Enhancing the performance and testability of the MI20 robot soccer system

Paul de Groot

Enhancing the performance and testability of the MI20 robot soccer system

Paul de Groot

A Master's thesis in computer science.
Twente University
Human Media Interaction
Date: 28 August 2006

Graduation committee:
Dr. Mannes Poel
Dr. Albert Schoute
Prof. dr. ir. Anton Nijholt
Ir. Thijs Verschoor

Computers use what mathematicians call the binary system, in which there are only two numbers, 0 and 1. In some ways, this is a disadvantage for computers.

For example, they are incapable of doing this cheer:

"Two, four, six, eight! Who do we appreciate?"

Instead, computers have to cheer thusly:

"One, zero! Who's our hero?"

This cheer is not nearly as effective, which is why, although computers are getting really good at chess, they still suck at football.

-- Dave Barry in Cyberspace, by Dave Barry

Abstract

For four years Twente University is a competitor in the FIRA robot soccer league. With a software system completely written from scratch, the team called MI20 has been competing and even producing some notable results.

However, factors such as the constant change of developers and the clear-cut nature of Master's assignments have resulted in the organic growth of the system, giving problems with manageability and introducing bugs. Also, students reported having difficulties with testing the system, due to the time-consuming nature and low reproducibility of such tests.

In this thesis, the efforts in redesigning and re-engineering the MI20 robot soccer system are described. Existing code has been scrutinized and adapted if necessary. The redesign has been done to ensure extensibility, usability and to make the system easier to grasp.

To further accommodate the manageability of the system, a set of tools was (re)introduced to facilitate the development process. The source code management system Subversion is now used, documentation is automatically generated and a bug tracker ensures that defects found are not neglected. A newly written tool called Builder tackles problems with Makefiles, whilst keeping every developer free in their choice of a development environment.

Testing the system has now become much easier due to built-in facilities for gathering and visualizing measurement data, which can be exported for further processing in Excel, Matlab or numerous other programs. Games played can be recorded, so that a means of using reproducible input for testing is provided and games can be reviewed to judge our strategy.

Finally, a simulator has been developed so that preliminary tests can be performed without having to set up the complete game. Also, simulation allows more control than a real-life game situation. The simulation has a realistic physics and collision model and can detect violations of game rules. Added benefit is that this simulator can be used for machine learning, allowing new strategies to be developed.

With this new design development on the MI20 system should be easier and less error-prone, hopefully making the team from Twente once again a force to be reckoned with.

Samenvatting

De Universiteit Twente neemt nu vier jaar deel aan de FIRA robotvoetbalcompetitie. Met compleet zelfgeschreven software strijdt het team MI20 mee en boekte daarbij zelfs enkele goede resultaten.

Echter, door het constante verloop van ontwikkelaars en de sterke afbakeningen van de afstudeeropdrachten is de groei van het systeem organisch van aard geweest. Dat blijkt nu problemen te geven met het onderhoud van het systeem en is de oorzaak van bugs. Ook ondervinden studenten problemen met het testen van het systeem, doordat tests vaak tijdsintensief en slecht reproduceerbaar zijn.

In dit verslag worden het herontwerpen en herontwikkelen van het MI20 robotvoetbalsysteem beschreven. Bestaande code is kritisch bekeken en waar nodig aangepast. Het nieuwe ontwerp legt de nadruk op uitbreidbaarheid, gebruiksvriendelijkheid en streeft ernaar dat nieuwe studenten sneller bekend raken met het systeem.

Om de hanteerbaarheid van het systeem nog verder te vergroten, zijn enkele tools (opnieuw) geïntroduceerd in het ontwikkelproces. Zo wordt voor het beheren van de broncode nu Subversion gebruikt, wordt automatisch documentatie van de code gegenereerd en wordt met een bug tracker gezorgd dat gevonden fouten niet genegeerd worden. Een nieuw geschreven programma, Builder genaamd, vormt een oplossing voor de problemen met Makefiles, zonder dat ontwikkelaars een bepaalde ontwikkelomgeving opgedrongen wordt.

Testen van het systeem is makkelijker geworden door de implementatie van functionaliteit om meetgegevens te verzamelen en te visualiseren. Deze data kan ook worden geëxporteerd zodat verwerken in Excel, Matlab of andere programma's mogelijk is. Wedstrijden kunnen worden opgeslagen, zodat deze als reproduceerbare invoer voor testen kunnen dienen, of om de gevoerde strategie te beoordelen.

Tenslotte is een simulator ontwikkeld zodat tests kunnen worden uitgevoerd zonder dat het gehele speelveld hoeft te worden opgezet. Ook biedt de simulator meer controle dan een spelsituatie in realiteit. De

simulator heeft een realistisch natuurkundig en botsings-model en is in staat om de navolging van de spelregels te controleren. Bovendien kan de simulator worden gebruikt voor machine learning, zodat nieuwe strategieën kunnen worden ontwikkeld.

Met dit nieuwe ontwerp moet de verdere ontwikkeling van het MI20 robotvoetbalsysteem makkelijker en minder vatbaar voor fouten worden, zodat het team uit Twente weer een tegenstander wordt om rekening mee te houden.

Table of Contents

CHAPTER

1 Introduction

An astonishing 1.2 billion people (being 17% of the world's population) watched Italy beat France during the FIFA World Cup football in 2006[1]. In comparison, a few dozen people watched the ruling world champion Socrates renew its world title in robot soccer.

Nonetheless, robot soccer forms an academic challenge for numerous teams in Europe and around the world, combining research areas like computer vision, artificial intelligence and control theory.

One of the teams that competes in this sport is the MI20 robot soccer team from Twente University. This thesis describes the work done on a new software architecture within this team.

1.1. What is robot soccer?

In June 1993 a group of Japanese researchers started the first robotic soccer league. Within a month, research groups from all over the world reacted enthusiastically on this initiative, signalling the start of a thriving competition. Its founders set an objective for the academic world: "Design a team of soccer-playing robots, that can beat the human World Cup finalists in 2050[1]." [2]

In 1995, another robot soccer association, the Federation of International Robotsoccer Association (FIRA), was established in Korea. This association too has grown to become a world-wide organization, and includes the MI20 team from Enschede.[3]

The two federations each hold competitions in various leagues, ranging from matches with one meter high, fully autonomous robots with pneumatic shooting mechanisms to robotic dogs and tiny robots of only 5.5 centimetres in height.

1 This then would have to be the winners of the World Cup in 2048, since the championship is held every 4 years, since 1930. Therefore, academics will have 2 years to tune their strategy to their competitors.

The MI20 team competes in the MiroSot league, playing with robots that cannot be larger than 7.5 x 7.5 x 7.5 cm. A team consists of 5, 7 or 11 players depending on the type of match. The pitch on a game of five against five measures 220 by 180 centimetres, for a 7x7 game the pitch is 280 x 220 centimetres.

Robots wear colour patches: pieces of cardboard with distinct colours so that the computer can identify the robots in the image easily. Regulations prescribe that the ball has to be an orange golf ball. Each team observes robots and ball using a camera placed above the field, sending its images to a computer.

This computer processes the images, makes strategical decisions about the robot's actions, and sends new commands to the robots using a radio frequency transmitter.

A game consists of two halves of 5 minutes of actual playing time each. During play no human intervention may take place, other than starting and stopping the game.[4]

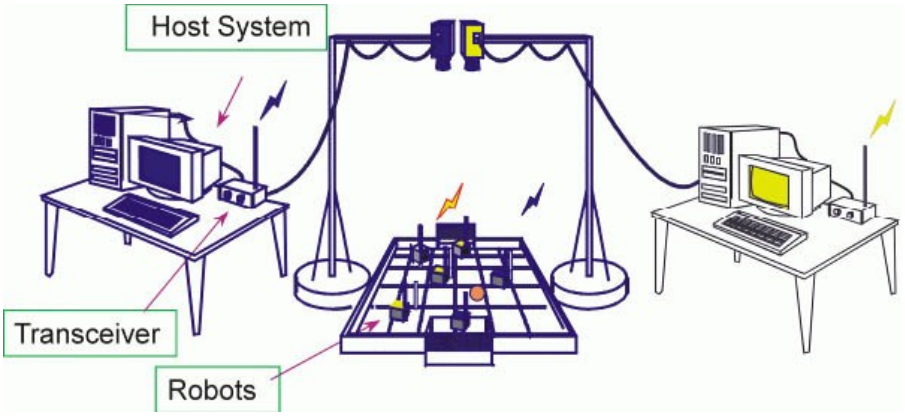


Figure 1.1. - Typical set up of a robot soccer game.

One of the first appearances of MI20 at a championship yielded quite good results: the team finished 4th at the European championship in Ljubljana. Sadly, at the World Championship of that year the third place in the group was all that could be reached (due to problems with the transmitters and two teams of the group later reached placed 2 and 3 in the tournament).[5][6]

1.2. Problem statement

In October 2005, it became clear that during more than three years of development of the MI20 robot soccer system, it had not only been extended with many useful features, but at the same time had become unmanageable.

In the course of the years, the code size has increased to 28,000 physical lines of code[2] (1.3 megabytes). Unfortunately, the *organic* characteristic of the growth introduced several subtle and not-so subtle bugs, resulting in an unacceptable high processing load and inexplicable errors. Consequently, it

2 A physical line of code is a non-empty, non-comment line.

became difficult to play a game of robot soccer, especially outside the standard situation of our lab.

One possible cause of the organic growth that hurt the manageability of the system so much, might be that students got increasingly slack about the development process. Although at first a source code management system was used, later every student worked in his own copy of the code. In many cases, documentation of the code was lacking.

Additionally, students reported having difficulty pin-pointing the source of the shortcomings observed. The system processed a wealth of information, but lacked a straightforward manner of viewing this information. Testing was done sparsely and with great difficulties because of the bad reproducibility of test cases.

These problems were the reason to write out an assignment, making sure the MI20 robot soccer system becomes once again a stable test bed for academic assignments and MI20 becomes a worthy adversary for other robot soccer teams.

1.3. Assignment

The purpose of this assignment is to make a complete revision of the system to solve the performance problems and restore a proper working environment for software development and testing.

The assignment consists of the following sub-goals:

1. A complete rewrite of the code starting from scratch, while gradually checking and optimizing code from the old system and incorporating it into the new. The modularity of the original system must be kept in tact, but to lower the multi-threading processing overhead, modules have to be merged into a single program instead of separate executables.
2. Introduce tools with respect to version control, software design, debugging, documentation, etc. to improve the software development process.
3. Provide facilities for testing, visualization and gathering data, inspired by the so-called Supervisory Control And Data Acquisition (SCADA) systems. Data logging must enable post-processing by Excel, Matlab or other applications.
4. Enable the integration of a simulation environment to ease the development process that allows for reproducible testing and playing games faster than real time.

1.4. Thesis outline

This thesis will describe the work done to fulfil the assignment above. First,

in chapter 2 the software architecture of version 2.0 of the MI20 robot soccer system will be presented.

In chapter 3 a description will be given of the tools and changes that were introduced to improve the software development process.

Then, in chapter 4, the data visualization features that have been added to the robot soccer system will be presented.

Furthermore, a simulation environment has been developed that enables students to test and create strategies more easily using machine learning. This simulator will be elaborated on in chapter 5.

Finally, chapter 6 contains conclusions about the work done and recommendations for further work and research.

CHAPTER

2 The MI20 application framework

In October 2006, the MI20 robot soccer system had seen such uncontrolled growth that it had become difficult to play a game of robot soccer at all. To remedy this, the system was rewritten and audited.

In this chapter, the redesign done and the rationale behind some of the decisions made is explained. Basic topics in this chapter are the several modules into which the system is divided, the inter-module communication and several other changes that were made to increase the system's maintainability.

2.1. Towards version 2.0

When the project started in 2002, the road to a working robot soccer system was still unpaved. Designing a system like this is a demanding task, because it touches upon a large number of research fields and it can be quite overwhelming to incorporate all these pieces into a well-working system. Nevertheless, the three first master students (Werner Dierssen, Remco Seesink and Niek Kooij) were successful, and a first version of the system was created.

During the years after that, many Master students worked on the program, adding new strategies and adapting existing algorithms. In September 2005, the system consisted of 1.3 megabytes of code, and had approximately 28,000 physical lines of code. Unfortunately, the system had grown organically, so that the code became somewhat messy. This made it more and more difficult to ensure the correct working of the program and to keep track of the big picture. Also, hard-coded values made it impossible to run the program on anything different than the current development machines.

In October 2005, the decision was made to do a complete rewrite of the code. A new program was started from scratch, gradually checking and optimizing all code from the old system and incorporating it into the new.

A rewrite like that can be compared to rebuilding a decrepit building. It is torn down and the foundations are inspected and – where necessary – patched up. Then you go through the building blocks that remained after tearing down the building and you build up the house; keeping the building blocks that are reusable and creating new ones where the old ones faltered. The decision to start completely from scratch was made to ensure that really every old piece of the system was reviewed and checked before placing it in the new version.

During this rewrite, the modularity of the original system was kept in tact, but the modules were merged into a single program instead of separate executables. Some of these modules still work just as they did in the first version of the system. In other cases, the software architecture within the module was changed radically. The current layout of the system is explained in detail in paragraph 2.4.

Inter-module communication was changed to make use of an event-based communication system, which reduced the needs for multi-threading and instead transformed the system into a *cooperative multitasking system*. The architecture and the benefits of such a system are elaborated on in paragraph 2.6.

In January 2006, main development was shifted from the old code base to the new version 2.0. Since then development was done on the new version, that now has reached version 2.2. The code now contains 46,000 physical lines of code and is more than 2 megabytes large.

The increase in lines of code has multiple reasons. First of all, during the past year development of the system has continued and now contains features like auto-positioning[3], an expanded user interface and data visualization (see chapter 4).

Another reason is the adoption of a code style in which the braces (`{` and `}`) that start and end a block of code are placed on a line by themselves. This hardly increases the byte count of the code, but creates a surge in the amount of physical lines of code.

2.2. Modularity

The MI20 system has always been designed with modularity in mind. This means that several parts of the system were identified that have a fairly self-contained function. The working of such a module can be regarded as a black-box: other modules do not and need not care about the inner workings of the module. All they require from the module is that it adheres to the rules that were set for inputs and outputs.

This modularity has a couple of advantages. First of all, it reduces the impact that changes in the code may cause. Because the functionality of a module is constrained to itself, when a change is made to a module, it can only affect the output of the module. Only the modules that use that information can be affected by a change of the module.[7]

3 Robots can be moved to predefined positions in response to a game situation without human intervention.

Also, the interfaces between the modules form good starting points to track errors. If the inputs of a module are correct, but the corresponding outputs are not, the error must be located within that module, reducing the amount of code that needs to be searched.
Finally, keeping modularity in the code makes it easier to perform regression and unit testing.

The following modules are currently identified in the MI20 system:

- **Vision** – Captures images from the camera, performs colour segmentation and tries to identify the robots and the ball. A snapshot will never contain anything time-related, like velocity or acceleration.
- **State estimator** – The state estimator uses the snapshots from the vision module to get the best possible estimation of the world state. This entails the filtering of noise from the measurements, and identifying the robots using a tracking algorithm by combining this measurement with previous measurements.
- **Strategy** – The strategy module decides what the actions of the robots should be by looking at the current world state. Different decision systems can be used by the strategy module to create different behaviour.
- **Motion** – The motion module bridges the gap between the higher level commands from the strategy module (e.g. "drive to (1800, 500)" or "shoot the ball in the goal") and robot commands. It uses these commands to generate linear and angular velocities every frame. It also contains a feedback controller to ensure the robot is driving the correct path.
- **RFComm** – The RFComm module is responsible for taking the desired speeds determined by the Motion module, and send the corresponding commands to the robots.
- **Joystick** – A joystick can be attached to the system and used to control the robots manually. This is useful for demonstrations or for quick testing.

In the original implementation of the system, all of the modules were separate executables. Starting the system was then done using a batch script, which started all of the separate executables. This situation was hardly ideal, since it not only made debugging harder, it also made sure inter-module communication was hardly trivial (see 2.6 Inter-module communication).
In version 2.0, this was changed so that the system has just one executable, in which modules can be chosen to be or not to be started.

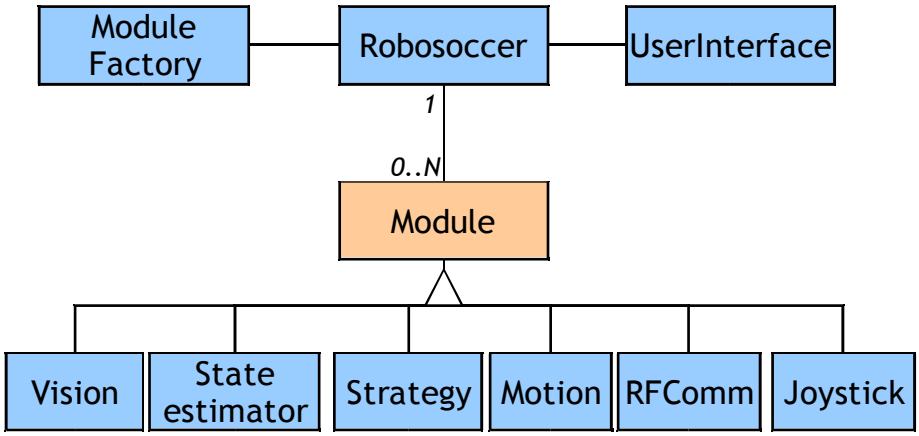


Figure 2.1. - The top level structure of the MI20 system version 2.0. The main class 'Robosoccer' creates a user interface, and any of the modules the user wishes to use.

This structure can be seen in figure 2.1. The class `Robosoccer` creates a so-called *execution environment* by setting the common needs for all modules: inter-module communication, the user interface, etc. etc. This category also includes the classes for reading and writing configuration files. Then, all modules selected by the user to run, are created and started. Creation of these modules is done using the *Factory* software design pattern[8], which hides the implementation of the modules from the Robosoccer class. That class has no knowledge whatsoever about the modules other than the fact that they have an `init()`- and `deinit()`-method, to be called at the start and end of the program.

Starting the program is now much easier, and attaching a debugger to this single executable is much more convenient than before. Modules can be stopped and restarted at run-time, adding a convenient way of resetting a module.

2.3. Model-view-controller architecture pattern

The whole MI20-system can be regarded as using the Model-View-Controller architectural pattern[9]. In such an application, the data model, user interface and control logic are separated into distinct components. This way, the impact of changes to one component is limited.

The user interface or *view* of the MI20 system is just that: the GUI module. It takes the world state of the data model and shows it to the user.

Commands, given by the user via the user interface, are delegated to the *controller* of the system: the combination of the Strategy and Motion modules. These modules take the input of the user and the current state in the data model to determine the way in which the system (and thus the robots) should be controlled.

Then there are three possible data models for the MI20 system. The most common is the combination of Vision, State Estimator and RFComm. The current state of the world in represented on an actual playing field and is

recorded by the camera. The state estimator enhances this data to create a usable data model.

The fact that RFComm is listed here as part of the data model might be a bit counter-intuitive. However, part of the functionality of the data model is to respond to instructions from the controller to change state. This implies that the Controller specifies *what* should be changed, and the data model knows *how to change that*.[10]

When using this definition, Motion specifies the changes that should be made to the data model (i.e. what velocities the robots should have) and RFComm knows how to change that (by sending commands to the robots).

Another possible data model is the simulator. The simulator receives the commands of the Controller (again Strategy and Motion), simulates a time step and notifies the View and the Controller of the new update.

A third form of data model stems from the fact that games can now be recorded to a file. When playing back the data in the file, the current world state found in that file forms a third possible data model.

2.4. Module descriptions

In this section, for each module will be explained what the module does and what other modules it communicates with. Also, for each of the modules a class diagram will be presented that shows the class hierarchy for that particular module. In some cases, this class hierarchy differs greatly from that in version 1.0. In other cases, the same set-up is used in both systems.

2.4.1. Vision

The responsibilities of the vision module are twofold: getting images from the camera and processing these images to detect the location of the robots and the ball.

The Camera class uses the *Factory* software pattern [8] to create the appropriate camera class for the system. A loop – run in a separate thread – waits for new camera images to become available, and feeds these to the Segmentator.

The Segmentator analyses the image by breaking it up into

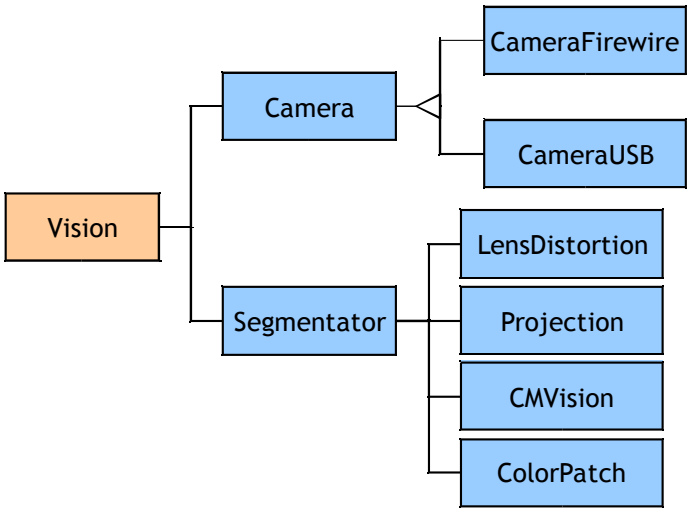


Figure 2.2. - Class diagram of the Vision module.

regions of a certain colour. To be contained in a region, the colour values of that pixel should fall into a colour cube in the YUV-colour space. The user specifies a series of these intervals, e.g. the colour of the ball or the opponent's team colour. The actual processing of the images is done by the `CMVision`-class, which was released under a GPL-license[11] by Carnegie Mellon University.

Using these regions, the Segmentator determines the (possible) locations of the robots and the ball. These locations are converted from image coordinates to world coordinates, using a projection matrix and the lens distortion. Then, these locations are placed into a data structure called a `Snapshot`.

Although, for example, it is assumed there is only one ball on the field, all regions that look like the ball are included into the Snapshot. The state estimator module then can use historical information to determine which of the regions is the most likely candidate to be the actual ball.

During the course of this assignment, the colour patches of the robots were changed from having 2 colours and being identical for each robot, to patches containing 3 colours, allowing us to identify each separate robot by its patch. This detection is facilitated by the `ColorPatch`-class.

Currently, we are using a Sony Firewire camera. Connecting to and retrieving images from this camera is done through two libraries called *libdc1394* and *libraw1394*. These libraries have been used in the MI20 robot soccer system since the beginning of the project, but during this code renovation the *libdc1394*-library was upgraded to version 2.0. Although at the time of writing this library is still in beta testing, the much better camera detection and other functionalities made interfacing with the camera much easier than using version 1.0 of the library.

2.4.2. State estimator

The state estimator uses the data from the Vision module to create the best estimation of the situation on the field.

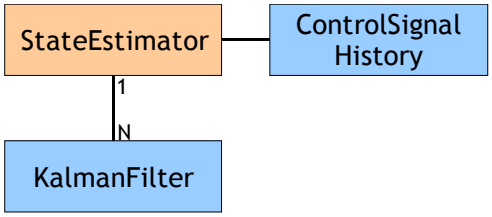


Figure 2.3. - Class diagram of the state estimator.

Vision handles each frame as a separate image, completely unrelated to the previous frame. The state estimator, however, uses historical information to determine the world state. This history is important, for example, when Vision has detected multiple candidates for the ball. The state estimator then uses the previous ball locations to select the region that is most likely the actual ball. Also, if the camera has not detected a robot or the ball (due to noise, obstructed view or shadows from the referee or audience), the state estimator can use the state history to make a prediction about the current location of that robot or ball.

The *ControlSignalHistory*-class buffers the commands that are sent to the robots. This is needed to make a more accurate prediction of the future world state. The MI20 system uses predictions of the world state fairly often. These can be long-term predictions (e.g. predict the location of the ball in 2 seconds), but the control signal history is of the utmost importance when making a short-term prediction (±100 ms).

This prediction is made every frame to compensate for a delay that exists between sending a command to a robot and receiving the measurement of the robot executing that command. If, during this delay, Motion would only react on the measurements it receives, it would assume the commands just sent were not executed. This would lead to unstable robot movement. Consequently, Motion uses a short-term prediction equal to the estimated command delay. The control signal history buffers these signals, and the prediction takes into account the delay. Effectively, Motion now determines its commands based on a future world state. This makes sense, since – due to that delay – the system is incapable of influencing the game situation between now and the time of that short-term prediction.

To reduce the effects of Gaussian noise, originating from noise in the camera images, the state estimator maintains an Extended Kalman Filter for each team robot.

2.4.3. Strategy

The strategy module is responsible for assigning each robot an action to perform. These actions are higher-order strategy decisions, such as blocking an opponent or trying to score a goal.

The actual decision making is a fairly difficult matter. Several Master students have tried different techniques in letting the computer decide the best action for any given world state, and several others will try to do so in the future. In the current framework, all these students will write their own type of *DecisionSystem*. At each update of the world data, the strategy

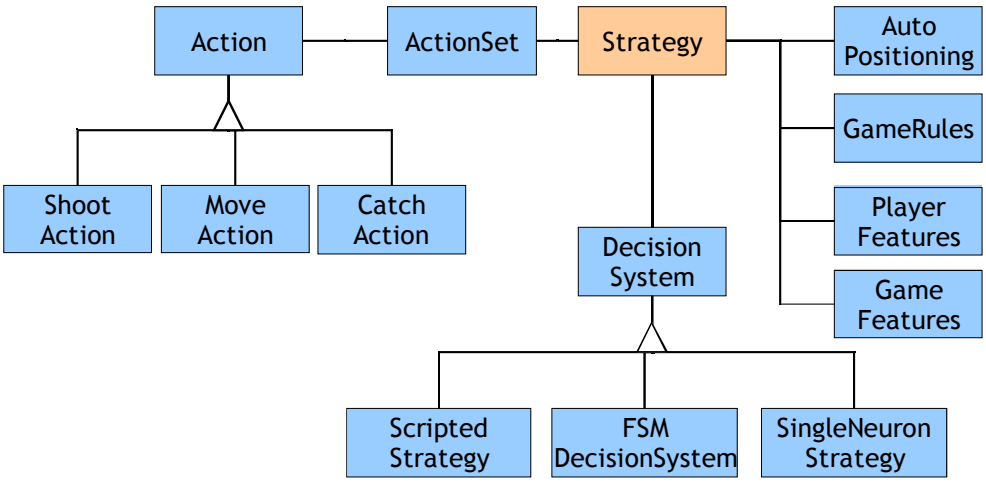


Figure 2.4. - Class diagram of the strategy module.

module will consult the currently chosen decision system to select the action the robots should currently perform.

The decision systems can make use of some standard utilities that are available to them. An example is the calculation of game and player features, which are values between 0 and 1, that form an abstraction of the current situation for a certain player or the whole game.[12]

After the decision system has decided what the robots should do, the Strategy module may override or adapt some of the actions. This should hardly ever occur, but may be needed to avoid collisions or to prevent game situations that are not allowed[4].

The decision system is only consulted in normal game play situations. When a special game situation has occurred (e.g. a goal kick), the Strategy module uses its auto-positioning functionality to let the robots drive to the correct positions.

The actions that the robots should perform are all grouped into an `ActionSet`. This set contains all the actions the robots are currently performing, including flags if the action is already finished. The ActionSet is shared between Strategy and Motion, by using the Singleton pattern. The Singleton pattern ensures that only one instance of a class ever exists within the application. This way, whenever Strategy or Motion request to access the ActionSet, they will always retrieve a reference to the single ActionSet within the application [8].

Changes to the ActionSet are made using the inter-module communication system. This makes sure that in case of a distributed system where Motion and Strategy are running on different machines, the ActionSet will always be updated correctly.

The class `Action` is an abstract base class and the actual events are subclasses of `Action`. This is done so that actions can contain parameters. For example, a move action contains the destination and the target orientation. Actions can also contain *optional* parameters. For instance, a move action can also contain a maximum speed. However, in most cases the exact top speed is irrelevant and Motion will revert to the default value: drive as fast as possible.

2.4.4. Motion

The Motion module takes the actions from the higher-order strategy and translates these actions into movement commands for the individual robots. Two types of motion are currently implemented: planned and reactive motion.

Planned motion is used for actions like shooting the ball to the goal. When the strategy decision is made to shoot the ball, the trajectory that the robot should drive is calculated completely. This trajectory is stored, and at every camera frame the robot velocities for the current part of the trajectory are

4 These features have not yet been implemented.

sent to the RFComm module. A so-called feedback controller uses the world state to check if the robot has deviated too much from its planned trajectory and will try to correct this error.[13]

Reactive motion determines the new robot velocities anew at every frame, and has no plan of the path that will be driven upfront. This makes it possible to react on moving obstacles (like opponent robots), or to perform actions such as blocking opponents.

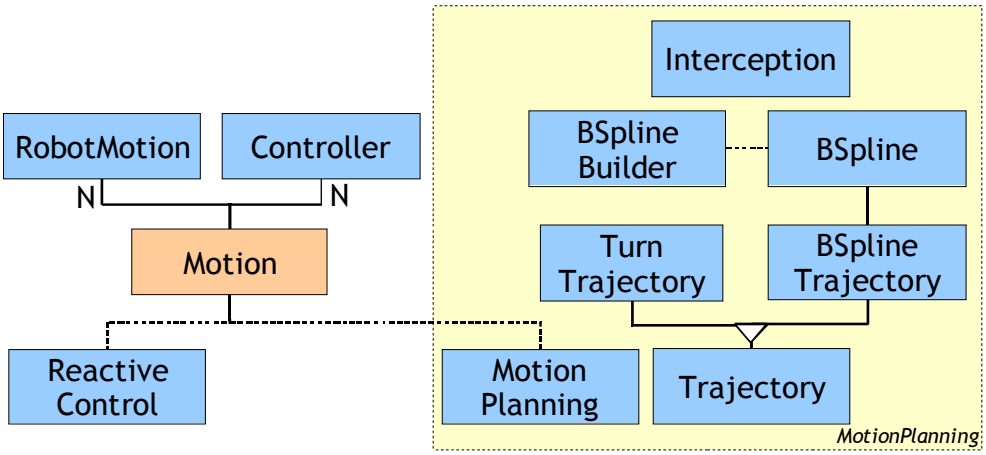


Figure 2.5. - Class diagram of the motion module. Two types of motion exist: reactive motion and planned motion. The planned motion part of the module, created by Maarten Buth, is placed in a separate directory and is indicated by the yellow rectangle on the right side of the diagram.

Whatever the type of action the robot should perform and the type of motion used, the result is always a linear and an angular velocity. These two velocities are sent to an instance of *RobotMotion*, a class that keeps track of the current velocities per robot. When the velocities requested by the Motion module differ too much from the current speeds of the robot, RobotMotion takes care of gradually increasing the speeds instead of sending the command directly to the robot. This is done for two reasons.

First, it is physically impossible for the robot to accelerate to the issued speed instantaneously. We can therefore be sure that in reality the robot will never reach the speed we send within the next frame. However, we use this command in our prediction of the world state. This prediction would be seriously hampered if we would use these velocities unaltered.

Second, the processor on the robot itself controls the wheels of the robot independently of each other. This means that, when instructing to turn while accelerating, each of the wheels independently will try to accelerate as fast as they can. This will result in a robot driving a straight line until it has reached its target linear velocity, only then to start turning. This is visible in the bottom graph of figure 2.6

This behaviour is circumvented by using the linear (v) and angular velocity (ω) to calculate left and right wheel speeds, like this:

$$v_l = v - \frac{\ell \cdot \omega}{2} \quad \text{and} \quad v_r = v + \frac{\ell \cdot \omega}{2}$$

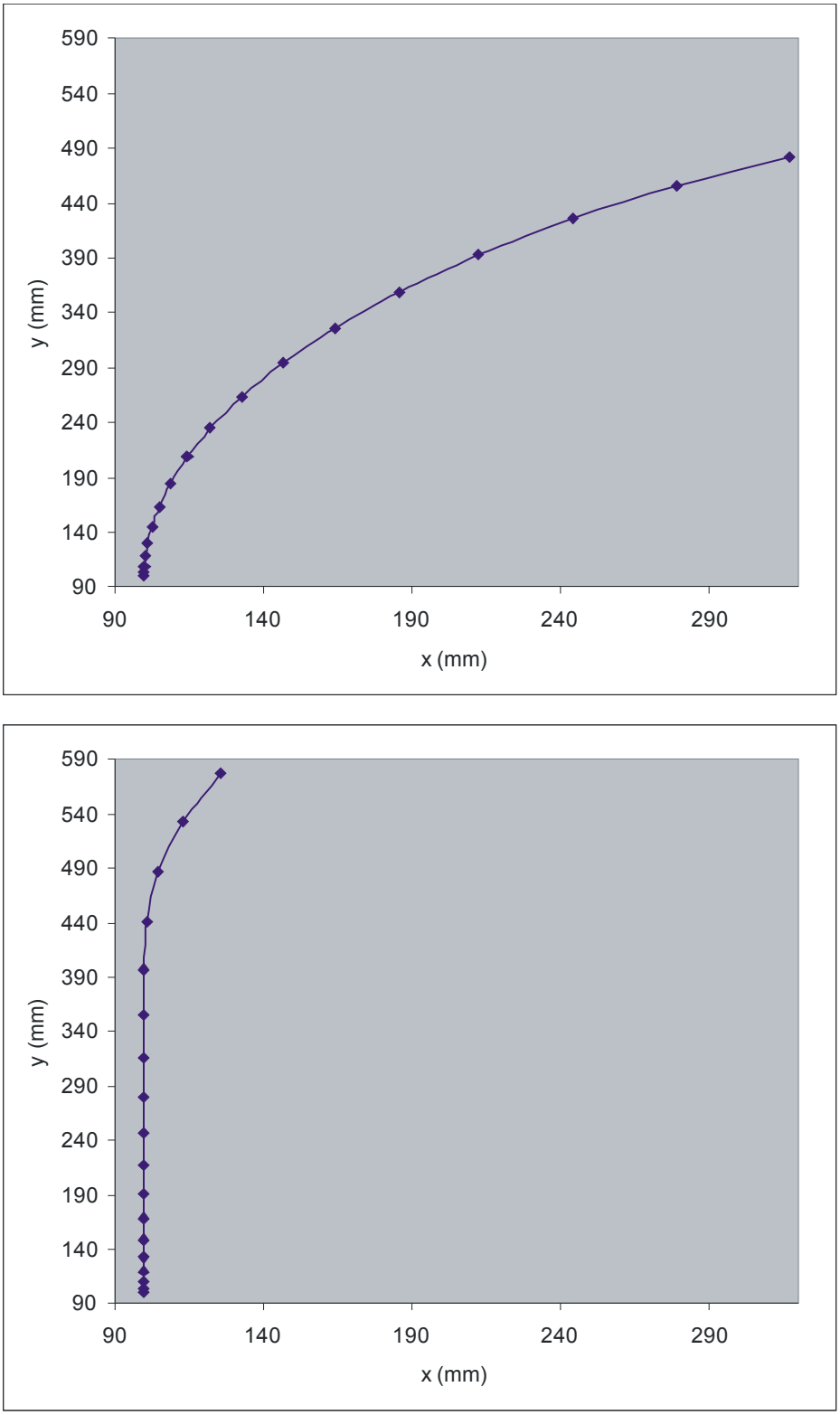


Figure 2.6. - Comparison between the paths driven by a robot with and without the RobotMotion class. An initially halted robot receives the command to drive with a linear velocity of 1400 mm/s and an angular velocity of -2.9 rad/s. In the bottom graph the robot drives forward until the right wheel reaches its target speed. In the top graph, RobotMotion makes sure that both wheel speeds are increased proportionally, giving a smoother path.

In this formula, ℓ is the wheel base of the robot, which in case of the Austro-bots, is 68 mm. From this, it is possible to see if one of the wheels would have to accelerate more than the maximum acceleration that is allowed. If so, the speeds of both wheels is reduced *proportionally*. This way, the curvature of the path the robot will drive is the same as the curvature in the command. The effect of this adaptation can be seen in the top graph of figure 2.6.

2.4.5. RFComm

The RFComm module is responsible for sending the commands issued by the Motion module to the actual robots.

Version 1.0 of the MI20 system only handled communication with the robots that were bought in Dortmund. However, when those robots started to slowly break down, new robots had to be bought. The first replacement robots were two MiaBots, robots from Merlin Systems Ltd. in England that use Bluetooth for radio communication. These robots performed adequately, but did not fulfil all of our requirements, so that in March 2006 a team of five robots were acquired from the university of Vienna.

Because of these different types of robots it became desirable to be able to use more than one type of robot at once during a game.

To be able to use multiple types of robots, the RFComm module has been set up as shown in figure 2.7. (In this figure, for simplicity, classes for only two types of robots are shown).

Each of the types of robots has their own class that inherits from `CommunicationSystem`. That latter class specifies an interface with all commands that any communication system should be able to handle.

For each of the robots, a communication handle is created at start-up by the appropriate communication system. This handle contains a reference to the communication system that handles this robot's communication and any information that the system would need to identify the robot (e.g. an ID or a Bluetooth address).

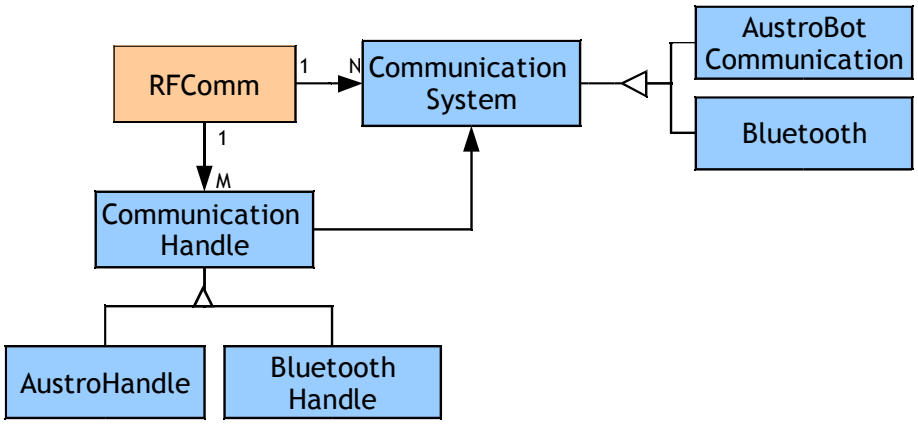


Figure 2.7. - Class diagram of the RFComm module.

RFComm listens for events from Motion containing commands that need to be sent to the robots. The handle for this robot is then used to delegate the command to the correct communication system. That system will then send the command to the correct robot, using the connection information contained in the handle.

This means commands can now be sent to robots of any type, and that these commands are sent transparently to whichever robot desired.

2.4.6. Joystick control

A fairly small module is the joystick control module. It tries to find up to 4 joysticks connected to the system that can be used to control robots on the field. This has proven to be quite popular with children that attend demonstrations, and for students during a coffee break.

Thirty times per second, the module polls the position of the joysticks, calculates the speeds to drive and injects the appropriate events in the event queue[5].

2.5. User interface

Revision of the MI20 robot soccer system has not been limited to the modules: the user interface has also been reworked. The first user interface for the system was built using a library called *Gtk+*, which is present on nearly every Linux system and available for Windows as well. This was later changed by Erik Schepers to a GUI based on a library called wxWidgets,

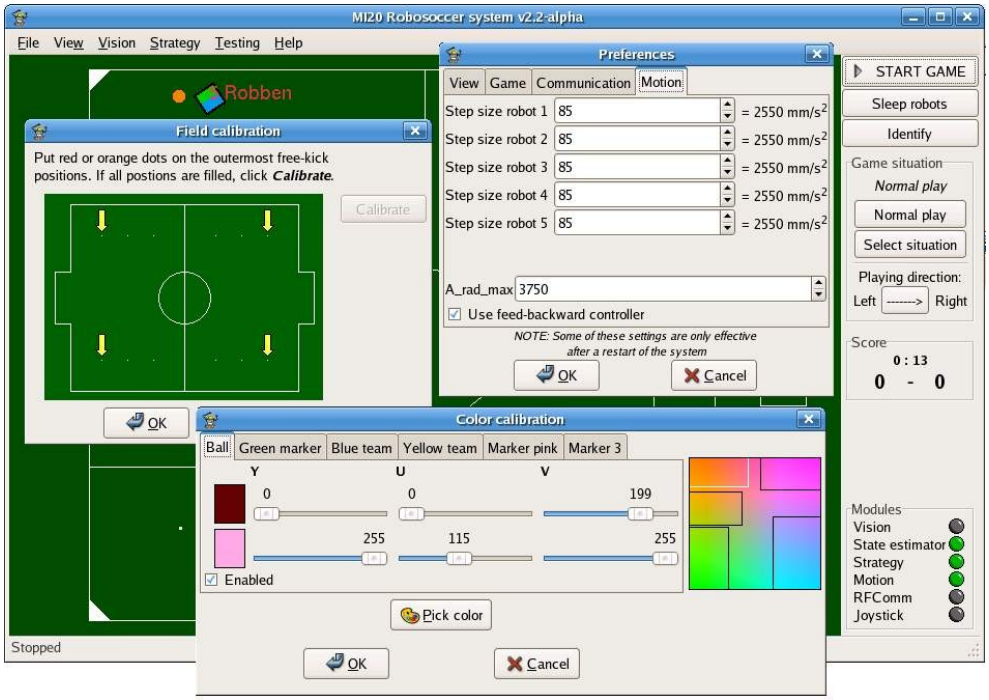


Figure 2.8. - The interface of the new MI20 system, showing the main GUI (background), field calibration (left), preferences dialog (right) and colour calibration dialog (lower).

5 Events are the new method of inter-module communication, see paragraph 2.6.2 below.

which had the advantage of a native look-and-feel across any platform.[14][15]

When designing the user interface of version 2.0 the decision was made to revert to the *Gtk+*-library. The main reason to do this was the existence of the *Glade* application, that offers a point-and-click interface to create and adapt interfaces. Using Glade, creating the new interface required little programming, and extending the interface requires little or no knowledge of *Gtk+*. Glade automatically generates the code needed for the interface created. The user only needs to write the code that ties the interface to the program itself (i.e. what method should be called when a certain button is pressed).

Main idea of the user interface redesign was that the main window should be simple and contain only the elements that are required during normal game play. Any other options should be conveniently stashed away in a menu bar at the top of the application. Also, most settings should be possible to change from within the application, i.e. without using a text editor to change the settings file.

The main canvas that shows the current world state to the user is drawn using OpenGL just as in the previous versions of the system. An addition, however, is the option the user has to use not a 2-dimensional, but a 3-dimensional view of the game. The actual use for this feature may be limited, but it sure makes simulation games look much more life-like.

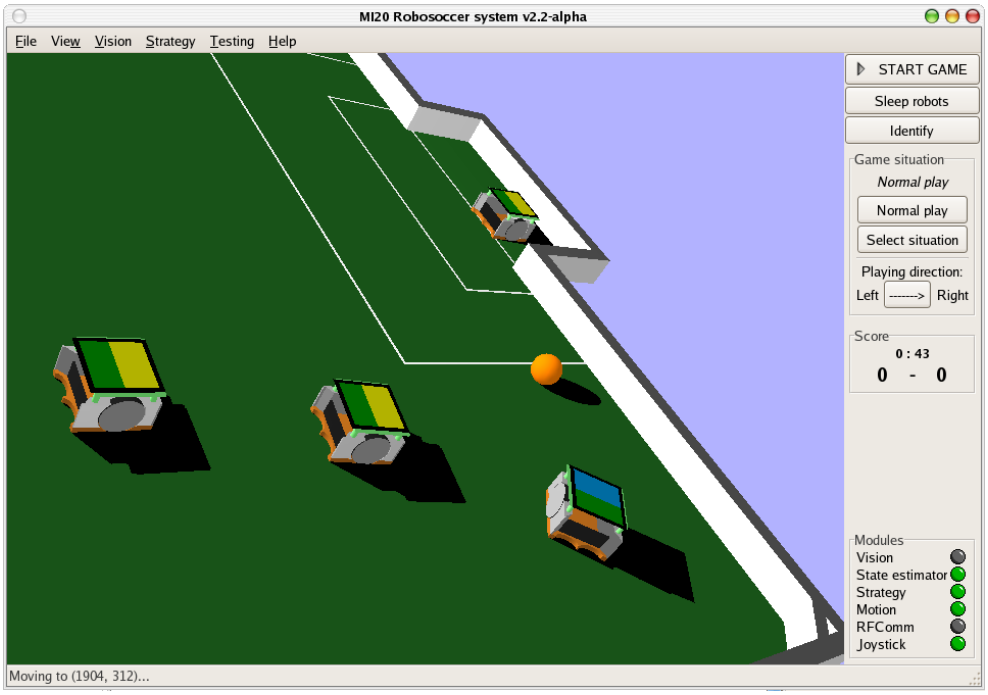


Figure 2.9. - The new version of the MI20 system even has the possibility for a 3-dimensional view on the field.

2.6. Inter-module communication

Although the idea of modularity is that communication between modules should remain limited, a module is of course totally useless when it has no input or output. Therefore, it is necessary to have some means of communication between modules.

Because of the different approaches in architecture of version 1.0 and 2.0 of the MI20-system, the approach for inter-module communication is quite different as well.

2.6.1. Sender-receiver threads (v1.0)

Inter-module communication in version 1.0 was implemented by using TCP/IP sockets. This made it possible to run a module of the system anywhere on the network, and still have the system run just as it would on a single computer.

For each piece of information a module wanted to receive from another module, it would create a receiver thread. That thread would wait for other modules to send the data, place this data in a shared global variable and then notify the module that new data had arrived. On the other side, the module wanting to send data would create a sender thread. The sender thread would wait indefinitely until new data came available, take this data from the global shared variable and send it over the network to the receiving module.

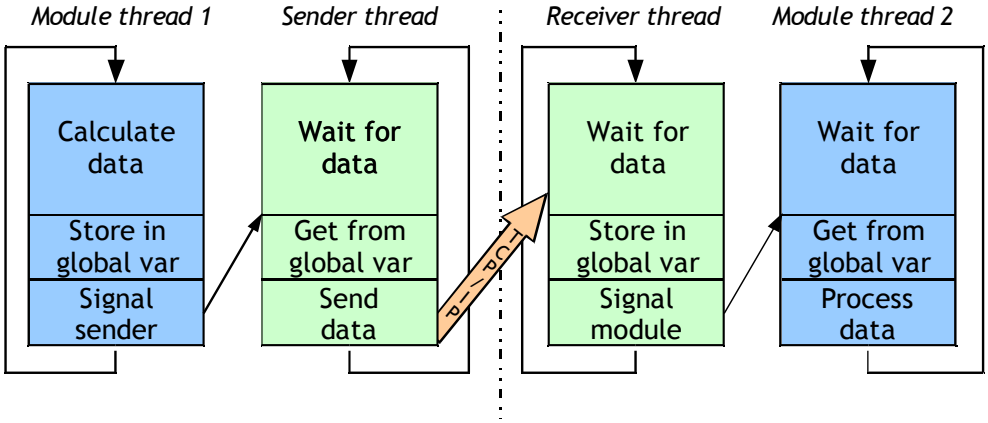


Figure 2.10. - Inter-module communication in version 1.0 of the MI20 robot soccer system. The first module thread performs its work, places its data on a common location and signals the sender thread. This thread takes the data and sends it to a receiver via TCP/IP. This thread places the data in a globally shared variable and signals the second module which processes the data.

Since these communication channels were dedicated (i.e. only one type of information was sent over each channel) the type of data sent could be determined from the channel it came in over, so that no overhead penalty is incurred and data transfer was as efficient as possible.

The advantage of using the data as it appears in memory has a drawback: all communicating modules should be run on a comparable platform. Compilers are allowed to rearrange the members of structs and classes so

that they benefit the most from memory alignment. If this rearrangement is different from the one in another module, data corruption and probably crashes will be the result. However, since we always used the GNU Compiler Collection on 32-bit 80x86 systems, this has never been a problem.

Another disadvantage is that this method relies fairly heavily on the use of multi-threading. Although communication between modules should be limited, there is still a substantial number of data types that will be transmitted. And because this method relies on the fact that data can be identified by the communication channel through which it was received, a simple game set-up already contains quite some channels. As can be seen from table 2.1, a simple game of robot soccer with 5 players, will need 72 communication channels, and a total of 165 threads.

# players	*# threads*	*# sockets*	*#channels*
5	165	144	72
7	213	188	94
11	309	276	138
N	45+24N	34+22N	17+11N

Table 2.1 - Number of threads, sockets and communication channels in use by the MI20 robot soccer system v1.0 as of September 2005.

The MI20 system has a fairly sequential manner of operation: a camera image is analysed, this data is used to estimate the current world state, a strategy decision is made, and velocities are sent to the robot. Then the system idles until another frame is received from the camera.

Since all modules are so dependant on each other, no module can run before another has finished. The threads in the system therefore have a limited task: wait for data, handle the data, and wake up the next thread that receives the new data. So, although the program has multiple threads, concurrency is minimal, since often only 1 thread is runnable at each time.

Even *if* more threads in the system are runnable, a thread will either run until its task is complete, or until it is pre-empted. However, the default pre-emption interval of the Linux kernel is 10 ms[6].[16][17] Currently the time spent to do *all* calculations is about 6 ms[7], making it likely that any task in the system will be completed within one time slice. So, even if multiple threads are ready to run, it is unlikely that interleaved execution will ever occur.

The only difference between a threaded approach and execution in a single thread is that, when multiple threads are in a runnable state, the order of execution of the two threads is non-deterministic. When using a single

6 Although some of the current desktop systems come with a kernel running with a 2.5 ms pre-emption interval to improve latency.

7 On a Pentium 4, running at 3.2 Ghz.

thread, a programmer has to prescribe which task will run first, creating a deterministic execution order. However, the fact that two threads might be runnable at the same time implies that their execution order is not an issue: the two parts of code must be able to run independently. Whichever order a programmer would prescribe, the code will run fine.

Now that we know that in practice there is no concurrency whatsoever in the MI20 system, it seems there is no need to use threading. Another argument for the removal of threading is that it would reduce the use of mutexes and condition variables, which are heavily used by the sender and receiver threads. So much even, that on a AMD Athlon 2200+, the CPU time spent on locking amounted to 7% when running at 30 frames per second.

Luckily, the decision to combine the seperate executables of version 1.0 into one single executable enables us to use another method of inter-module communication.

2.6.2. Events (v2.0)

The inter-module communication in version 2.0 of the MI20 robot soccer system is done using an event system.

One of the key design considerations for the new version of the system was to keep modularity high. In many communication systems, including the one used by the old version of the systems, modules are sending data to another module they know of. However, the fact that sending modules need to know about the other modules in the system does not combine well with the concept of modularization. To satisfy this goal, modules should only have contact with one ominous entity for communication.

In the new version of the system, communication is done by sending events to a system-wide event queue. Examples of events that typically occur in the system are[8]:

- A new snapshot arrived
- Robot 3 should try to shoot the ball
- Robot 2 should move to position (1300, 500) on the field
- The referee has given a free kick

Modules that are interested in a certain event can register with the event manager. A module sends to the event manager the type of event it is interested in and a function pointer to the method that should be called every time this type of event was received.

A large number of subclasses exist of the abstract base class `Event`, so that events can contain data that further specifies the event. For example, a `VelocitiesEvent` commands a robot to drive at a specified speed. This type of event contains the ID of the robot, the linear velocity and the angular

8 A more complete list of events can be found in appendix A.

velocity to drive.

When a module wants to send an event, it creates an instance of the correct event subclass and sends it to the event manager. Meanwhile, on a separate event handling thread, the event manager continuously walks through the event queue. It looks up the listeners that have registered themselves for this type of event and then - sequentially - dispatches the event to those listeners.

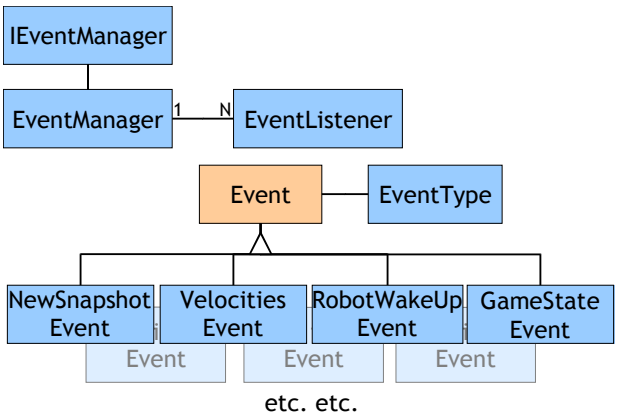


Figure 2.11. - Class diagram of the classes necessary for handling events.

It should be noted that the sending module does not specify the module it wants to send the events to. Any module can register to listen to a type of event, and the sending module does not know who will receive the event. The reverse is true as well: a module that receives an event does not know from which module it originates.

Because the sending module needs only to communicate with the event manager, it has no need for knowledge about the receiving module. In code, this means none of the headers of the other modules needs to be included, thereby decreasing compile times and guaranteeing modularity.

This feature is quite handy for debugging purposes, since an extra listener can be registered to see which events are being sent. For that, there is no need to change the sending module.

To dispatch events to the correct event listener, the event manager needs to determine the type of the event that it is processing. One approach would be to create an enumeration of all event types in the system and add to each event the identifier that belongs to the type of event.

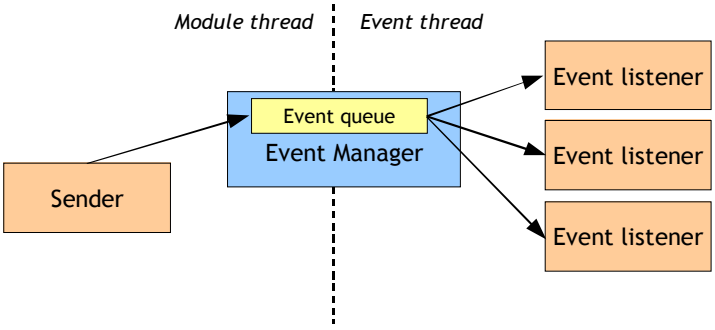


Figure 2.12. - The event manager maintains an event queue. Modules send an event to the event manager which places it in the queue. Meanwhile, a separate event thread continuously takes events from the queue and dispatches them to the registered listeners.

A large drawback to this approach is that the header containing this enumeration would be included

by nearly every source file. A change in the header would trigger a recompile of the whole system for anyone who is working on it. This is not desirable and can be circumvented by having each event contain a reference to a static `EventType`-instance. That instance is constructed using a simple name for the event (e.g. '`new_worlddata`').

To speed up comparisons[9], this name is then hashed and stored. The hash algorithm can be found in listing 2.1. The definitions of these event types can now be placed in a source file, so that the addition or modification of events would only require recompilation of the event definitions and the users of this specific event.[18]

Since all communication is now centralized at a single part of the system – the event manager – we can now track the communication within the system better. An example of this is the watchdog thread that is currently implemented in the system. This thread wakes up twice a second and checks if the event that is currently handled is not the same as the one during the previous cycle. If that would be the case, the event listener would have been running for more than 500ms and is probably hanging. This would indicate a bug in the code of an event listener. Since the event manager can report the function pointer of the currently executing event listener, the offending code can be found quite quickly.

Also, we can keep statistics about the handling of the events. As every listener is called, the time that the listener took to process the event is tracked. At runtime, a list can be printed showing the number of calls, the minimal, maximal and average processing time, including the standard

```
1   unsigned int EventType::getHash( const char * const name )
2   {
3      const unsigned long BASE = 65521L; // Largest prime < 65536
4      const unsigned long NMAX = 5550;   // Max. number of chars that
5                                         // would not overflow hash
6      unsigned long s1 = 0;
7      unsigned long s2 = 0;
8      const char * p = name;
9
10     for( size_t len = strlen(name); len > 0; )
11     {
12          unsigned long k = (len < NMAX ? (unsigned long)len : NMAX);
13          len -= k;
14          while( k > 0 )
15          {
16              s1 += tolower(*p);
17              s2 += s1;
18              p++;
19              k--;
20          }
21          s1 %= BASE;
22          s2 %= BASE;
23     }
24     return ( 0 | (s2 << 16) | s1 );
25   }
```

Listing 2.1: Hashing algorithm that calculates a hash for an event name.

9 This has another use than just a speed increase. See paragraph 2.6.3 Networking below.

deviation. This information can be used to spot bottlenecks in the program, by finding event listeners where the large processing time is not justified by the work done.

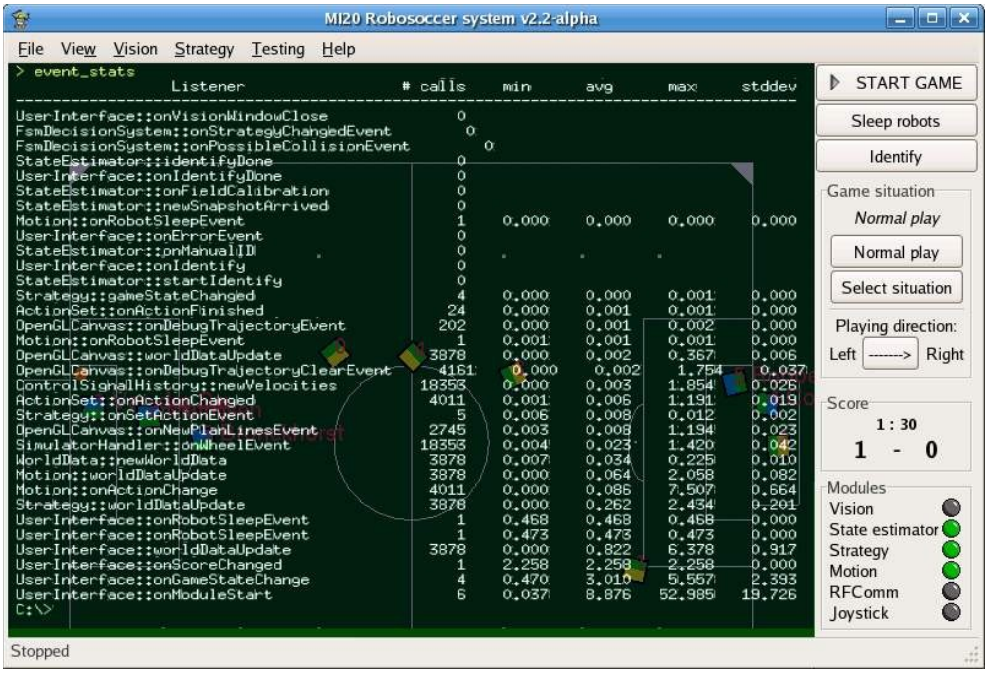


Figure 2.13. - A list showing the registered event listeners, the number of times they were called, and statistics about the number of ms processing the event took.

2.6.3. Networking

The original design of the MI20 robot soccer system was a school book example of a distributed system. A module could run anywhere on a network, either in the lab or anywhere else on the world. Communication via those modules took place through the network, even if all modules were running on the same machine.

With the new system, modules on the same system no longer communicate with each other using sockets, but with the event system explained above. This makes the MI20 no longer a distributed program per se. In other words, the standard means of communication shifted from inter-*process* communication to inter-*module* communication.

However, the centralized nature of the event queue design makes it not very difficult to reimplement networking. All network communication can now be limited to one communication channel, and needs only one extra thread: the receiving network thread.

Using a peer-to-peer architecture, several instances of the program can connect which each other. Connecting to another system could be done by specifying *one* other system. The receiving program instance will check if the modules running at the connecting system are no duplicated and, in that case, will accept the connection. The receiving system will then send all other known peers to the connecting system, allowing it to connect to these instances of the program as well.

When an event listener registers itself with the event manager of that program instance, it should also notify any other instances of the system that someone there is interested in that type of event. The networking module on the other end will then register as an event listener for that type of event at the local event manager. When that type of event is sent from that program, the networking module will be called and will send the event over the network. The other side will receive the event and inject it transparently into the event queue. There, the actual listener will receive the event, having no idea the event was actually generated on another machine. This way, effectively, one distributed event queue has been created.

When sending an event over the network, a network packet is created. This packet begins with the hash that identifies the type of event (as described on page 34). After the hash, all data contained in the event is sent. To do this, all data should be converted to a single stream of bytes; a process called *serialization*. To make this possible, each subclass of `Event` must implement a method called `serialize` that converts the data into a stream of bytes.[10] Code has been written to make this as easy as possible, as can be seen in an example implementation, shown in listing 2.2:

```
1    int m_blueScore;
2    int m_yellowScore;
3
4    void serialize( StringOutputStream &out ) const
5    {
6        out << m_blueScore << m_yellowScore;
7    }
```

Listing 2.2: Serialization of the data of an event.

When the network packet is received by the program at the other end of the network, the event needs to be reconstructed. First, the hash at the beginning of the packet is read. To be able to construct the right type of packet from this hash, at the start of the program a map is created that maps a hash to the corresponding event type.

To make deserialization possible, subclasses of `Event` must also implement a constructor that can use a data stream to initialize its data members. The code for such a constructor looks quite similar to that of the serialization as seen in listing 2.2. When the event was successfully constructed, it is placed in the local event queue, just as if the sender of the event was a local module.

2.7. From enhanced C to actual C++

Although the programming language that was used to develop the MI20 system has been C++ from the start, many of the features of this language

10 Also, several data types like `WorldData` and `Snapshot` contain such serialization methods. These are used by the serialization code of the events (when events contain such data), but also for writing data to file like in the game recording files (see paragraph 4.3 and appendix B).

were not used. Only *one* C++-feature was actually used quite extensively, namely the separation of functionality using classes. The rest of the code, however, could very well have been simple C. Throughout the rewrite of the MI20-system, more of the useful C++-features have been used.

2.7.1. Object-orientated features

As seen in paragraphs 2.2 and 2.4, the design of the MI20 system is object-oriented in nature. To make the source code tightly reflect this design, the C++-support for inheritance and comparable features are widely used.

Quite some classes in the system, like `Action` and `Module`, are abstract base classes: classes that cannot be instantiated, but do define the structure of their subclasses.

2.7.2. STL containers

In a system like the MI20 robot soccer system, the use of data structures is a necessity. Structures like arrays or linked lists have all kinds of uses, in all kinds of programs. The designers of the C++-language recognized this and added the most frequently used data structures and algorithms in what they called the Standard Template Library (STL). This set of functions and data types is implemented as a set of headers. These headers contain so-called template definitions, another C++-feature, making it possible to create linked-list implementations that can contain any data type. For any type the user specifies, the compiler will generate the code that will implement a data structure, using the user's type.

This means custom-made implementations that previously existed in the system (e.g. skill queues) have been abandoned and replaced by STL-containers.

Also, the old implementation used quite some arrays where the number of elements was not clear upfront. In those cases often fixed-sized arrays were used, with an upper bound chosen high enough that it would be highly improbable (read: still possible) that the array would be too small. These occurrences have been replaced by the use of `std::vector`, a dynamic array that adapts to the number of elements in it.

2.7.3. The const keyword

For efficiency reasons it is recommended that when an object is a parameter to a method, that a pointer or reference is passed. However, the called method is then able to change the passed object. Quite often a method needs an object because it *uses* the data, opposed to *altering* the data. Moreover, to limit the possibility of error we want to guarantee that the method will not alter the data indeed.

Luckily, C++ allows us to do that. By adding the `const`-keyword in the method specification, we tell the compiler that the parameter is not to be altered. Doing so anyway will result in a compiler failure.

To be able to discern if a method call will alter the object or not, the `const`-

keyword can also be used with methods. When the declaration of a method is followed by `const`, the programmer promises that a call to this method will not alter the object. Again, doing so will generate a compiler error.

Finally, just as parameters can contain `const`, return types can be `const` as well. This is useful when a reference to part of a data structure is returned.

```
1   class WorldData {
2       void calculateSomething( WorldObject const &object );
3       void setZeroVelocity( WorldObject &object );
4       WorldObject const &getBall() const;
5   };
6
7   WorldData wd;
8   WorldData const &wd2;
9
10  wd.setZeroVelocity( wd2.getBall() );    // ERROR, getBall() is const
11  wd.calculateSomething( wd2.getBall() ); // OK
12  wd2.calculateSomething( wd.getBall() ); // ERROR, wd2 is const
```

Listing 2.3: Some examples of the const keyword.

2.7.4. Exceptions

To prevent the need for checking return values of methods, *exceptions* were introduced to indicate an error. When an error occurs, one of a collection of exceptions is thrown, including a human-readable error message. This error message can then be shown to the user.

Exceptions that are thrown but not caught, are propagated all the way up to the starting function of the program. There, a catch-all exception handler is included, so the the user will see a descriptive error message, instead of experiencing a weird crash because a programmer did not check a return value.

2.7.5. C++ for Java and C-programmers

Some of the mistakes that were found in the original code are very common errors. Quite a few are simple to make and their presence is understandable. Moreover, one source for these errors might remain a concern, even in this new version.

Currently, the programming languages taught in our computer science curriculum are Java and C. A fair amount of students start their Master's assignment being mainly proficient in those two languages. Although C++ is very similar to those languages, it has some subtle differences which can be quite misleading. Consider the code in listing 2.4.

Most programmers that know C and Java will spot no apparent errors. Nevertheless, there are some caveats present in these few lines of code.

The first problem arises in line 5, where a `WorldData`-instance is passed as a parameter to the method. In Java, when passing objects to a method, under the hood a reference to the object is passed to the method and not the object itself. In C++ this line of code will pass by value. This means that line 13 will have no effect whatsoever, since `data` will be deleted as soon as the end of the method is reached. Second, to follow pass-by-value semantics, the `WorldData`-instance has to be copied. Especially in cases where the

```
1   /**
2    * Checks for collisions in the current WorldData and changes
3    * the WorldData to reflect this collision.
4    */
5   void Simulator::detectCollision( WorldData data )
6   {
7        CollisionDetector *detector = new CollisionDetector();
8        detector->setWorldData(data);
9        if( detector->collisionDetected() )
10       {
11            RobotWheels *wheels = data.getRobotWheels();
12            wheels->spin();
13            data.addCollisionPoint(detector->getPoint());
14       }
15  }
```

Listing 2.4: Small piece of source code that will probably look correct programmers that are proficient in C and Java, but not in C++.

objects are big, this can have a surprisingly large influence on performance. The solution to these problems is to explicitly pass a reference to the instance.

A fairly common construction in C is the one in line 11. A pointer to an instance of `RobotWheels` is retrieved from the `WorldData`. However, we can't determine if the pointer retrieved is a valid pointer. The implementer of `getRobotWheels()` might have decided that NULL will be returned in certain cases. In that case, the dereferencing of the pointer in line 12 will lead to a segmentation fault – a crash.

To enforce the condition that `getRobotWheels()` should always return a valid pointer, we can let the method return a reference. A reference is guaranteed to point to a valid object, making a segmentation fault impossible[11].

Finally, the construction of the `CollisionDetector` in line 7 is somewhat problematic. Java-programmers will constantly use the `new`-keyword to construct an object. When the method ends, the Java garbage collector will automatically delete the object again, since it is then unused. In C++ this is not (by default) the case. Here, the object is allocated in line 7, but is never destructed, effectively creating a *memory leak*. When this method is called very often (i.e. every frame or even more), it is likely that the computer will run out of memory and the program will crash. To prevent this, one might call the `delete` operator on the object at the end of the method to clean-up the object, but it is both simpler and faster to allocate the object on the stack, as is done in listing 2.5.

11 Strictly speaking, making a reference to an invalid object *is* possible, but you need to construct such a reference knowingly.

```
1   /**
2    * Checks for collisions in the current WorldData and changes
3    * the WorldData to reflect this collision.
4    */
5   void Simulator::detectCollision( WorldData &data )
6   {
7        CollisionDetector detector;
8        detector.setWorldData(data);
9        if( detector.collisionDetected() )
10       {
11            RobotWheels &wheels = data.getRobotWheels();
12            wheels.spin();
13            data.addCollisionPoint(detector.getPoint());
14       }
15  }
```

Listing 2.5: The same piece of code in listing 2.4, but now adapted to cope with the pitfalls of C++.

2.8. Other changes

A few other changes have been made to the code base to help the extensibility and manageability of the program.

2.8.1. Settings

A system like the MI20 robot soccer system contains all kinds of settings, like the values of the colour calibration and team settings. In v1.0 of the system, these settings were kept in a configuration file that was read at start-up. These settings were to be changed directly in the configuration file and not while the program was running.

To enhance the usability of the system, many of the options that previously could be changed in the settings file, can now be changed within the user interface of the system. A configuration file does still exist, but it is now read and written by the system and is not meant to be changed by humans any more.

Many of the settings are also changeable at run-time. A Singleton instance [8] called `Settings` keeps track of all the settings and their default values. It does so by means of a map, where a human-readable setting name (e.g. '`ui.renderer3d.shadow`') is coupled to an actual setting. Modules are encouraged to store the pointer to the setting and use this pointer when the value of the setting is needed. This way, whenever the user changes the setting using the user interface, the new value for this setting is instantly used by the implementation.

2.8.2. Assertions

All throughout the program, assertions are made about the state of certain variables etc. For example, a robot ID can only be a positive number. By explicitly checking these assertions, the cause of a problem can be spotted even before it would normally cause trouble.

However, the processing power that would be needed to constantly perform these checks could be quite large. As a golden mean, one can (and should)

use the assert-macro. In debug modes this check is enforced, and when it fails the source file and the line of code is reported to the programmer. In release mode these checks will not be compiled into the application.

Another form of assertion is the *compile time assertion*. This is a precondition just as the normal assertion, but can already be checked by the compiler and therefore has no performance cost. Examples of these assertions can be seen in listing 2.6.

```
1    assert( robotID >= 0 );
2    assert( vector.length() > 0 );
3    COMPILE_TIME_ASSERT( sizeof(int) == 4 )
```

Listing 2.6: Examples of the two types of assertions that can be specified.

2.8.3. Memory leak detection

C++, unlike languages as C# and Java, does not use garbage collection by default. This means that any memory that is claimed by the application must also be explicitly returned to the operating system. Sadly, making the call to free memory is extremely easy to forget. Especially when the claim for memory happened in a tight loop, this *memory leak* will cause the operating system to eventually run out of memory, and the application will most likely crash.

To prevent this scenario, code was added to the system that keeps track of the memory allocations and deallocations. It does this by overriding the global `new` and `delete` operators, replacing them with functions that record the allocation of memory, but also the source file and line in which the allocation was done. At the exit of the application, a check is done if all claimed memory has also been released. If not, the offending line of code is reported to the user.

This only happens in the debug builds of the program, so that in releases the inevitable overhead of this method disappears.

2.9. Platform independence

In October 2005, the MI20 robot soccer system had become so dependant on the build environment in our lab, that it would only run correctly on Debian Linux systems, running a 2.4 kernel.

During the reorganization of the code, care was taken in writing code that would compile and run on a multitude of systems. It is possible to run the current system on both Linux and Windows (again). However, some features like the code for acquiring images from the camera, and communicating with the Bluetooth MiaBots has not yet been written for Windows. Running the system in combination with the simulator works fine.

The code has been checked to compile with the following compilers:

- GCC 3.2.2 for Linux (no warnings)
- GCC 4.0.2-8 for Linux (no warnings)
- GCC 4.1.1 for Linux (3 harmless warnings)

- GCC 3.4.4 on Windows XP (no warnings)
- Microsoft Visual C++ 2005 (no errors, much warnings)

Although is has not been attempted, it is believed that the code will also run without problems on 64-bit systems.
However, in the serialization and deserialization code, currently little-endianness is assumed. This means that this code should be adapted before it would run without problems on a big-endian machine, like a Mac.

2.10. Conclusions

It is safe to say that the move from version 1.0 to 2.0 encompassed a lot of changes. Although the modularity of the system was kept in tact, the modules were all combined in one executable. The main program now has the task to create an execution environment in which any amount of the modules can run independently.

Also the modules themselves were scrutinized and, where needed, redesigned. Some of the modules have only undergone a little polishing and were kept largely as before. Others were completely reworked, most noticeably the Strategy and RFComm modules. Also the user interface was rebuilt using the Glade interface designer.

Because of the little concurrency in the operation of our robot soccer system, the introduction of the event system gave us a communication scheme that, in effect, is very similar to the old system using threads and sockets. However, there are a few extra benefits:

- There is no need for synchronization in many of the modules themselves anymore. This eliminates the possibility of the obscure bugs that can result from a programmer forgetting to lock a specific mutex variable. Also, the overhead from locking and unlocking mutexes has been largely removed from the system.
- The centralized nature of the event system makes it easier to implement networking, recreating the distributed system we had before. Network communication can then take place over a single connection, instead of the multitude of network connections in the old system.
- Because of this centralization, we can also keep track of the processing power certain parts of the system take up. If one part of the system needs an excessive amount of CPU-time, this can be spotted and dealt with.

The fact that the event system is a *cooperative* multi-tasking system, other processes depend on the fact that each listener should return the control of the program within a small amount of time. If a listener misbehaves and does not yield control for several seconds, other listeners suffer a big latency hit and there is no authoritative system that steps in and corrects

this behaviour[12].

One could, however, actually regard this as an advantage, because long response times often result from errors in the event listener. Although this error might long go undetected in a multi-threaded environment, in a cooperative multitasking system these errors will be spotted immediately.

Furthermore, during the rewrite of the system much more use was made of all the features that the C++-language has to offer. Although this is advantageous in most aspects, it should be noted that little C++ is currently taught to students in the current curriculum. Although, students may experience difficulties in grasping C++-constructs at first, it also might make them more aware of the differences between C and C++.

All in all, the system now runs smoothly, using only 20% of the CPU on a Pentium 4, at 3.2 Ghz. Furthermore, the two Master students that have used both version 1.0 and version 2.0 found the revised system easier to extend and to grasp.

12 Although the system's *watchdog* will signal an uncooperative event listener and reports the listener's name to the user.

CHAPTER

3 Development process

Quality of code is not only influenced by a programmer's proficiency, but also by the development process as a whole. Although the original developers of the MI20 system used the source code management system CVS (Concurrent Versioning System), three years later students worked in their own separate copy of the system and hardly shared their changes.

Also, as the magnitude of the code had grown, it became impossible to oversee all of it. Without tools like browsable documentation this can become quite a problem.

Finally, the source code over the years became riddled with comments that indicated problems that should be fixed when somebody had the time. However, quite often nobody had that time, or judged the problem was too far from the scope of his Master's assignment. To keep track of those issues found, a bug tracker was introduced.

3.1. Source code management

In a project where multiple people are working on the same code, it is important to use a source code management system to store the code base. Source code management systems make sure two people (or more) can work on the same source file, ensuring that none of the changes will be overwritten. Another important benefit is that old versions of the code remain available, making it possible to retrieve an old version if changes turn out not to be beneficial after all.

Previously, CVS was used for source code management. After three years of programming, however, CVS was hardly used any more. Master students starting on the project created a copy of the source code from one of the other students and started programming. Then, when they finished, they uploaded the copy with their changes in a new directory on the CVS-server. Therefore, many copies of the system were around and nobody could really

tell what code was in which version.

When starting on version 2.0 it was possible to start with a clean slate and use source code management correctly again. Also, a switch was made from CVS to Subversion.

3.1.1. Subversion

Subversion is sometimes described as: "*CVS done right*". It offers all features that are present in CVS, and adds some extra. Most notably of those extra features is the ability to move files and directories, whilst retaining the change history.

Also, changes are not tracked on a per-file basis, but Subversion uses a concept called *revisions*. Every time a developer commits the changes to Subversion, all changes are packed into one revision.[19][20]

How often changes should be committed depends on the task at hand. When writing a new feature, commits should happen about once a day. When hunting bugs that number is likely to be about 10 times more. A few things to keep in mind when deciding if changes should be committed:

- Changes should never hinder other peoples' work. This means that the code should at least compile cleanly. It also means that new features that are not finished and tested should never be started automatically.
- Committing changes is also necessary when you start working on something totally different, so that the changes you commit are related. If a bug was fixed in the StateEstimator, commit that fix, and only then proceed to make a change in Motion. The reason for this is simple. Suppose those changes above are made in version 2.3. However, version 2.2 is considered stable and is used for matches and demonstrations. The, now fixed, bug in the StateEstimator is present in version 2.2 as well. When the fix is self-contained in a single revision, Subversion can be instructed to apply these changes to version 2.2 as well, thereby fixing the bug there too.
- Work should be commit at least once a day. This way, your hard day's work is included in the backup that is made of the repository. Secondly, it prevents that the code bases of students grow too much apart. It is much harder to merge changes if there is three months worth of changes in them, than when it is just the work of one day.

Now, the first and the last suggestion might be in conflict with each other. A change might be that extensive that it cannot be completed within a day. In that case, it is advisable to make a so-called branch. How branching works is explained in the next paragraph.

3.1.2. Branching

A feature that was already available in CVS, but was not used by the

MI20-team, is branching. When branching the source code, the current code base is split up in two parts.

This, for example, was used prior to the European Championship 2006 in Vienna and the World Championship in Dortmund. A week before departure, a branch was made. In one of these branches – called the release branch – no changes were made other than bug fixes. Addition of new features and elaborate changes were made to the main line of development, called the *trunk*. This made sure that none of the possible side effects of changes made in the last week endangered the stability of the application in Vienna.

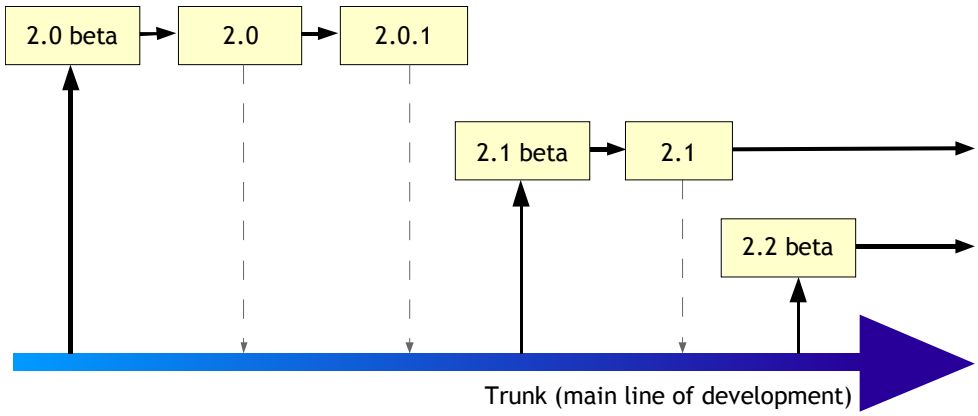


Figure 3.1 - Branching ensures that development can continue even when preparing for a championship. Stability of the branched version is ensured, and developers have no need to stop making radical changes.

Branching is also useful when making big interface breaking changes to the system. By creating a private branch, a developer can change the system in any way he likes. When he or she commits the code to Subversion, the changes will only be made to the private branch and none of the other students will be hindered if the code does not compile or work correctly yet.

3.1.3. Vendor drops

Not all of the software that is part of the MI20 system is written by students on the project. Some of the libraries used, like the Bluetooth interface or the dynamics library used for the simulator, are open source libraries that were made available through the Internet.

To use the libraries exactly like we would like to, there were some changes made to them, for example to change the way the library behaves when errors are encountered. The LGPL- and BSD-licenses under which the code was released allows us to make those changes.[21][22]

However, these libraries are under active development, meaning they will change over time, fixing bugs and adding new features. If we would simply update our source tree with the new library code, our changes would be lost.

To remedy this problem, Subversion supports *vendor drops*. This means that the library code is put under version control in a separate directory,

and then copied to our system (Subversion performs a cheap copy here, only copying the references not the files themselves). We can change the library code that lives within our copy. When an update of the library is released, we place this new copy in the directory of the vendor drop. Because the original library code still exists there, Subversion can determine the changes made by the library maintainers and apply only these changes to our copy, effectively merging our changes and those made by the library maintainers.

3.2. Documentation

Currently, most of the MI20 code has been documented using the Javadoc method. Every documented method is prefixed by a block comment, explaining the use of the method and the input and output variables.

```
1    /**
2     * Calculates the length of the hypothenusa of a triangle from
3     * the length of the two other sides. For this, Pythagoras'
4     * theorem is used, which is:
5     * \f[ c = \sqrt{ a^2 + b^2 } \f]
6     *
7     * @param a The length of the bottom side of the triangle.
8     * @param b The length of the right side of the triangle.
9     * @return   The length of the hypothenusa.
10    *
11    * @throws TriangleException   if either a or b is negative.
12    *
13    * \todo   Use the GPU for a more efficient calculation.
14    */
```

Listing 3.1: Example of the Javadoc documentation above a method.

A tool called Doxygen is used to automatically generate HTML-pages from these comments, providing developers with a browsable reference manual for the system. Doxygen even has the ability to include LaTeX in its documentation, so that method descriptions can contain the formulas used in the implementation. Figure 3.2 shows the HTML-documentation that is generated from the comment block in listing 3.1.

float hypothenusaSide ( )

Calculates the length of the hypothenusa of a triangle from the length of the two other sides.

For this, the Pythagoras theorem is used, which is:

$$c = \sqrt{a^2 + b^2}$$

Parameters:

a The length of the bottom side of the triangle.

b The length of the right side of the triangle.

Returns:

The length of the hypothenusa.

Exceptions:

TriangleException if either a or b is negative.

Todo:

Use the GPU for a more efficient calculation.

Figure 3.2. - The documentation generated from the code in the previous listing.

Doxygen can use the same comments to generate documentation in RTF, LaTeX, hyperlinked PDF, XML, Unix man page format and CHM[13].[23] This documentation, however, tends to become quite large and these off-line formats are a lot less convenient than the browsable HTML-documentation. As an example, the reference manual in LaTeX created from the code in August 2006 encompassed a whopping 1117 pages.

3.3. Bug tracker

Previous students all concluded their theses with a chapter called *Conclusions and recommendations*. Because all of these writers had 9 or more months of experience with the system, these recommendations often contained very valuable advice.

However, often – despite the usefulness – this advice was not followed. Many of the recommended tasks were not large enough to justify a whole Master's assignment. Also, if a task was not directly related to another students' assignment, they were not eager to take on this extra task. As a result, these recommendations never got a follow-up and were forgotten.

Also, the comments in version 1.0 of the system showed that quite some defects were actually detected, but never fixed. This is understandable, since these bugs are often found during other work. Immediately fixing the bug would distract from that task, making development chaotic. Furthermore, the developer that found the bug might not have full knowledge about that part of the code. Bugs were therefore marked by a

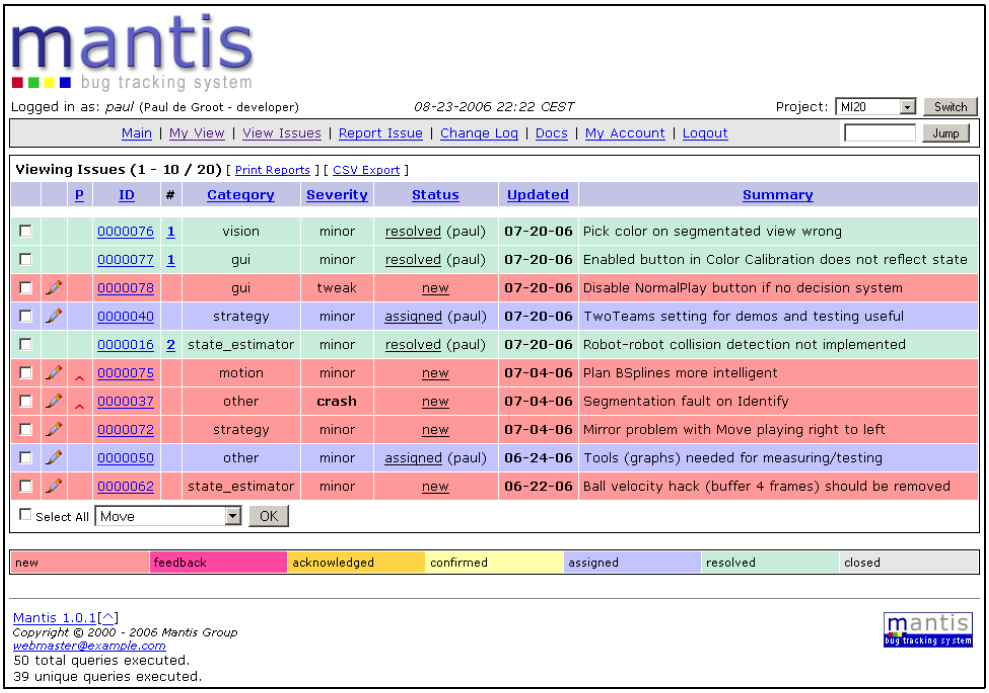


Figure 3.3. - Bug tracker Mantis can be used to keep track of bugs that were found, but are yet to be fixed. By keeping track of defects this way, it it less easy to refrain from fixing them.

13 Compressed HTML, commonly used for help files for Windows programs.

comment (something like: “Fix me!”), but time-constraints often ensured these bugs were forgotten as well.

In March 2006, a so-called bug tracker was introduced. The web application Mantis (an open source project started by a student at Twente University), enables us to keep a handy overview of the tasks that should be performed. It can also store additional information about the tasks and who is currently executing that task.[24]

Students that encounter a bug that they do not wish to fix right away, or do not know how to fix, are encouraged to enter the bug in Mantis. This way, a database of known defects exists, so that a bug will always be tracked.

Of course, keeping track of bugs is useless when they are never fixed. It is therefore advisable to pick a time of the week in which the bugs in Mantis will be fixed. When taking an afternoon to fix tracked bugs on a regular basis, not only will the software become more stable, it can also be a welcome distraction from one's Master's assignment.

3.4. Builder

Version 1.0 of the system was mainly developed on Linux systems. Therefore, compiling and linking the whole system was done using the standard way on Unix-like systems: with Makefiles. Unfortunately, when dependencies within the systems change, *make* does not automatically detect this.

Large programs are often split up in multiple source files. The types and functions defined in these source files are contained in header files. These header files get included by every source files that makes use of the types and functions. When a header changes, any source file that includes this header file should be recompiled. This is needed because changes, e.g. a change in the size of a data type, can result in different assembly code in the functions. When these source files are not recompiled, weird behaviour and crashes may be the result.

Well-maintained Makefiles contain a list of the dependencies of each source file. For each file, the file upon which it depends are stated. On execution of *make*, if any of the dependant files has been changed, the source file will be recompiled.

Sadly, it is easy to forget to adapt the Makefile when you include another header in a source file. Therefore, it was possible that – upon recompilation – strange, inexplicable errors occurred. Students quite often just removed all temporary files and recompiled the whole system, to see that their problems had miraculously disappeared.

To tackle this problem, a tool called *Builder* was created by the author. Upon every compilation, Builder quickly scans the source files for lines beginning with '`#include`', identifying a dependency on another header. This annihilates the need to specify dependencies manually, or at least to manually instruct a tool to update the dependencies. Other tools (like `gcc` with the `-M` flag) actually parse the source files, making them perfectly

accurate, but also much slower than Builder. In most cases, results from Builder are equally accurate and when this is not the case, source files can easily be adapted to make Builder give the correct result.

Another small, but handy feature is that Builder shows the progress of the compilation. This is nice, since compilation of the complete MI20 system can take up to five minutes depending on the CPU.

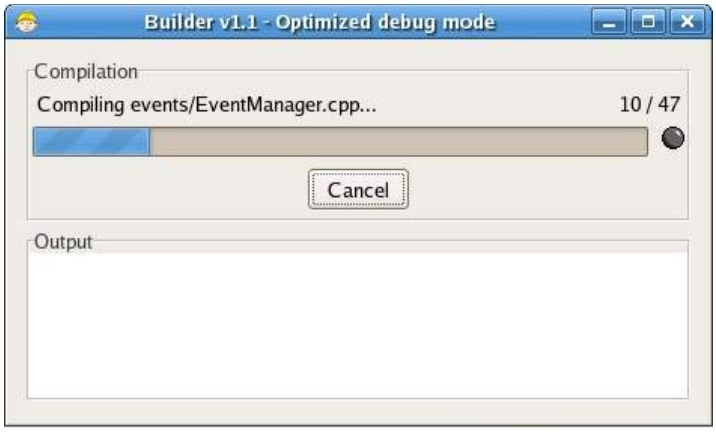


Figure 3.4. - The Gtk+ interface of the Builder program.

Builder also has three modes of compiling the program. These modes differ in the level of optimization performed. These modes are:

- **Release** – In release mode the program is compiled to run as efficiently as possible. The optimizer tries to use the CPU's registers for most operations, simplifies calculations, removes unreachable code etc. etc. Testing code such as assertion checking and memory leak detection will not be compiled in.
- **Debug** – In debug mode, the developer gets as much help from the system as possible. Functions are compiled as-is, so not optimized or inlined. At every function call the memory stack is checked for corruption and function names and symbols are included in the executable, so that you can use a debugger like `gdb` to debug your program. Also, Builder defines the DEBUG preprocessor variable, making sure that memory leak detection and assertion are compiled in. As a result of all these extras, the executable will be huge and most likely will run three to four times slower.
- **OptDebug** – Optimized debug mode is a combination of the two modes explained above. The DEBUG preprocessor variable is defined, so assertions and memory leak detection are used. However, the executable is optimized, running at nearly full speed, and symbols are removed from the program. This is the mode which is recommended to use during main development. Compile your program in optimized debug mode and test it by running. If you encounter a problem for which you need the help of the debugger, like a segmentation fault or a failed assertion, recompile the program in debug mode and feed that executable to `gdb`.

3.5. Coding style

Throughout the redesign of the system an attempt has been made to maintain a consistent coding style all throughout the application. Coding style is a matter of taste and somewhat personal. It is impossible to identify a *correct* coding style, only the one you like best. However, working with code that has a style that is somewhat different from your own preference is always easier than working on code with so many different style that you can never get used to one style.

The coding style that is used for the majority of the MI20 system follows chapter 7, 9 and 10 of the Java code conventions[25]. Names of variables should have a capitalized first letter for each internal word (i.e. `aLongName`), class names begin with a capital and names should preferably be spelled out completely (so `segmentationValue` and not `segVal`) . To easily distinguish between member variables and local variables, member variables are prefixed with `m_`. No Hungarian notation is used[14].

A method should be either so small that the comment above the method can explain the working of the entire method, or the method should contain inner comments. Comments should be written so that another developer (or you, later) can see in a glance what the method does. Also, describe rationales in comments: why is, say, a hyperbolic tangent function used?

The most controversial coding style decision that anyone can make is always the placement of braces. In the case of the MI20 system, the decision has been made to put each brace at a line of it's own, because this makes a screen full of code look less cluttered and more readable early in the morning or late at night. This change is also a large contributor to the increase in lines of source code between version 1.0 and version 2.0 of the source code.

3.6. Conclusions

The most significant change made in the software development process was the reinstatement of a source code management system. With the change from CVS to Subversion, it became possible to use several useful features. Features like branching are now used (although they were available in CVS), thereby increasing the confidence in the code that is used in at championships..

Most of the code is now documented using the Javadoc-style, which should be familiar to any student that works on the system. This documentation can then be automatically read by the tool *Doxygen*, which will generate HTML-documentation from it. This makes the use of the documentation convenient, since it has become searchable and hyper-linked.

14 Hungarian notation describes a convention introduced by a Hungarian engineer at Microsoft, Simonyi Károly. In this convention a variable name is prefix by either its function (Apps Hungarian) or its data type (Systems Hungarian). For example, many Windows APIs contain identifiers like `szName`, showing that the variable is a **s**tring that is **z**ero-terminated.[26][27]

A bug tracker was introduced, that keeps track of defects that were found (or suspected) in the code. It is not always possible or desirable that a bug is fixed as soon as it is detected. Also, the developer that finds the bug may have little knowledge of the part of the system that contains the bug. Using a bug tracker, these bugs can be fixed later, but will not be forgotten. It also might be a good tool for supervisors to get an idea of the state of the code.

A tool called *Builder* was introduced to help compilation and linking of the code. Builder automatically determines dependencies in the source files, making it easier in use than the standard *make*, but still retaining the freedom to use whichever development environment the programmer desires.

CHAPTER

4 Test environment

When working on an academic assignment it is often of utmost importance to perform tests to confirm that the adaptations made are in fact beneficial. Therefore, it is very useful to have a convenient way of gathering system data.

The usual way of gathering data from the system was to include debug messages in the code where the required information was available. Then, this data could be captured and used for processing in Excel, Matlab or other applications.

To limit this practice, functionality was added to the system to gather and visualize data, inspired on the so-called *Supervisory Control and Data Acquisition systems* (SCADAs), used to monitor machinery.[28] Data can be visualized on screen or output to a comma-separated file to enable processing in other applications.

4.1. Graphing system data

Two Master students were observed as they were performing test to support the conclusions of their theses. These observation inspired the creation of a test environment, for which the following user requirements were determined:

- The user should have convenient access to the measurement data contained in the system state.
- A basic visualisation of the data gathered should be possible, to get a quick overview of the behaviour of certain variables within the system. It should be possible to create simple graphs that are suitable for inclusion in a thesis without further processing.
- System running speed should not be impaired by the inclusion of the test environment. Only when tests are actually used, is a performance hit acceptable.

- It should be possible to export data from the MI20 system in a format that is readable by external applications, such as Matlab, Microsoft Excel or OpenOffice Calc.

Using these requirements, a new option has been developed in the MI20 system that allows the user to create graphs of certain data, such as robot orientation or ball velocity. The data used for the graphs is gathered as soon as the graph dialog opens and graphs are updated real-time. The user can stop measurements by clicking a button, allowing him/her to study the graph, take a screen shot or export the data.

The *type* of graph determines what is measured. The following graph types can be used in the system:

- X-position (in mm)
- Y-position (in mm)
- Linear velocity (in mm/s)
- Angular velocity (in rad/s)
- Orientation (in rad)
- Predicted linear velocity (in mm/s)
- Predicted angular velocity (in rad/s)
- Linear control signal (in mm/s)
- Angular control signal (in rad/s)

The *target* of the graph determines which object is tracked. Possible targets currently are robots 1 to 5 and the ball. Opponent robots cannot be a target for the graph, since there is no data for the control signals.

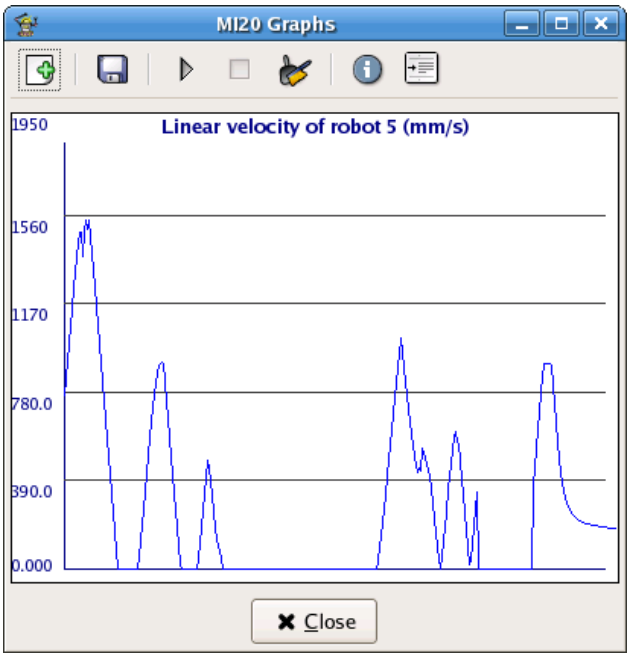


Figure 4.1. - Screenshot of the new graph dialog.

Data can be exported to a CSV-file, which can be read by Excel, Matlab and several other programs. CSV literally means *Comma Separated Values*, but is also the name Excel gives to files that contain values that are delimited by tabs. The CSV-files contain a header, describing the type of measurements that are in the file and then lines containing the time and value of the measurement. It is possible to create a CSV-file that contains the measurements of a single graph, or a large file that contains the measurements of all possible graph types and targets.

4.2. Statistical tools

During the gathering of data, some statistical values are calculated on the fly. To do so, we keep track of the measurement count, the sum and the squared sum. Whenever a new measurement becomes available, these values are updated:

$$s_n = s_{n-1} + val \quad \text{and} \quad sq_n = sq_{n-1} + val^2$$

Also, a check is performed if the new value exceeds the previous maximum or is smaller than the previous minimum.

Figure 4.2. - The statistics dialog shows some basic statistics about the current graphs.

Then, at any time, the average and standard deviation of the measured values can be calculated, as follows[29]:

$$avg_n = \frac{s_n}{n} \quad \text{and} \quad \sigma_n = \sqrt{\frac{1}{n-1} \cdot \left(sq_n - \frac{s_n^2}{n} \right)}$$

Furthermore, a statistical tool was developed to help set the correct values for the Kalman filter that filters the measurements in the state estimator of the system. The Kalman filter models the measurement noise as a multivariate Gaussian distribution with a certain covariance matrix.[30] This covariance matrix can be determined by placing a (non-moving) robot on the field and start the measurements. The system can then determine the covariance matrix. The sample period can be as long as the user wishes, but as with all statistics, the accuracy of the measured covariance is better with a large number of samples. After a measurement, the user can stop the data gathering, clear the data and start a new measurement.

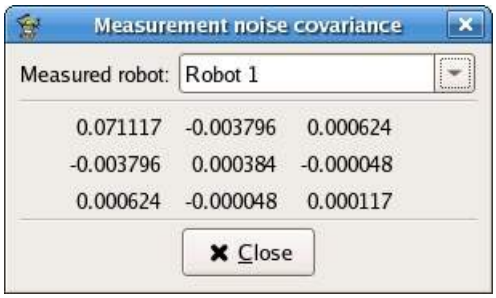


Figure 4.3. - The dialog showing the measured noise covariance.

The observation state in the MI20 system is a column vector of 3 variables:

$$z = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

being the x-position, y-position and orientation of the robot. Therefore, the covariance matrix is a 3x3 matrix. The maximum likelihood covariance matrix is defined as:

$$\hat{\Sigma} = \frac{1}{n} \sum_{i=1}^{n} (X_i - \bar{X})(X_i - \bar{X})^T$$

where n is the number of samples, and $\bar{X}$ is the average value of the state variables. The matrix calculated is a biased estimation, since its expectation is:

$$E(\hat{\Sigma}) = \frac{n-1}{n} \Sigma$$

This fact gives us a simple adaptation to find the formula for an unbiased sample ML-covariance matrix:

$$\hat{\Sigma} = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \bar{X})(X_i - \bar{X})^T$$

For each frame, the values of the observation state are measured. For each of the samples we can calculate the following matrix:

$$\Sigma_i = \begin{bmatrix} (x-\bar{X})^2 & (x-\bar{X})(y-\bar{Y}) & (x-\bar{X})(\theta-\bar{\theta}) \\ (y-\bar{Y})(x-\bar{X}) & (y-\bar{Y})^2 & (y-\bar{Y})(\theta-\bar{\theta}) \\ (\theta-\bar{\theta})(x-\bar{X}) & (\theta-\bar{\theta})(y-\bar{Y}) & (\theta-\bar{\theta})^2 \end{bmatrix}$$

and at any time find the covariance matrix for use in the Kalman filter with:

$$\hat{\Sigma} = \frac{1}{n-1} \sum_{i=1}^{n} \Sigma_i$$

4.3. Game recording

When trying to improve our system, it is probably very useful to analyse previous games. To make this possible, the MI20 system contains an option to record a game.

These recordings can be very useful during testing of the system. Recorded snapshots can be used during the development and testing of systems that are influenced by noisy measurements. An example of this might be the adaptation of the Kalman filtering. By using recordings, the data that is used as an input remains predictable and reproducible.

Game recordings could also be used for analysing the effectiveness of our strategy. Reviewing what happened during the games played at championships might offer insight in possible improvements that can be made.

Finally, game recordings might just be used to view a memorable game again.

A first attempt at including game recording in the MI20 system was made just before the European Championship in Vienna, for the reasons listed above. The recording was implemented in the Vision module, that stored each snapshot as it became available. Using a command line option, the Vision module could be instructed not to get images from the camera, but read the file containing the snapshots.

Later, a second version was implemented. This format is quite a bit more flexible. The user can choose to record world data, snapshots and the commands that were sent to the robots. It is also possible to add comments to the recording, such as the time and place of the match and the teams involved.

The implementation of game recording again showed the advantages of using the event system for inter-module communication. When the user starts recording a game, event listeners are registered according to the selected options regarding what should be stored in the recording (i.e. world data, snapshots or robot commands). Whenever the corresponding events flow through the system, the game recording code will also receive this event and can write it to disk.

Similarly, when reading data from the recording file, snapshots and world data can be fed into the event queue transparently. The listening modules have no idea this data was recorded and did not originate from the Vision or StateEstimator module respectively.

Reading the more flexible game recording format has not yet been implemented into the system. However, the deserialization of the data types used has already been written, because of the use in the networking code. Therefore, it should not be difficult to implement this. A specification of the file format can be found in appendix B.

4.4. Conclusions

To accommodate testing, it is now possible to graph much of the data that is handled by the MI20 system. Data can be exported to CSV-files, making it possible to use this data in Microsoft Excel, OpenOffice, Matlab or numerous other programs.

Tools for determining minimum, average, maximum and standard deviation for all these data types have been included, as well as the possibility to determine a covariance matrix for use in the Kalman filter.

Also, functionality has been included to record games played, inserting measurements into the system as if those were done at that particular moment. This makes it possible to reproduce tests, and compare results accurately. An added advantage is the possibility to review games played at championships, to find flaws in our strategy.

When the test environment will be actually used by Master students, it is likely that this students will find the need for additional features. Therefore

it is likely that this environment should be expanded, according to those new user requirements.

One possible addition might be the option to specify an expression that will be used to create a new graph type. For example, if a student has devised an algorithm to minimize the distance between a robot and the ball, an expression like '`(ball.location - robot1.location).length`' would show a graph of this distance. The student could then quickly check if the algorithm works satisfactory.

CHAPTER

5 Simulation

Development of the robot soccer system will be an ongoing process. Algorithms will be changed and new decision systems will be implemented. When developing, testing newly written code is – of course – crucial. The easiest way of checking correctness is simply running the system. Unfortunately, even this can be quite time-intensive: robots have to be placed on the field, cameras and lighting need to calibrated, and so on. This problem can be overcome by performing the preliminary tests in a simulated environment.

Another important advantage of using a simulator for testing is the possibility to create a reproducible situation. This would make it possible that, when an error is spotted, the experiment can be repeated to check that the mistake has indeed been corrected.

Simulation can also make it easier to determine the cause of an error. For example, it might be possible that an error only occurs when the measurements contain noise. This can be determined by switching on and off the noise in the snapshots that are fed into the system; something that would have never been possible without simulation.

Finally, the simulator can be used to play games faster than real time, so that using machine learning (e.g. neural networks or genetic algorithms) becomes feasible.

5.1. Physical model

To model a robot soccer game, quite some physics are needed to provide a realistic model of the world. The current MI20 simulator contains realistic friction, collision detection and a model of the robot motors.

5.1.1. Air resistance

In cars, the major component of the friction that is experienced is air resistance. The air resistance can be determined with the following

formula:

$$F_{air} = \frac{1}{2} \cdot \rho \cdot c_d \cdot A \cdot v^2$$

where ρ is the density of the fluid or gas the object travels through (for air this is 1.29 kg/m^3), A is the area of the object that is perpendicular to the movement direction and v is the velocity of the object.

The drag coefficient (c_d) is a factor that describes how aerodynamic an object is. For a box, which the front of the robot essentially is, 1.95 is a good approximation [31].

When we calculate this value for an Austro-bot travelling at 3 m/s – which is practically the maximum speed reached on a 7x7 field, we get:

$$F_{air} = \frac{1}{2} \cdot \rho \cdot c_d \cdot A \cdot v^2 = \frac{1}{2} \cdot 1.29 \cdot 1.95 \cdot \left(7.5 \cdot 10^{-2} \cdot 4.8 \cdot 10^{-2}\right) \cdot 3.00^2 = 4.1 \cdot 10^{-2}\ N$$

Although air resistance at the maximum speeds is not completely negligible, it is largely outweighed by static and dynamic friction. Currently, the simulator neglects air resistance for a robot travelling below 100 mm/s. Above that speed, air resistance is added to the robot.

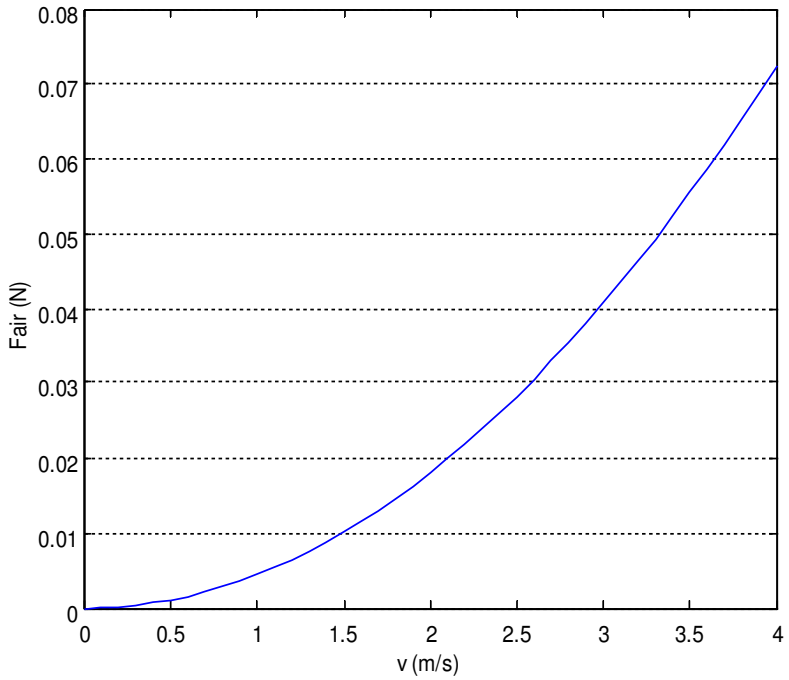


Figure 5.1. - Air resistance of an Austro-bot. It is clear from the graph that air resistance is only a minor factor in the friction model. Only at high speeds this force becomes slightly significant.

5.1.2. Rolling resistance and sliding friction

Rolling resistance is the friction that occurs when an object rolls over the ground. It is caused by the deformation of the wheels under the pressure of the weight of the vehicle. The force acts in a direct opposite direction to the rolling direction and its magnitude is a factor of the normal force of the object. In formula:

$$F_{rr} = \mu_{rr} \cdot N = \mu_{rr} \cdot m \cdot g$$

The factor μ_{rr} is a unit-less factor that determines the amount of friction, m is the mass of the robot and g is the gravitational acceleration.

It should be noted that the friction that can be calculated with this formula is the *maximum* rolling resistance. If the force that is responsible for the rolling motion of the robot is less than the maximum rolling resistance, then the actual rolling resistance will be equal to the applied force, resulting in zero movement.

A comparable force is *sliding friction*, which is the force that acts on a robot when it is pushed laterally, or when its wheels are locked. The formula for sliding friction is the same as that for rolling resistance, only the factor μ is different.[32]

In the simulator, it is assumed that the wheels of a robot can always roll freely. Therefore the frictional forces that act on the robot can be split into two parts. In figure 5.2 we see a robot that is moving along the velocity vector v. Although the engine force of the robot can only be exerted in the direction of the robot's orientation (this is called a non-holonomic constraint), the actual velocity of the robot may have been influenced by collisions or skidding.

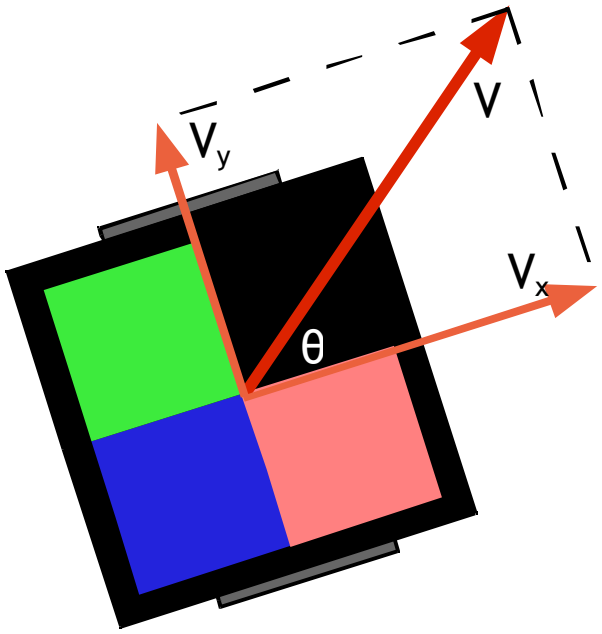


Figure 5.2. - The robot's velocity vector is split up in a vector that points in the robot's orientation and a vector that moves the robot sideways.

The velocity is split up in two components: one in the direction of the robot (denoted v_x in the figure) and a lateral component (v_y in figure 5.2). These velocities are found using two dot products:

$$|v_x| = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} \cdot \vec{v} \quad \text{and} \quad |v_y| = \begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix} \cdot \vec{v}$$

The velocity component v_x is subject to rolling resistance, while v_y is used for the sliding friction.

As said, the friction is limited to the force that would completely halt the motion in that direction. Therefore, it becomes necessary to calculate that force. This can be done by using Newton's Second Law:

$$v = a \cdot \Delta t = \frac{F}{m} \cdot \Delta t \Leftrightarrow F_{max} = \frac{v \cdot m}{\Delta t}$$

In this formula Δt represents the time step of the simulation, which is

currently set at 1 millisecond.

The forces calculated above dampen the linear velocity of the robot. A similar effect occurs for the angular velocity. This friction delivers a *torque*, which can informally be seen as rotational force.

Just as with the linear velocities we can also calculate the torque that would be necessary to reduce all angular velocity to 0:

$$\omega = \alpha \cdot \Delta t = \frac{\tau}{I_{zz}} \cdot \Delta t \Leftrightarrow \tau_{max} = \frac{\omega \cdot I_{zz}}{\Delta t}$$

Here, τ is the torque, ω stands for the angular velocity, α is the angular acceleration resulting from the friction torque and I_{ZZ} is the component of the inertia tensor that describes the preservation of rotation around the z-axis. This inertia equals [33]:

$$\frac{1}{12} \cdot m \cdot \left({s_x}^2 + {s_y}^2 \right)$$

where s_x and s_y are the lengths of the sides along the x- and y-axis.

To determine the values to use for μ_{rr} and similar constants, a series of experiments was set up.

The simplest experiment was determining the frictional constant of the ball. To determine this constant, the ball was rolled over the field a number of times while its location was recorded by the camera. From this data, the deceleration was determined and the constant could be calculated using Newton's Second Law:

$$F_{rr} = m \cdot a \Leftrightarrow$$

$$\mu_{rr} \cdot m \cdot g = m \cdot a \Leftrightarrow$$

$$\mu_{rr} = a / g$$

A series of measurements was done, resulting in:

$$\mu_{rr} = 4.731 \cdot 10^{-3} \pm 1.51 \cdot 10^{-4}$$

To determine the friction torque, a similar experiment was done. The robot was given a large rotational velocity by hand and the camera registered the angular deceleration. This yielded:

$$\tau = 1.606 \cdot 10^{-2} \pm 8.52 \cdot 10^{-4}\ Nm$$

The friction coefficients for the robots were determined with another experiment. A robot was placed on a slope, as can be seen in figure 5.3. The angle of the slope was then gradually increased until the robot started to

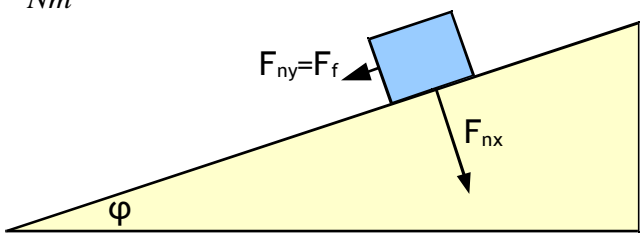


Figure 5.3. - The linear friction coefficients were measure by placing the robot on a slope and measuring the angle at which it started to move.

move. At first, the friction will stop the robot from moving. In that case, the frictional force is equal to the component of the normal force that pushes the robot down the slope (F_{ny} in the figure). As the angle of the slope increases, so will this normal force. Once it is larger than the maximal frictional force, the robot will start to move. At that point the angle of the slope was measured, and the friction coefficient could be calculated with the following derivation:

$$F_{ny} = F_{fr} \Leftrightarrow m \cdot g \cdot \sin(\varphi) = \mu \cdot m \cdot g \Leftrightarrow \mu = \sin(\varphi)$$

The value for the sliding friction found was:

$$\mu_{sliding} = 4.81 \cdot 10^{-1}$$

A similar experiment was done to determine the rolling resistance. The difference was that now the robot was given a small velocity. The camera above the robot soccer field was used to see if the robot would retain that speed. In that case the normal force cancels out the rolling resistance and μ_{rr} can be determined. It should be noted that various factors make it hard to measure exactly if the robot's velocity has really remained constant. The value found for μrr therefore can only be considered an estimate:

$$\mu_{rr} = 1.71 \cdot 10^{-2}$$

5.1.3. Collision model

One of the most important features of simulation is collision detection and collision handling. The whole game of robot soccer can be seen as the deliberate engaging or avoiding of collisions. After all, the robots' only way of scoring a goal is to collide with the ball in such a manner that the ball will roll into the goal.

A first iteration of the MI20 simulator modelled the robot as a simple box. However, it was observed that – although collisions with the back and the sides of the robot appeared quite realistically – when the ball was hit on the front, it bounced away too much. The actual robot is formed at the front in such a manner that it is able to keep the ball near, so it can dribble the ball. This feature is used often in shooting the ball, and increases the chance of a goal considerably.

When using the simulation for machine learning purposes it might be important that the chances of a successful attack resemble reality quite closely.

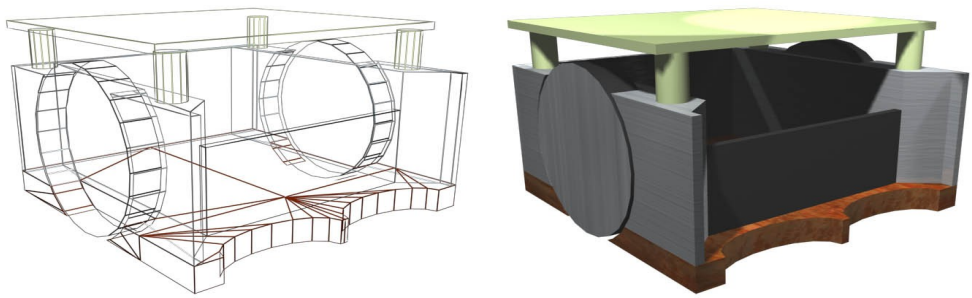


Figure 5.4. - Model of the Austro-bot in wireframe version and with solid rendering and texturing

Therefore, it was considered that the behaviour observed using a box as collision geometry was not desirable, so a more accurate model of the robot was created in a program called Blender. Blender is a 3D-modelling application that is available under a GPL-license, for both Windows and Linux.[34][11] The constructed model was exported in an easily parseable file format called *ASCII Scene Export* (ASE), and using a small self-written program converted to a format that could be easily used by OpenGL (the API used for rendering) and ODE (a dynamics library, see paragraph 5.2).

The model shown in figure 5.4 is currently used in the rendering of the 3D-view of the user interface, but using it for collision detection gave some problems. As can be seen from the figure, the model is hollow; a space that in the real robots contains the motors and battery pack. Because the detection of collisions happens only on discrete time steps, it was very well possible that the ball was outside the robot at one time step, but was in the hollow space the next time step. No collision would then be detected, and the ball would become confined in the hollow space.

One solution to this problem, of course, is to use smaller time steps. A large disadvantage for this is that the computation time required for the simulation will increase linearly with the decrement of the time step. Therefore, the choice was made to adapt the model of the robot, so that it does not contain this hollow space. See figure 5.5. Using this model results were much better[15].

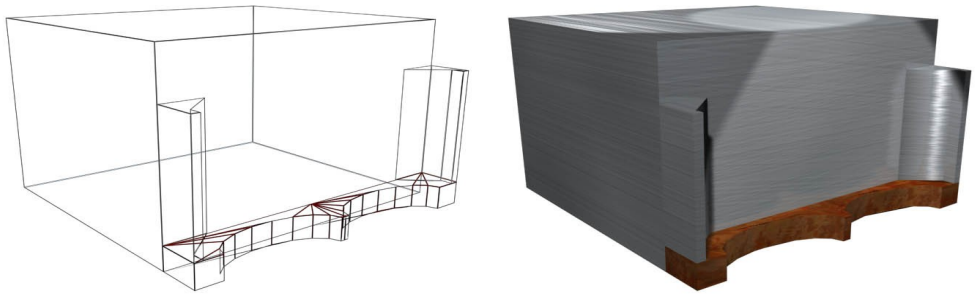


Figure 5.5. - Adaptation of the robot model to accommodate for problems that were encountered when using a more realistic model in collision detection. This model does not contain hollow shapes, still following the actual front of the robot quite closely.

In every time step of the simulation, it should be tested if a collision occurred. There are four types of collisions that should be detected by the simulation[35]:

- A collision of a robot with the wall
- A collision of a robot with the ball
- A collision of a robot with another robot
- A collision of the ball with the wall

Even when we do not consider collisions with the walls, the amount of tests

15 Simulation, however, is quite a bit slower than with the box model. Therefore, the accurate collisions are currently switched off until optimizations have been realized.

is quite large. A simple 5x5 game contains 11 objects that need pairwise testing to see if they intersect. In a naïve implementation, this would result in 66 collision tests in every time step (of which there are 33 per frame, giving 2178 tests per frame). In a 7x7 game, that number would increase even to 120 x 33 =3960 tests.

To reduce the amount of collision tests, two optimizations have been made. First of all, when testing for a collision with a robot, a first quick collision test is done by representing the robot as a box. If there is no collision with this proxy collision mesh, there is no need to check for collisions with the more accurate model.

Secondly, the field has been divided using a quad tree. The quad tree is a space partitioning, where the root node is a rectangular area that contains the entire field and contains four evenly-sized children that each contain a quarter of the root node's area. This partitioning into smaller areas is stopped when the tree has a depth of 4. An object will be listed in the smallest cell than can completely contain the object.

The advantage of this way of organizing the collision objects, is that when checking for possibly colliding objects, it is possible to quickly discard all objects that are nowhere near the object that is tested.

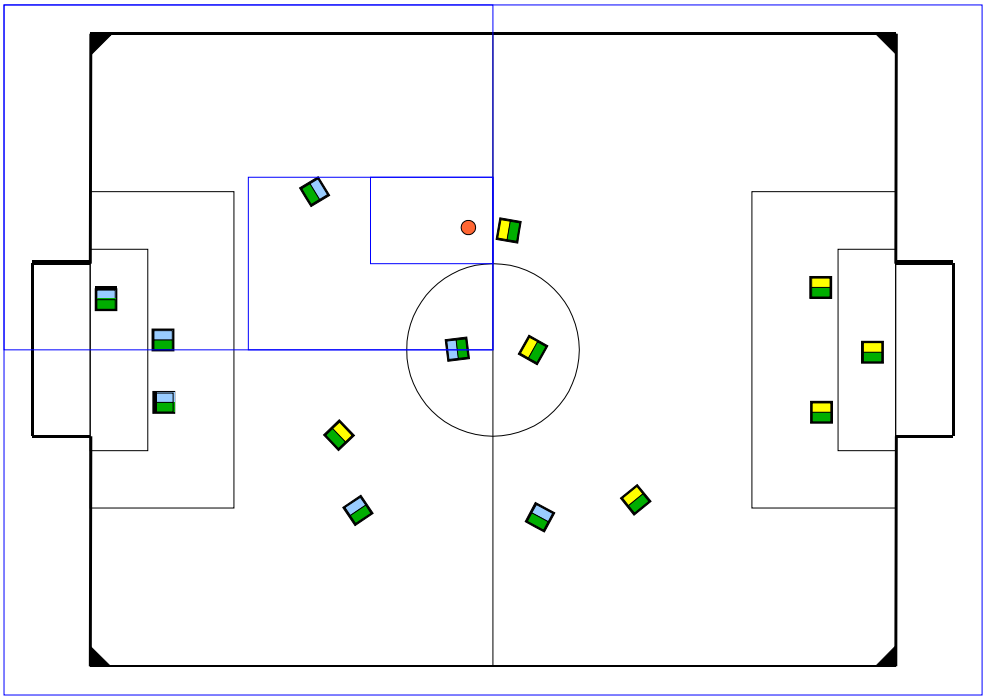


Figure 5.6. - Quad tree decomposition of a 7x7 robot soccer game. The collision detection mechanism can determine from the quad tree that there are no potential candidates for colliding with the ball.

Consider the 7x7 game that is depicted in figure 5.6. When determining if the ball is colliding with another object, a naïve collision detection algorithm would perform collision tests with 14 objects, excluding the walls. The quad tree implementation, however, can quickly determine that there is no object in the same cell as the ball, so that no collision can possibly have occurred.

5.1.4. Robot motor model

When the simulator receives a robot command, two different courses of action can be taken, depending on a user setting.

If the user chose for an unrealistic motor modelling, the robots will immediately have the speeds sent to the simulator. This can be very useful for open-loop motion testing.

A more realistic model can also be chosen. In this model, each wheel is controlled separately, just as is done in the Austro-bots. The velocities in the command are first converted to left and right wheel speeds. (The formula to do this can be found on page 25). For each motor, these speeds are set as the target speed.

Then, every time step (of 1 ms) a discrete PID-controller tries to minimize the error between the current wheel speed and the target wheel speed (err_k), as is done on the Austro-bots[36]:

$$out_k = out_{k-1} + err_k \cdot \alpha - err_{k-1} \cdot \beta + err_{k-2} \cdot \gamma$$

where α, β and γ are tuning parameters that are functions of the normal gains found in a PID-controller. The tuning of these parameters was done by the designers of the Austro-bots, which yielded: α = 4512, β = 4192 and γ = 512.

The output signal is clamped between -256 and 256, since this is the range of the encoder used, the Faulhaber IE2-512[37].

Now that the control signal for the motor has been determined, we need to find the resulting new wheel speed. Our robots are using DC-motors to move. These motors use a magnetic field to rotate a coil connected to the drive shaft. As with all electric devices the voltage through the coil (V) can be determined using Ohm's law:

$$V = I \cdot R$$

where I is the current through the coil, and R the resistance. However, in a DC-motor the magnetic flux of the magnetic field generates an induction current. The voltage generated by this induction is called back-electric motor force (back-EMF), so the previous formula becomes[38]:

$$V = I \cdot R + V_{ind}$$

The induction current is proportionate to the rotation speed (ω) of the coil. Besides, the current flowing through the coil is proportionate to the amount of torque (rotational force, τ) delivered by the motor. The formula therefore can be written as:

$$V = \frac{\tau}{k_\tau} \cdot R + k_e \cdot \omega$$

The values k_τ and k_e are constants that are characteristics of the specific motor used. It can also be proven that under normal circumstances k_τ and k_e are equal. Substituting these values for k and rewriting the formula so

that it provides the engine torque, gives:

$$\tau = \frac{V \cdot k}{R} - \frac{k^2 \cdot \omega}{R}$$

The values for k and R are known. Those used in the simulation can be found in appendix C. The right part in the formula is the cost in torque of the induction current. The motor can be controlled using the voltage on the coil. This can vary from -6 to 6 V.

At each simulation step the current rotation speed of the engine coil is calculated, using the current wheel velocity[16]:

$$\omega_{engine} = \frac{v_{wheel} \cdot N}{r_{wheel}}$$

In this formula, N is the ratio of the gears that connect the motor with the wheel. On the Austro-bots, this ratio is 25:3 [39]. r_{wheel} is the radius of the wheel. Then the output signal from the PID-controller (out_{PID}) is taken and used to calculate the engine torque that is delivered by our motor:

$$\tau_{engine} = \frac{\frac{1}{256} out_{PID} \cdot V \cdot k}{R} - \frac{k^2 \cdot \omega_{engine}}{R}$$

This torque is exerted on the ground via the wheel, resulting in a driving force, calculated with:

$$F = \frac{\tau_{engine} \cdot N \cdot \eta}{r_{wheel}}$$

The factor (η) here is the motor efficiency. This force can then be used to determine the new wheel velocity, using Newton's Second Law. Since the robot contains two motors, the mass of the robot should be halved:

$$v_k = v_{k-1} + \frac{F}{\frac{1}{2} m}$$

5.1.5. Measurement noise

To test the ability of the system to cope with noisy measurements, it is possible to add noise to the snapshots that might be generated by the simulator. As seen in paragraph 4.2, the measurement noise is modelled as a multivariate Gaussian distribution. So, for each measurement we need a noise vector X:

$$X \sim N_3(0, \Sigma)$$

where Σ is the covariance matrix of the noise distribution.

16 This velocity can originate from movement of the robot, but also from collisions of other robots.

To find a random vector X that comes from the normal distribution above, the unique lower triangular matrix L needs to be found that satisfies:

$$L L^T = \Sigma$$

This matrix L is known as the Cholesky decomposition (or the matrix square root) of Σ. The values of the elements of this matrix can be found with [40]:

$$l_{ii} = \sqrt{s_{ii} - \sum_{k=1}^{i-1} l_{ik}^2} \quad \text{and} \quad l_{ji} = \frac{\sqrt{s_{ji} - \sum_{k=1}^{i-1} l_{jk} l_{ik}}}{l_{ii}}$$

The most convenient way of calculating the elements of this matrix is by starting at the upper left corner of the matrix, and calculating the values column by column. This is known as the Cholesky-Crout algorithm.

A noise vector can now be created by multiplying the matrix L by a vector of independent Gaussian distributed values. These values are obtained by using a Mersenne Twister pseudo-random number generator[41]. This type of generator gives uniformly distributed values, which can be transformed to Gaussian distributed values using the Box-Muller transform[42].

Let r_1, φ_1, r_2 and φ_2 be four random values between 0 and 1 from the random number generator, we can now calculate a noise vector with the correct multivariate Gaussian distribution with:

$$X = L \cdot \begin{bmatrix} \cos(2\pi\phi_1) \cdot \sqrt{-2 \cdot \ln r_1} \\ \sin(2\pi\phi_1) \cdot \sqrt{-2 \cdot \ln r_1} \\ \cos(2\pi\phi_2) \cdot \sqrt{-2 \cdot \ln r_2} \end{bmatrix}$$

This noise vector can then be added to the observation state to get the measurement that is placed in the snapshot.

To make sure the reproducibility of the simulator results is left intact, the Mersenne Twister random number generator is seeded with a value that is specified in the simulator configuration. Therefore, when the user does not change this seed, there will be no differences in the noise added between two runs of the simulator.

5.2. Implementation

The simulator is currently implemented as a server and the MI20 program functions as a client. This client-server architecture makes it simple to let two teams play against each other: one simulator can then listen for two MI20 applications, each determining the actions for one team.

Communication between the two programs makes use of the networking code as described in paragraph 2.6.3. A `SimulatorHandler` within the client program transmits events over the network that are important for the simulator (like robot commands). The simulator, in turn, generates events that are placed in the event queue when they are received by the simulator handler.

The state of the world is contained in a `PhysicalModel` class. This class is

responsible for the simulations described in paragraph 5.1 and makes heavy use of a library called Open Dynamics Engine (ODE). ODE is an open-source physics library, targeted at rigid body simulation and collision handling [43]. Using this has saved a lot of time that could now be spent on the actual simulation, instead of spending time on integration methods and collision algorithms.

Every frame, the `Referee`-class looks at the world state and decides if the game rules require intervention. These events might be a scored goal, a stalemate (the ball is not touched for more than 10 seconds), or one of the possible fouls. Some violations of the game rules are very difficult to detect (e.g. a collision should influence the game significantly to be a foul), so these are not recognized. When a foul could be detected, the client should be asked to supply the targets of its auto-positioning system and the game recommences.

5.3. Conclusions

The new simulator that has been added to the MI20 system makes testing a lot easier. It is no longer necessary to set up the field just to test newly written code.

A physical model has been developed for simulation that takes into account air resistance, static and dynamic friction, collisions between all objects and the inner workings of the robot motors. It is even possible to simulate the noise that is experienced when using the camera to observe the world state.

Since the model used for simulating the game situation is deterministic, games played with the simulator are reproducible.

The fact that games can now be simulated, makes it possible to use machine learning to develop new game strategies.

To really make machine learning feasible, though, some small enhancements to the simulator are recommended. Although the simulator's `Referee`-class checks for violations of the game rules, it does not yet stop the game and communicate these violations to the clients. The framework to this is available[17], only the specific implementation has yet to be created.

It is important to note that testing using the simulator should never replace testing in real-life situations. Although the model of the simulation gives quite realistic results, it still remains a model, i.e. a *simplified version* of reality. It would be a pity to do all tests in the simulator, only to discover at a championship that performance is not that good.

17 When a goal is scored, this fact *is* currently sent to the clients. In a same manner all other communication can be handled.

CHAPTER

6 Conclusions

The previous chapters described in detail the changes made to the software and development process of the MI20 robot soccer system. In this chapter, the balance sheet of these advancements will be drawn up, and some recommendations about further work will be given.

6.1. Conclusions

With all the development done in the past few months, it is believed that a large step has been taken to make the MI20 system more manageable.

Development should be easier now. A more clear-cut design and the documentation (in the form of both this thesis and documentation generated automatically from the source code) should make it less overwhelming to start working with the system, which is, mark you, a larger program than most students will have ever worked on. It is believed that students can now get an overview of the entire system more easily, so that they can better assess the impact of their changes.

Debugging has become more convenient as well. The unwieldiness of a system with multiple executables led to *caveman debugging*[18], which is quite time-intensive. Now, using tools like `gdb` is much more convenient and rewarding.

An end has also been made to the messy way of gathering test data. Often, students would just place log messages throughout the code, outputting the data they needed and redirecting this output to a text file. Now, all data is available and can be exported to CSV, or graphed directly.

All in all development of the MI20 system should now be possible in a more rapid manner, delivering better tested and more stable code, and hopefully resulting in more victories.

18 Caveman debugging is debugging by placing log messages throughout the code in which a bug is suspected. Then, by running the program and noting the log messages, the location of a defect or crash can be found. However, multiple runs of the software are typically needed to find the error.

6.2. Recommendations

Working eleven months on this project, led to some ideas about the direction further development in the project could take. These recommendations are split in two parts. Paragraph 6.2.1 contains recommendations about further work that can be done on the system. In paragraph 6.2.2 recommendations are given on how the development process might be improved.

6.2.1. Further work

Most of the designs described in this thesis have actually been implemented in the MI20 system. Unfortunately, there are still some features left unimplemented. Also, other features not yet designed will be mentioned in this paragraph, that might be good ideas.

When combining the individual executables of version 1.0 of the system into one monolithic program, the distributed nature of the system disappeared. Currently, this does not seem a problem, since current computers have sufficient processing power to run the entire robot soccer system on just one PC. However, there might be a need to implement networking soon.

The advantage of our current camera is that it is using Firewire, and that this device can be *daisy-chained*. This means that the data of the camera can be made available to other computers, simply by connecting the systems: they all share the same bus.

A new camera has been bought that communicates with the host computer using USB. Daisy-chaining is not possible with a USB camera, so that only one machine in the lab would have access to the camera images. A solution to this problem might be to have a sole Vision module running on the system that the camera is connected with. Other MI20 systems on other system could then connect with the Vision module, to receive the snapshots from there.

The games that are currently recorded using the second – more flexible – game recording format cannot yet be read by the system. The files from the first attempt to record games, containing only snapshots, can still be read though. Including this functionality in the system should not be hard, since all data types that are used in the format already contain methods for deserialization.

During the design of the new version of the MI20 system, emphasis was placed on manageability and testing. Tools have been added to make the system more manageable, and to make *acceptance testing* easier.

Another good way of guaranteeing that the code works as intended, is the use of *unit testing*. With unit testing a small piece of software is tested using simple test cases.

After checking code with unit tests, these tests should not be removed. Quite the contrary, they should be run on a regular basis, guaranteeing that changes made to the system have not broken existing code (this is called *regression testing*).

Currently a small unit testing framework is included in the source code of the system, but little tests have been written using this framework. It might be beneficial to include unit and regression testing more solidly in the development process.

The user requirements of the data acquisition and visualisation features were based on the few observations made of the two Master students that graduated while this redesign was done. It is very well possible that the actual use of these facilities will bring to light the need for extra options and features.

It is very likely that many more decision systems will be added over the years. When the amount of different decision systems grows, it might be beneficial to compile these into dynamic linked libraries (i.e. DLL-files or the Linux-equivalent, so-files). Then, the main executable of the system remains small and the system can scan the working directory for possible strategy libraries. The clear-cut boundary between the decision system and the rest of the system should make implementing this scheme not too difficult.

6.2.2. Process recommendations

Currently, only Master students are working on the robot soccer project. These students work on the project for a relatively short time[19], taking their built-up knowledge with them after graduation. This means that the people working on the system will always, on average, have little experience with it.

To increase the continuity within the project, it might be beneficial to include somebody in the team with hands-on experience, but with a more long-term commitment, for example a student assistant. This person could then support Master students with the practical side of their assignments. He[20] could also ensure that the use of development tools like Subversion will not come to a dead end again. He could keep track of the changes made to the system by the students, and detect possible problems. This way, he would function somewhat like a project leader. Furthermore, this team member could be responsible for fixing the defects found in the software, so that the students can focus completely on their assignments.

Other robot soccer teams around the world tackle the continuity problem by creating PhD-positions within the team. In that case continuity is maintained for at least three years. Besides, a PhD could perfectly

19 Especially when students from the Bachelor-Master track will work on the project, since their Final Project is only 30 credits, representing 21 weeks. Students from previous generations had a project length of 30 weeks.

20 Or she, of course.

(co)supervise the Master students on the project.

Of course, problem with the above solutions is the fact that they cost money. Solving that problem is left conveniently out of the scope of this thesis.

Furthermore, the project might benefit from setting more long-term goals. By keeping an overview of the current bottlenecks and shortcomings of the system, more long-term planning can be done and a road map can be established. Currently, students are kept relatively free in deciding the type of assignment they want to do. Having a long-term plan should lead to more concrete assignments, solving the current issues with the system, and progressing the software development in a more focussed manner.

Bibliography

[1] World cup final peaks at 4 million viewers at CTV, http://www.channelcanada.com/Article1453.html

[2] RoboCup official site, http://robocup.org/

[3] About FIRA - Overview, http://fira.net/about/overview.html

[4] FIRA MiroSot Middle league rules, http://fira.net/soccer/mirosot/MiroSot.pdf

[5] FIRA European Championship 2003 - Results, http://robotsoccer.fe.uni-lj.si/Results.htm

[6] FIRA World Cup 2003 - Results, http://www.ihrt.tuwien.ac.at/FIRAWM03/english/results/miro5/res_miro5_C.html

[7] Pressman, R.S., *Software engineering: a practioner's approach*, 1997, McGraw-Hill, New York, ISBN 0-07-709411-5

[8] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design patterns: Elements of reusable object-oriented software*, 1994, Addison-Wesley, , ISBN 0-201-63361-2

[9] Lethbridge, T.C. & Laganière, R., *Object-oriented software engineering*, 2001, McGraw-Hill, Maidenhead, ISBN 0-07-709761-0

[10] Model View Controller - Microsoft, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/DesMVC.asp

[11] The GNU General Public License (GPL), http://opensource.org/licenses/gpl-license.php

[12] Seesink, R.A., *Artifical intelligence in a multi agent robot soccer domain*, Master's thesis, Twente University, 2003

[13] Buth, M.D., *Ball-handling motion control for soccer playing mini-robots*, Master's thesis, Twente University, 2006

[14] Schepers, E.M., *Improving the vision of a robot soccer team*, Master's thesis, Twente University, 2004

[15] About wxWidgets, http://wxwidgets.org/about/

[16] Understanding the Linux kernel: process scheduling, http://www.oreilly.com/catalog/linuxkernel/chapter/ch10.html

[17] Cornes, P., *The Linux A-Z*, 1997, Prentice Hall, Hemel Hempstead, ISBN 0-13-234709-1

[18] McShaffry, M., *Game coding complete, second edition*, 2005, Paraglyph Press, Scottsdale, AZ, ISBN 1-932111-91-3

[19] Version control with Subversion, http://svnbook.red-bean.com/nightly/en/svn-book.pdf

[20] Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M., *Version control with Subversion*, 2004, O'Reilly Media, Cambridge, MA, ISBN 0-596-00448-6

[21] The GNU Lesser General Public License, http://www.opensource.org/licenses/lgpl-license.php

[22] The BSD license, http://www.opensource.org/licenses/bsd-license.php

[23] Doxygen features, http://www.stack.nl/~dimitri/doxygen/features.html

[24] About Mantis, http://mantisbt.org/about.php

[25] Code convetions for the Java programming language, http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

[26] Hungarian notation - Wikipedia, http://en.wikipedia.org/wiki/Hungarian_notation

[27] Hungarian notation, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs600/html/hunganotat.asp

[28] SCADA - Wikipedia, http://en.wikipedia.org/wiki/SCADA

[29] Meijer, T.M.J., *Syllabus 'Kansrekening en statistiek voor INF'*, Master's thesis, Twente University, 1999

[30] Kalman filter - Wikipedia, http://en.wikipedia.org/wiki/Kalman_filter

[31] Gillespie, T.D., *Fundamentals of vehicle dynamics*, 1992, Society of Automotive Engineers, Inc., Warrendale, PA, ISBN 1-56091-199-9

[32] Feynman, R.P., Leighton, R.B., Sands, M.L., *The Feynman lectures on Physics*, 1963, Addison-Wesley, Reading, MA, ISBN 0-201-02010-6

[33] List of moments of inertia - Wikipedia, http://en.wikipedia.org/wiki/List_of_moments_of_inertia

[34] Blender.org, http://blender.org/cms/Home.2.0.html

[35] Grond, S., *State estimation of colliding objects in a robot soccer environment*, Master's thesis, Twente University, 2005

[36] This formula was sent to us by the manufacturer of the Austro-bots. The exact formula sent contains a division by 8192, because the measured velocities have a range from -32767 to 32768 as opposed to the -4.0 to 4.0 in m/s.

[37] Faulhaber IE2-512 data sheet, http://www.faulhaber-group.com/uploadpk/d_IE2512_DFF.pdf

[38] Motors for mobile robots - A gentle and quick introduction, http://courses.csail.mit.edu/6.141/spring2006/pub/lectures/Actuation-Lecture.pdf

[39] Roby-Speed technische Details, http://www.roboterfussball.at/robygo/tech-deu.html

[40] Multivariate normal distribution - Wikipedia,

http://en.wikipedia.org/wiki/Multivariate_normal_distribution

[41] Matsumoto, M., Nishimura, T., *Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator* ,1998, ACM Trans. on Modeling and Computer Simulation, 8, No.1, January, p. 3-30

[42] Box-Muller transform - Wikipedia, http://en.wikipedia.org/wiki/Box-Muller_transform

[43] Open Dynamics Engine, http://www.ode.org

[44] Technische details Faulhaber 2224 006 SR DC-Kleinstmotor, http://www.faulhaber-group.com/uploadpk/d_2224SR_DFF.pdf

APPENDIX

A Network packet format

This appendix describes the format of the networking packets that are currently used in the communication between simulator and robot soccer system, but could also be used in a distributed set up of the system. A packet always has the following format:

Field	*Size*	*Description*
Signature	1 byte	Always 0x20. Can be used to check if this is indeed the beginning of a network packet.
Version	1 byte	The version number of the protocol. Currently 0x01.
Checksum	1 byte	Checksum of all bytes following this, using a CRC-8 checksum with polynomial $x^8+x^2+x^1+x^0$.
Type	1 byte	The type of packet this is. The following packet types are defined: 0x01 Hello 0x02 Connection refused 0x03 Connection accepted 0x10 Register event. 0x11 Unregister event 0x20 Event 0x30 Setting changed Currently the only type 0x20 is used. A real implementation of networking might change this.
Data length	2 bytes	The number of bytes of data following this header.

As shown in the table, currently only the event packet (0x20) is used for network communication. The payload of this packet has the following format:

Field	*Size*	*Description*
Event type ID	4 bytes	The hash code of the event type (see paragraph 2.6.2).
Data	-	Serialized data of the event. This part of the packet is not more and not less than the bytes stream generated by `event.serialize()`.

The event type ID can be used to create the right type of event from the network packet. The `EventManager` maintains a list that matches ID's to event types. This list is set up during the start up of the program. The currently used event type ID's are:

`0x20e20519`	NewSnapshotEvent
`0x2662056b`	NewWorldDataEvent
`0x0fab037c`	ShutdownEvent
`0x17390437`	VelocitiesEvent
`0x0ed8035c`	IdentifyEvent
`0x259d0561`	IdentifyDoneEvent
`0x16d00429`	SetActionEvent
`0x26f5058f`	VisionWindowEvent
`0x40d2071e`	VisionWindowClosedEvent
`0x398806aa`	VisionWindowClickedEvent
`0x25540548`	ColorChangedEvent
`0x207904f5`	ColorReloadEvent
`0x120f03a7`	CalibrateEvent
`0x24ba0550`	CalibrateResultEvent
`0x3dfd06df`	GameStateChangedEvent
`0x314a0639`	PossibleCollisionEvent
`0x25f7056f`	NewPlanLinesEvent
`0x31330627`	ActionFinishedEvent
`0x24f20543`	ActionChangeEvent
`0x0b7502e7`	IdSwapEvent
`0x20cd0513`	ModuleStartedEvent
`0x1be4049e`	RobotSleepEvent
`0x20f50512`	RobotWakeupEvent
`0x0667022a`	ErrorEvent
`0x1f6f04d1`	ClippingChangedEvent
`0x1b0b047e`	ClippingReloadEvent
`0x0bc902e8`	SetPIDEvent
`0x0b8d02e8`	PIDSetEvent
`0x201904e1`	SimScoreChangedEvent

APPENDIX

B Game recording file format

In this appendix the file format that is used for the second version of the game recordings is described. The format is set up so that it can be expanded while retaining backward and forward compatibility.

Each recording file begins with a 15-byte header file with the following format:

Field	*Size*	*Description*
Signature	4 bytes	Always 'MI20'. In hexadecimal format: 4D 49 32 30
File version	1 byte	Version of the recording file format. Currently 0x03.
Field size	1 byte	The size of the field. Can be one of: 0x05 A 5x5 field (220 x 180 cm) 0x07 A 7x7 field (280 x 220 cm) 0x11 A 11x11 field (400 x 280 cm)
Team colour	1 byte	The team colour on the system that recorded this file. 0x01 Blue 0x02 Yellow
Start time	4 bytes	The number of seconds since 1 January 1970, 0:00:00 UTC. This can be found using the `time()`-function
File length	4 bytes	Floating point number, representing the number of seconds of data in this recording.

After this header follow chunks of data. A data chunk always has the following format:

Field	*Size*	*Description*
Chunk type	1 byte	The type of chunk. The following chunk types are used: W (0x57) World data S (0x53) Snapshot C (0x43) Comment G (0x47) Game state change V (0x56) Robot commands
Chunk length	2 bytes	Number of bytes of the chunk data. If the chunk type is not recognized, this value can be used to skip the data and proceed to the next chunk.
Chunk data	n bytes	The data of the chunk.

More chunks can be added to the format without breaking compatibility with old recording files. These augmented files can even be read by older version of the program, since it knows the length of the chunk and can simply skip it.
The chunk data for a Comment (C) chunk is as follows:

Field	*Size*	*Description*
Comment type	1 byte	The type of comment. The following comment types are used: B (0x42) Blue team name Y (0x59) Yellow team name D (0x44) Game description
Length	2 bytes	Number of bytes of the comment.
Comment	n bytes	The comment itself. (No zero-termination used)

The robot command chunk (V) has the following format for the chunk data:

Field	*Size*	*Description*
Time stamp	4 bytes (`float`)	The time stamp (number of seconds since program start) when these commands were generated.
Robot ID	1 byte	The ID of the robot this command is for
Linear speed	4 bytes	The linear velocity of the robot command in mm/s.
Angular speed	4 bytes (`float`)	The angular velocity of the robot command in rad/s. Negative velocity is clockwise movement.

The game state chunk (G) has the following format:

Field	*Size*	*Description*
Time stamp	4 bytes (`float`)	The time stamp (number of seconds since program start) when the game state was changed.
Game state	4 bytes	The game state as serialized by the method `GameState::operator<<`.

The game state format is:

Field	*Size*	*Description*
Running	1 byte	True (0x01) if the game is running, false (0x00) otherwise.
Situation	1 byte	The current game situation: 0x00 Normal play 0x01 Kick off 0x02 Free kick 0x03 Goal kick 0x04 Penalty kick 0x05 Free ball
Playing direction	1 byte	The direction were are playing in: 0x00 Left to right 0x01 Right to left
Location	1 byte	The location of a game situation. Can be: 0x00 We are the attacking team (In free ball situations, we attack at the position with the lowest Y-coordinate) 0x01 We are the defending team (In free ball situations, we defend at the position with the lowest Y-coordinate) 0x02 Free ball: attack at highest Y-position 0x03 Free ball: defend at highest Y-position

The data of the world data chunks (W) and the snapshot chunks (S) are the exact data as the serialization methods `WorldData::operator<<` and `Snapshot::operator<<` returned. The world data format is found in the following table:

Field	*Size*	*Description*
Time stamp	4 bytes	Time stamp of the world data. Is the number of seconds since program start, as `float`.

Field	*Size*	*Description*
# robots	1 byte	The number of robots per team.
Ball state	29 bytes	The state of the ball. See next table for the format of world objects.
Team robots	N * 29 bytes	The state of the team robots, in order. See the next table.
Opponent robots	N * 29 bytes	The state of the opponent robots, in order. See the next table.

World data objects are stored as:

Field	*Size*	*Description*
On field	1 byte	Boolean describing if an object was on the field (0x01) or not (0x00).
Location X	4 bytes	The X-coordinate of the object's location, as `float`.
Location Y	4 bytes	The Y-coordinate of the object's location, as `float`.
Orientation	4 bytes	The orientation of the object, as `float`. Value lies between π and -π.
Velocity X	4 bytes	The X-component of the object's velocity vector, as `float`.
Velocity Y	4 bytes	The Y-component of the object's velocity vector, as `float`.
Angular velocity	4 bytes	The angular velocity of the object in rad/s.
Acceleration	4 bytes	The acceleration of the object in mm/s, as `float`.

Snapshots are stored using the following format:

Field	*Size*	*Description*
Time stamp	4 bytes	Time stamp of the snapshot. Is the number of seconds since program start, as `float`.
# balls	1 byte	The number of balls in the snapshot.
Ball location	N * 16 bytes	The location of the possible balls. Stored as two consecutive doubles, representing the X- and Y-location respectively.
# team robots	1 byte	The number of team robots in the snapshot.
Team robots	N * 25 bytes	The team robots in the snapshot. See the next table.
# opponent	1 byte	The number of opponent robots in the snapshot.

Field	*Size*	*Description*
robots		
Opponent robots	N * 25 bytes	The opponent robots in the snapshot. See the next table.

The robots in the snapshots are stored using the following format:

Field	*Size*	*Description*
Location X	8 bytes	The X-coordinate of the robot location, as `double`.
Location Y	8 bytes	The Y-coordinate of the robot location, as `double`.
Orientation	8 bytes	The orientation of the robot, as `double`. Value lies between π and -π.
Robot ID	1 byte	The ID of the robot that was detected if the multiple-colour patches were used. A value of 0xFF means the identify could not be determined.

APPENDIX

C Simulation constants

This appendix lists some of the constants that were used in the physics simulation, including the motor model. Source of most of these values are either the manufacturer of the hardware [39][44].

Robot properties

Length (s_x)	75	mm
Width (s_y)	75	mm
Height (s_z)	48	mm
Mass (m)	608.7	g
Wheel base (ℓ)	68	mm
Wheel radius (r_{wheel})	22.5	mm

Motor constants

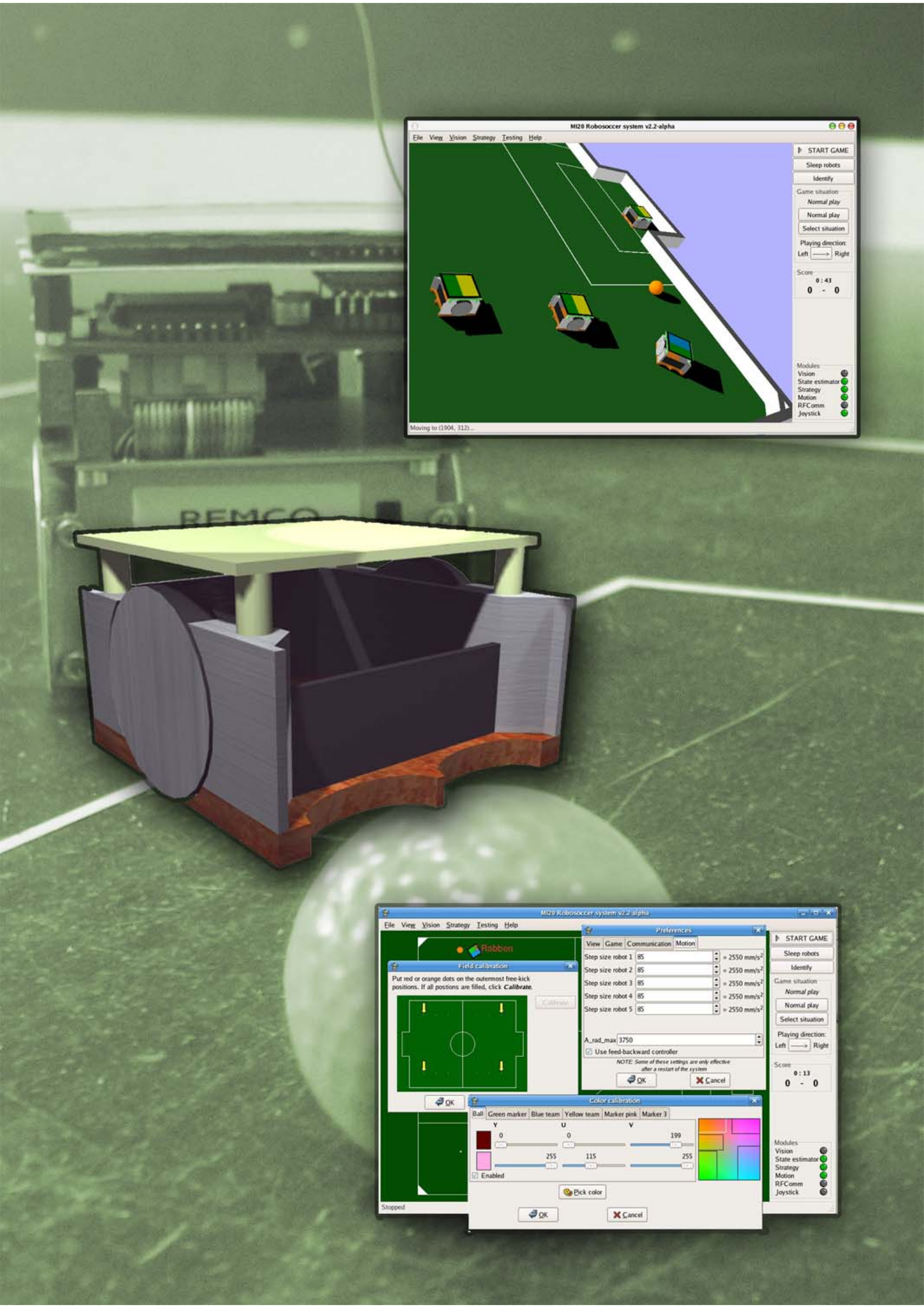
Electrical voltage (V)	6	V
Electrical resistance (R)	1.94	Ω
Stall torque (τ_S)	21.2	mNm
Minimal torque (τ_F)	0.2	mNm
EMF constant (ke)	0.725	mV/rpm
Torque constant (k_τ)	6.92	mNm/A
Motor effeciency (η)	82	%
Gear ratio (N)	25: 3	

Ball properties

Ball radius (r_{ball})	21.35	mm
Ball mass (m_{ball})	45.9	g

Friction coefficients

Ball rolling resistance ($\mu_{rr,ball}$)	$4.731 \cdot 10^{-3}$	
Robot rolling resistance ($\mu_{rr,robot}$)	$1.71 \cdot 10^{-2}$	
Robot sliding friction ($\mu_{sliding}$)	$4.808 \cdot 10^{-1}$	
Robot friction torque (τ_f)	16.06	mNm


MI2O Robosoccer system v2.2-alpha
File View Vision Strategy Testing Help
START GAME
Sleep robots
Identify
Game situation
Normal play
Normal play
Select situation
Playing direction:
Left Right
Score
0 : 43
0 - 0
Modules
Vision
State estimator
Strategy
Motion
RFComm
Joystick
Moving to (1904, 312)...
MI2O Robosoccer system v2.2-alpha
File View Vision Strategy Testing Help
Robben
Field calibration
Put red or orange dots on the outermost free-kick positions. If all postions are filled, click Calibrate.
Calibrate
OK
Preferences
View Game Communication Motion
Step size robot 1 85 = 2550 mm/s²
Step size robot 2 85 = 2550 mm/s²
Step size robot 3 85 = 2550 mm/s²
Step size robot 4 85 = 2550 mm/s²
Step size robot 5 85 = 2550 mm/s²
A_rad_max 3750
Use feed-backward controller
NOTE: Some of these settings are only effective after a restart of the system
OK
Cancel
Color calibration
Ball Green marker Blue team Yellow team Marker pink Marker 3
Y U V
0 0 199
255 115 255
Enabled
Pick color
OK
Cancel
START GAME
Sleep robots
Identify
Game situation
Normal play
Normal play
Select situation
Playing direction:
Left Right
Score
0 : 13
0 - 0
Modules
Vision
State estimator
Strategy
Motion
RFComm
Joystick
Stopped