Improving self-localisation and behaviour for Aibo's soccer-playing robots (Implementation according to the new RoboCup rules of 2005)

Woiyl Hammoumi, Vladimir Nedovic, Bayu Slamet and Roberto Valenti

Intelligent Autonomous Systems Group, Informatics Institute
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

{whammoum,vnedovic,baslamet,rvalenti}@science.uva.nl

February 4th, 2005

Abstract

Every year the RoboCup organization [1], which organizes the annual robot world championships in soccer, changes the rules of the various leagues involved to work towards increasingly natural soccer playing. In the four-legged league, which is played with Aibo robots, some new rules have been introduced as well [2], [3]. Among others, the white wooden borders that so far surrounded the field will be removed, the coloured marker poles that stand on the field boundaries will be moved towards the centre of the field, and the field size will be larger.

To deal with these changes, the Aibos' software needs to be updated. The potential problems concern mostly localisation and behaviour control, while perception is not affected significantly. In this project, we set out to solve the issues that relate to the changing field specifications; we have programmed the new field specifications within the C++ layer and we have scaled-up the behaviour for the new field size.

Unfortunately there appear to be many more dependencies towards the field dimensions than we have been able to find in our time window. We have delivered the improved behaviour specifications and our modifications to the C++ code. In addition to this, we pointed out where additional changes need to be made before improved performance can be observed.

We suggest a central field dimensions module to be implemented instead of the hard-coded values, in order to be able to deal with changing dimensions more easily in the future.

We also found out that there is an undocumented light-source detector which can be exploited to further improve localization in the future by making use of the lamps that are usually at the ceiling above the soccer field.

1. Introduction

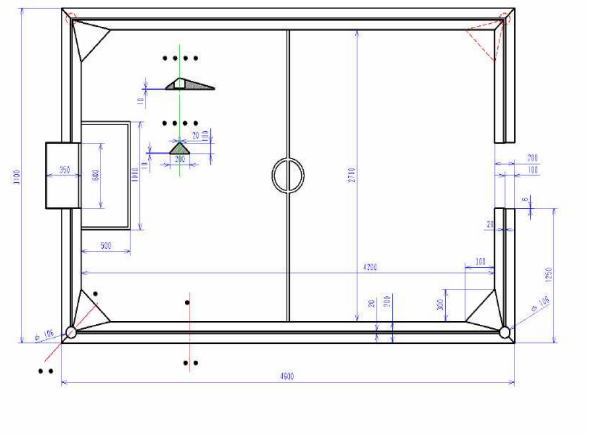
Since 2004, the Dutch Aibo Team, which is a joint effort of various Dutch universities, participates in the four-legged league of the international RoboSoccer competition [1]. During the year, various tournaments are held all over the world with the main annual event being the international RoboCup world championships organised by the RoboCup organisation. The four-legged league is played with teams of four where each player is one of the well known Sony Aibo robot dogs. These robots have a programming interface and various national teams that compete at the RoboCup matches win or lose with the quality of their software.

The Dutch Aibo Team

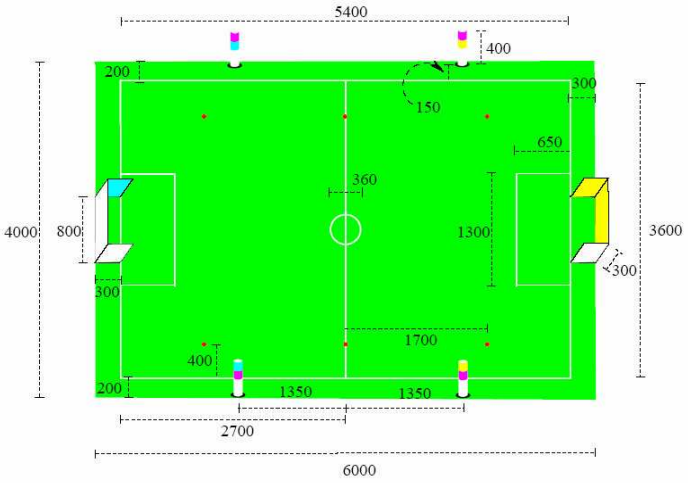
Each year, after the world championships, the participating teams have to publish their software. This way the teams can learn from each other and innovations can be based on the best implementation available. This serves the ultimate purpose of the RoboCup organisation which is to improve on the various fields of science that correlate to intelligent autonomous systems, so that by 2050 a team of robots can be developed that can play a reasonable soccer match against a human team.

1.1 New rules for the RoboCup 2005

The RoboCup organisation poses its participants with additional challenges every year. In doing so, they work from highly constrained environmental conditions towards an increasingly 'natural' situation. The robots' software should evolve accordingly so that we will end up with human-level soccer capabilities in 2050.



Rules 2004, the field has a size of 2.7 x 4.2 meter surrounded with low triangular walls.



Rules 2005, a field of 3.6 x 5.4 meter surrounded with a small green border

Figure 1: Sony four-legged league fields with old and new dimensions and flag positions

This year, in the four-legged league, the white wooden borders that so far surrounded the field will be removed from the sidelines and the coloured poles that served as markers in the corners of the playing field will move closer to the centre [2], [3] (see Figure 1). While in the past the dogs and the ball were unable to leave the field because of the boundaries, they now have to take this new variable into account. The dogs will be punished with a 30 seconds time-out when they cross the outer boundaries of the playing field and the ball will be brought back into the game at one of the predefined locations when it leaves the inner boundaries of the field. In addition to this, the dimensions of the playing field will be increased.

These changes pose various new challenges for the Aibo software which have to be dealt with before the next RoboCup in July 2005. It is obvious that the dogs need to be aware of the fact that they should not leave the field and that they should also show different behaviour when the ball leaves it. On top of this they also have to be programmed to take the new dimensions of the field into account.

The tougher problem that follows from these new challenges is an internal one. While playing soccer, Aibo dogs continuously estimate their location based on current observations and some knowledge of past locations and movements [4], [5]. Localisation algorithms currently rely significantly on the detection of white borders and marker poles. The removal of these will cause serious performance losses on the current localisation algorithm. As the localisation software is fundamental to almost all behaviours, this certainly is to become the weakest link. Improvements on localisation will have positive effects on most behaviours.

Another side-effect of removing the walls is that the robots are no longer concealed from distracting stimuli from outside the playing field. The robot could now by mistake classify certain clothing of supporters or other distractions as objects relevant to the soccer game.

1.2 Problem statement

In short, in order to compete at a desired level in 2005, the following problems, presented here in the prioritised order, have to be dealt with adequately:

1. localisation is for a big part based on the detection of the white borders which are no longer in place
2. the robots are hard-programmed against old (wrong) field dimensions
3. the robots expect the marker poles to be at different positions
4. the robots are unaware of the fact that they will be punished if they get outside of the boundaries of the playing field
5. the robots are unaware of the fact that when a ball leaves the playing field it will be brought back into the game by the referee at the nearest predefined location
6. the robots are now confronted with a larger amount of distracting sensor input from outside the playing field

2. Current implementation

In 2004, the Dutch Aibo Team used the German software from 2003 (GT2003) as a basis for their software development [4], [8], [9]. In retrospect, this was a good choice, as the Germans won in 2004. This year the Dutch team wants to merge the German software of 2004 (GT2004) with some aspects of their own software of 2004 (DT2004) and some improvements that are issued by Floris Mantz from the Technical University Delft [5] (see Figure 2).

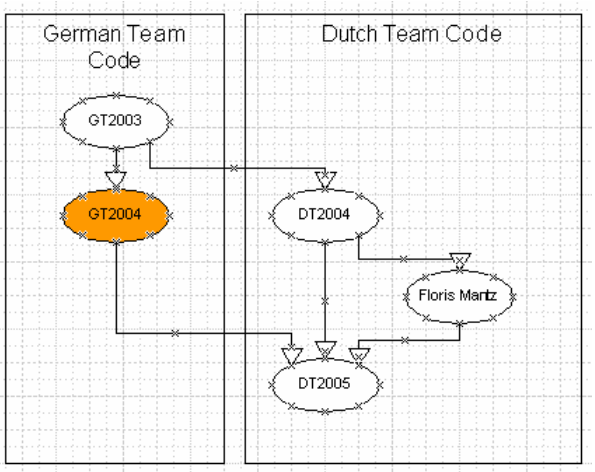


Figure 2: *The 'evolution' of the software based on German GT2003 code*

The Masters thesis by Floris Mantz [5] proposes a fundamentally different approach to the robots' vision system, which would be behaviour-based. Currently the Aibo is equipped with a single colour table which is used in every image processing task. Floris claims that making use of multiple colour tables, which are object and behaviour-dependent, can dramatically improve the robustness of the Aibo's vision system, and thus lead to a significantly better overall performance.

2.1 Code architecture

The architecture of the German software, which was the basis for DT2004, is very complex [4] (also see Figure 3). The core of the software is written C++ which is compiled to run on multiple platforms. The code is neatly structured in a very modular architecture. Many key parameters can be set using external configuration files. The behaviour control module makes use of an XML-based configuration system called XABSL (eXtensible Agent Behaviour Specification Language).

At the perception level all internal sensor data of the Aibo is processed. The direction of the camera is determined using the information on the current states of the joints, and the image that was provided by the camera is searched for objects that are known to exist in that direction. The detection of the field, field lines, goals, other players and so on is used to update the world model, which is shared among all robots during the game and maintains the integrated world-state information that surpasses the currently visible part of it.

The world model not only models the stationary objects like the goals and the marker poles, but also the dynamic objects like other players and the ball. The result is an estimated world state.

The behaviour control bases its reasoning on the world state. Depending on the role of the robot, its current position, perceptions, and the state of the game, the behaviour-control module decides which actions the robot has to take. The selected motions are then sent as requests to the motion level.

The motion control level translates requests issued by the behaviour level into the actual joint parameters and head movements that correspond to the requested action. A dedicated kick-selection module is used to perform all the possible kicks the Aibo robot can execute.

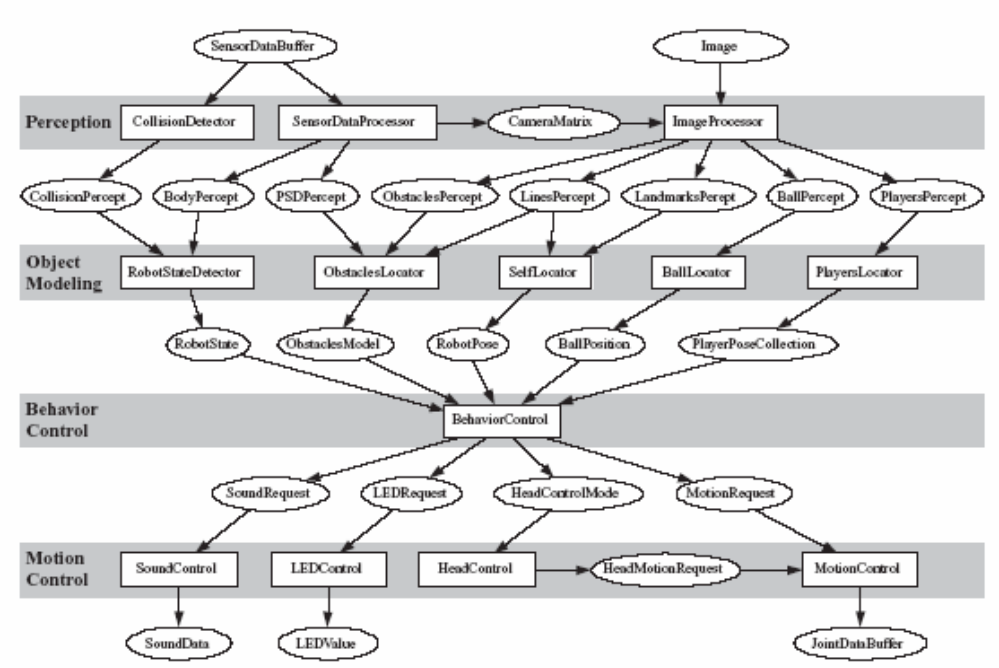


Figure 3: *Overview of the German code modules*

2.2 RobotControl

RobotControl is a full-fledged debugging interface to the Aibo software [4]. It was developed together with the Aibo software and its main purpose is to increase the speed and comfort of the software development process. In *RobotControl*, using wireless LAN, it is possible to visualise almost all internal representations of the Aibo online: images, joint values, sensor data, the world state and so on. In addition to this, the software that runs on the Aibo is also linked into *RobotControl*. Using this, *RobotControl* can simulate the physical Aibo offline.

It is also possible to use *RobotControl* to control or manipulate the internal representations of the Aibo. This allows the testing of every particular module independently. For example, the parameters for certain motion requests that are normally issued by the behaviour control can be set to test the motion control or kick selection module independently.

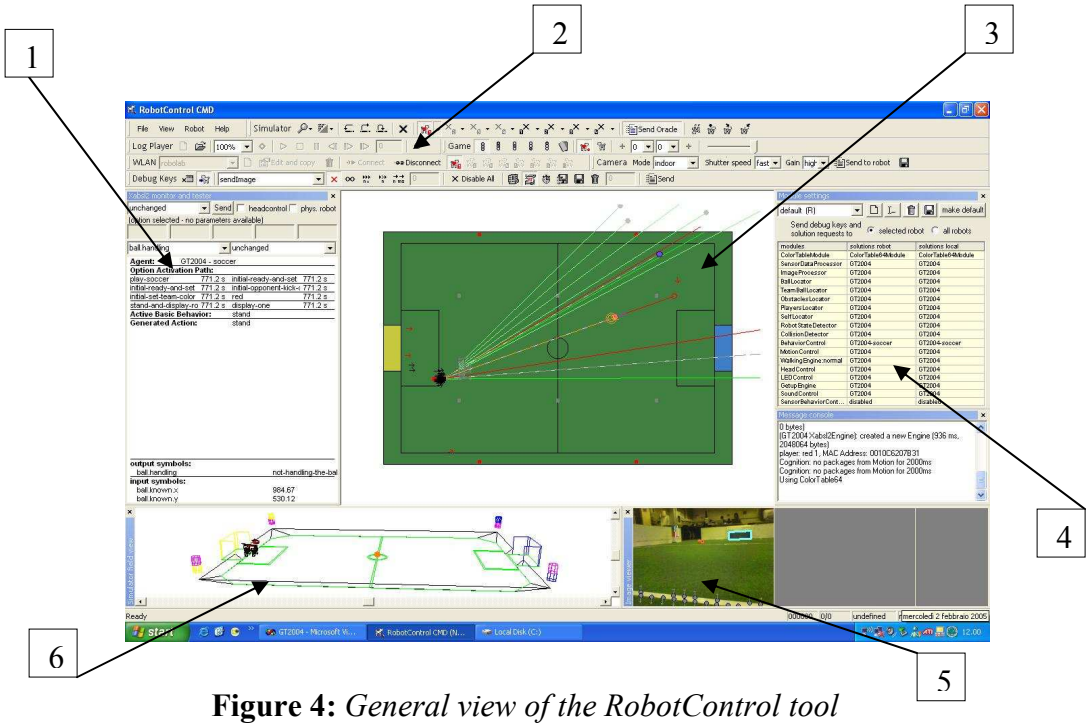


Figure 4: *General view of the RobotControl tool*

1. The XABSL behaviour tester dialog - the debugging interface to the behaviour modules (gives an overview of the internal states of the engine)
2. The toolbars providing different helper tools for colour calibration, for copying data to the memory stick, and for wireless network configuration …
3. The new robot control's field view and radar viewer - draws the percepts based on robot's localization on the field, and the world state.
4. The settings dialog - permits switching between the solutions of modules running on the robot or on the simulator.
5. Image viewer - displays images and debug drawings from the robot.
6. The simulator - offers a lot of possibilities to simulate and debug algorithms.

2.3 Localization

The Aibo is equipped with a *Self-Locator,* which implements a Markov-localization method employing the so-called Monte-Carlo approach [10], [14], [4]. In this approach, the current position of the robot on the field is modelled as the density of a set of particles. Each particle represents a possible position of the robot on the field. Therefore a particle mainly consists of a vector representing the hypothetical x and y coordinates of the robot in millimetres and its rotation in radians.

The localization technique first positions all particles based on the motion model of the previous action of the robot. Then it computes the probability of each particle based on the current perceptions of the robot. The particles are resampled towards the locations with higher probabilities and then the average of the resampled probability distribution is taken as the estimate for current pose of the Aibo.

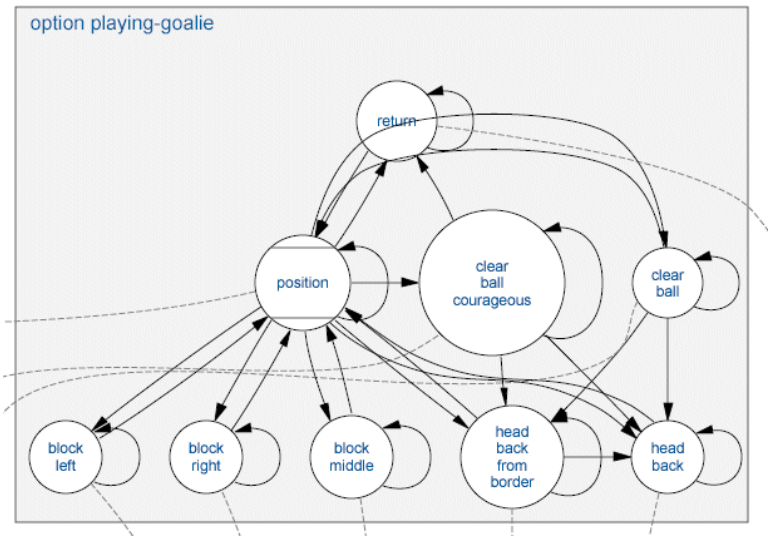
2.4 Behavior Control

The Behavior Control module is responsible for making decisions based on the world state, the current game state and the behaviour that is executed currently by the Aibo [4], [9]. The Behaviour control outputs the following to the motion control layer:

- a motion request that specifies the next motion of the robot
- a head motion request that specifies the mode how the robot's head is moved,
- a LED request that sets the states of the LEDs,
- a sound request that selects a sound file to be played by the robot's loudspeaker,
- a behaviour team message that is sent to other players by wireless communication.

In the German Team code Aibo's behaviour is specified using a very sophisticated XML-based system called XABSL, the eXtensible Agent Behaviour Specification Language [4]. At the basis of the XABSL definition, the C++ layer provides several so-called basic symbols and basic behaviours. The basic symbols represent run-time values which are provided by the C++ layer during execution on the Aibo. The basic behaviours define units of movement which can be executed by the Aibo. These basic symbols and behaviours are also written to XABSL compliant files with each code compilation. Following the XABSL specification, it is then possible to take these basic symbol definitions as a starting point to constructing increasingly complex behaviour patterns. In fact it is possible to use XABSL to define finite state machines where in each state a basic behaviour is executed. The reasoning, by which the Aibo chooses among possible successive states, can be based on the provided symbols. Many aspects of soccer playing are put in XABSL symbols, and thus regenerated with each software compilation.

On the right, a visualisation of the finite state machine that specifies the behaviour of the goalie can be seen. All transitions from one state to another are regulated by decision trees.



3. Our project

The problem statement mentioned in chapter 1 can be translated into a mission statement quite straightforwardly. By doing so, we formulated an overall mission statement for the Dutch Aibo Team. However, in consultation with our supervisor and some experienced members of the Dutch Aibo Team, we have defined a more specific focus of our project [6], [7]. At the University of Utrecht a larger project will start in February 2005. To be of most help to this team, we focused on solving one problem adequately (namely change the dependencies on the field dimensions and solve implied problems in self-localisation and motion, i.e. parts of item 1 listed below) instead of getting involved with multiple issues, possibly without producing satisfactory results in any of them.

3.1 Overall mission statement

From the problem statement we derived the following mission statement. We ordered the problems by their importance:

1. improve localisation [4], [9]
   - make localisation independent of the white borders
   - find and implement alternative means to improve localisation, perhaps make use of colour detection improvements that are suggested by Floris Mantz [5]
   - take changed field specifications into account
     - ◊ preferably make field specifications parameterised properties of the Aibo behaviour
     - ◊ otherwise hard-code new specifications
2. improve behaviour [4], [9]
   - make the robot aware of the fact that they should stay inside the field boundaries
   - make the robot aware of the concept of an 'out-ball'
     - ◊ they should show improved behaviour on 'out-ball' situations
     - ◊ the possible locations for the ball to be brought back into play should be parameterised
3. make the robot ignore irrelevant input stimuli from outside the playing field

3.2 Our project goals

A great help in analysing the listed problems was the fact that the German Team code of 2004 is very well-structured and extensively documented [4]. This enabled us to quickly get a reasonable impression of how the problems relate to the various software modules in C++. Using the insight that we acquired from the documentation and from a closer look at the code, we were able to evaluate the problems in terms of how much degradation in overall performance they would cause, whether it would be feasible to solve the problem in a one-month time-frame and whether solving one problem could be done without first solving some other one.

We decided to focus our project entirely on changing the field specifications and on updating their dependencies. We envisioned that a larger field and repositioned marker poles

would have a great impact on localisation performance [7]. As localisation is fundamental to almost all of the Aibo's behaviour, the performance degradation due to inaccurate localisation propagates through most of the exposed behaviour. Because of this dependency of behaviour on localisation it is also rather difficult to get an accurate analysis of the behaviour-related problems and it is possible to end up designing solutions in terms of behaviour patterns while in fact that is only compensating for the uncertainty in localisation. Reasoning along this line led us to the logical decision that we should to try to design and implement a solution to localization first in which we make the Aibo aware of the new field specifications.

3.3 Approach

We expected there to be many modules in the C++ code that make use of information about the field dimensions. At the perception level, we have among others the *SelfLocator,* which makes use of the field specifications in the Monte Carlo algorithm implementation. It is clear that this algorithm cannot generate sensible hypothetical robot positions in its particle distribution when it is supplied with the wrong field size. Information about the field dimensions and marker pole positions is also necessary to be able to relate perceptions to a correct world model.

The field dimensions are also used by the Aibos for Behavior Control. Many decisions about which behaviour to execute next are based on specific distances, e.g. the distance to the own goal or that to the centre line.

We decided to approach the problem from multiple directions simultaneously; we used a bottom-up approach in parallel with a top-down approach [7]. Besides the fact that this looked like the most efficient method, we also chose it for some additional benefits. As we were inexperienced in this field of applications, we were likely to come across several practical difficulties; by approaching the problem from multiple directions, we aimed at detecting these issues as soon as possible so that we could build up the relevant experience and deal with them early in the process. Another pursued positive side-effect of executing multiple approaches in parallel was that we would acquire knowledge both on a specific and on a more conceptual level of the different software modules and the various concepts involved.

3.3.1 Bottom-up approach

The bottom-up approach implied that we should first pin-point the hard-coded field specifications in the code; then we would track all dependencies we could find and make the necessary changes. We expected that there would also be dependencies with regards to field dimensions and marker positions, which we could not find directly by searching through the code and reading the documentation. We assumed that we would either see a significant improvement by just modifying the visible dependencies or that the invisible dependencies would become paramount along the way as we got more insight and experience. In order to identify at least some of these invisible dependencies and reduce the problems with the latter scenario, we were in contact with members of both the Dutch Team in Utrecht and the German Team in Bremen; their valuable information helped us with making the changes in the code, as well as debugging the resulting configuration.

3.3.2 Top-down approach

The top-down approach started at the conceptual level, which is the high-level behaviour exposed by Aibo robots. All behaviour is defined in XABSL files. At the basis, the C++ layer provides the basic symbols and behaviours. Many aspects of soccer playing are put in XABSL symbols, and thus regenerated with each software compilation. Therefore, the symbols will always include the latest software changes. Unfortunately, the field dimensions were not represented by XABSL symbols; the values were instead hard-coded in the decision trees that are used in the behaviour specifications. So we would have to scrutinize every rule that is defined in XABSL and made all appropriate changes.

4. Implementation

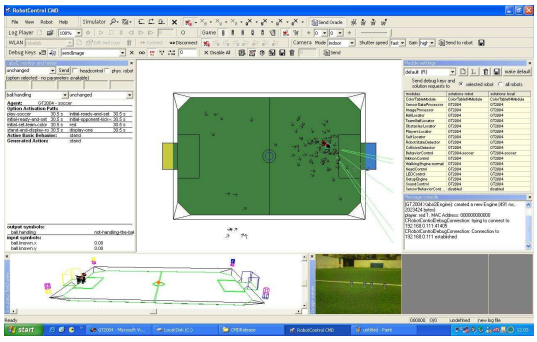
Our team was divided into two teams, each following one of the approaches. The team that followed the bottom-up approach used the low-level C++ implementation as a basis for their modifications. The other team took the XABSL definition files which are used by the behaviour control module as a starting point for the top-down approach.

4.1 Changes in C++

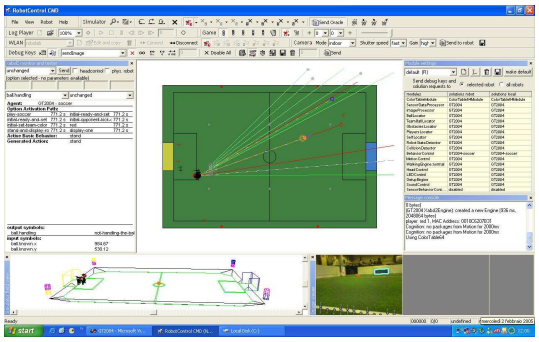
We determined the main location in the software code where the field specifications are defined: the *FieldDimensions* class [7]. Every documented or otherwise determined dependency towards this was followed and the necessary modifications were implemented. It appeared as if most modules in the perception layer that need field dimensions for their algorithms make use of this class.

However, we cannot know for sure that all dependencies on field dimensions were solved while executing this procedure. Even though the German code is well-structured and well-documented, we found it surprising that there existed several instances of files where robots' localisation or motion had hard-coded field-dependencies. Since it was not always easy to know what the actual numbers were supposed to represent (i.e. whether the developers desired a certain distance from the goal, the halfway line, the border lines, or something else), a logical change in the code would be to parameterise all these distances and store them all in some header or library file (just as in the case of *FieldDimensions.h* on which many modules depended, there could be another file, say *FieldDistances.h* from which specific player positions could be read). However, given our time constraints, we decided not to make these changes, but instead to update the hard-coded values wherever we encounter them.

In addition, the field-view in *RobotControl* was updated to reflect the changed field size, moved marker poles and removed borders. In the figure below a difference between the old and the new field-view can be clearly seen; in the new field-view, Monte Carlo localization makes use of the new dimensions.



Old view: a field with a size of 2.7 x 4.2 metres, flags in the corners and wooden borders.



New view: a field of 3.6 x 5.4 metres, poles moved to the centre and no wooden borders.

Figure 6: The old and the new field view in RobotControl.

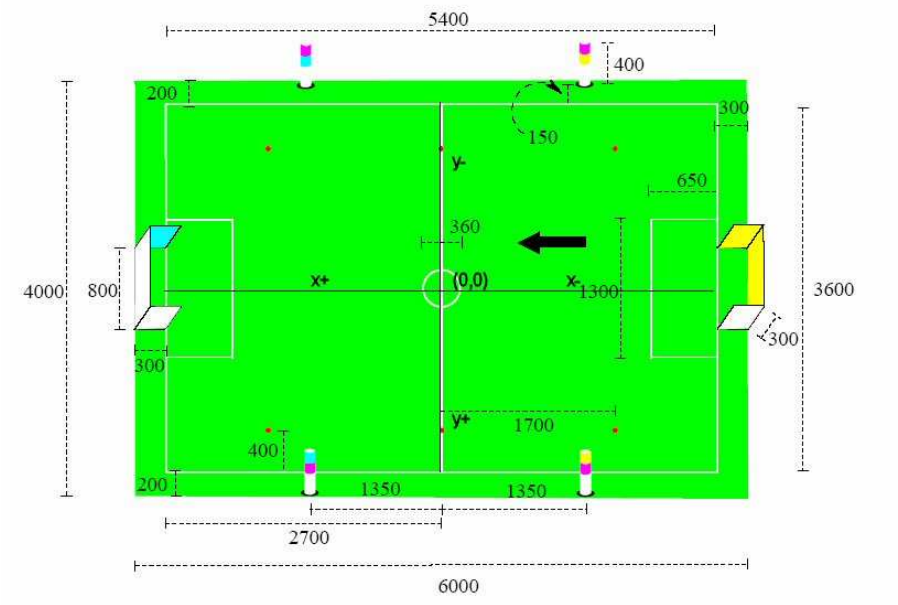
4.2 Changes in XABSL

One part of the software that clearly had the field dimensions programmed in without using the *FieldDimensions* class (directly or indirectly) was the behaviour-control module. Throughout the XABSL definitions, hard-programmed field positions are used in decision trees that control the selection of subsequent states in the finite state machines. We checked every rule in the XABSL and made all appropriate changes. Using the documentation of the old behaviour and using common sense we were able to 'scale up' the behaviour to the now larger field.

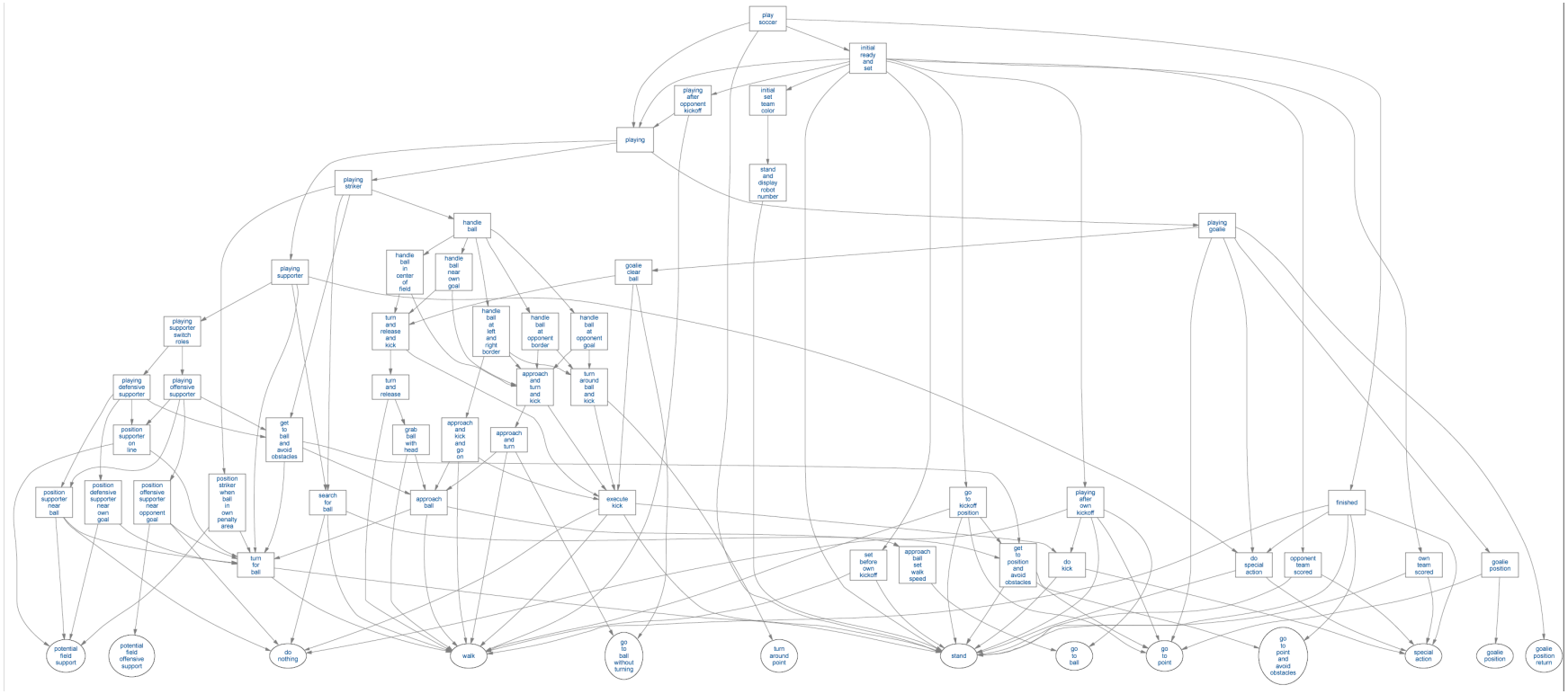
```
<state name="position">
  <subsequent-option ref="goalie-position"/>
  <set-output-symbol ref="head-control-mode" value="head-control-mode.s
  <decision-tree>
    <if>
      <condition description="ball was not seen for 3 seconds but in last 3
        <and>
          <greater-than>
            <decimal-input-symbol-ref ref="ball.time-since-last-seen"/>
            <decimal-value value="3000"/>
          </greater-than>
          <less-than>
            <decimal-input-symbol-ref ref="ball.time-since-last-seen"/>
            <decimal-value value="3500"/>
          </less-than>
          <greater-than>
            <decimal-input-function-call ref="abs">
              <with-parameter ref="abs.value">
                <decimal-input-symbol-ref ref="robot-pose.y"/>
              </with-parameter>
            </decimal-input-function-call>
            <decimal-value value="400"/>
          </greater-than>
        </and>
      </condition>
```

On the right an excerpt from a decision tree defined in XABSL is shown; it makes use of specific values that correspond to certain distances on the playing field. For example, *robot-pose.y* value of 400 refers to a distance of 400 millimetres from the halfway line going through the centre. In another figure below, the coordinate system that is defined on the field is shown (for the player attacking the blue goal on the right):

- the origin is at the centre of the field
- x coordinate is in the direction of line of sight, with positive values on the opponent side, and negative ones on the own side
- y is in the direction orthogonal to the line of sight, with positive values indicating left of the centre and negative ones-right of the centre.



On the following page, a complete option-hierarchy for Aibo's behaviour is displayed. The hierarchy of square boxes start with the *play-soccer* behaviour, while the basic behaviours provided by the C++ software are shown at the bottom and indicated with ellipses. The arrows indicate the lower-level behaviour that may be executed when the higher-level behaviour gets activated. Every displayed option internally has a finite state machine with one or more internal states. Depending on the current state, lower-level behaviours are activated until the activation chain ends at the bottom with the execution of a basic behaviour. It is in these finite state machines where the decision trees reside and where we have made all the necessary changes.


play soccer
initial ready and set
playing after opponent kickoff
initial set team color
playing
stand and display robot number
playing striker
handle ball
playing goalie
handle ball in center of field
handle ball near own goal
playing supporter
goalie clear ball
turn and release and kick
handle ball at left and right border
handle ball at opponent border
handle ball at opponent goal
playing supporter switch roles
approach and turn and kick
turn around ball and kick
turn and release
playing defensive supporter
playing offensive supporter
get to ball and avoid obstacles
grab ball with head
approach and kick and go on
approach and turn
position supporter on line
position striker when ball in own penalty area
search for ball
approach ball
execute kick
go to kickoff position
playing after own kickoff
finished
position supporter near ball
position defensive supporter near own goal
position offensive supporter near opponent goal
turn for ball
set before own kickoff
approach ball set walk speed
get to position and avoid obstacles
do kick
do special action
opponent team scored
own team scored
goalie position
potential field support
potential field offensive support
do nothing
walk
go to ball without turning
turn around point
stand
go to ball
go to point
go to point and avoid obstacles
special action
goalie position
goalie position return

5. Results

Note: we will refer to the original code of the German Team with GT2004, while the software with our modifications included is referred to as DT2005.

All types of players (goalie, defensive supporter, offensive supporter, striker) were able to find their kick-off positions both with the GT2004 and DT2005 code. The behaviour *go-to-kick-off-position* is executed by all players during the initiation phase of the soccer game. We observed that this initial positioning is based purely on perception. This makes a lot of sense as Aibos cannot make any assumption on where they will be placed on the field.

When the game starts, the GT2004 Aibos do just fine. Despite the fact that they have the wrong field dimensions programmed in throughout the code, the code appears to be very robust and especially the players such as goalie or striker do not appear to be bothered too much with a larger field.

In case of DT2005 Aibo, we analyzed different behaviours one at a time and found that the robot has no problem finding the centre of the field or any of the goals, since these behaviours are based solely on perception. However, when asked to move to an arbitrary position in the new field (specified by exact x and y coordinates), the DT2005 robot positions itself incorrectly. This happens because the DT2005 version of the code has accurate field dimensions and corresponding distances in *FieldDimensions* and in the Behavior Control, but not in the Perception modules (since we discovered these dependencies too late in the process). Therefore the robot is able to find any of the markers, but its internal representation of the field remains wrong, based on the old field dimensions.

When the Aibo is configured with the old field-dimensions and the new behaviour specifications (using the new field dimensions) things do not get better. When we configured the Aibo with the new field dimensions using the old behaviour we also saw no improvements. So we figured that the source of the problem is not in our modified code, but somewhere in a hidden dependency towards the old field dependencies. We found that there are multiple XML files, all in the directory
\GT2004\Src\Modules\BehaviorControl\GT2004BehaviorControl\PotentialFields\Common,
which all map certain visual perceptions to positions on the field. In these files we found statements similar to 'when you perceive a part of the white line together with a marker pole you are on this position'. Some of these could refer to positions on the old field, but we could not make sense of most of the numbers.

We noticed that the goalie forces itself to stay inside the penalty area by constantly looking to the surrounding lines and the poles, while the other players do not constrain themselves and follow the ball outside of the field. The goalie's example can therefore be used to implement a desired XABSL *ball-out-of-field* behaviour linked with team ball-locator (a module for communicating the ball position among the dogs. This behaviour would then be extended to position the dogs in the direction where the ball will be placed by the referee.

A surprising fact that we observed was that when the Aibo was really lost and we commanded it to get back to the centre of the field, it starts looking upwards, at the ceiling. In *RobotControl* we could see that the Aibo makes use of the TL-lamps attached to the ceiling to position itself. Apparently the Aibo software has a light-source detection algorithm programmed in and makes use of this as a fall-back methodology to localise itself. We observed that the Aibo assumes the TL-lamps to be attached in symmetry with the field as it positions itself precisely between them; it also assumes only two lamps. In the RoboLab we had three, but the Aibo would

always position itself between the two of them. As neither the Germans nor the Dutch make any mention of this algorithm in any of their documentation, we did not know the Aibo made use of this kind of information for localization.

6. Conclusions

For the most part the Germans did a good job when developing their Aibo software. The behaviour control, which is a very complex subject when the relations between certain behaviours and low-level motion specifications are considered, was made very flexible and easy to modify using the XABSL engine. The C++ code also looks neatly structured for the biggest part and *RobotControl* is an amazing debugging interface.

However, when it comes to the field dimensions, things start to get really messy. We have changed a big number of files, checked many dependencies and read all documentation available, but we obviously haven't covered all the dependencies. We think that the problem resides somewhere at the perception level in the C++ code. We have thoroughly checked our modified behaviour specifications and we do not think that only a syntax error in those would cause unsatisfactory results when the game starts.

As the RoboCup organization is likely to change the field dimensions another few times as they work towards human-level soccer playing, the software needs to be modified to make these changes easier to implement. There already is a *FieldDimensions* class, but for some unknown reason it is not linked to XABSL. It is very well feasible to export the field dimensions as an XABSL basic symbol, which is the way many other variables have been used. In that case, the decision trees could be made more self-explanatory in the XABSL definitions, which would automatically be updated as the field dimensions are changed in the C++ code.

It is likely that the FieldDimensions class does not yet provide enough information to be of good use for the perception modules. The *FieldDimensions* class could therefore be accompanied by a *FieldDistances* class to make all necessary information easily available. In any way, it would be a great improvement if all layers, namely perception, behaviour control and motion control would tap from the same source when it comes to field dimension information.

To improve localisation, the Aibo could also make extensive use of the lamp detector, which is apparently already programmed in. Now that the white borders are removed, the lamps could be a good alternative, as they are also visible from any position and give a good indication on the orientation of the Aibo. We don't know whether it is possible to calibrate the relative position of the lamps in relation to the field; this technique was not documented and we did not have enough time to explore this issue further. We think that taking a "screenshot" of the light position in the beginning of the game could help to improve the Monte-Carlo self localization when the dogs are completely lost.

In conclusion, even if we did not accomplish all of our goals, we understood the structure of the code and pointed out where the problems are coming from. Furthermore, we discuss and propose some solutions for the future problems, building a solid base that the Dutch team can use as a starting point for further improvements and implementations.

7. References

[1] RoboCup official website, http://www.robocup.org/
[2] RoboCup Technical Committee: "Sony Four-Legged Football League Rule Book 2005". Available online: http://www.tzi.de/4legged/pub/Website/History/Rules2005.pdf
[3] RoboCup Technical Committee: "Sony Four-Legged Football League Rule Book 2004". Available online: http://www.tzi.de/4legged/pub/Website/History/Rules2004.pdf
[4] Thomas Roefer et al., "German Team – Robocup 2004", Technical Report 2004. Available online: http://www.germanteam.org/GT2004.pdf
[5] Floris Mantz, "A Robust Behaviour-Based Hierarchical Vision System with Local Colour-Tables and Use of Behaviour and Location Information", Thesis Report, Technical University Delft, January 2005.
[6] UvA's Aibo Soccer Team Project Page, http://student.science.uva.nl/~baslamet/wiki
[7] UvA's Aibo Soccer Team LabBook, http://student.science.uva.nl/~baslamet/labbook
[8] Stijn Oomes et al. "The Dutch Aibo Team 2004", http://aibo.cs.uu.nl/articles.htm
[9] Thomas Rofer et al. "German Team 2004 – The German National RoboCup Team", In *8th International Workshop on RoboCup 2004*, Lecture Notes in Artificial Intelligence, Springer-Verlag 2004.
[10] S. Oomes, P. Jonker, M. Poel, A. Visser, M. Wiering "The Dutch AIBO Team 2004", in *8th International Workshop on RoboCup 2004*, Lecture Notes on Artificial Intelligence, Springer-Verlag 2004.
[11] Thomas Rofer and Matthias Jungel, "Vision-Based Fast and Reactive Monte-Carlo Localisation", *IEEE International Conference on Robotics and Automation,* 2003.
[12] Brammert Ottens et al. "Aibo Project 2004 – German Team Report", University of Amsterdam, January 2004.
[13] Patrick de Oude et al. "Evaluation of CMPack 2003", University of Amsterdam, January 2004.
[14] Frank Dallaert et al. "Monte Carlo Localisation for Mobile Robots", *IEEE Conference on Robotics and Automation (ICRA99)*, May 1999.
[15] D. Fox et al. "Monte Carlo localization: Efficient position estimation for mobile robots", In *Proc. of the National Conference on Artificial Intelligence*, 1999.
[16] Martin Lotzsch, "XABSL - a behavior engineering system for autonomous agents - Diploma thesis", Humboldt Universitat zu Berlin, 2004. Available online: http://www.martinloetzsch.de/papers/diploma-thesis.pdf
[17] Martin Lotzsch et al. "Designing agent behavior with the extensible agent behavior specification language XABSL", In *7th InternationalWorkshop on RoboCup 2003 (Robot World Cup Soccer Games and Conferences)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2004.
[18] Sony Corporation, "OPEN-R documentation – OPEN-R Internet protocol version 4", Technical Report, 2004. Available online: http://openr.aibo.com/openr/eng/index.php4
[19] Thomas Rofer, "An architecture for a national RoboCup team", In *RoboCup 2002 Robot Soccer World Cup VI*, Lecture Notes in Artificial Intelligence, Springer-Verlag, 2003.
[20] Tim Laue and Thomas Rofer, "A Behavior Architecture for Autonomous Mobile Robots Based on Potential Fields", In *8th International Workshop on RoboCup 2004*, Lecture Notes in Artificial Intelligence, Springer-Verlag, 2005. (to appear)

Appendix A: C++ changes

A.1 FieldDimensions

First we changed FieldDimensions.h and FieldDimensions.cpp to make it reflect the new field size and marker pole positions. We also tracked and modified several dependencies.

File	Description
FieldDimensions.h	changed all hard-coded dimensions added new variables according to rules of 2005 removed variables not used anymore
FieldDimensions.cpp	updated the methods to use new dimensions or new variables
Field.cpp	replaced SideCorner and Sideline by Groundline

A.2 Behavior Control

We had to change some basic symbols in the XABSL files, so we changed their C++ counterparts which originate the XABSL basic symbols.

File	Description
GT2004PlayersLocator.cpp	replaced SideCorner and Sideline by Groundline
BallSymbols.cpp	added a template for new behaviour 'ball-out-of-field'
AngleSymbols.cpp	Sideline replaced by Groundline
LinesTables2004.cpp	removed comment from '#IFDEF SAVE'
DrawingMethods.cpp	changed method 'paintFieldPolygons' replaced SideCorner and Sideline by Groundline removed code not necessary anymore added outside lines and new points where out balls will be placed
GT2004StandardConverter.cpp	changed players positioning
GT2004ConfigurationSymbols.cpp	changed various kick-off positions which were defined in c++

Appendix B: XABSL changes

We found many XABSL files which were using integer values that correspond to a certain region or position on the field which were based on the old field dimensions. The table below only lists the files and states within these files which we have changed. The state definitions are very self-explanatory.

XABSL File	**States**
Simple-basic-behaviors → goalie-position	Changed the cut-y range
Handle-ball	ball-in-center-of-field ball-at-left-border ball-at-right-border ball-at-left-opponent-border ball-at-right-opponent-border ball-near-own-goal
Search-for-ball	Go-to-left-side Go-to-right-side
Playing-goalie	Return Clear-ball Clear-ball-courageous Position Head-back Head-back-from-border
Goalie-clear-ball	Walk
Goalie-position	Ball-just-seen-not-moving Ball-just-seen Ball-not-seen
Playing-defensive-supporter	Ball-in-opponent-half
Position-defensive-supporter-near-own-goal	Position
Playing-offensive-supporter	Changed the common decision tree
Position-offensive-supporter-near-opponent-goal	Position-left Position-right
Position-supporter-near-ball	Choose-side
Position-striker-when-ball-in-own-penalty-area	Position