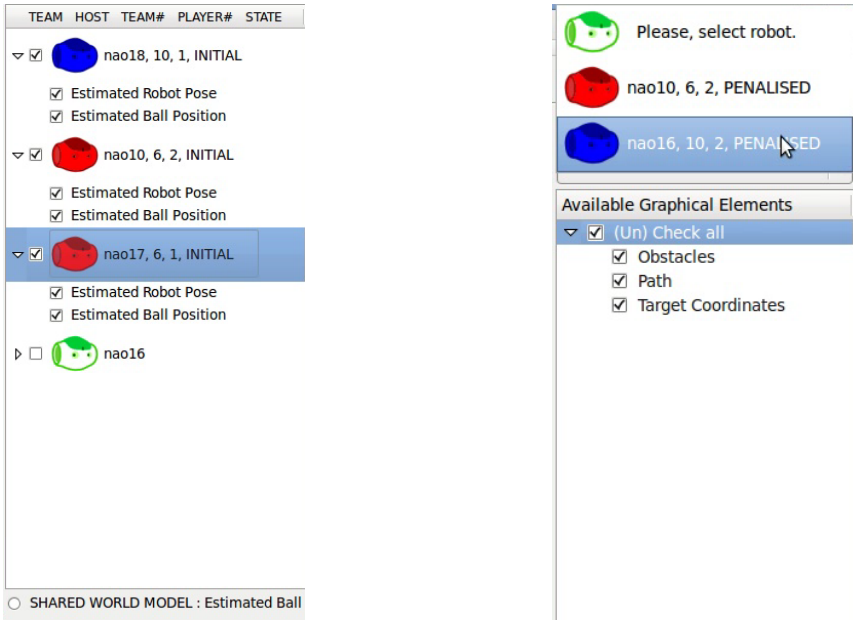

Figure 5.2: Visualization of the known hosts as tree (left) and as combo box (right)

Furthermore, the graphical representation of each host provides information about the identity of the host and the current game state, so as to facilitate the user in distinguishing the different hosts. This information includes the team's color (currently red or blue), the host's name (`nao10`, `nao14`, etc.), the team's number, the player's number, and the player's state (`INITIAL`, `SET`, `READY`, `PLAYING`, `PENALISED`, `FINISHED`, etc.).

A basic requirement for KMonitor is the precise visualization of the soccer field, where most of the provided dynamic graphical elements are depicted. According to the Robocup SPL Rule Book 2012, the soccer field and its static elements must meet certain specifications, as shown in Figure 5.3. As mentioned, configurability is a desired feature for KMonitor in order to conform to future changes in the specification of the field. In order to achieve configurability, all the dimensions and sizes of the field elements, that is side lines, end lines, halfway line, center circle, and the lines surrounding the penalty areas, are parameterized through an XML file. Thus, the corresponding constructor of the graphical 2D field scene loads the parameters from the field configuration XML file, scales their values to fit within the window where the 2D field is visualized taking into consideration the real field dimensions ratio, and positions the required graphical elements to the resulting coordinates. Another functionality which KMonitor provides is

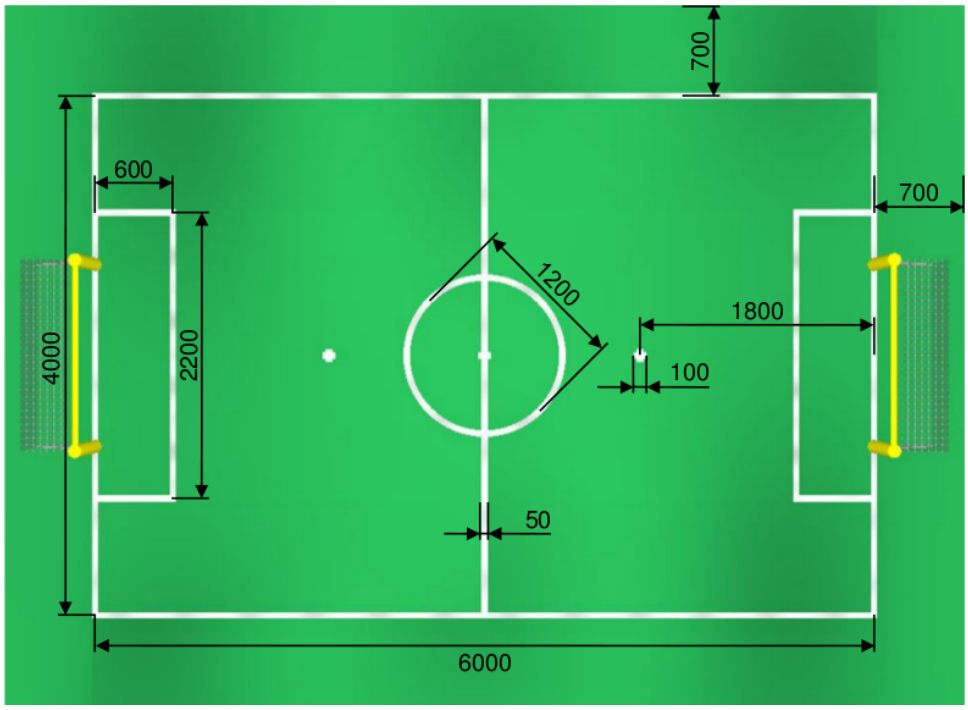

Figure 5.3: The field dimensions (in mm) according to the RoboCup SPL Rules 2012

the adjustment of the graphical field scene to any window resize request posted by the user. Every time a resize to the width or height of the application window takes place, the corresponding slot which is responsible for the recomputation of the coordinates of all the field elements is executed.

The first tab (Global World State) visualizes the global world state in the field and is described in detail in Section 5.2. A snapshot of the first tab in its initialized view is shown in Figure 5.4. It is composed of two graphical sections, the 2D field scene and the dynamic tree widget of detected hosts (robots). KMonitor detects the existence of active robots and, if any, it creates dynamically a tree widget item for each one of them, which contains information about identity and game state. The user can choose any number of robots to monitor and more specifically can choose to visualize two elements for each robot: the estimated robot pose and the estimated ball position. The selected elements are depicted graphically on the field scene.

The second tab (Local World State) visualizes the local world state of a robot and is described in detail in Section 5.3. A snapshot of the second tab in its initialized view

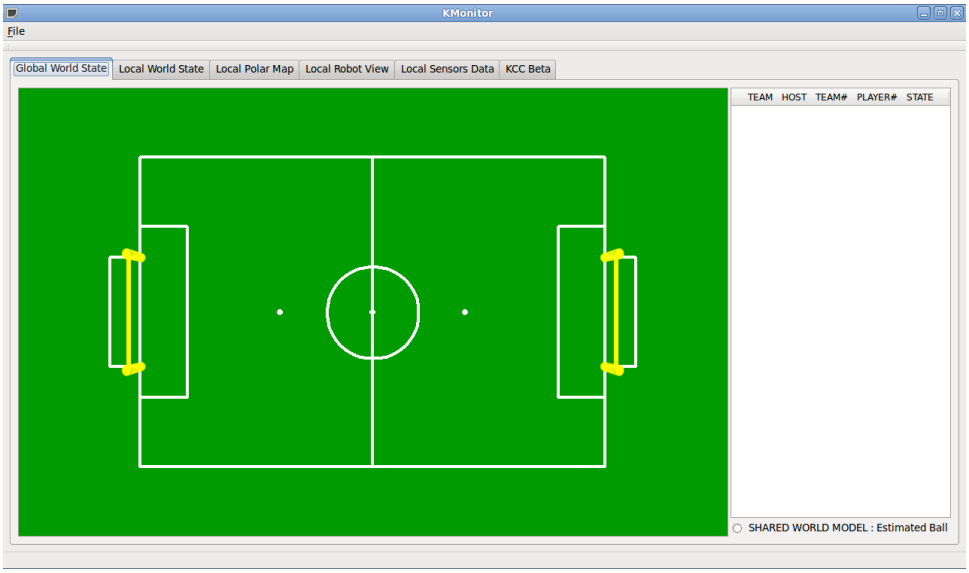

Figure 5.4: The Global World State tab (initialized view)

is shown in Figure 5.5. It is composed of three graphical sections, the 2D field scene, the dynamic combo box of detected hosts (robots), and the widget of available graphical elements. The user is able to select only one robot from the corresponding combo box

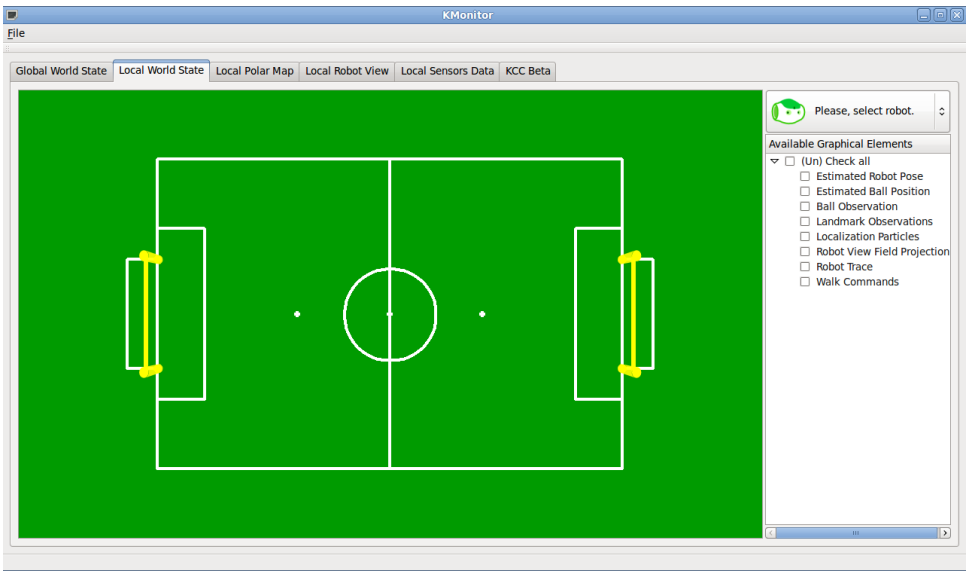

Figure 5.5: The Local World State tab (initialized view)

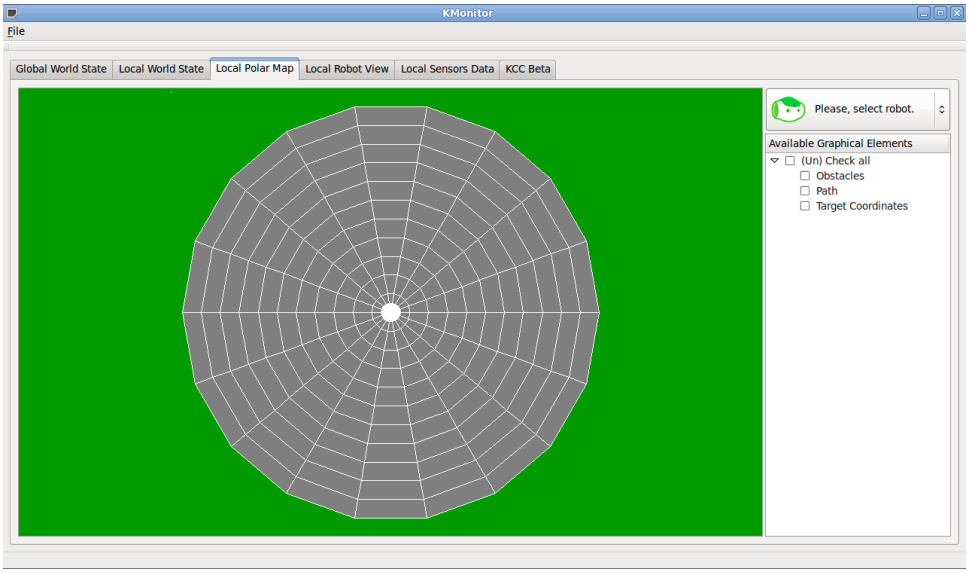

Figure 5.6: The Local Polar Map tab (initialized view)

and any number of elements to monitor, such as the estimated robot pose, the estimated ball position, the ball observation, the landmark observations, the localization particles, the robot view field projection, the robot trace, and the walk commands.

The third tab (Local Polar Map) visualizes the local polar map of a robot and is described in detail in Section 5.4. A snapshot of the third tab in its initialized view containing only the static graphical elements is shown in Figure 5.6. This tab is designed with the same graphical sections as the second tab, except for the content of the widget of available graphical elements. The user can select only one robot to monitor and any of the available elements: the obstacles, the path, the target coordinates.

The fourth tab (Local Robot View) provides the local view from the camera of a robot. A detailed description about this tab is given in Section 5.5. A snapshot of the KMonitor's fourth tab in its initialized view is shown in Figure 5.7. The tab consists of three graphical sections, the graphical label, the dynamic combo box of detected robots and the widget of available graphical elements. The user is able to visualize in real time either the raw or the color-segmented image received by the camera of the chosen robot.

The fifth tab (Local Sensor Data) represents the sensor data measurements of a robot. This tab is described in detail in Section 5.6. A snapshot of the fifth tab in its initialized view is shown in Figure 5.8. This tab is composed of 75 graphical sections: the dynamic

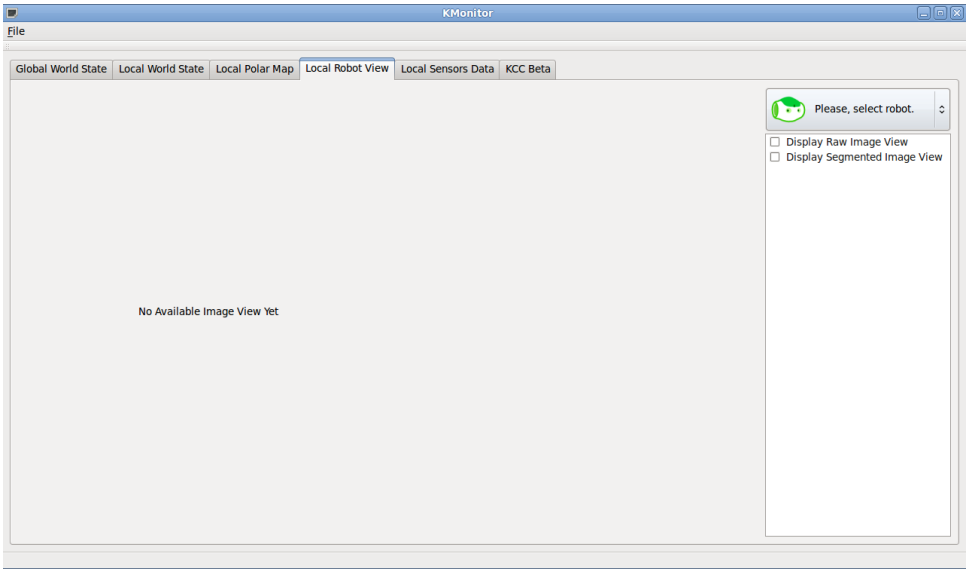

Figure 5.7: The Local Robot View tab (initialized view)

combo box of detected robots, 37 combo boxes, and 37 labels, one for each of the 37 sensors. The user is able to select only one robot from the corresponding combo box and automatically the combo boxes which hold the most recent joint encoder readings,

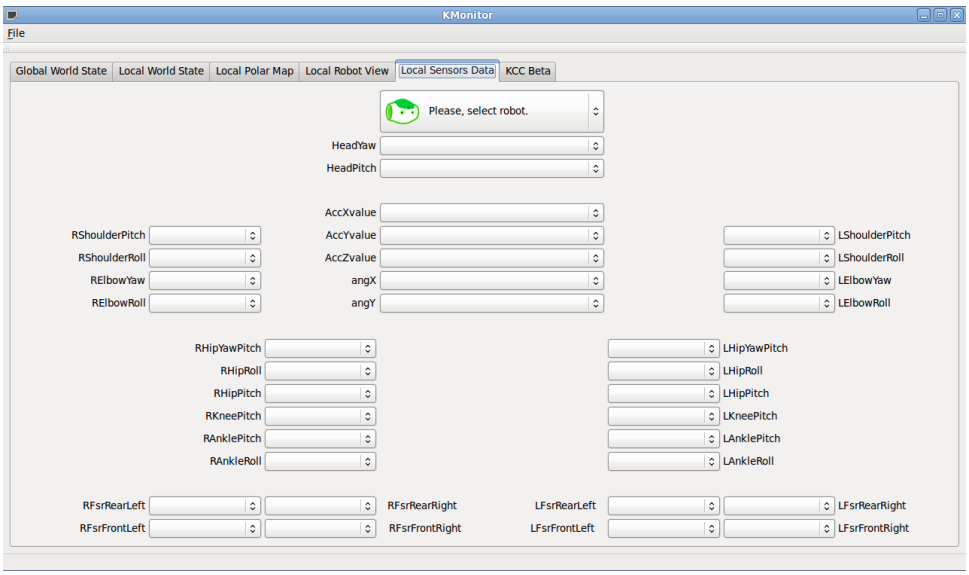

Figure 5.8: The Local Sensors Data tab (initialized view)

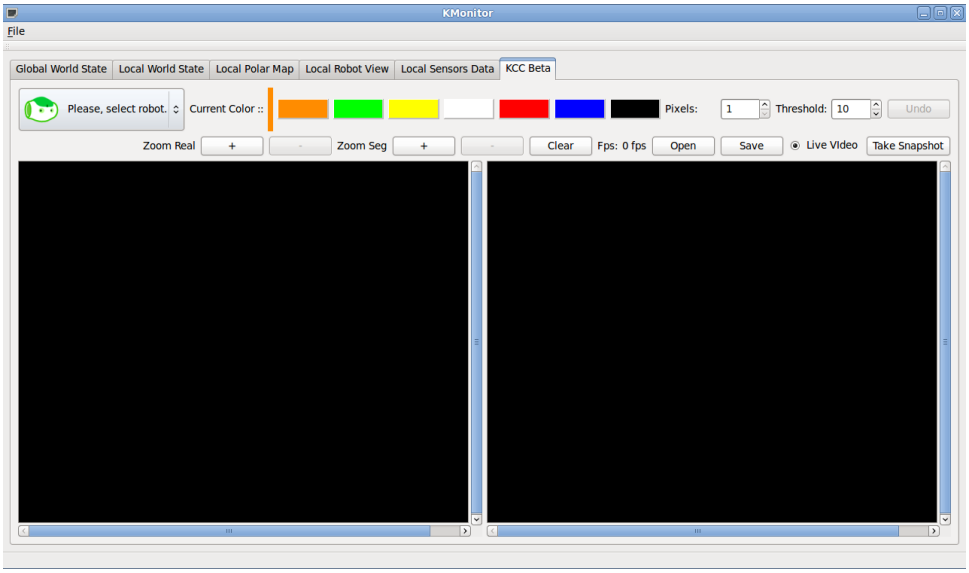

Figure 5.9: The KCC Beta tab (initialized view)

inertial measurements, and values of the FSRs are updated. In addition, the user is able to monitor the 10 latest measurements for each of the sensors by opening the corresponding combo box.

The sixth tab (KCC Beta) provides access to the beta version of the Kouretes Color Classification (KCC) tool [20], which is used for creating color tables. The KCC tool was initially designed as an independent tool, but was recently integrated into KMonitor to offer uniform user access to all debugging facilities. Since it was not developed as part of this thesis, it is not described further. A snapshot of this tab in its initialized view is shown in Figure 5.9.

5.1.3 The View-Controller Module

The `View-Controller` module is responsible for transforming the current user requests from the graphical user interface into actual visualizations of the received debugging data. Whenever a choice is made to the current tab, the corresponding signal is emitted, then the `View-Controller` of that tab is responsible for creating a new object for the requested host, if it does not already exist, call the corresponding functions which compute the position of the element based on the latest data, and visualize the requested graphical element of the robot to the graphical scenes, labels, or views. The graphical

depiction of a robot is a collection of graphical items. Various ellipse, line, and polygon items are utilized to visualize the body of robot, its orientation, the shape of the observed and estimated ball, and many other elements which are examined and described thoroughly in the Sections below. The `View-Controller` module is responsible for updating the corresponding structures with the latest data received from the `Message Allocator` module for each of the `Graphical Robot Elements`. Furthermore, it is responsible for managing the visibility of all the depicted elements.

5.2 The Global World State

The first tab, called the Global World State, provides global monitoring of all active robots on the soccer field. It visualizes the estimated robot pose and the estimated ball position for each robot. It can be used mainly for debugging software modules related to behavior and coordination. These modules combine the individual robots' beliefs about their pose and the position of the ball in the field to infer a shared world model, which is subsequently used to make decisions at the team level. Therefore, it is important for the developer to be able to monitor multiple robots concurrently and their individual beliefs to understand and debug the shared world model and the related decisions (roles, formations, etc.). In order to avoid confusing the user with multiple robots and balls simultaneously in the field, a connecting line is set visible automatically between the two elements (pose and ball) of a single robot, if the user has requested the visualization of both elements. Furthermore, if the user points the mouse pointer to the graphical depiction of a robot or ball in the virtual field, a tooltip pops out containing the player number of the corresponding robot.

5.2.1 Visualization of the estimated robot pose

The debugging message about the world state sent periodically by the robot via the UDP channel contains the belief of the robot about its pose as a probability distribution over the 3-dimensional space of (x, y, θ), where x, y, and θ refer to the global coordinate system of the real field shown in Figure 5.10 (left). In order to monitor the current estimated robot's position (x, y), the corresponding slot scales the (x, y) coordinates from the real field coordinate system to the virtual field coordinate system shown in Figure 5.10 (right)

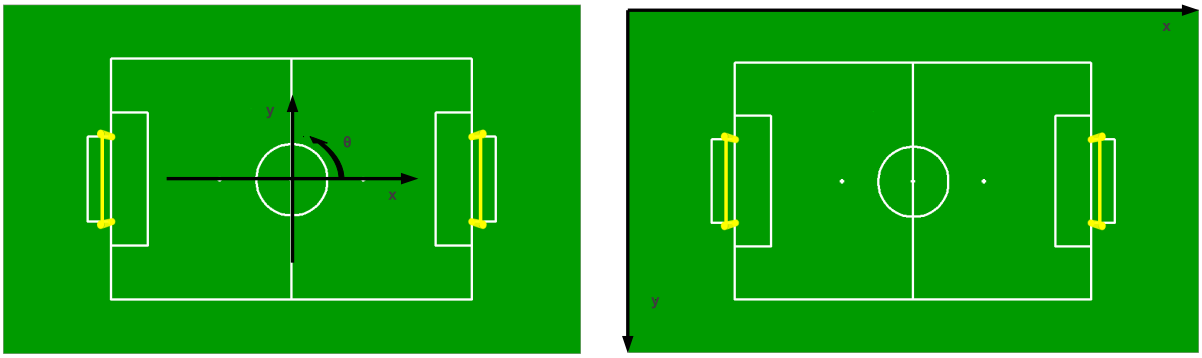

Figure 5.10: Global coordinate systems of the real (left) and the virtual (right) field

and draws the graphical robot element in the same color as the current team color of the robot. In addition, the slot draws a small line segment at angle θ to indicate the current robot orientation. A snapshot of the visualization of the robots' poses is shown in Figure 5.11.

5.2.2 Visualization of the estimated ball position

The debugging message about the world state also contains information about the estimated ball position; this message may be empty, if the ball model has been reset due to lack of recent ball observations. If the estimated ball position is not empty, the message provides data for the relative ball position (rbx, rby) with respect to the robot's pose. Thus, the handling function computes the actual ball position in the real field coordinate system by combining the estimated robot pose (x, y, θ) with the received (rbx, rby) coordinates, scales the output to the virtual field coordinate system, and draws graphically the ball element in the same color as the current team color of the corresponding robot. In case the user has requested the visualization of both the robot's pose and the ball position, a connecting line that joins the two graphical elements is drawn too, as shown in Figure 5.11.

5.3 The Local World State

The second tab, called the Local World State, provides monitoring of a single active robot in the soccer field. It visualizes the estimated robot pose, the estimated ball position,

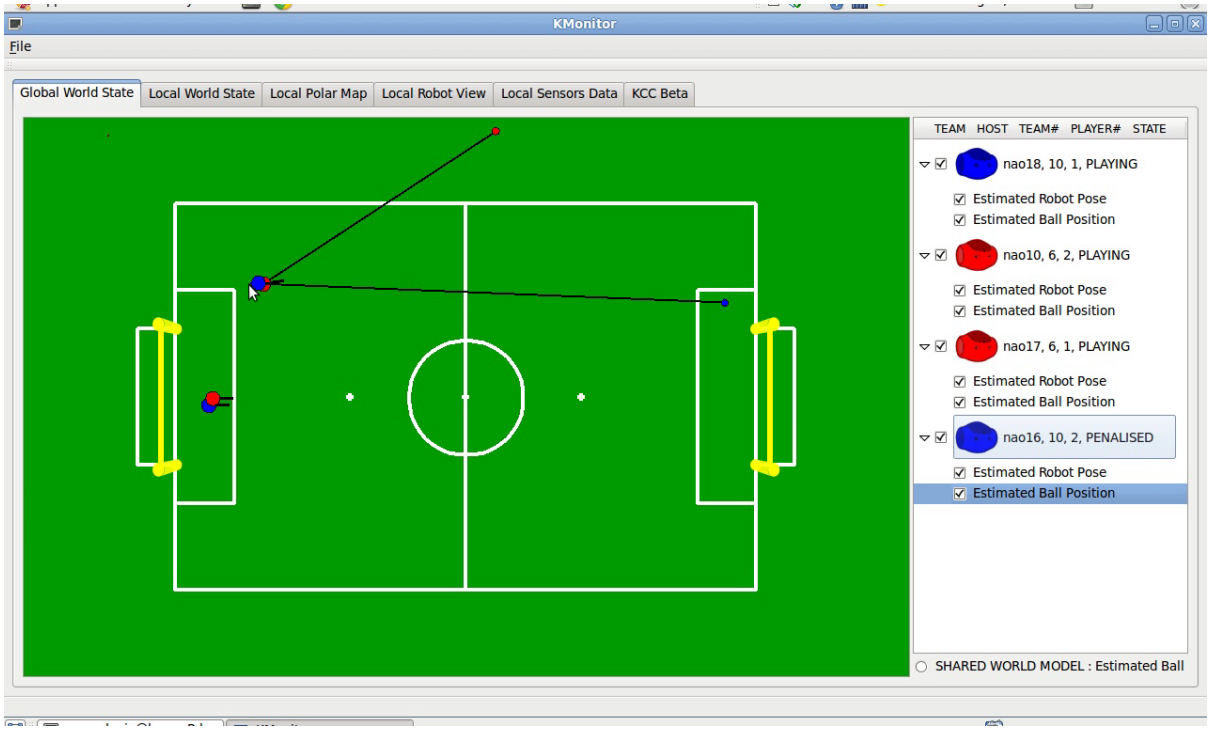

Figure 5.11: Visualization of the estimated robot poses and ball positions

the ball observation, the landmark observations, the localization particles, the robot view field projection, the robot trace, and the walk commands. It can be used for debugging software modules related to a single robot. The estimated robot pose and the estimated ball position have been implemented in the same way as described in Section 5.2.1 and Section 5.2.2 respectively.

5.3.1 Visualization of the ball observation

The instantaneous observations of the ball through the camera are significant data for maintaining a reliable belief about the position of the ball. The debugging message about recognized objects delivers the current ball observation, if any, in the egocentric polar coordinate system of the robot in terms of distance d and bearing ϕ. Thus, the handling function computes the actual position of the observed ball in the real field coordinate system by combining the estimated robot pose (x, y, θ) with the received (d, ϕ) coordinates, scales the output to the virtual field coordinate system, and draws

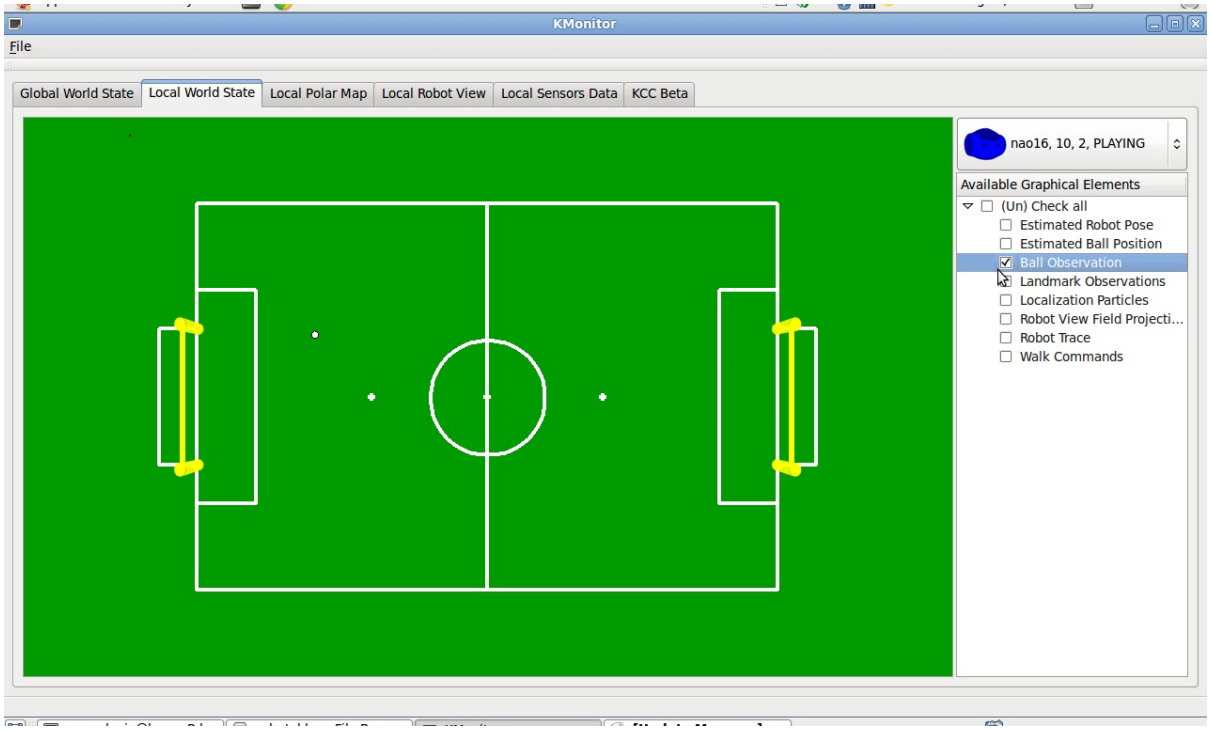

Figure 5.12: Visualization of the instantaneous ball observation

graphically the ball element in white color to differentiate it from the graphical element of the estimated ball position as shown in Figure 5.12.

5.3.2 Visualization of the landmark observations

The instantaneous observations of field landmarks through the camera are significant data for maintaining a reliable belief about self-location. The current layout of the SPL field offers several landmarks (goals, goalposts, lines, center circle, penalty marks) which can be used for self-localization in the field. Our current vision module detects only goalposts. In certain situations, a goalpost can be identified as a left or as a right goalpost, but in many cases it is only identified as an ambiguous goalpost without any indication of left or right. In addition, since in the current SPL field both goals are yellow, there is no way to break the symmetry and distinguish between the own and opponent goal. The debugging message about recognized objects delivers the current landmark (goalpost) observations, if any, in the egocentric polar coordinate system of the robot in terms of

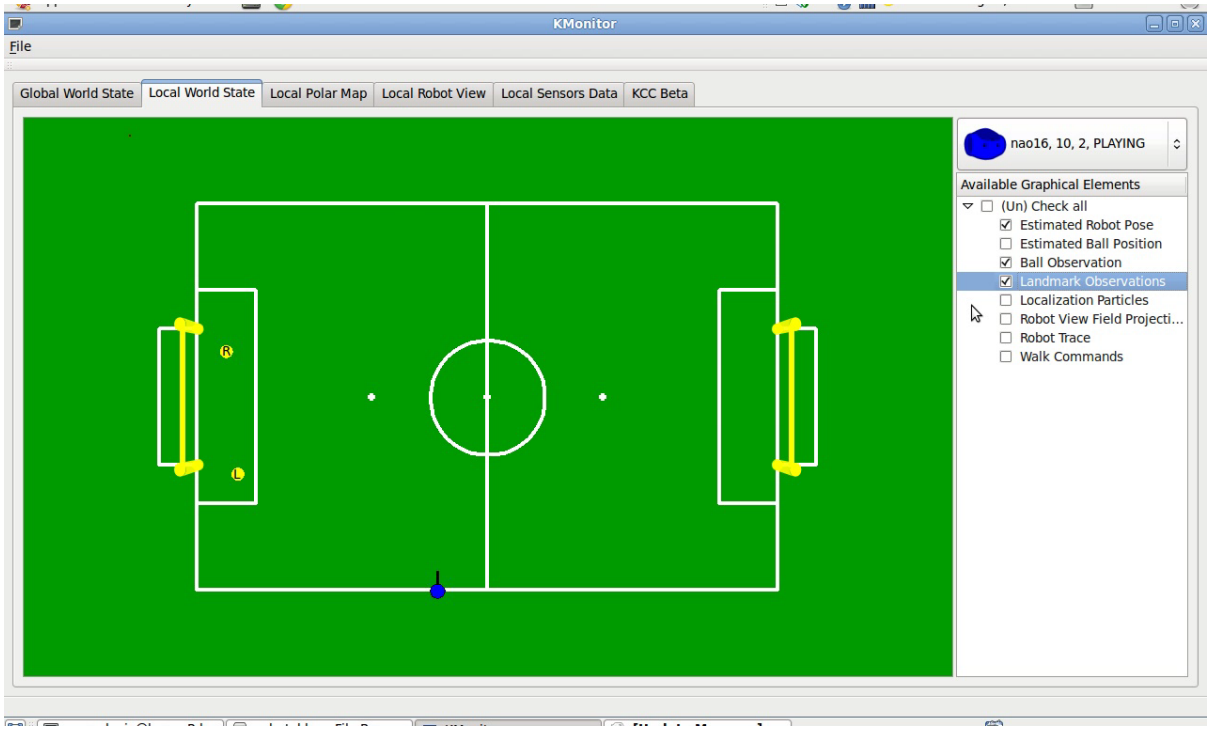

Figure 5.13: Visualization of the instantaneous landmark observations

landmark type (`YellowLeft`, `YellowRight`, `Yellow`), distance d, and bearing ϕ. Thus, the handling function computes the actual position of the observed landmark in the real field coordinate system by combining the estimated robot pose (x, y, θ) with the received (d, ϕ) coordinates, scales the output to the virtual field coordinate system, and draws graphically the appropriate goalpost element according to its type:

- `YellowLeft`, a circular item tagged with an `L` indicating a left goalpost
- `YellowRight`, a circular item tagged with an `R` indicating a right goalpost
- `Yellow`, a circular item with an `A` indicating an ambiguous goalpost

A visualization of the landmark (goalpost) observations is shown in Figure 5.13.

5.3.3 Visualization of the localization particles

In order to monitor the auxiliary particle filtering which implements the update of the belief about self-location, visualization of the population of particles has been realized.

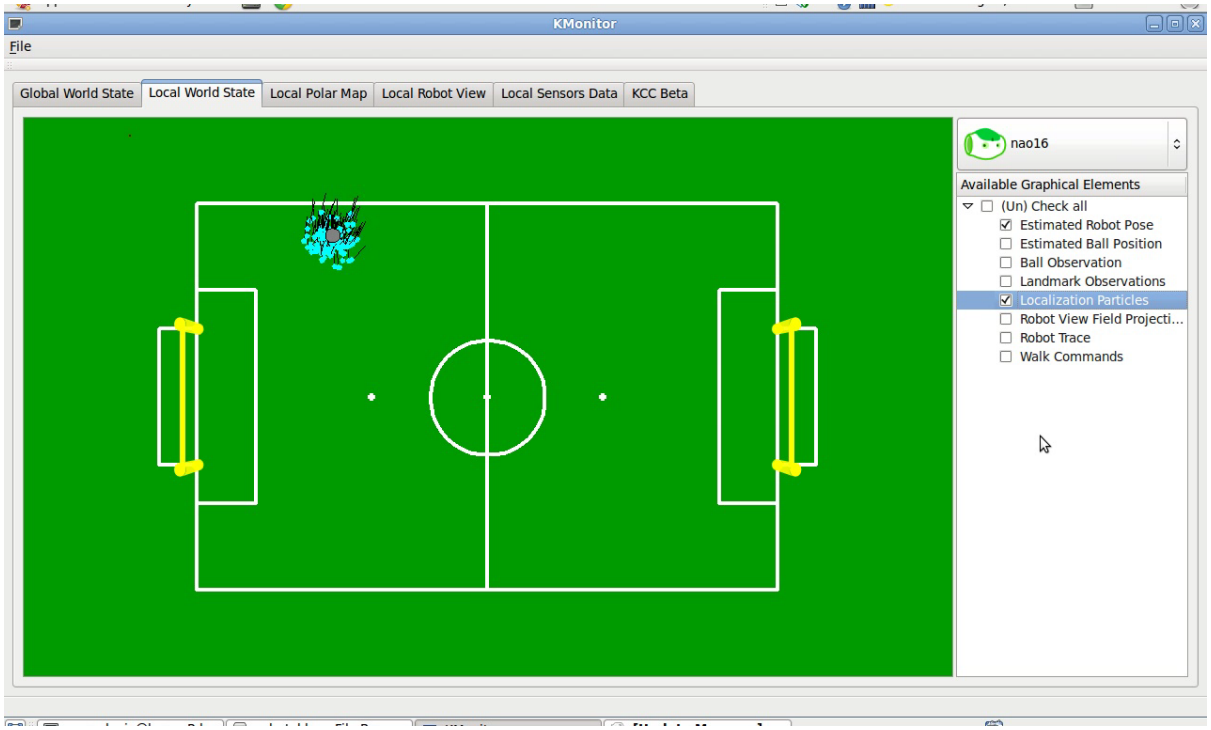

Figure 5.14: Visualization of the localization particles

The number of particles used and the pose (x, y, θ) for each particle are provided by the debugging message about localization which is received periodically. Since the data received for each particle are the same as those for the estimated robot pose, the handling function treats each one of them exactly in the same way drawing one graphical element for each of them with the difference that the circular element is much smaller and has a distinct color (cyan). Furthermore, the handling function can handle any number of particles, since the size of the population is provided dynamically through the debugging message. A visualization of the population of particles is shown in Figure 5.14.

5.3.4 Visualization of the robot view field projection

The robot's current visual field is crucial for the detection and identification of known objects, such as the ball and the goalposts. Its visualization aims in monitoring the robot's view as a 2D field projection, which is approximated by a graphical polygon (trapezoid) item. The received debugging message provides information about the distance

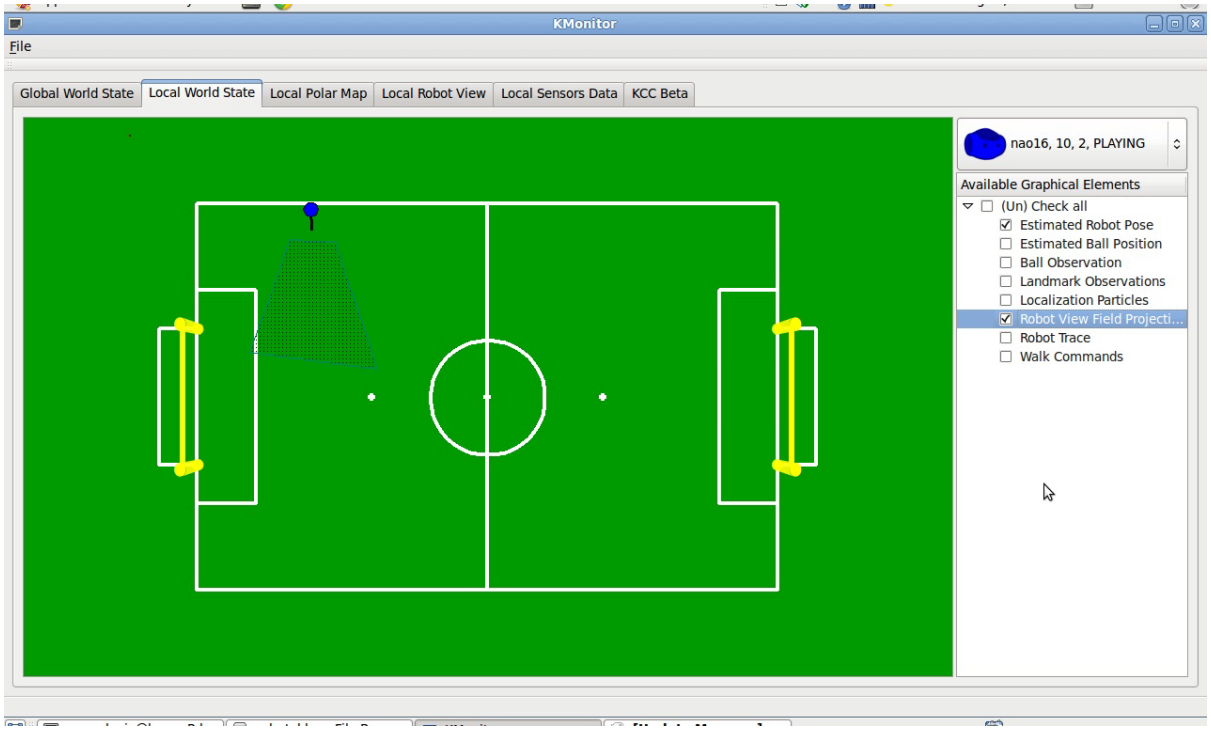

Figure 5.15: Visualization of the robot view field projection

and the bearing of the four vertices of the polygon in the egocentric coordinate system of the robot. The handling function computes the position of all vertices in the real field coordinate system by combining the estimated robot pose (x, y, θ) with the received coordinates, scales the output to the virtual field coordinate system, and draws graphically the appropriate polygon element by providing the four vertices. A visualization of the robot view field projection is depicted in Figure 5.15.

5.3.5 Visualization of the robot trace

The sequence of the most recent estimated robot positions is referred to in this thesis as the robot trace. Monitoring the robot trace is essential in assessing the consistency of the estimated robot poses and interpreting the spontaneous outlier observations. In order to monitor the robot trace, the handling function stores the most recent (by default, 100) estimated robot poses to represent them graphically whenever the user makes the corresponding request. For simplicity, only the estimated robot positions are visualized

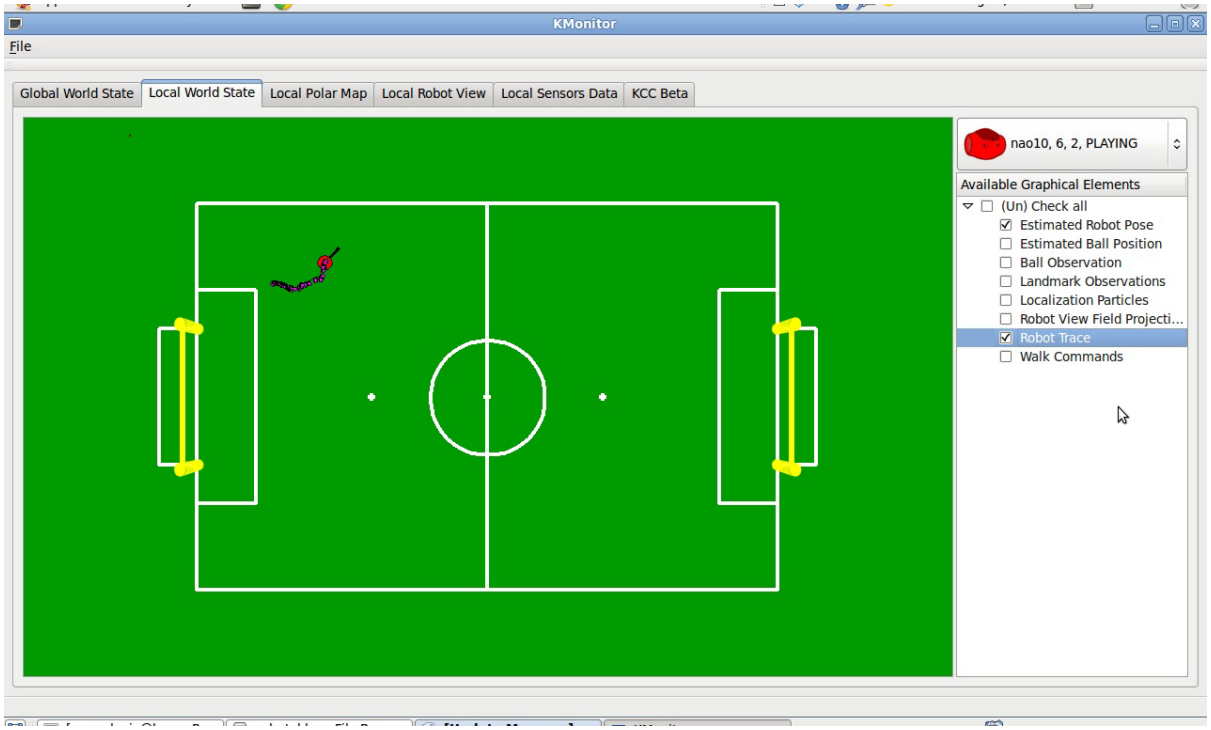

Figure 5.16: Visualization of the robot trace

(not the orientation), connecting successive positions with line elements, so that the user forms a view of how the robot's belief evolves over time. The handling function iterates through the trace list, computes the virtual field global coordinates from the stored real field global ones, draws the positions, computes the connecting line coordinates between successive positions, and visualizes them, as shown in Figure 5.16.

5.3.6 Visualization of the walk commands

The behavior module collects data from almost all the other modules and makes decisions concerning the actions of the robot. The majority of these decisions are walk commands for locomotion. An omni-directional walk command is specified as a vector $(vx, vy, v\theta)$, which indicates the desired velocity in each of the three dimensions; vx for longitudinal translation, vy for lateral translation, and $v\theta$ for rotation. The debugging message about walk contains these three values normalized to $[-1, +1]$. In order to visualize a walk command, we depict the translational velocities (vx, vy) as a 2D vector, whose length and

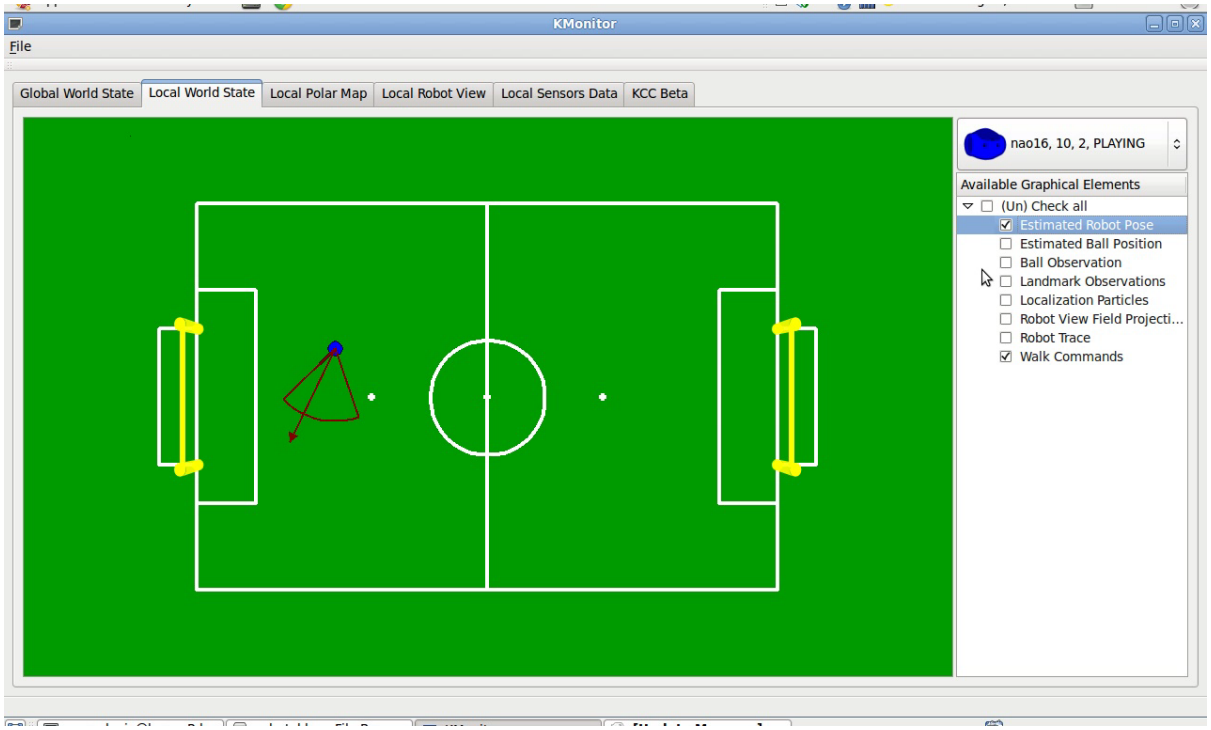

Figure 5.17: Visualization of walk command

direction is proportional to (vx, vy), and the rotational velocity $v\theta$ as an arc, whose length is proportional to $v\theta$, both in the egocentric coordinate system of the robot. To facilitate the user intuition about the value of $v\theta$, its visualization is implemented using a circle segment. Its position implies the sign of $v\theta$; if the segment is positioned on the left side of the robot's orientation, it means that the sign is positive (counter-clockwise rotation), otherwise, if the segment is positioned on the right side of the robot's orientation, the sign is negative (clockwise rotation). A visualization of a walk command is shown in Figure 5.17.

5.4 The Local Polar Map

The third tab, called the Local Polar Map, provides monitoring of information related to obstacle avoidance for a single active robot in the soccer field. The robot builds a local polar obstacle occupancy map, which holds the robot's belief about the existence of obstacles. The map is updated constantly using real-time measurements from the

ultrasonic range sensors and is transformed taking into account the robot's locomotion. Given a target pose, an A* search algorithm is used for path planning and the outcome is an obstacle-free path for guiding the robot to the target. This information that needs to be visualized to facilitate software development and debugging includes the probabilistic polar occupancy map, the coordinates of the target pose, and the obstacle-free path to the target.

5.4.1 Visualization of the occupancy map

The area around the robot is represented by a discrete $n \times k$ polar grid with n rings and k sectors (currently, 12×18). The robot is always located at the center of this grid. The areas defined by the concentric circles of the polar grid are referred to as rings, the slices in which the diameters cut the circle are referred to as sectors, and the polygons that are formed by the conjunction of rings and sectors of the grid are referred to as cells. The debugging message which is being sent periodically via the UDP channel contains one value for each cell of the grid indicating the probability that the cell is occupied by some obstacle. The maximum value for each cell is 1.0, meaning that the cell is occupied with high confidence, and the minimum value is 0.08, meaning that the cell is empty with high confidence. In order to visualize the obstacle occupancy map, each cell is colored with a level of gray proportional to its probability value. The higher the value, the closer its color to black, whereas the lower the value, the closer its color to white. The handling function iterates over the polar coordinates the cells and for each cell it selects its pre-calculated scaled cartesian coordinates in the graphical representation of the polar map, computes the appropriate color value, and updates the corresponding graphical element. The robot is constantly located at the center of the graphical representation of the polar map facing upwards, as indicated by the green arrow. An example of the visualization of the occupancy map is shown in Figure 5.18.

5.4.2 Visualization of the target coordinates

The most common action taken by each active robot is to move to a target pose. Note that the target includes a desired position in the field and a desired orientation. The debugging message about a target (if any) contains the target coordinates $(tx, ty, t\theta)$, where (tx, ty) is the desired position and $t\theta$ is the desired orientation. The handling function simply

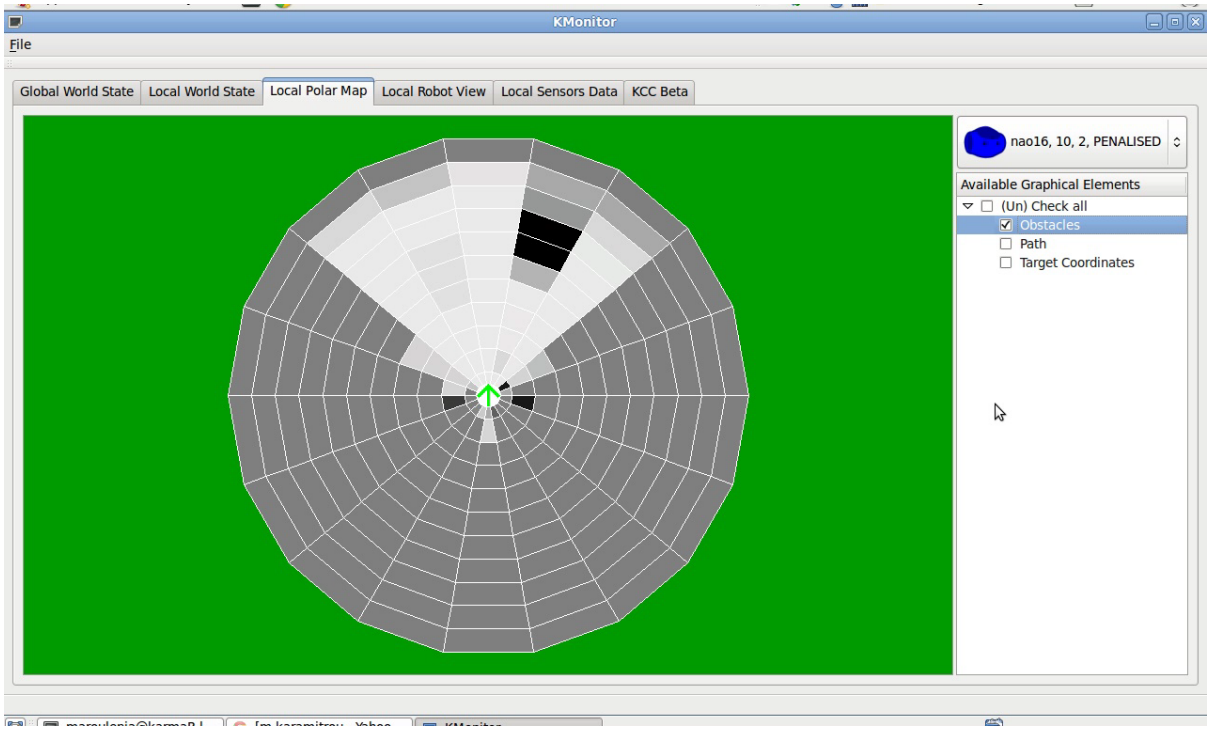

Figure 5.18: Visualization of the obstacle occupancy map

scales the received target position coordinates to the graphical representation of the polar map and draws a circular element for the position of the target and a small line segment indicating the received target orientation (discretized to the closest multiple of $\pi/4$). If the target falls outside the area covered by the polar map, then the target is drawn on the outer ring in the closest sector. An example of target visualization is shown in Figure 5.19.

5.4.3 Visualization of the obstacle-free path

Given a target pose and a map, the A* heuristic search path-planning algorithm delivers an obstacle-free path from the robot to the target. The search algorithm takes into account not only the adjacency of cells in the map, but also the orientation of the robot in each cell, as well as the rotational adjacency constraints, by considering only eight discrete directions (multiples of $\pi/4$). The debugging message about paths contains the entire path delivered by the A* algorithm. The path is described as a sequence of selected

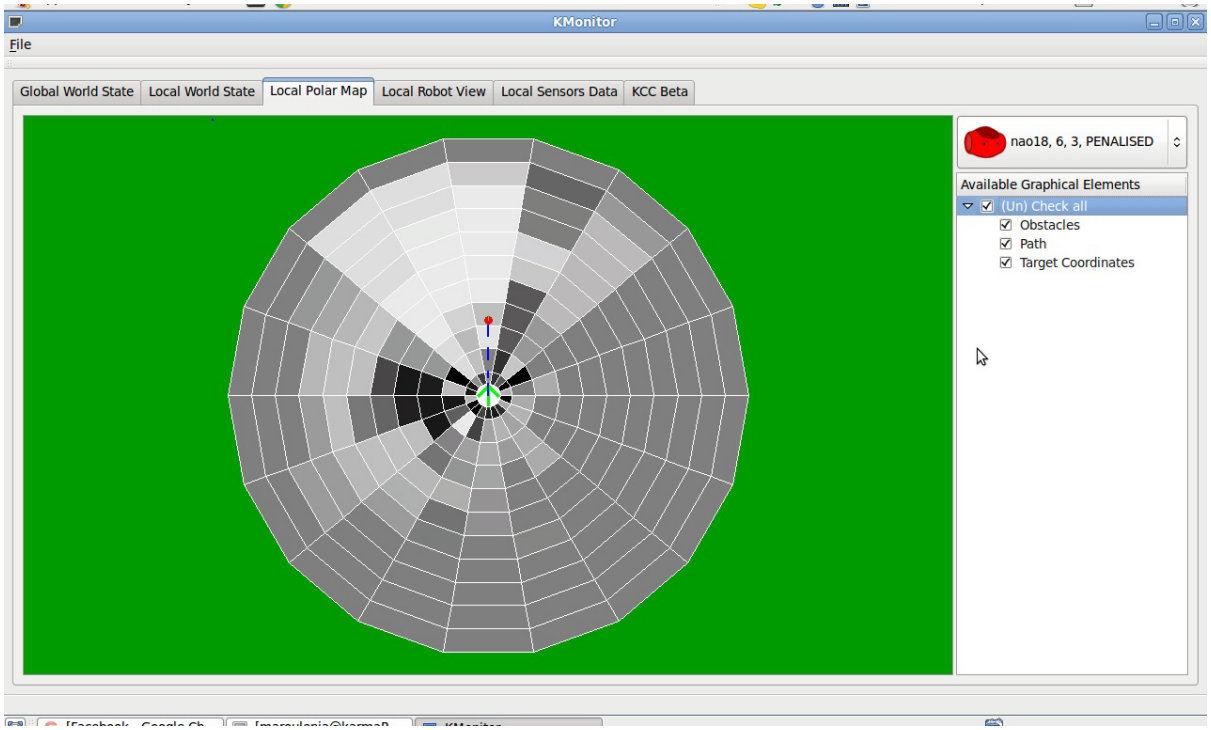

Figure 5.19: Visualization of the target coordinates and the obstacle-free path

cells starting from the center of the polar map, each one of which is specified as (r, s, o), where r is the ring, s is the sector, and o is the orientation. The handling function iterates over the cells of the path and for each cell it selects its pre-calculated scaled cartesian coordinates in the graphical representation of the polar map and draws a small line segment indicating the orientation in that cell. Therefore, the final path consists of a collection of line segments. An example of the visualization of an obstacle-free path is shown in Figure 5.19.

5.5 The Local Robot View

The fourth tab, called the Local Robot View, provides monitoring of the robot camera for a single active robot in the soccer field. The robot camera is the main source of information about the environment. Visual object recognition based on color, shape, and size is accomplished by KVision, a light-weight image processing method for humanoid robots. KVision relies on identifying regions of interest on sampled pixels of the image

utilizing an auto-calibrated color recognition scheme. Therefore, the developer needs to monitor the raw camera image to verify proper operation of the image provider and calibrate the camera and the color-segmented image to verify that color recognition works properly under the current lighting conditions.

5.5.1 Visualization of the raw camera image

The robot camera provides images of resolution 640×480 pixels at 30 frames per second and the native colorspace is YUV422. The debugging message about the camera image contains information about the luminance (Y channel) and chrominance (U, V channels) for each pixel of the image in a compressed form due to YUV422. In particular, there is a Y value for each pixel, but U and V values are provided every other pixel; in other words, every two pixels, the U and V values are the same. Due to the locality of chrominance, this lossy compression is not visible to the eye, but reduces significantly the size of the image (and the message). For visualization, the image must be converted to RGB colorspace. Thus, the corresponding handling function decompresses the message, extracts the Y, U, V components for every pixel of the image and computes the R, G, B components using the transformation of RGB to YUV as defined in the JPEG standard:

$$
\begin{aligned}
R &= Y + 1.402 \times (V - 128) \\
G &= Y - 0.34414 \times (U - 128) - 0.71414 \times (V - 128) \\
B &= Y + 1.772 \times (U - 128)
\end{aligned}
$$

Finally, it creates a new graphical image item, defining the pixel coordinates and the computed value in the RGB colorspace for every unit of the image and outputs the raw image on the graphical label section, as depicted in Figure 5.20.

5.5.2 Visualization of the color-segmented camera image

KVision employs a color recognition process in order to categorize image pixels into discrete colors, so as to identify regions of interest corresponding to colored objects. Consequently, a visualization of the recognized colors should be a proper way to inspect the recognition process. The corresponding handling function decompresses the message containing the raw image, extracts the Y, U, V components of each pixel in the image,

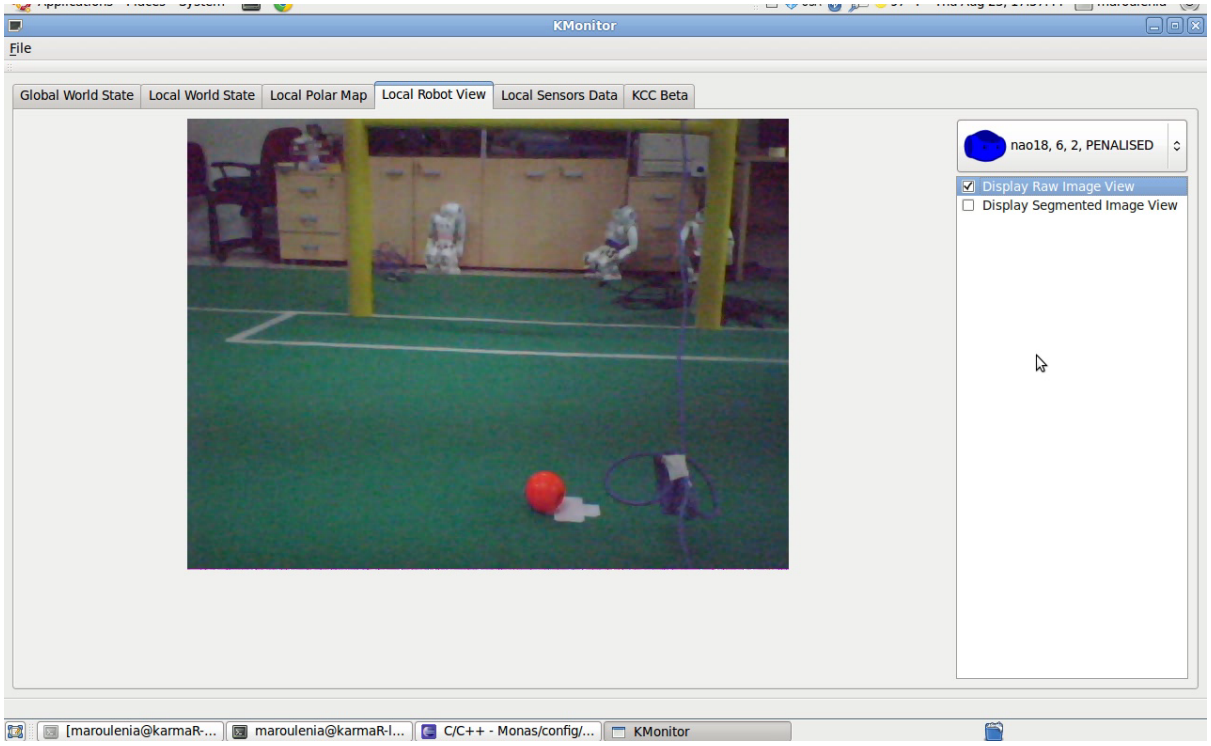

Figure 5.20: Visualization of the raw camera image

recognizes the color of each pixel based on its Y, U, V values, and creates a new graphical image item, defining the pixel coordinates and the recognized color in RGB for every unit of the image, and outputs the color-segmented image on the graphical label section, as depicted in Figure 5.21. It should be noted that color segmentation of the entire image is executed only locally by KMonitor for debugging purposes; the robot applies color recognition to selected pixels only and never transmits color-segmented images. KMonitor and KVision share the same XML configuration files to ensure that the same color map is used on both sides.

5.6 The Local Sensors Data

The fifth tab, called the Local Sensors Data, provides monitoring of the sensors of a single active robot, namely the joint encoders, the accelerometer, the gyroscope, and the force sensitive resistors. The measurements of these sensors constitute a significant source of information for the developer for a number of useful tasks, such as inspecting

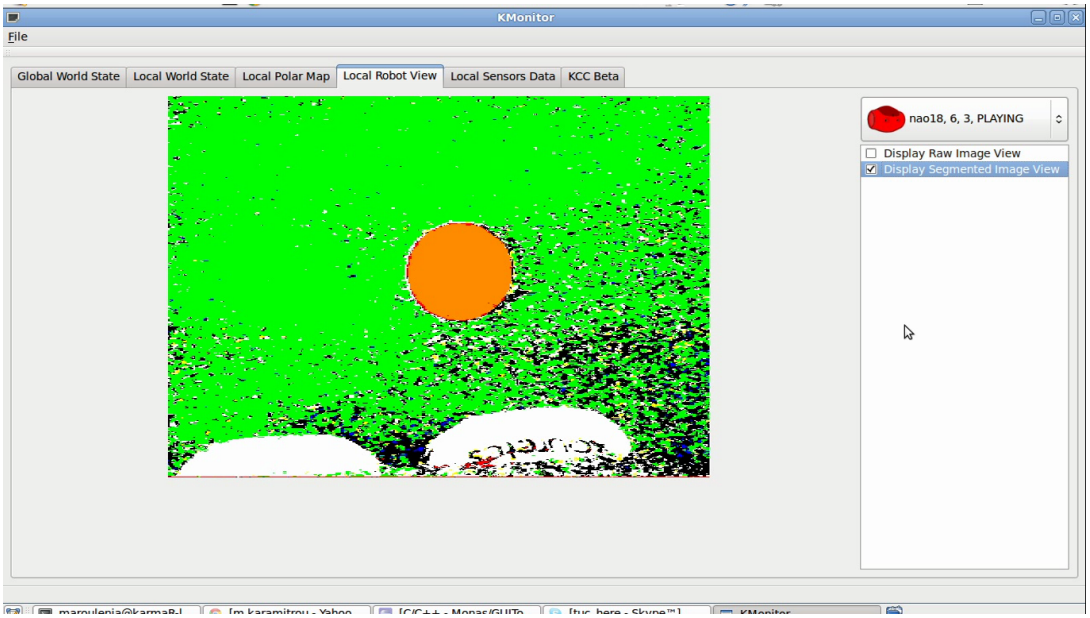

Figure 5.21: Visualization of the color-segmented camera image

the performance of a predefined special action, verifying the robot kinematics, testing a dynamic robotic walk, testing the detection of a fall, etc. As mentioned in Section 2.1.2, Nao RoboCup edition has 21 degrees of freedom and therefore 21 joint encoders, each one of which provides a 12-bit value indicating the current angle of the corresponding joint. The three-axis accelerometer provides three values indicating the acceleration along each of the x, y, and z axes. The gyroscope yields two values indicating the angular velocity about the x and y axes. Finally, the force sensitive resistors deliver four values for each foot indicating the pressure applied to each corner of the foot. The debugging message about sensors contains all the values mentioned above. The handling function reads these values and displays them in combo boxes using an intuitive layout that resembles the robot body, as shown in Figure 5.22. Additionally, the handling function keeps a history of the most recent debugging messages (default: 10); the developer can inspect the most recent values of any sensor by simply opening the corresponding combo box.

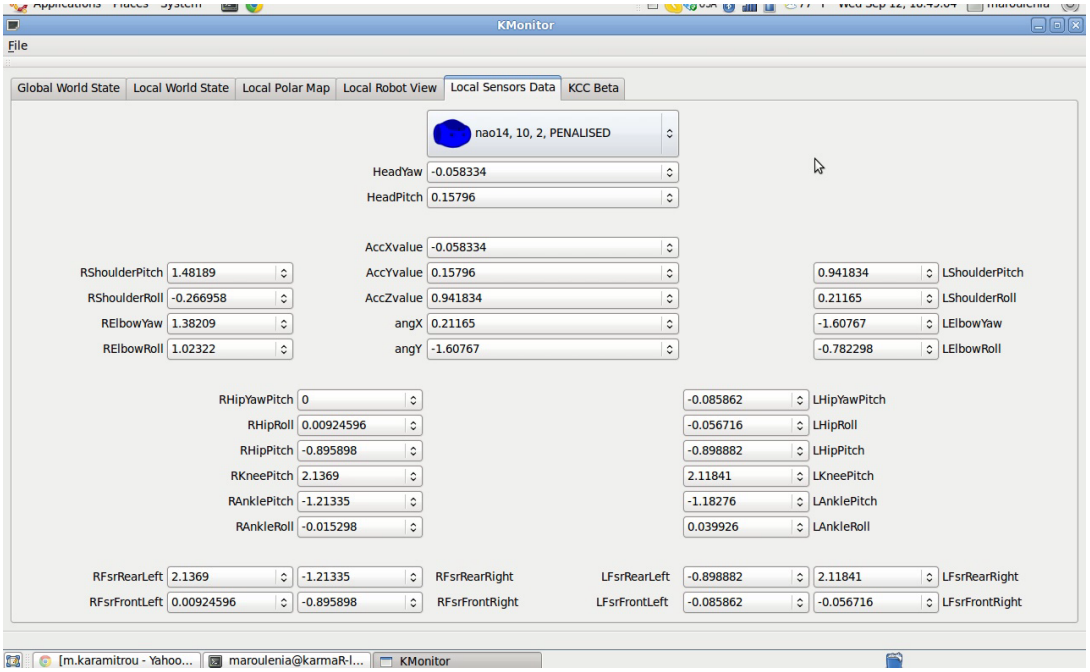

Figure 5.22: Visualization of the data from the sensors

Chapter 6

Implementation

KMonitor was developed based on the Qt4 application framework, the Qt4 Designer, the C++ and Python plugins which we integrated in the Eclipse Indigo SDK to take the full supervision and editor advantage of our robotic software Monas (basically C++ and Python code) and the functionalities which the Qt4 Designer provides. In this chapter we thoroughly document the implementation of the components of KMonitor, the complete structure of all classes, the signals and slots, and the interaction among them from a technical point of view. Figure 6.1 shows the UML class diagram containing all classes we implemented for KMonitor and their structure.

6.1 `MessageAllocator` Class Reference

The `MessageAllocator` class inherits from Qt4's `QObject` class in order to implement the required signals and slots. As mentioned in Section 5.1.1, the `MessageAllocator` module is responsible for manipulating the flow of the data received by the multicast network by (un)subscribing (from)to certain topics and allocating the received data streams among the corresponding graphical interfaces using the Qt's Signal/Slot mechanism.

The `multicast` property, an `EndPoint` class pointer, points to the `MulticastPoint` object. The `timer` property, a `QTimer` class pointer is used to define the periodic interval for the inspection of the write `MessageBuffer` of the `MulticastPoint`. The `myGWRequestedHosts` property, a `QStringList` object holds the list of the multiple robots requested for visualization by the user in the Global World State tab. The `myLWRequestedHost`, `myLMRequestedHost`, `myLVRequestedHost` properties are `QString`

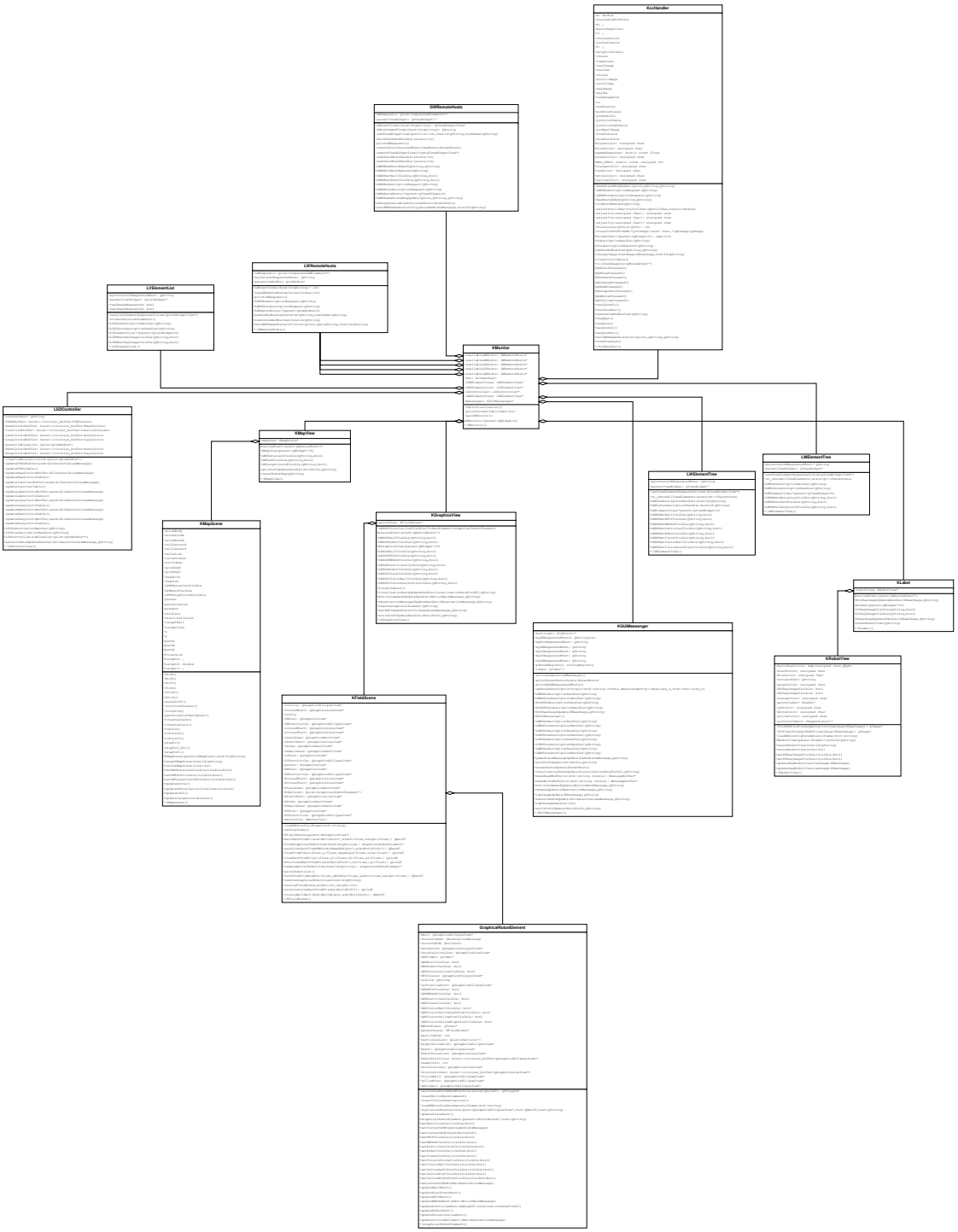

Figure 6.1: The UML Class Diagram of our KMonitor Implementation

objects that hold the current user Nao requests in the Local World State tab, the Local Polar Map tab, the Local Robot View tab respectively.

The MessageAllocator() public function or else the constructor of the corresponding class, loads the required parameters of the network configuration, creates the MulticastPoint object and starts the multicast thread. Furthermore, it instantiates the QTimer class, defines its interval and connects the timeout() signal of the timer to its private slot allocateReceivedMessages(). The makeWriteBuffer() and makeReadBuffer() functions create write and read MessageBuffers respectively and attach them to the MulticastPoint object.

The public slots GWRHSubscriptionHandler() and GWRHUnsubscriptionHandler() update the myGWRequestedHosts property, send the (un)subscription requests for the worldstate topic of the current selected hosts and are executed whenever the user checks or unchecks some of the available remote hosts of the dynamic hosts tree widget on the first tab. The void LWRHSubscriptionHandler(QString) and the LWRHUnsubscriptionHandler() are executed whenever the user selects a host in the dynamic hosts combo box on the second tab, update the myLWRequestedHost, and send the (un)subscription requests for the worldstate, vision, debug and motion topics of the currently selected host. Furthermore, the LMRHSubscriptionHandler(), LMRHUnsubscriptionHandler(), LVRHSubscriptionHandler(), LVRHUnsubscriptionHandler() slots update the corresponding properties that hold the currently selected hosts on the third, fourth, and fifth tabs in an identical way. The last public slot is tabChangeHandler() and implements the filtering of the requested data via the UDP network based on the currently active tab. When the Global World State tab is active, unsubscription requests are sent for the vision, debug, motion, obstacle, and image topics; when the Local World State tab is active, unsubscription requests are sent for the obstacle and image topics and subscription requests are made to the vision, debug, and motion topics; when the Local Polar Map tab is active, unsubscription requests are sent for the vision, debug, motion, and image topics and a subscription request is made to the obstacle topic; finally, when the Local Robot View tab is active, unsubscription requests are sent for all the topics except the image topic for which a subscription request is made.

The signal knownHostsUpdate(KnownHosts) is emitted, whenever a message of type KnownHosts is received from the multicast network. The signal gameStateMessageUpdate (GameStateMessage, QString) is emitted, whenever a message of type GameStateMessage

is received and contains the corresponding `GameState` data structure and the publisher host. The signal `worldInfoUpdate(WorldInfo, QString)` is emitted, whenever a message of type `WorldInfo` is received and contains the corresponding `WorldInfo` data structure and the publisher host. The signal `localizationDataUpdate(LocalizationDataForGUI, QString)` is emitted, whenever a message of type `LocalizationDataForGUI` is received and contains the corresponding `LocalizationDataForGUI` data structure and the publisher host. The signal `obsmsgUpdate(ObservationMessage, QString)` is emitted, whenever a message of type `ObservationMessage` is received and contains the corresponding `ObservationMessage` data structure and the publisher host. The signal `motion-CommandUpdate(MotionWalkMessage, QString)` is emitted whenever a message of type `MotionWalkMessage` is received and contains the corresponding `MotionWalkMessage` data structure and the publisher host. The signal `gridInfoUpdate(GridInfo, QString)` is emitted, whenever a message of type `GridInfo` is received and contains the corresponding `GridInfo` data structure and the publisher host. The signal `rawImageUpdate(KRawImage, QString)` is emitted whenever a message of type `KRawImage` is received and contains the corresponding `KRawImage` data structure and the publisher host.

The protected function `updateSubscription()` is called, whenever any function has to sent (un)subscription requests. It creates a message entry and defines its topic, host, and message class. The message entry is added to the read `MessageBuffer` of the multicast in order to be sent to the UDP network. Finally, the protected slot `allocateReceivedMessages()` which is connected with the `timeout()` signal of the timer, removes the incoming messages from the write `MessageBuffer` of `MulticastPoint`. For each one of the receives messages, it checks its type, if the publisher host has been requested by the user, and, if true, it creates the corresponding data structure, loads it to the corresponding signal, and emits the signal.

6.2 Global World State Tab's User Interface

As shown in Figure 6.2 the predefined user interface for the Global World State tab consists of a promoted `KGraphicsView` on the left pane, where the 2D field scene is visualized, and a `QTreeWidget` on the right pane, where the available remote hosts and their features are represented. When the KMonitor application is executed the pointers of

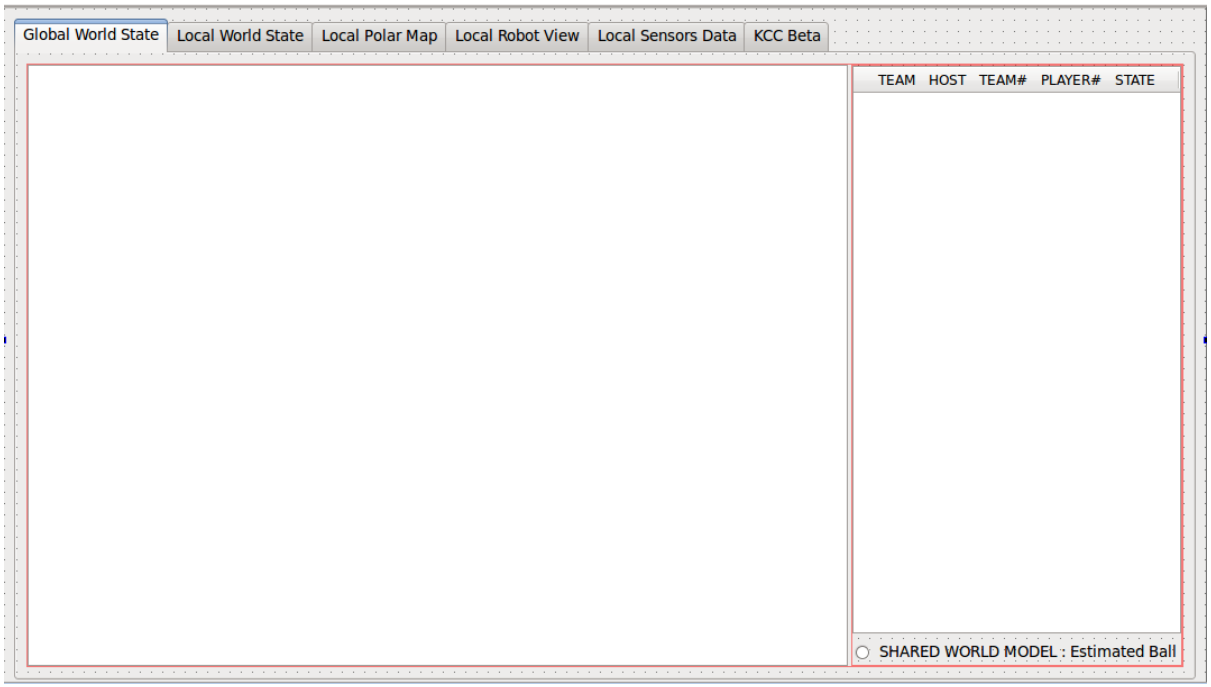

Figure 6.2: Global World State's User Interface

these items pass objects to the corresponding controller, so that the desired functionalities are accomplished.

6.2.1 `GlobalRemoteHosts` Class Reference

The `parentTreeWidget` property, a `QTreeWidget` class pointer is used for holding the predefined in the `KMonitor.ui` tree widget of the Global World State tab. A helper structure has been implemented, called `requestedElements` having the properties `hostId`, `hostName`, `requestedPosition`, and `requestedBall`, which is used to store the current user requests for an active robot in the field. Thus, the property `GWRequests` is a `QList` object of pointers to `requestedElements` structures where all user requests from the tree widget are stored and visualized properly.

The public function `GWRemoteHosts()`, the constructor of the class takes as parameter the `QTreeWidget` pointer, stores it to the `parentTreeWidget` property, and simply initializes the properties of the class.

The public slot `emergeAvailableHosts(KnownHosts)` is connected with the signal

knownHostsUpdate() of the MessageAllocator class, whenever an update for all the existing online active robots is received by the multicast thread. The slot firstly removes the disconnected hosts, those that exist in the GWRequests list, but do not exist in the received KnownHosts message, then for every host in the KnownHosts message it searches if the host already exists in the GWRequests list, and if it does not, it creates a new TreeWidgetItem for the corresponding host.The public slot setGWRHGameStateInfo (GameStateMessage, QString) is connected with the signal gameStateMessageUpdate (GameStateMessage, QString) of the MessageAllocator class and visualizes the game state information on the corresponding TreeWidgetItem that represents the received host. It searches for the host from the received QString parameter by iterating over the parentTreeWidget, and, if the item is found, it sets the current team color by altering the item's icon, sets the player number, the team number, and the player state as a QString text of the TreeWidgetItem. Finally, it emits the signal LWRHGameStateMsgUpdate(QIcon, QString, QString) which informs the LocalRemoteHosts class for a game state update in the RemoteHosts list.

The signal GWRHSubscriptionRequest(QString) is emitted whenever the user checks one or more features of an active robot to visualize from the TreeWidget and is connected to the GWRHSubscriptionHandler() slot of the MessageAllocator class. The GWRHUnsubscriptionRequest(QString) signal is emitted whenever the user unchecks both of the two subcheckboxes of the TreeWidgetItem that represents the host and is connected to the GWRHUnsubscriptionHandler() slot of the MessageAllocator class. The signals GWRHSetRobotVisible(QString, bool) and GWRHSetBallVisible(QString, bool) are emitted whenever the user (un)checks any of the two subcheckboxes of the TreeWidget-Item that represent the selections of the visualization of the estimated robot pose or estimated ball position and are connected to the ViewController object of the Global World State tab. The signal GWRHNewHostAdded(QString, QString) contains information about the host ID and the host name of any new host added to the TreeWidget and is connected to the slot addComboBoxItem(QString, QString) of the LocalRemoteHosts class, whereas the signal GWRHOldHostRemoved(QString) contains information about the host ID of any old host that has been removed from the TreeWidget and is connected to the slot removeComboBoxItem(QString) of the LocalRemoteHosts class. The last signal LWRHGameStateMsgUpdate(QIcon, QString, QString) is emitted whenever a game state update is accomplished in any of the TreeWidgetItems of the TreeWidget

and is connected to the slot `setLWRHGameStateInfo(QIcon, QString, QString)` of the `LocalRemoteHosts` class.

The protected function `GWhostNameFinder(QString hostId)` iterates through the `GWRequests` list, checks if the received host ID exists and returns its host name. The `GWhostFinder(QString hostId)` iterates through the `parentTreeWidget`, checks if the host ID exists and returns a `QTreeWidgetItem` pointer to it. The protected function `removeDisconnectedHosts( KnownHosts newHosts)` accepts the `KnownHosts` argument and compares each of the host in the `parentTreeWidget` with those of the `KnownHosts` message. If the host is found, it means that the host is still connected, so it remains in the `parentTreeWidget`, but, if the iteration is over and the host is not found, it means that the host is disconnected and is removed from the `TreeWidget` and the `GWRequests` list. The protected function `addTreeWidgetItem(int position, QString hostId, QString hostName)` creates and customizes a `TreeWidgetItem` for the newly connected host. The default visualization of the `TreeWidgetItem` is a green icon for the main checkbox; the text that follows it contains simply the host name of the newly connected host. The two subcheckboxes of the main checkbox stand for the estimated robot pose and the estimated ball position respectively. The user is able to (un)check one or both of the subcheckboxes and if he/she (un)checks the main checkbox then automatically the two subcheckboxes are (un)checked too.

The protected slot `mainCheckBoxHandler(int)` manipulates the main checkbox of the customized `TreeWidgetItem` that represents an active robot. It receives as a parameter the current state of the check box and after having found the corresponding host it sets the same checkbox state to the subcheckboxes of the `TreeWidgetItem` and updates the `GWRequests` list. The same logic applies also to the slots `subCheckBox1Handler(int)` and `subCheckBox2Handler(int)`, which are executed whenever the user (un)checks the two subcheckboxes of any of the customized `TreeWidgetItems` of the `TreeWidget`.

6.2.2 `KFieldScene` Class Reference

The `KFieldScene` class is responsible for visualizing the two-dimensional field and all its static graphical items. It inherits from the Qt4's `QGraphicsScene` class. The property `parent` is a `KGraphicsView` pointer that holds the parent view of the scene. The properties `LSide`, `RSide`, `LSmallArea`, `RSmallArea`, `LGoalArea`, `RGoalArea` are `QGraphicsRect-`

`Item` pointers that are used for the creation of all the field lines, the properties `LCrossHPart`, `LCrossVPart`, `RCrossHPart`, `RCrossVPart`, `CCrossHPart` are `QGraphicsLineItem` pointers that are used for the creation of the field penalty marks (crosses). The `QGraphicsEllipseItem` pointer `CCircle` holds the center cirle and the `QGraphicsItem` pointers `LTPost`, `LBPost`, `RTPost`, `RBPost` hold the three-dimensional goalposts. A helper struct has been implemented to hold significant field points whose coordinates are transformed from the real SPL field to the 2D virtual field coordinate system (Figure 5.10).

The constructor of the class loads the XML file with the field parameters and updates the helper struct. It also sets the field scene's background color and creates all its static graphical items, the lines, the penalty marks, the circle, and the 3D goalposts. The most important protected function of the class is the `resizeFieldScene()`, which is executed whenever a resize to the 2D scene window is requested by the user, takes as arguments the new width and height, and rearranges the graphical items according to the new size of the scene window and the specified by the SPL rules positions of the items on the field.

6.2.3 `GraphicalRobot` Class Reference

The mentioned class is the one that holds all the dynamic graphical items of the robot that are able to be visualized in real time execution of the Monas code. The user makes his/her requests through the `Remote Hosts Widget` and the `Available Elements Widget`, the `View-Controller` interprets the user requests to actual visualizations of the robots' or robot's features on the 2D Scenes of the first and second tab. The documentation of the `GraphicalRobot` class follows.

The property `parentScene` is a `KFieldScene` class pointer which holds the address of the corresponding scene that is created from the `View-Controller` of the Global/Local World State tabs. The `hostId` is a QString that holds the unique host ID of the `GraphicalRobot` object. The `currentWIM` is a `WorldInfo` reference that aims to store the most recent message of that type, in order to be used as a data base for the majority of the visualized elements on the 2D field. The protocol buffer message, `WorldInfo`, holds the data for the robot's and ball's pose and the definition of its data structure is the following:

```
message WorldInfo{
    required RobotPose myPosition = 1;
```

```
repeated Ball Balls = 2;
...
}

message RobotPose{
required float X = 1;
required float Y = 2;
required float phi = 3;
...
}

message Ball{
required float relativeX = 1 [default = -100000];
required float relativeY = 2 [default = -100000];
...
}
```

where:

- `wim.myPosition.X`, is the estimated position of the robot in the x-axis of the global coordinate system,
- `wim.myPosition.Y`, is the estimated position of the robot in the y-axis of the global coordinate system,
- `wim.myPosition.phi`, is the estimated orientation of the robot in the θ-axis of the global coordinate system,
- `wim.Balls.relativeX`, is the estimated relative distance of the ball from the robot's pose in the x-axis,
- `wim.Balls.relativeY`, is the estimated relative distance of the ball from the robot's pose in the y-axis.

The `RobotVisible` is a boolean variable that holds if the user has requested the robot's pose visualization. The `Robot` is a `QGraphicsEllipseItem` class pointer that represents the estimated robot position on the 2D field as a colored circle. The `RobotDirection`

is a QGraphicsLineItem class pointer that represents the estimated robot orientation as a straight line segment. In order to visualize the robot's pose the received data has to be transformed and scaled to fit in the 2D field coordinate system and the geometry of the utilized graphical items has to be computed. As mentioned above an ellipse item is used for visualizing the robot's position. To set the item's ellipse geometry a QRectF item has to be defined and passed as argument to the setRect() function of QGraphicsEllipseItem class. The rectangle's left edge defines the left edge of the ellipse, whereas the height and width of the rectangle describe the height and width of the ellipse. The calculation of the QRectF item's properties are computed by the following code lines:

```
rect.x = sceneRect().width() * (config.xCentre - width/2 + xMiddle)
                                                * (1 / config.totalCarpetAreaWidth);

rect.y = sceneRect().height() - sceneRect().height()
             * (config.yCentre + height/2 + yMiddle) / config.totalCarpetAreaHeight);

rect.width = sceneRect().width() * width / config.totalCarpetAreaWidth;

rect.height = sceneRect().height() * height / config.totalCarpetAreaHeight;
```

A line item is used for visualizing the robot's orientation which is calculated from the robot's position and orientation by the following code lines:

```
point.x = wim.myPosition.X + distance * cos(wim.myPosition.phi);
point.y = wim.myPosition.Y + distance} * sin(wim.myPosition.phi) + 1;
```

To set the item's line geometry a QLineF item has to be defined and passed as argument to the setRect() function of QGraphicsLineItem class. The QLineF item is defined by a start point and an end point. The calculation of the QLineF item's properties are computed by the following code lines:

```
line.x_0 = sceneRect().width() * (config.xCentre+x)/config.totalCarpetAreaWidth;

line.y_0 = sceneRect().height() - sceneRect().height() * (config.yCentre+y)
                                                        /config.totalCarpetAreaHeight;

line.x_1 = sceneRect().width() * (config.xCentre+x)/config.totalCarpetAreaWidth
```

```
            + sceneRect().width()* size*cos(degAngle)/config.totalCarpetAreaWidth;

line.y_1 = sceneRect().height() - (sceneRect().height() * (config.yCentre+y)
        * ( 1/config.totalCarpetAreaHeight + sceneRect().height()*size*sin(degAngle)
        * (1 / config.totalCarpetAreaHeight));
```

The `BallVisible` is a boolean variable that holds if the user has requested the ball's position visualization. The `Ball` is a `QGraphicsEllipseItem` class pointer that represents the estimated ball's position on the 2D field with respect to the robot's pose as a colored circle. Before estimating the graphical item's geometry, we compute the absolute coordinates of the ball's position on the real field with the following code lines:

```
ball.x = wim.myPosition.X + wim.Balls(0).relativeX * cos(wim.myPosition.phi)
                                    - wim.Balls(0).relativeY*sin(wim.myPosition.phi);

ball.y = wim.myPosition.Y + wim.Balls(0).relativeX * sin(wim.myPosition.phi)
                                    + wim.Balls(0).relativeY*cos(wim.myPosition.phi);
```

The calculation of the ball item's `QRectF` properties are then computed with the same equations used for the robot's position item geometry. The `UnionistLine` is a `QGraphicsLineItem` class pointer which joins the graphical elements that represent the robot's pose and ball's pose in order to avoid confusion in cases where there are multiple robots on the 2D field.

6.3 Global World State Tab's `View-Controller`

The `View-Controller` of the first tab is implemented through the `KGraphicsView` class and is responsible for manipulating all the graphical robot elements on the 2D field scene according to the user's requests made through the `GlobalRemoteHosts` widget and for updating the position of those elements based on the received protocol buffer messages from the `MessageAllocator` module.

The `KGraphicsView` inherits from the Qt4's `QGraphicsView`. The property `childScene` is a `KFieldScene` pointer that holds the address of the 2D field scene. The constructor of the referring class creates a new `KFieldScene` object. The protected function `resizeEvent(QResizeEvent* event)` reimplements that of the `QGraphicsView` one and

handles the resize of the field scene caused by the user. Whatever the increase/decrease of the view window might be, the ratio of the field scene is preserved.

The public slot `GWSGVRobotVisible(QString, bool)` is connected with the GlobalRemoteHosts signal `GWRHSetRobotVisible(QString, bool)`. Whenever it is executed, it searches the list of `GraphicalRobots` for the corresponding host, if it does not exist, it creates a new `GraphicalRobot` object and it adds it to the list, and stores its visibility based on whether the user has checked or unchecked the robot's pose checkbox for the robot. The same procedure is followed by the public slot `GWSGVBallVisible(QString, bool)` which is connected with the `GlobalRemoteHosts` signal `GWRHSetBallVisible (QString, bool)` according to the ball's pose checkbox. The slot `worldInfoUpdate-Handler(World- Info, QString)` is connected with the signal `worldInfoUpdate(World-Info, QString)` of the `MessageAllocator` class. Whenever it is executed, it searches the host in the list of `GraphicalRobots` and, if the host exists, it updates the `currentWIM` property, it checks the visibility of the robot's and ball's pose and if the user has requested them to be visible, it updates their current positions as described in the `GraphicalRobot` class. If the user has requested the visualization of both the robot and the ball then automatically, the slot makes the `UnionistLine` visible too. The slot `setKGFCGameStateInfo(GameStateMessage, QString)` is connected with the `Message-Allocator`'s signal `gameStateMessageUpdate(GameStateMessage, QString)` and, whenever it is executed, it sets the corresponding team color to the ellipse item that represents the robot's position and sets the robot's number as tooltip. The last slot `removeGraphical-Element(QString)` is connected with the `GlobalRemoteHosts` signal `GWRHOldHostRemoved (QString)` and removes from the list of `GraphicalRobots` the received one.

6.4 Local World State Tab's User Interface

The Qt4 designer was used to predefine the second tab's user interface as shown in Figure 6.3. The tab consists of a promoted `KGraphicsView`, at the left pane, similar to that of the first tab's, a `QComboBox` at the right top pane, where the active remote hosts are visualized, and a `QTreeWidget` at the right pane, where the available features are represented.

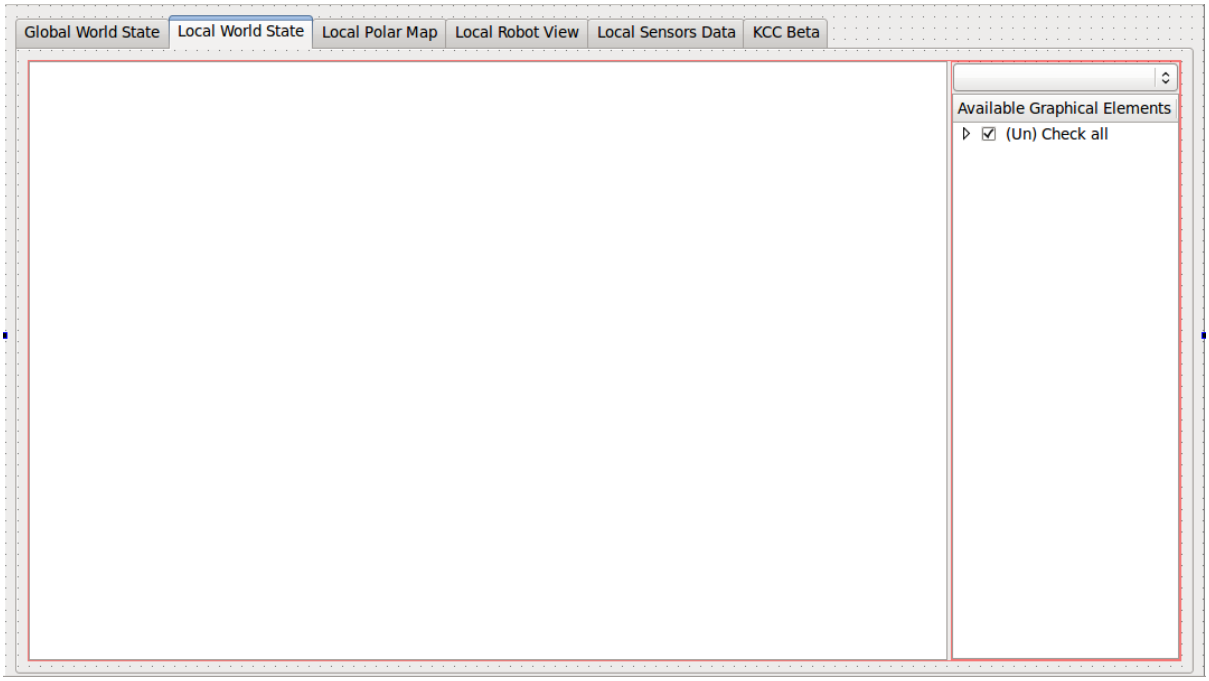

Figure 6.3: Local World State's User Interface

6.4.1 `LocalRemoteHosts` Class Reference

The `LocalRemoteHosts` class inherits from Qt4's `QComboBox` class in order to implement a customized combobox which shall visualize all the current connected hosts of the network, but it shall limit the user to select only one host for monitoring.

The `parentComboBox` property, a `QComboBox` class pointer is used for holding every predefined in the `KMonitor.ui` combobox. Currently, the predefined comboboxes are added on all the existing tabs of the KMonitor tool apart from the Global World State tab. A helper structure has been implemented, called `requestedLWElements` having the properties `hostId`, `hostName`, and `hostSelected`, which is used to store whether or not the current user requests the corresponding active robot in the field. Thus, the property `LWRequests` is a `QList` object of pointers to `requestedLWElements` structs for the user request from the combobox to be stored and visualized properly. The `myCurrentRequestedHost` property is a `QString` which holds the current user local host selection.

The constructor `LWRemoteHosts()` defines some layout values for the visualization of the items of the `parentComboBox` and connects the inherited signal `activated(int)`,

which is emitted whenever an item of a QComboBox object is selected, with the protected slot newLWRemoteHostSelected(int). Furthermore, it creates the initial prompting comboBoxItem to urge the user to select an available host.

The public slot addComboBoxItem(QString, QString) is connected with the GlobalRemoteHosts signal GWRHNewHostAdded(QString, QString) and it creates a customized QComboBoxItem which visualizes the received newly connected host with default parameters, updates the LWRequests list and adds the item to the parentComboBox. The removeComboBoxItem(QString) is connected with the GlobalRemoteHosts signal GWRHOldHostRemoved(QString). It searches for the position of the host which has to be removed, and if the current requested host is the disconnected one, the on screen parentComboBoxItem is set to be the initial prompting one.The slot setLWRHGameStateInfo(QIcon, QString, QString) is connected with the LWRHGameStateMsgUpdate (QIcon, QString, QString) of the GlobalRemoteHosts class and updates the combobox item with the received information for the current game state of the robot.

The signals LWRHSubscriptionRequest(QString) and LWRHUnsubscriptionRequest (QString) are connected with the void LWRHSubscriptionHandler(QString) and the LWRHUnsubscriptionHandler(QString) of the MessageAllocator class and are emitted whenever the user selects a host to monitor from the parentComboBox.

6.4.2 LWElementTreeWidget Class Reference

The LWElementTreeWidget class inherits from Qt4's QTreeWidget class in order to implement a customized tree widget which shall visualize all the available graphical robot elements for monitoring. It contains properties for storing the current user selected host and the parent tree widget. The constructor LWElementTree() unchecks all the tree widget's items and connects the QTreeWidget's signal itemChanged(QTreeWidgetItem*, int) with the class's slot newTreeElementRequested(QTreeWidgetItem*).

The signal LWRHSetRobotVisible(QString, bool) contains information about the user's request for the robot pose visibility. The LWRHSetBallVisible(QString, bool) contains information for the ball's pose visibility and the LWRHSetVisionBallVisible (QString, bool) for the ball observations visibility. In the same way the LWRHSetVisionGoalPostsVisible(QString, bool), LWRHSetParticlesVisible(QString, bool),

`LWRHSetHFOVVisible(QString, bool)`, `LWRHSetTraceVisible (QString, bool)`, `LWRHSetMWCmdVisible(QString, bool)` are responsible for the landmarks observations, the particles, the robot view, the trace, and the motion command visibilities.

The private slot `newTreeElementRequested(QTreeWidgetItem* item)` is executed whenever the user makes a different selection on the customized tree widget, it checks which item of the widget has changed and it emits the corresponding signal.

6.4.3 `LocalRobot` Class Reference

The `LocalRobot` class is the one that holds all the dynamic graphical items of the robot that can be visualized on the 2D scene. For the visualization of the static field elements the `KFieldScene` is instantiated. The mentioned class inherits from the `GlobalRobot` class. The `currentObsm`, an `ObservationMessage` reference is used to hold the most recent message of that type. The definition of the referring data structure is the following:

```
message ObservationMessage {
...
optional BallObject ball=2;
repeated NamedObject regular_objects=3;
...
repeated PointObject view_limit_points=11;
}

message BallObject{
required float dist =1 [default = 0.0];
required float bearing =2  [default = 0.0];
...
}

message NamedObject {
required string object_name=1 [default =""];
required float bearing=2 [default=-1];
required float distance=3 [default=-1];
...
}
```

```
message PointObject{
required float distance =1;
required float bearing =2;
}
```

The boolean `LWSVisionBallVisible` holds the user's request for the ball observation's visibility, whereas the `VisionBall`, a `QGraphicsEllipseItem` pointer is used for the visualization of the ball observation received from the vision module. Before estimating the graphical item's geometry, we transform the received polar coordinates to cartesian ones with the following code lines:

```
ball.x = wim.myPosition().x() + bob.dist()
                            * cos((wim.myPosition().phi() + bob.bearing());

ball.y = wim.myPosition().y() + bob.dist()
                            * sin((wim.myPosition().phi() + bob.bearing());
```

The `LeftYellowPost`, `RightYellowPost`, `YellowPost` are `QGraphicsEllipseItem` pointers that are used to hold the corresponding addresses of the graphical elements that visualize the goalposts observations received from the vision module. Similar to the transformation of the coordinates of the ball observation, we compute the actual global field coordinates for the goalposts as below:

```
post.x = wim->myposition().x() + nob->distance()
                          * cos((wim->myposition().phi() + nob->bearing());

post.y = wim->myposition().y() + nob->distance()
                          * sin((wim->myposition().phi() + nob->bearing());
```

The calculation of the observation ball, the goalpost items' `QRectF` properties are then computed with the same equations of those used for the robot's position item geometry. The boolean `LWSHFOVVisible` and the `HFOVLines`, a `QGraphicsPolygonItem` pointer are used to visualize the robot's view projection on the 2D field as received from the vision module.

The definition of the `LocalizationDataForGUI` data structure is the following:

```
message LocalizationDataForGUI{
repeated RobotPose Particles = 1;
}
```

The boolean LWSParticlesVisible and the ParticlesList, a QList of pointers to Particles structures are used to hold the required addresses and information for the visualization of the particles received from the world state module. The estimation of the position of each particle and the geometry of the graphical item which represents it, is made following the same procedure with that for the robot's pose. The bool LWSTraceVisible, the RobotPositions, a circular buffer of QGraphicsEllipseItem pointers and the UnionistLines, a circular buffer of QGraphicsLineItem pointers are used for the visualization of the robot's trace. The boolean LWSMWCmdVisible, the GotoPositionLine, a QGraphicsLineItem pointer, the GotoArrow, a QGraphicsPolygonItem pointer, the zAxisArc, a QGraphicsEllipseItem pointer and the zAxisArcArrow, a QGraphicsPolygonItem pointer are used to visualize the robot's motion command on the 2D field as received from the behavior module.

6.5 Local World State Tab's View-Controller

Similar to the View-Controller of the first tab, the View-Controller class of the second tab is responsible for manipulating all the graphical robot elements on the 2D field scene according to the user's requests made from the user interface and for updating the position of those elements based on the received protocol buffer messages from the MessageAllocator module.

The public slot LWSGVVisionBallVisible(QString, bool) is connected with the LWSElementTree signal LWRHSetVisionBallVisible(QString, bool) and whenever it is executed, it checks if the already created LocalRobot item has the same host ID with the host ID of the requested host of the signal; if not, it removes the old one and creates a new LocalRobot item with the currently requested one and sets the observation ball's visibility according to the user's request. The same procedure is followed for the following public slots. The LWSGVVisionGoalPostsVisible(QString, bool) is connected with the LSWElementTree signal LWRHSetVisionGoalPostsVisible(QString, bool). The LWSGVParticlesVisible(QString, bool) is connected with the signal

`LWRHSetParticlesVisible(QString, bool)` and sets the visibility of the particles list. The `LWSGVHFOVVisible(QString, bool)` is connected with the signal `LWRHSetHFOV-Visible(QString, bool)` and sets the visibility of the polygon that visualizes the robot view as a field projection. The `LWSGVTraceVisible(QString, bool)` is connected with the signal `LWRHSetTraceVisible(QString, bool)` and updates the visibility of the ellipse and line items in the circular buffers that implement the graphical trace. The `LWSGVMWCmdVisible(QString, bool)` is connected with the signal `LWRHSetMWCmdVisible (QString, bool)` and sets the visibility of the items that visualize the motion command.

The public slot `observationMessageUpdateHandler(ObservationMessage,QString)` is connected with the `MEssageAllocator` signal `obsmsgUpdate(ObservationMessage, QString)`. During its execution procedure, it checks the `LocalRobot`'s `hostId` and if it matches the one in the argument, it updates the `currentObsm` property. Then, it checks the visibility booleans of the ball observation, the goalposts observations, the robot's view projection, and, if the user has requested specific elements, it updates their position on the 2D field scene. The `localizationDataUpdateHandler(LocalizationDataForGUI, QString)` is connected with the MEssageAllocator signal `localizationDataUpdate(LocalizationDataForGUI, QString)`; it follows a similar procedure with the above, but the update stands for the position of the particles.The last slot, `motionCommandUpdateHandler(MotionWalkMessage, QString)` is connected with the `MEssageAllocator` signal `motionCommandUpdate(MotionWalkMessage, QString)` and updates the position of the graphical items that visualize the motion command.

6.6 Local Polar Map Tab's User Interface

The Local Polar Map tab was predefined so as to consist of a promoted `KMapView`, a `QComboBox` at the right top pane, where the active remote hosts are visualized, similar to that of the second tab's item, and a `QTreeWidget` at the right pane, where the available features are represented. The predefined user interface is shown at Figure 6.4.

6.6.1 `LMElementTreeWidget` Class Reference

The `LMElementTreeWidget` Class has been implemented similarly to the class in Section 6.4.2, with the only difference in the items of the tree widget it provides. More

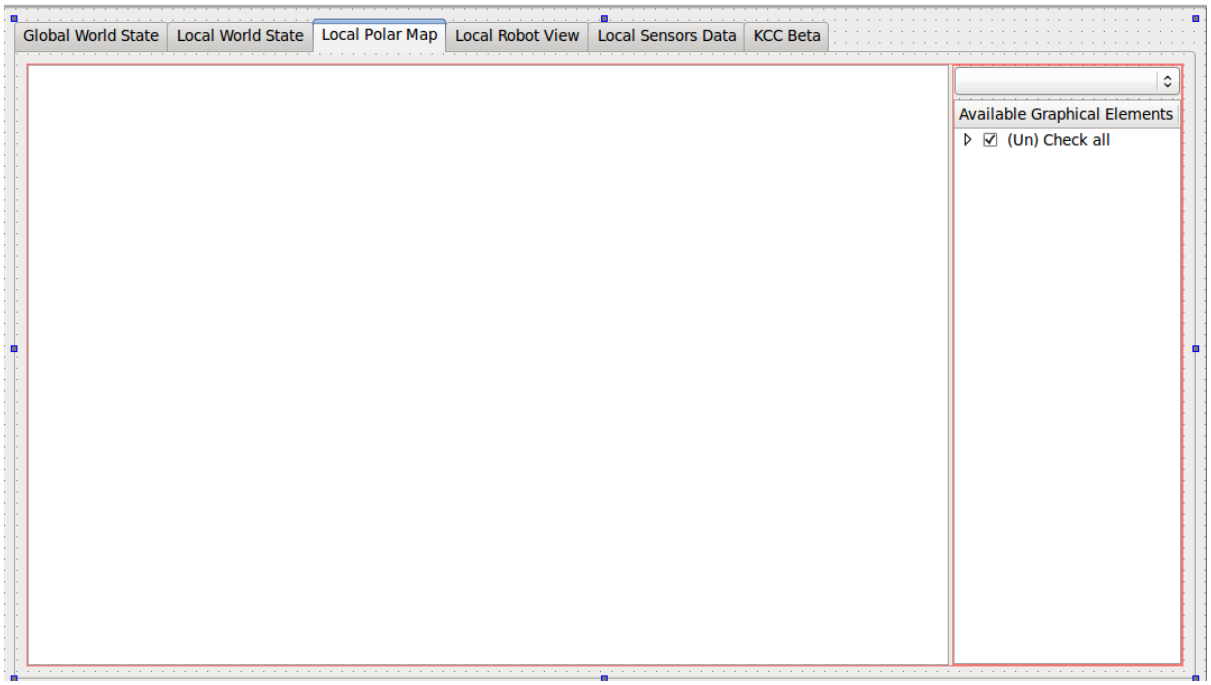

Figure 6.4: Local Polar Map's User Interface

specifically, the graphical robot elements the user can check or uncheck are the obstacles, the target coordinates, and the obstacle-free path. Thus, corresponding signals are emitted whenever the user selects or deselects any of the aforementioned features.

6.7 Local Polar Map Tab's View-Controller

`KMapView` class, a customized Qt4's `QGraphicsView` class, implements the controller of the Local Polar Map tab. The main functionality that the controller provides is the control of the visibility of the available graphical elements according to the user's requests made from the `LocalRemoteHosts` and the `LMElementTree` widgets and for updating the `MapScene` with the `GridInfo` data received by the `MessageAllocator` module.

`KMapView` inherits from `QGraphicsView` class and a `KMapScene` pointer holds the corresponding scene which the view visualizes and controls. The public slot `LMObstaclesVisible(QString, bool)` is connected with the signal `LMRHSetObstaclesVisible(QString, bool)` and, whenever it is executed, it checks the host ID of the current `MapScene` object and, if the host is not the same, it resets the `MapScene` and sets the boolean that holds the

visibility of the item that visualizes the obstacle map to the received value. Similar procedure is followed by the slot `LMPathVisible(QString, bool)`, which is connected with the signal `LMRHSetPathVisible(QString, bool)` and sets the visibility of the path's items and the slot `LMTargetCoordVisible(QString, bool)` which is connected with the signal `LMRHSetTargCoordVisible(QString, bool)` and handles the visibility of the target item.

The slot `gridInfoUpdateHandler(GridInfo, QString)` is connected with the `MessageAllocator` signal `gridInfoUpdate(GridInfo, QString)`. Whenever it is executed it updates the `PolarGrid` array of the `KMapScene` object with the probability values for each grid cell, it updates the `targetX`, `targetY`, and `targetA` values and the arrays `pathR`, `pathS`, and `pathO`. Then, it checks the visibility booleans for each of the available elements, updates the position, and sets visible the requested elements.

6.8 Local Robot View Tab's User Interface

The fourth tab's user interface was designed as depicted in Figure 6.5. The tab consists of a promoted `KLabel` at the left pane, that constitutes a customized `QLabel` for the needs of the image visualization in Qt4, a `QComboBox` at the right top pane, where the active remote hosts are visualized and its handler is a `LocalRemoteHosts` object, and a `QListWidget` at the right pane, where the available features are represented.

6.8.1 `LVElementList` Class Reference

The `LVElementList` class visualizes the two available to the user features, the display of the raw images of the robot and the display of the color-segmented images. The user is able to check only one of the two available elements to monitor. The mentioned class inherits from the Qt4's `QListWidget` class. A `QString` reference holds the host ID of the requested robot, which the user has selected from the customized `LocalRemoteHosts` combo box. A `QListWidget` pointer holds the parent `QListWidget` from the user interface and two booleans hold the current user request for the visualization of the raw or the segmented image. The constructor of the mentioned class sets the `parentListWidget` pointer, calls a private function which unchecks the two items of the `ListWidget`, and

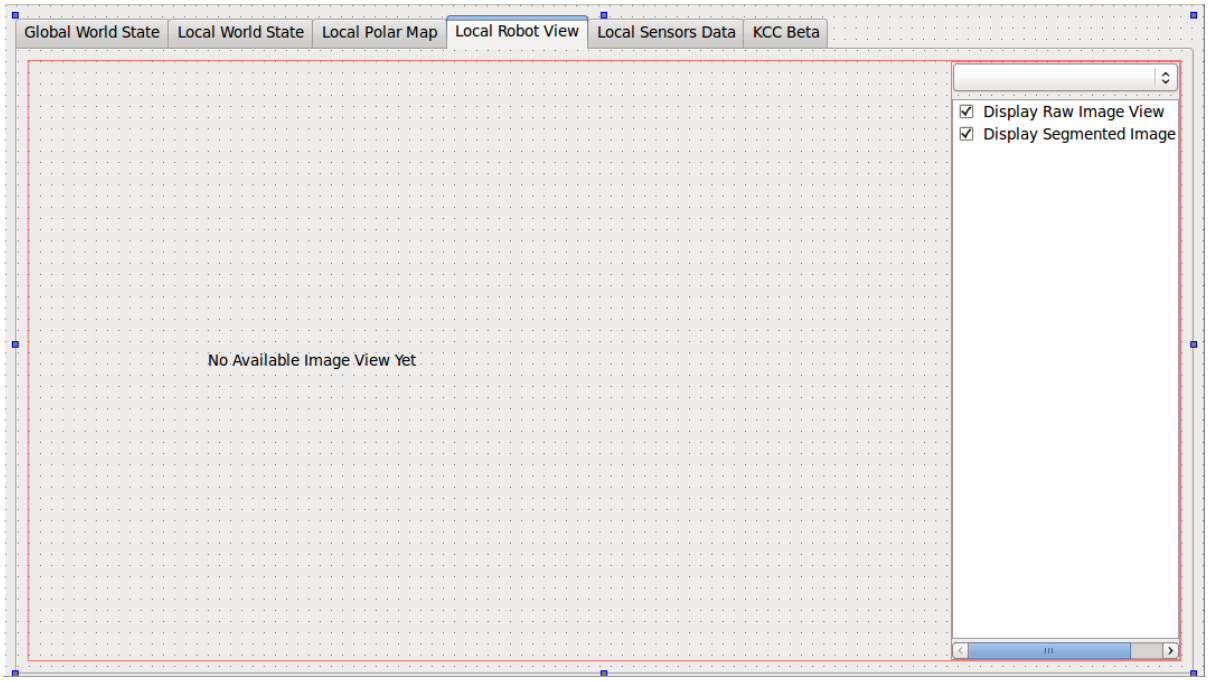

Figure 6.5: Local Robot View's User Interface

connects the `QListWidget` signal `itemChanged(QListWidgetItem*)` with the private slot `newListElementRequested(QListWidgetItem*)`.

The last mentioned slot is executed whenever the user (un)checks any of the two list widget items. During its execution, it checks which of the two items was triggered, and the check state of the item. If the user has checked the raw image checkbox, then the raw image visibility boolean is set to true, if the visibility boolean of the segmented image is also true, then is set to false and finally the signal `LVRHSetRawImageVisible(host, true)` is emitted. If the user has unchecked the raw image checkbox simply the visibility boolean is updated and the signal `LVRHSetRawImageVisible(host, false)` is emitted. Similar procedure is followed, if the triggered item is the segmented image checkbox. The public slots `LVELSubscriptionHandler(QString)` and `LVELUnsubscriptionHandler(QString)` are connected with the `LocalRemoteHosts` signals `LWRHSubscriptionRequest(QString)` and `LWRHUnsubscriptionRequest(QString)`. Whenever they are executed they update the current host variable and uncheck the two list element items whenever it is needed. The signals `LVRHSetRawImageVisble(QString, bool)` and `LVRHSetSegImageVisible (QString, bool)` are connected with the slots `LVRawImageVisible(QString, bool)`

and `LVSegImageVisible(QString, bool)` of the `KLabel` class respectively and are emitted whenever the user (un)checks any of the two list widget items.

6.8.2 `RobotView` Class Reference

The `RobotView` class implements the creation of the raw or the color segmented RGB image that is displayed on the corresponding label of the Local Robot View tab's user interface. A `KLabel` pointer and a `QString` reference hold the parent label and the current requested host respectively. The definition of the `KRawImage` data structure which is received by the `MessageAllocator` module is the following:

```
message KRawImage{
...
required uint32 bytes_per_pix = 1 ;
required uint32 width = 2 [default = 0];
required uint32 height = 3 [default = 0];
required bytes image_rawdata = 5;
...
}
```

where:

- `krim.bytesperpix`, is the number of the channels utilized to store the YUV components (currently 2),
- `krim.width`, is the received image width in pixels,
- `krim.height`, is the received image height in pixels,
- `krim.imagerawdata`, is the raw image data.

The function `updateRawRobotView(KRawImage rawImage)` calls the private `YUV2RGB (KRawImage rawImage)` function which returns a `QImage` pointer to a `QImage` item with pixel format transformed from the received YUV colorspace to the RGB one, transforms the `QImage` to `QPixMap` and displays the image on the label. The private function `YUV2RGB(KRawImage rawImage)` iterates over the received raw data and extracts the values for the y, u, v components of every pixel. Then the r, g, b components are

computed based on the transformation equations in Section 5.5.1 and every pixel of the newly created QImage is set by defining its position and the values of the RGB components. The function updateSegRobotView(KRawImage rawImage) follows similar procedure with that followed by the updateRawRobotView(KRawImage rawImage) function except for calling the private YUVSeg2RGB(KRawImage rawImage). The private function YUVSeg2RGB(KRawImage rawImage) iterates over the received raw data, extracts the values for the y, u, v components of every pixel and provides these values to the KSegmentator which accomplishes the color segmentation. The outcome is mapped to the RGB values and every pixel of the newly created QImage is set by defining its position and the values of the r, g, b components.

6.9 Local Robot View Tab's View-Controller

The View-Controller of the fourth tab is implemented through the KLabel class, a customized QLabel class and is responsible for controlling the visibility of the available displayed images according to the user's requests made from the LocalRemoteHosts and LVElementList widgets and for updating the RobotView with the KRawImage data received by the MessageAllocator module.

The aforementioned class inherits from the QLabel class. A KRobotView pointer holds the handler's address.The public slots LVRawImageVisible(QString, bool), LVSegImageVisible(QString, bool) are connected with the LVElementList signals LVRHSetRawImageVisible(QString, bool), LVRHSetSegImageVisible(QString, bool) respectively and whenever they are executed, they handle the booleans that hold the requested visibility of the raw and segmented image. The slot kRawImageUpdateHadler(KRawImage, QString) is connected with the MessageAllocator signal kRawImageUpdate(KRawImage, QString). It checks the compatibility of the received host and the currest RobotView host and if they are the same, it calls the updateRawRobotView (KRawImage) or the updateSegRobotView(KRawImage rawImage) function of the RobotView class according to the user request.

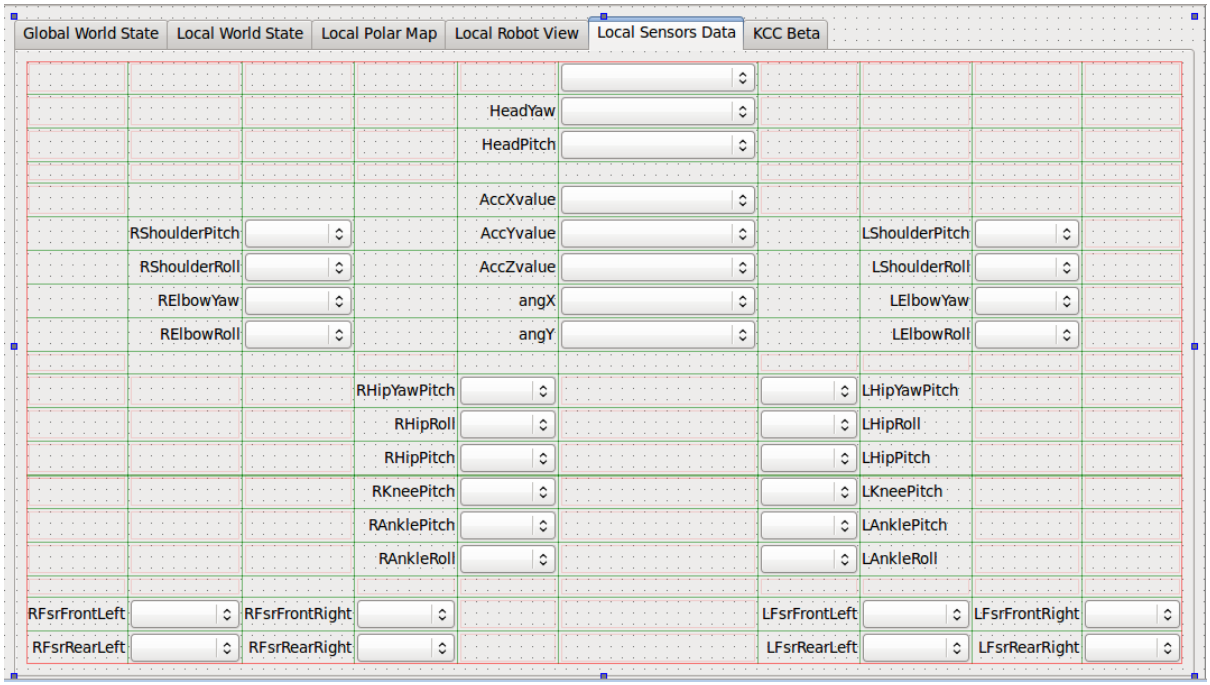

Figure 6.6: Local Sensors Data's User Interface

6.10 Local Sensors Data Tab's User Interface

Because of its provided functionality, the Local Sensors Data tab was predefined so as to be a collection of Qt4's `QComboBox` and `QLabel` widgets. A total of 38 `QComboboxes` were utilized in order to visualize the current sensors measurements and 37 `QLabels` close to each combobox item to depict the sensor name. Additionally each combobox holds the ten latest values of the sensor which visualizes. The `QComboBox` on the top center pane visualizes the active remote hosts and the rest of the utilized comboboxes were placed in such a way so as to depict the actual position of each sensor on the robot's body, as shown in Figure 6.6.

6.11 Local Sensors Data Tab's `View-Controller`

The `View-Controller` of the fifth tab called the `LSDController`, a customized `QObject` class, is responsible for updating the comboboxes with the `AllSensorValuesMessage` received by the `MessageAllocator` module, based on the user's host request made from the

`LocalRemoteHost` widget. Seven circular buffers were used to hold the ten latest sensor values of each kinematic chain, of the inertial unit and the FSRs. The constructor of the class initializes the structures and sets the capacity of the buffers to ten. The public slot `sensorsDataUpdateHandler(AllSensorValuesMessage, QString)` is connected with the `MessageAllocator` signal `sensorsDataUpdate(AllSensorValuesMessage, QString)` and whenever it is executed, it updates the buffers with the received sensor message and calls the corresponding function which connects the buffers contents with the `QComboBox`es of the Graphical User Interface of the tab.

Chapter 7

Results

In order to present the functionality of KMonitor five different scenarios were implemented, executed, and recorded. In the sections below we briefly describe each scenario, depict representative snapshots of their progress, and discuss the outcomes. In the figures below, snapshots on the left were taken from a regular video camera and show the real SPL soccer field, whereas snapshots on the right were taken from KMonitor; side-by-side snapshots are synchronized.

7.1 Monitoring the Global World State

The first scenario concerns the execution of a real SPL game, with the only difference that the two competing teams consist of two robots each (one goalkeeper and one attacker). Thus, we present the functionality of the Global World State tab and examine the global visualization and monitoring process offered by KMonitor. Initially, the four players are switched off and placed at the side lines of their own half of the field, as shown in the first row of Figure 7.1 (left), therefore the Global World State tab of KMonitor is in its initial visualization without any available active robots to monitor, as shown in the first row of Figure 7.1 (right). After the robots successively boot up, KMonitor detects the four active robots and visualizes them in the tree widget, whereas their graphical elements are placed at the predetermined positions in the virtual field (second row of Figure 7.1). Note that only three robots are shown on the 2D virtual field scene, since we have selected to monitor only three of the four available robots; this is the reason why in the hosts widget the last robot contains no game state information and is shown in green.

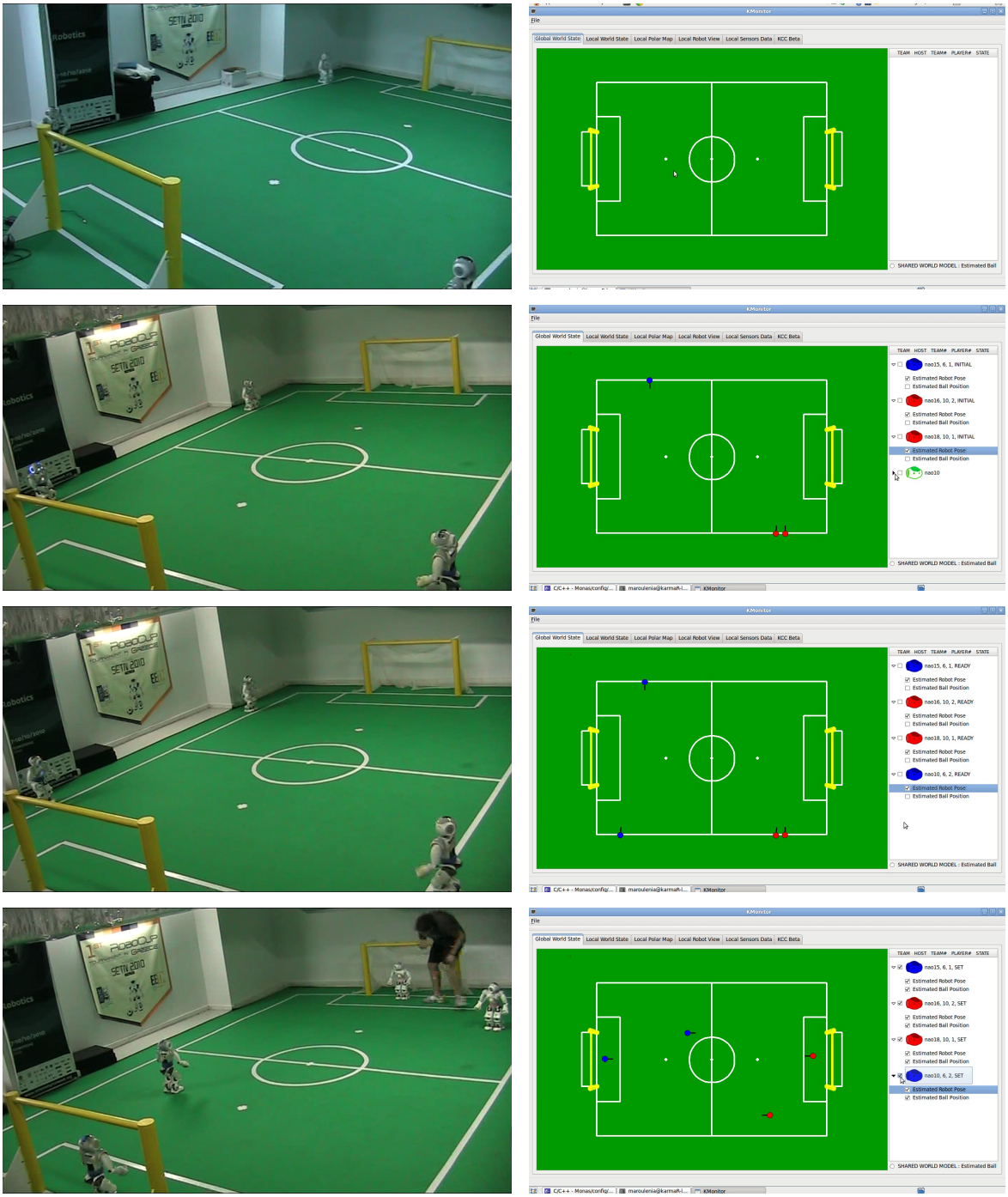

Figure 7.1: Monitoring the Global World State during a real SPL game

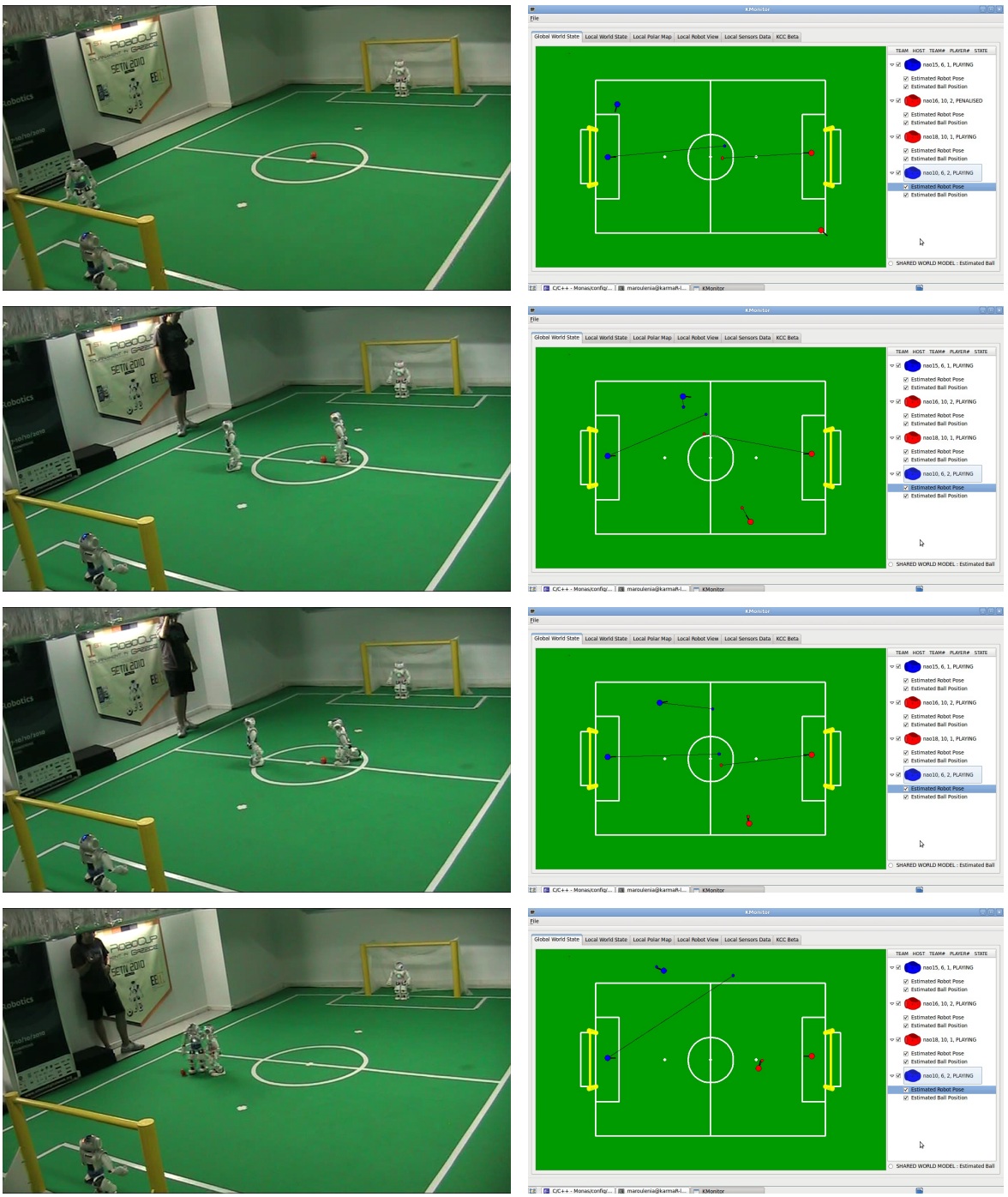

Figure 7.2: Monitoring the Global World State during a real SPL game (cont'd)

In the third row, using the Game Controller, we set the players to the `READY` game state, where autonomous placement to kick-off positions takes place. Having selected all active hosts in KMonitor we are able to monitor their estimated robot pose, inferring that the beliefs of three players match their real poses, whereas one red player has converged at a wrong estimated robot pose. After the end of the autonomous placement phase (`READY` state), the game moves to the `SET` state and the estimated robot poses match the real ones (fourth row of Figure 7.1). After the ball is placed by the referee at the center of the field, the game begins and we select to monitor both the estimated robot pose and the estimated ball position of all players (Figure 7.2). We can see that both goalkeepers have accurate estimated poses and the error in their estimated ball positions is less than $0.3m$. In contrast, the blue attacker has a wrong estimated pose which is off by $0.45m$ and about $20°$. The red attacker has been penalized as indicated in the tree widget. As the game continues, we see that the error in the estimated poses and ball positions increases, indicating potential problems in the localization software module. On the other hand, the vision software module seems to be working quite well, since all the robots have accurate knowledge of the position of the ball, as shown on the virtual field scene of KMonitor; the mismatch between the estimated ball positions is due to mistakes in the estimated robot poses.

7.2 Monitoring the Local World State

The second scenario concerns the behavior of a single player in the soccer field, which starts from the predefined initial position on the side line and tries to score a goal with a ball placed at the center of the field. In this scenario, we examine the functionality provided by the Local World State tab of KMonitor. As shown in the first row of Figure 7.3, we have selected to visualize the estimated robot pose, the player's view field projection, the landmark observations, and the population of the localization particles to inspect the single robot behavior. We can see that the initial estimated robot pose converges to the real pose and the particles are equally distributed along the side lines of the own half of the field. In the second row, we can see the distribution of the population of particles in the field and the final estimated robot pose, as well as an erroneous right goalpost observation inside its camera view. In the third row, we have selected to visualize, apart from the aforementioned features, the estimated ball position

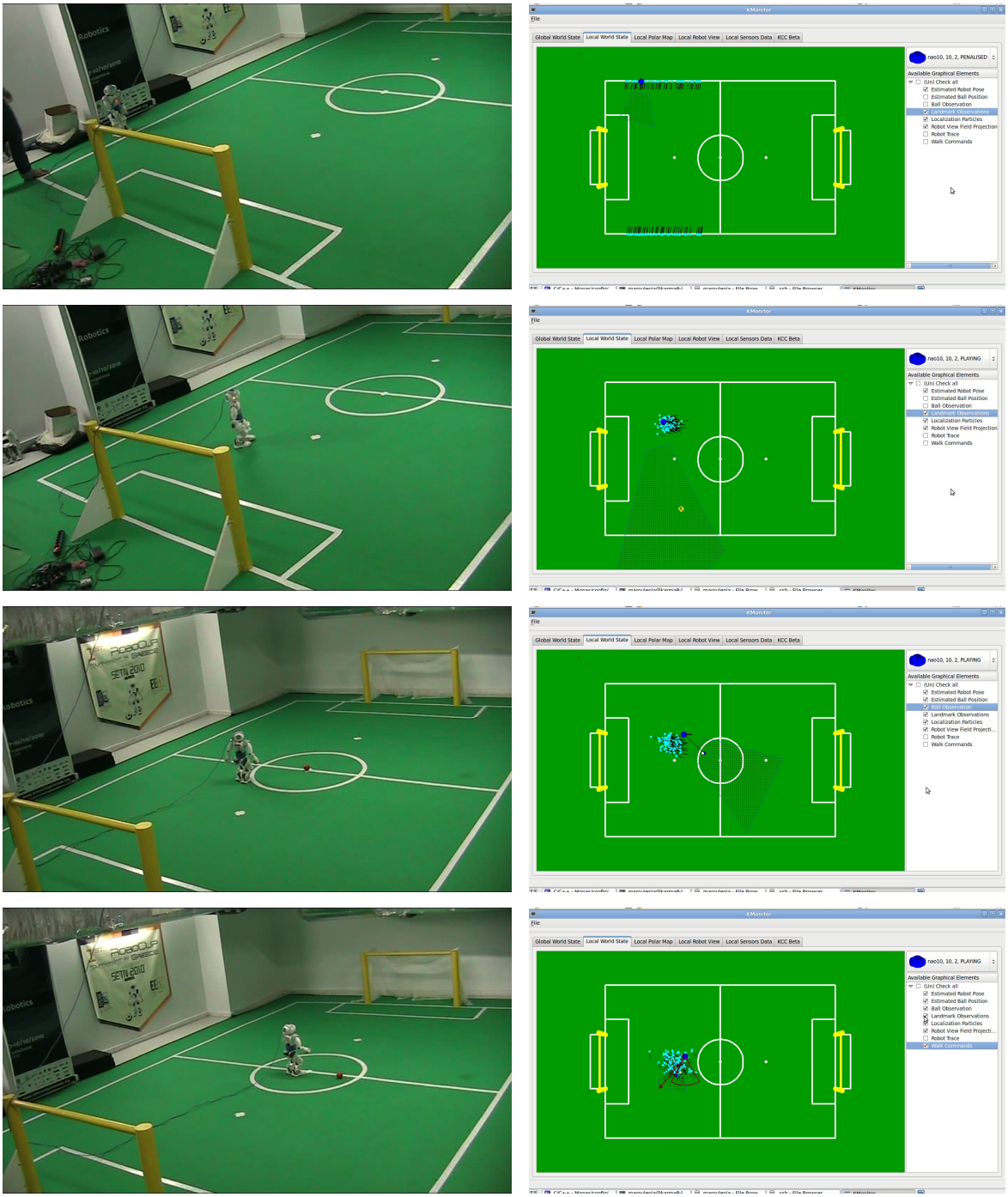

Figure 7.3: Monitoring the Local World State

and the instantaneous ball observations. We can see from KMonitor that there is an error of about $0.6m$ in the estimated robot pose, but the robot has a very good estimate of the ball position, which matches both the current ball observation and the real ball position on the real field. Finally, in the fourth row, we have added to the monitoring process the motion commands. As we can see, despite the error in the estimated robot pose, the decided motion command dictates the robot to move laterally to the right with maximum speed and rotate clockwise by about 60 degrees, that is, towards the ball.

7.3 Monitoring the Local Polar Map

In the third scenario, the player is placed at the center of the field, the ball is placed in front of the opponent goal, and another player (in `PENALISED` game state) is placed in front of the player slightly to the left. After the player has detected the ball and starts walking towards it, an additional player (in `PENALISED` game state) is placed in front of him, as an unexpected obstacle, to test the performance of the obstacle avoidance module via KMonitor. In all rows of Figure 7.4 we have selected to visualize all the relevant elements. In the first row, we can see the correct estimation of the obstacle position in the front left area of the player; there is no target and path, because the player has not seen the ball and has not set a target yet. In the second row, the player is moving towards the ball and we can see how the obstacle has been shifted to the left side of the player due to map transformation as the robot moves. In the third row, the unexpected obstacle appears and we observe that the probability of occupancy of the map cells in the right front area of the player increases and the derived path suggests a 45-degrees rotational movement to avoid the obstacle and reach the target. As the robot gets closer to the obstacle (fourth row), the obstacle is detected by both sonars and is placed right in front of the player, as indicated by the increased probability of occupancy of the corresponding cells. Path planning now derives another obstacle-free path which begins with a lateral movement to the left to avoid the obstacle and then a curved movement to reach the target.

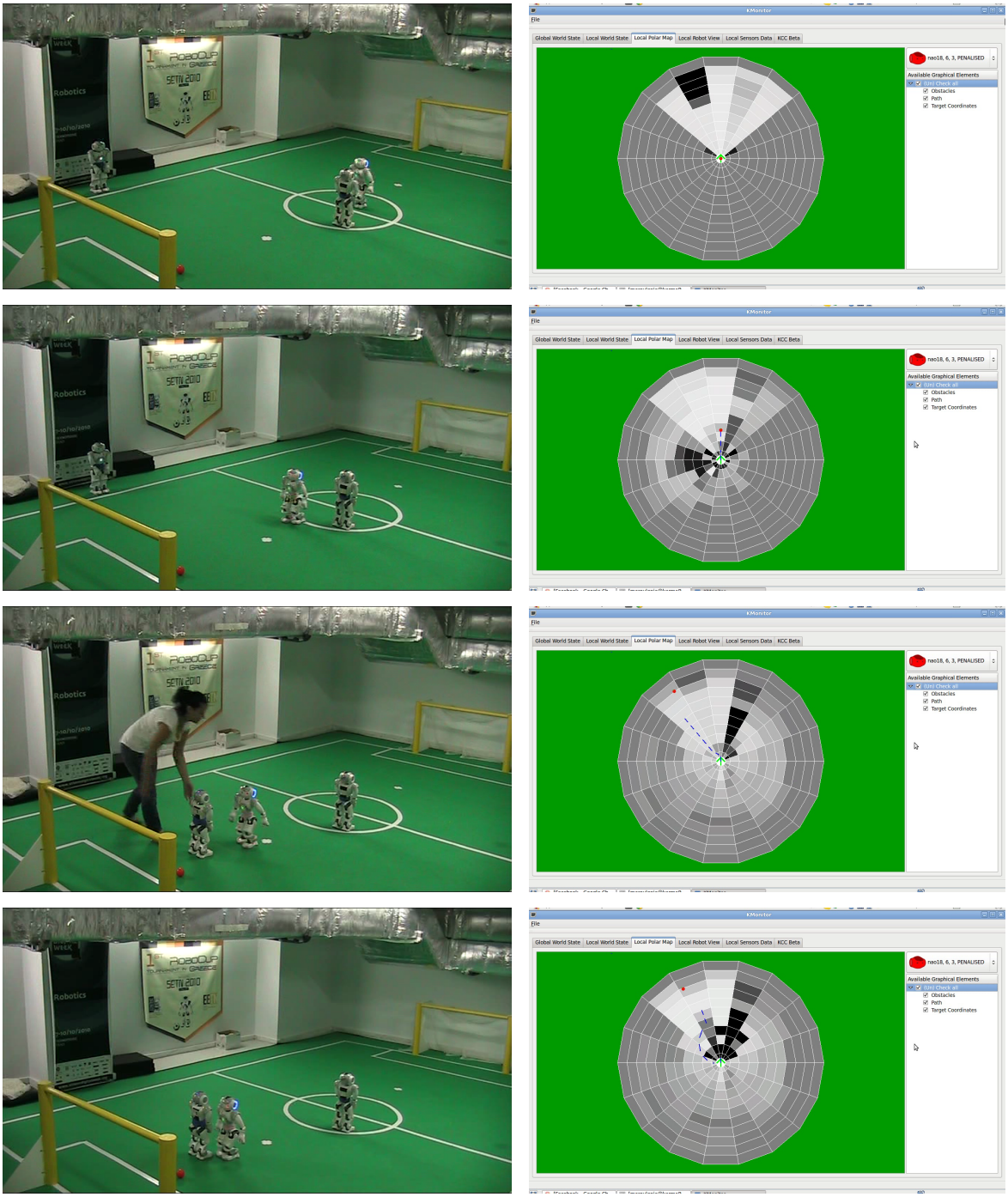

Figure 7.4: Monitoring the Local Polar Map

7.4 Monitoring the Local Robot View

In this scenario, we demonstrate the performance of the Local Robot View tab. A single player starts from the predefined initial position on the side line, walks towards the ball, which is placed at the center of the field, and kicks it. We monitor the image acquired by the bottom robot camera, which is the one used exclusive by our vision module. Note that in this scenario the robot is connected to KMonitor via a wired network link due to the extreme load on the network when transferring images. In the two first rows of Figure 7.5, we have selected to monitor the raw image from the robot camera, whereas in the last two rows we monitor the color segmented-image. We can observe that color recognition works quite well on the robot with the loaded color map. Almost all pixels of the ball are correctly recognized as orange. This is also true for the penalty mark and the robot body parts, which are recognized as white. Several pixels of the field carpet are misidentified as irrelevant (black) due to shades, however at this proportion there is no negative impact on ball recognition. Finally, from the last row, it is easy to see that the pixels of the goal are correctly recognized as yellow, whereas the background yields mostly irrelevant pixels.

7.5 Monitoring the Local Sensors Data

Finally, in order to present the functionality of the Local Sensors Data tab, we switch off the robot's stiffness on all joints and we visualize the sensor readings through KMonitor as we manually move the joints. We present the readings, after having moved the head left and right, the left hand, the right hand, the right leg, and the left leg respectively (Figure 7.6). We can easily observe the change in the values of the encoders of the corresponding kinematic chains according to the moved part of the robot's body.

7.6 Usability

Kmonitor is already being tested by all members of team Kouretes in various tests in the lab, but also in actual team deployments with the most recent one being the RoboCup 2012 competition in Mexico City. As evidenced by all team members, the team setup and

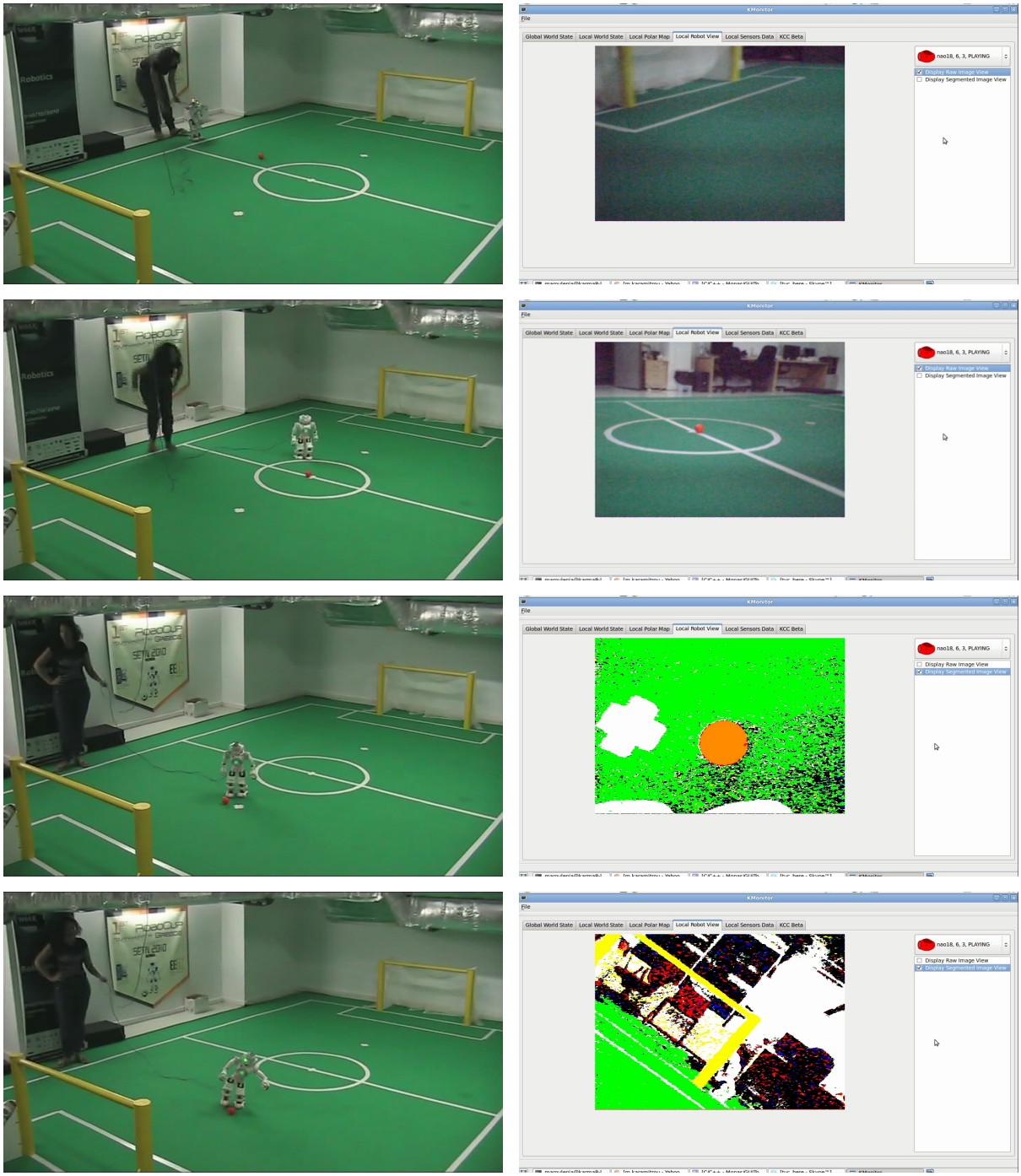

Figure 7.5: Monitoring the Local Robot View

Figure 7.6: Monitoring the Local Sensors Data

fine tuning on site was highly facilitated by the use of KMonitor, in contrast to previous years, where such a monitoring application was missing.

An advantage of KMonitor is that multiple users can run their own instance of KMonitor to visualize and inspect the same or different robots in the field simultaneously without any conflict or additional overhead. The underlying KNetwork mechanisms ensure that topic subscriptions and unsubscriptions are done properly, so that there are no duplicates, if several users choose to monitor the same data, and no useless network traffic, when all users have unsubscribed from a certain topic. This feature allows all members of Kouretes to use KMonitor on their laptops simultaneously to monitor the data most related to their software modules and area of responsibility.

Chapter 8

Conclusion and Future Work

Conclusively, debugging robotic software can be challenging because of the continuous interaction between the agent and the environment and the real-time aspect of this interaction. KMonitor, our integrated real-time visualization and monitoring application, facilitates the debugging of the entire robot code and allows the developer to not only assess each module individually, so as to test the performance of isolated functionality, but also jointly as components of a larger integrated software system, so as to test its overall performance. Furthermore it integrates the required functionality and views under a single intuitive graphical user interface, facilitating the user in switching quickly between different views in order to monitor different aspects of the robot software.

8.1 Future Work

The current Kouretes robotic software provides each robot with XML-based configuration data concerning required parameters for each module of the existing code. The modification of the aforementioned configuration is required frequently to make the code modules fit the different game environments and improve their performance. Currently, the configuration loaded at boot time is preserved unchanged until the next restart of the NAOqi middle-ware. Any change in configuration requires no compilation, however it still requires modification and upload of XML-files and a time-consuming restart of NAOqi. With the appropriate configuration handling process from the robot software, a new functionality can be implemented and integrated to KMonitor to allow for remote

real-time visualization and modification of the robot configuration without the need of stopping the code execution or restarting NAOqi.

The sixth tab (KCC Beta) of KMonitor represents one example of how other graphical tool can be integrated into KMonitor. In addition to KCC, there are two more useful tools developed within our team, which can integrated into KMonitor in the future: the Kouretes Motion Editor (KME) [21], an interactive graphical tool for designing, testing, and executing complex motion patterns, and Kouretes Statechart Editor (KSE) [10], a CASE (Computer-Aided Software Engineering) tool for model-based development of robot team behaviors using statecharts.

Currently, KMonitor supports only online monitoring. However, KMonitor could be extended to support also offline monitoring, provided that full data and video logs can be recorded from the robots with an appropriate synchronization mechanism. Additionally, some kind of log replay functionality must be added to KMonitor to allow for easy browsing of recorded log at variable speeds.

References

[1] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E., Matsubara, H.: Robocup: A challenge problem for AI. AI Magazine **18**(1) (1997) 73–85 5

[2] RoboCup SPL Technical Committee: Standard Platform League rule book (2012) Only available online: www.tzi.de/spl/pub/Website/Downloads/Rules2012.pdf. 6

[3] Gouaillier, D., Blazevic, P.: A mechatronic platform, the Aldebaran Robotics humanoid robot. In: Proceedings of the 32nd IEEE Annual Conference on Industrial Electronics (IECON). (2006) 4049–4053 6

[4] Aldebaran Robotics: Nao documentation (2011) Only available online: www.aldebaran-robotics.com/documentation. 7

[5] Paraschos, A.: Monas: A flexible software architecture for robotic agents. Diploma thesis, Technical University of Crete, Greece (2010) 11

[6] Orfanoudakis, E.: Reliable object recognition for the RoboCup domain. Diploma thesis, Technical University of Crete, Greece (2011) 11

[7] Chatzilaris, E.: Visual-feature-based self-localization for robotic soccer. Diploma thesis, Technical University of Crete, Greece (2009) 12

[8] Kyranou, I.: Path planning for nao robots using an egocentric polar occupancy map. Diploma thesis, Technical University of Crete, Greece (2012) 12

[9] Tzanetatou, D.: Interleaving of motion skills for humanoid robots. Diploma thesis, Technical University of Crete, Greece (2012) 12

[10] Topalidou-Kyniazopoulou, A.: A case (computer-aided software engineering) tool for robot-team behavior-control development. Diploma thesis, Technical University of Crete, Greece (2012) 12, 98

[11] Vazaios, E.: Narukom: A distributed, cross-platform, transparent communication framework for robotic teams. Diploma thesis, Technical University of Crete, Greece (2010) 13

[12] Röfer, T., Laue, T., Müller, J., Fabisch, A., Feldpausch, F., Gillmann, K., Graf, C., de Haas, T.J., Härtl, A., Humann, A., Honsel, D., Kastner, P., Kastner, T., Könemann, C., Markowsky, B., Riemann, O.J.L., Wenk, F.: B-Human team report and code release 2011 (2011) Only available online: www.b-human.de/downloads/bhuman11_coderelease.pdf. 25

[13] Tasse, S., Kerner, S., Urbann, O., Hofmann, M., Schwarz, I.: Nao Devils Dortmund team report 2011 (2011) Only available online: www.irf.tu-dortmund.de/nao-devils/download/2011/TeamReport-2011-NaoDevilsDortmund.pdf. 27

[14] Barrett, S., Genter, K., Hester, T., Khandelwal, P., Quinlan, M., Stone, P.: Austin Villa 2011 technical report UT-AI-TR-12-01. Technical report, The University of Texas at Austin (2011) Only available online: www.cs.utexas.edu/~pstone/Papers/bib2html-links/UTAITR1201-sbarrett.pdf. 27

[15] Michel, O.: Webots: Professional mobile robot simulation. Journal of Advanced Robotics Systems **1**(1) (2004) 39–42 27

[16] Tilgner, R., Reinhardt, T., Borkmann, D., Kalbitz, T., Seering, S., Fritzsche, R., Vitz, C., Unger, S., Eckermann, S., Müller, H., Bellersen, M., Engel, M., Wünsch, M.: Nao-Team HTWK team research report 2011 (2011) Only available online: http://robocup.imn.htwk-leipzig.de/documents/report2011.pdf. 28

[17] Visser, A., ten Velthuis, D., Verschoor, C., Wiggers, A., Bodewes, B., Cabot, M., van Egmond, E., Fodor, E., Gieske, S., Iepsma, R., Jetten, S., Jozefzoon, O., Keune, A., Koster, E., van der Molen, H., Nugteren, S., van Rossum, T., Rozenboom, R., van Zanten, J., van Bellen, M., Laan, S.: Dutch Nao Team technical report 2011 (2011) Only available online: www.dutchnaoteam.nl/wp-content/uploads/2011/10/TechnicalReport.pdf. 29

[18] UPennalizers: The University of Pennsylvania Robocup 2011 SPL Nao soccer team (2011) Only available online: www.seas.upenn.edu/~robocup/files/upenn_team_desc_spl_2011.pdf. 30

[19] Burkhard, H.D., Holzhauer, F., Krause, T., Mellmann, H., Ritter, C.N., Welter, O., Xu, Y.: NAO-Team Humboldt 2010 the RoboCup NAO team of Humboldt-Universität zu Berlin (2010) Only available online: www.naoteamhumboldt.de/wp-content/uploads/2010/02/NaoTH10Description.pdf. 31

[20] Panakos, A.: Efficient color recognition under varying illumination conditions for robotic soccer. Diploma thesis, Technical University of Crete, Greece (2009) 42

[21] Pierris, G.: Soccer skills for humanoid robots. Diploma thesis, Technical University of Crete, Greece (2009) 98