RoboSoc a System for Developing RoboCup Agents for Educational Use

Fredrik Heintz
frehe@ida.liu.se

March 23, 2000

Abstract

This report describes RoboSoc, a system for developing RoboCup agents designed especially, but not only, for educational use.

RoboSoc is designed to be as general, open, and easy to use as possible and to encourage and simplify the modification, extension and sharing of RoboCup agents, and parts of them. To do this I assumed four requirements from the user: she wants the best possible data, use as much time as possible for the decision making, rather act on incomplete information than not act at all, and she wants to manipulate the objects found in the soccer environment.

RoboSoc consists of three parts: a library of basic objects and utilities used by the rest of the system, a basic system handling the interactions with the soccer server and the timing of the agent, and a framework for world modeling and decision support. The framework defines three concepts, views, predicates and skills. The views are specialized information processing units responsible for a specific part of the world model, like modeling the ball or the agent, controlled by events generated by the basic system. The predicates can be used either by the decision maker or the skills to test the state of the world. The skills are specialized, short-term planners which generate plans for what actions the agent should do in order to reach a desired goal state.

The whole RoboSoc system, implemented in C++ in the Solaris Unix environment, is working and have been used in a course on AI-progamming.

Contents

Chapter 1

Introduction

This chapter contains a short description of the problem I have studied, namely how to create a system for developing RoboCup agents, and the outline of my solution, RoboSoc. The purpose of this chapter is to motivate and define the conditions for the research, but also to state the assumptions I had to make in order to solve it. The last section of this chapter describes the structure of the report. But first a short introduction to RoboCup for those who are not familiar with it.

RoboCup is an attempt to foster artificial intelligence (AI) and intelligent robotics research by providing a standard problem, soccer, where a wide range of technologies can be integrated and examined. The reason for choosing soccer is that it is played in a highly dynamic environment, with both teammates and opponents, where independent agents must collaborate in order to beat the other team [14]. At the same time it is easy to understand the problem and most people know how soccer is played and what a team should do to be successful. These properties make RoboCup an excellent environment for students to, in a playful fashion, create and test different AI-strategies. It is also easy to motivate them by providing competitions where they can test their teams against each other.

RoboCup is divided into different leagues, real robots leagues and a simulated software agents league. This report only consider teams of simulated software agents. The simulation is done by a special server called the Soccer server, which is described in section 2.1. Anybody can download the necessary software and start developing their own soccer agents [28]. Descriptions of the teams that participated in the RoboCup world cup competitions held 1997, 1998 and 1999 can be found in [2, 13, 35]. Many of them can also be downloaded from [22] and tested. More information about RoboCup can be found on web-site of the RoboCup organization [21].

1.1 The Problem

A major problem when constructing RoboCup agents for the simulation league is to implement the basic functionality of the agent. This is a general problem when developing multi-agent systems, Wooldridge and Jennings write “one of the greatest obstacles in the way of the wider use of agent technology is that there are no widely-used software platforms for developing multi-agent system” [37].

This is due to a number of problems, for instance interacting with the Soccer server, and developing the basic skills of a soccer player like finding the ball, move to the ball and kick it in the desired direction. The development of a new RoboCup team from scratch usually takes at least six months, often more. Since we want students to be able to use RoboCup to implement and test different AI algorithms we need some way to make the students start from a much higher level so they can spend their time implementing the AI parts and skip most of the tedious low-level programming needed to create a functional RoboCup team. Therefore a platform for creating new teams is needed. This platform can also be used by non-students creating their first team but I think the educational aspects put higher demands on the ease of use, clean design and sharability so that the teams can improve from year to year even with different students and different approaches.

The platform should first of all take care of the most basic tasks of a RoboCup agent namely receiving data from the server, interpret this sensor data and sending commands back to the server. To be able to do this the platform must have a well defined work cycle since timing is very important and events are asynchronous. It also needs data structures for storing the results from the interpretation of the sensor data. Finally it needs some basic actions or skills the agent can perform.

Since different teams use different theories and models the platform must be able to work with all kinds of different agent architectures and cope with other user defined design issues. The platform should also support, encourage and promote the sharing of different parts of the agent, like skills and information processing algorithms.

1.2 The Solution

Since this is a very open problem I had to make some assumptions. The assumptions I made about what the user wants for their agents are:

1. accurate, complete and consistent data, in that order of importance;

2. use as much time as possible for the decision making;

3. rather act on incomplete information than not act at all;

4. soccer objects the agent can manipulate, like the ball and the players.

Since I want to make as few assumptions as possible I do not make any unnecessary estimations since they can be done in many different ways. The few estimations I do are well documented and easy to change if the user finds a need for it.

My solution to this problem is a system called RoboSoc. It includes three parts: a library of utility classes, a basic system for taking care of the timing and the interaction with the server, and a framework for information processing and decision support. The basic system provides a well defined work cycle by which the agent acts. It takes care of the communication with the Soccer server and it does the basic information processing, mainly parsing the messages sent by the Soccer server but also some feed back from the actuators. The information processing framework consists of a collection of specialized units that process the raw data obtained from the basic system and from other information processing units into information. There are for example units specialized in the ball, the agent, and the players. This system of specialized units makes it easy for the user, i.e. the agent programmer, to either use the units as they are or modify them to suit her needs. RoboSoc provides two more frameworks, one for defining skills and one for defining predicates. The decision making is almost completely done by the user but it is guided by events generated by the basic system.

The purposes of the frameworks are to simplify the extension, modification and sharing of the units. In fact the whole system is designed to be as general and open as possible to encourage and simplify the modification, extension and sharing of RoboCup agents created by RoboSoc.

1.3 The structure of the report

This report is organized as follows: Chapter 2 contains the background material needed for this report. In section 2.1 the Soccer server is discussed, in section 2.2 the educational aspects, and finally in section 2.3 related work.

In chapter 3 RoboSoc is described in detail starting with an overview in section 3.1 where the overall design of RoboSoc is discussed. The library is described in section 3.2, the basic system in section 3.3, and the framework in section 3.4. The chapter is concluded with section 3.5 describing the educational value of RoboSoc.

In the last chapter of the report, chapter 4, contains some concluding remarks. First a summary is given in section 4.1 and then a discussion about future work in section 4.2.

There is also an appendix with a limited user's manual.

Chapter 2

Background

2.1 The Soccer server

The Soccer server is the software developed for RoboCup by Itsuki Noda to do the soccer simulation [20]. It is a distributed simulation server to which 11 players and 1 coach from each of the two teams can connect. One of the players is special, the goal keeper. The goal keeper is the only player who is allowed to catch the ball. There is also a simulated referee who is responsible for enforcing the rules of the game. The referee and the mechanics of the simulation are described in section 2.1.2.

The communication between the server and the clients is done with the UDP/IP protocol, which is a connectionless and unreliable internet protocol [29]. Each agent (player or coach) is a separate program with no direct communication with the other agents. All communication has to go through the Soccer server.

When the agent is connected to the server it receives sensor data continuously as they are available. Visual data is available approximately each 150 milliseconds, physical data roughly each 100 milliseconds and aural data directly when either the referee, one of the players or one of the coaches say something. The sensor data is neither complete nor perfect. The sensors are discussed in detail in section 2.1.3.

The server also accepts commands from the agent describing what action it wants to perform. The commands and their properties will be further discussed in section 2.1.4. Since the UDP/IP protocol is unreliable, data to and from the server will be lost in the network. This together with the noisy sensors and the fact that players can act as often as 10 times a second makes the simulation highly dynamic and uncertain. This also makes the task of developing RoboCup agents very interesting.

Apart from the simulation server there is also a visualization tool called the Soccer monitor where you can watch the games as they are played. In Figure 2.1 you can see what it looks like. The players are the bigger circles

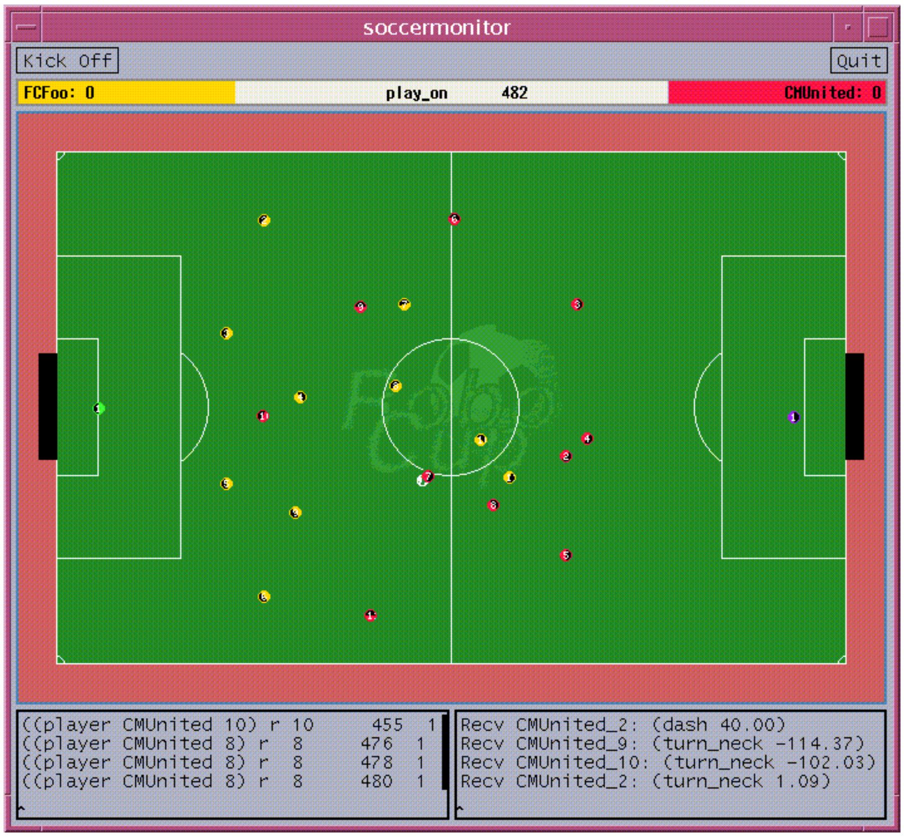

Figure 2.1: The Soccer monitor.

and the small circle, next to the dark player below the middle circle, is the ball. The players are facing the direction of the light half of the circle. The black bars on the left and right side of the field are the goals. The text in the three shaded areas on top of the screen is from the left: the name of the team playing on the left side and its score, the current play mode and the current time, and the name of the team playing on the right side and its score. In this example FCFoo is playing on the left side and has scored zero goals, the current play mode is "play on" and the current time is 482, the team on the right side is CMUnited and they have also scored zero goals.

The rest of this section will describe important parts of the simulation in more detail. Most of the material comes from the Soccer server manual [8], where more information can be obtained.

2.1.1 The server parameters

The Soccer server has many different parameters which control how the simulation works. Most of them can be changed by editing a configuration

Table 2.1: The Soccer server parameters used in the report with their default values. The table is adapted from the parameter table in the Soccer server manual [8]

Parameter name	Default value	Description
simulator_step	100	Length of each simulation cycle in milliseconds
sense_body_step	100	Length of interval, in milliseconds, between `sense_body` informations
send_step	150	Length of interval, in milliseconds, between sending visual information to a player in the standard view mode
recv_step	10	Length of interval between server polling sockets to clients (milliseconds)
visible_angle	90	Angle of the view cone of a player in the standard view mode
visible_distance	3.0	Maximum distance to an object out of the view cone a player can see
audio_cut_dist	50	Maximum distance a spoken message can be heard
ball_size	0.085	The radius of the ball in meters
ball_decay	0.94	Decay rate of speed of the ball (1 = no decay, 0 = all decay)
ball_rand	0.05	Amount of noise added in the movements of the ball
ball_speed_max	2.7	Maximum speed of the ball during a simulation cycle (meter/cycle)
player_size	0.3	The radius of a player (meter)
player_decay	0.4	Decay rate of speed of a player (1 = no decay, 0 = all decay)
player_rand	0.1	Amount of noise added in players' movements and turns
player_speed_max	1.0	Maximum speed of a player during a simulation cycle (meter/cycle)
stamina_min	0.0	Minimum stamina of a player
stamina_max	3500.0	Maximum stamina of a player
stamina_inc_max	35.0	Maximum amount of stamina that a player gains in a simulation cycle
recover_dec_thr	0.3	Decrement threshold for players' recovery
recover_dec	0.002	Decrement step for a player's recovery
recover_min	0.5	Minimum recovery of a player
effort_dec_thr	0.3	Decrement threshold for a player's effort capacity
effort_dec	0.05	Decrement step for a player's effort capacity
effort_inc_thr	0.6	Increment threshold for a player's effort capacity
effort_inc	0.1	Increment step for a player's effort capacity
effort_min	0.6	Minimum value for a player's effort capacity
effort_max	1.0	Maximum value for a player's effort capacity
kickable_margin	0.7	The maximum distance the ball can be from a player and still be kickable
kick_power_rate	0.016	Rate by which the *Power* argument in `kick` commands is multiplied
dash_power_rate	0.006	Rate by which the *Power* argument in `dash` commands is multiplied
inertia_moment	5.0	Inertia moment of a player, affects its movements
maxpower	100	Maximum value of *Power* in `dash` and `kick` commands
minpower	-100	Minimum value of *Power* in `dash` and `kick` commands
maxmoment	180	Maximum value of *Moment* in `turn` and *Direction* in `kick` commands
minmoment	-180	Minimum value of *Moment* in `turn` and *Direction* in `kick` commands
maxneckmoment	180	Maximum value of *Moment* in `turn_neck` commands
minneckmoment	-180	Minimum value of *Moment* in `turn_neck` commands
maxneckang	90	Maximum value of *Moment* in `turn_neck` commands
minneckang	-90	Minimum value of *Moment* in `turn_neck` commands
say_msg_size	512	Maximum length of a message a player can `say`

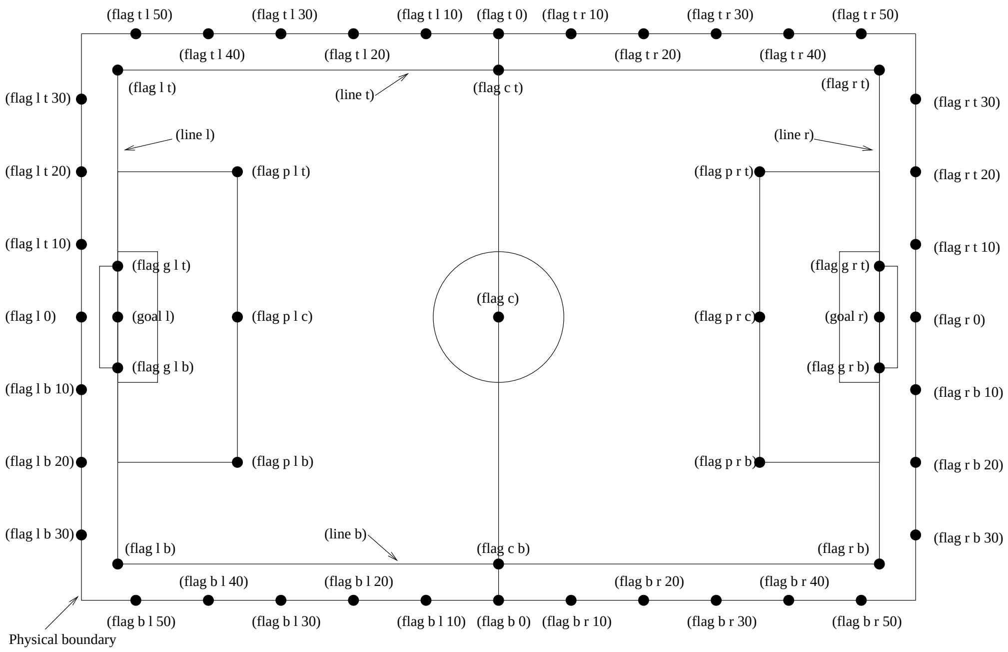

Figure 2.2: The markers and the lines in the simulation.

file and load it with the server. The parameters mentioned in the report are described in Table 2.1. For a complete list see the Soccer server manual [8].

2.1.2 The simulation

Since the simulation is discrete all moves within one time step occur simultaneously at the end of the step. The length of a step is set by the `simulator_step` parameter. At the end of the step, the server takes all action commands received and applies them to the objects in the field, using the current position and velocity information to calculate a new position and velocity for each object.

The soccer field

The soccer field and all objects in it are 2-dimensional, so there is no notion of height of any object. The field is 105 meters long and 68 meters wide. The goals are 14.02 meters wide, about twice the size of a normal soccer goal since it was too difficult to score otherwise. The field and the different markers and lines used by the agents to navigate are shown in Figure 2.2. The players are bound by the outer flags which are 5 meters outside the playing field. The markers on the field have a known absolute position which is independent of the side the team is currently playing on. The

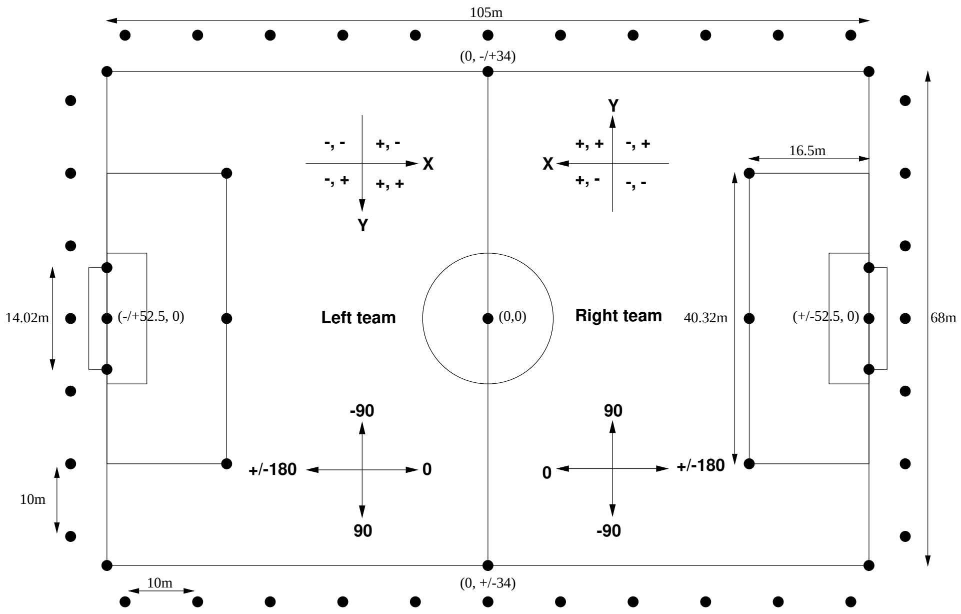

Figure 2.3: The coordinate system for the two teams. The coordinates are given in the form (x, y), if two signs are shown then the first sign is for the left team and the second sign is for the right team. The two diagrams at the top shows the sign of the coordinates for each team and the two at the bottom shows the directions of each team.

coordinate system and the directions are on the other hand dependent on what side the team is playing on, see Figure 2.3. The origo is in the center of the field; negative x-values are always found on the team's own side of the field and positive on the opposite side; negative y-values are found on the team's left part of the field and positive on the right, when standing in origo facing the opponents' goal. Directions are also based on the side of the team with straight ahead (0 degrees) from the origo towards the opponents' goal. Positive angles are to the right of the agent (standing in origo facing the opponents' goal) and negative angles are to the left.

All movable objects on the field, i.e. the players and the ball, are treated as circles. All distances and angles are to the center of the circles.

The referee

The simulated referee controls the play mode of the game. It decides when a team scores a goal, when the ball is out of bounds and when to call an offside. There are also rules which the simulated referee can not enforce since they concern the "intention" of the players. Some of these rules are: surrounding the ball, not putting the ball into play, and intentionally blocking the movement of other players. Therefore a human referee also has the possibility to call free-kicks.

Movements of objects

At the end of each time step the server updates the current state of all the objects in the world. To quote the the Soccer server manual the movement of each object is calculated in the following manner [8]:

$$\begin{aligned}
(u_x^{t+1}, u_y^{t+1}) &= (v_x^t, v_y^t) + (a_x^t, a_y^t)\text{: accelerate} \\
(p_x^{t+1}, p_y^{t+1}) &= (p_x^t, p_y^t) + (u_x^{t+1}, u_y^{t+1})\text{: move} \\
(v_x^{t+1}, v_y^{t+1}) &= \mathit{decay} \times (u_x^{t+1}, u_y^{t+1})\text{: decay speed} \\
(a_x^{t+1}, a_y^{t+1}) &= (0,0)\text{: reset acceleration}
\end{aligned}$$

where (p_x^t, p_y^t), and (v_x^t, v_y^t) are position and velocity of the object in timestep t, *decay* is a decay parameter specified by `ball_decay` or `player_decay` depending on what object is being updated. (a_x^t, a_y^t) is the acceleration of the object, which is derived from the *Power* parameter in `dash` (in case the object is a player) or `kick` (in case the object is the ball) commands in the following manner:

$$(a_x^t, a_y^t) = \mathit{Power} \times \mathit{power\_rate} \times (\cos(\theta^t), \sin(\theta^t))$$

where *power_rate* is either `dash_power_rate` or is calculated from `kick_power_rate` as described in section 2.1.4, and θ^t is the direction of the object in timestep t. In the case of a player θ^t is the direction the player is facing. In the case of the ball the direction is:

$$\theta^t_{\text{ball}} = \theta^t_{\text{kicker}} + \mathit{Direction}$$

where θ^t_{ball} and θ^t_{kicker} are the directions of the ball and the kicking player at time t, and *Direction* is the second parameter of the `kick` command.

Collisions

At the end of the simulation cycle if two objects overlap then a collision occurs and the objects are moved back until they do not overlap. Then the velocities are multiplied by -0.1. Note that it is possible for the ball to go through a player as long as the ball and the player never overlap at the end of a cycle.

The stamina model

To simulate the endurance of the players each player has a limited amount of stamina. The stamina is used when the player dashes and is regained a little each cycle. The maximum *Power* the player can dash with and the amount of stamina regained each cycle is depending on the three variables *stamina*, *effort*, and *recovery*. Where *stamina* is the total amount of energy the agent has left, *effort* determines how effective the agent's dashes are and *recovery* controls how much stamina is regained each cycle. According to the Soccer server manual the stamina model works as follows:

> When the client dashes its stamina is updated like this:
> If $Power \geq 0$:
>
> $$\begin{aligned} EffectivePower &= effort \times \min(Power, stamina) \\ stamina &= \max(stamina - Power, \texttt{stamina\_min}) \end{aligned}$$
>
> If $Power < 0$:
>
> $$\begin{aligned} PowerUsed &= \min(-2 \times Power, stamina)/-2 \\ EffectivePower &= effort \times PowerUsed \\ stamina &= \max(stamina + 2 \times PowerUsed, \texttt{stamina\_min}) \end{aligned}$$
>
> The *Power* of the dash actually executed is *EffectivePower*.
>
> Every cycle (whether the client dashes or not), the three variables are updated in the following way:
>
> - **Stamina:** Stamina increases slightly every cycle. When *recovery* decreases less stamina is recovered.
>
> $$stamina = \min(\texttt{stamina\_max}, stamina + recovery \times \texttt{stamina\_inc\_max})$$
>
> - **Effort:** The basic idea is that if *stamina* gets low, *effort* decreases (with a minimum value given by `effort_min`) and if *stamina* gets high enough, then *effort* increases with a maximum of `effort_max`. Specifically:
>
> $$effort = \begin{cases} \max(\texttt{effort\_min}, effort - \texttt{effort\_dec}) & \text{if } stamina \leq \texttt{effort\_dec\_thr} \times \texttt{stamina\_max} \\ \min(\texttt{effort\_max}, effort + \texttt{effort\_inc}) & \text{if } stamina \geq \texttt{effort\_int\_thr} \times \texttt{stamina\_max} \\ effort & \text{otherwise} \end{cases}$$
>
> - **Recovery:** This is similar to effort except that *recovery* never increases.
>
> $$recovery = \begin{cases} \max(\texttt{recover\_min}, recovery - \texttt{recover\_dec}) & \text{if } stamina \leq \texttt{recover\_dec\_thr} \times \texttt{stamina\_max} \\ recovery & \text{otherwise} \end{cases}$$
>
> The order in which the different variables are changed are important. First the *stamina* value is decreased (if there is a dash). Next, the *recovery* value is changed, then the *effort*. Finally, *stamina* is recovered.

Figure 2.4 visualize the different thresholds in the stamina model.

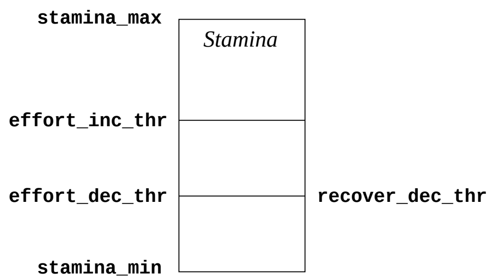

Figure 2.4: The different thresholds used in the stamina model.

Table 2.2: Data given by the visual sensor.

Data	**Description**
ObjectName	What object the data describes
Distance	The distance to the object
Direction	The direction to the object
DistChng	The change in distance to the object
DirChng	The change in direction to the object
BodyDir	The body direction relative the facing direction of the agent, only if the object seen is another player
HeadDir	The head direction relative the facing direction of the agent, only if the object seen is another player
TeamName	The name of the team of the seen player, only if the object seen is another player
UniformNumber	The uniform number of the seen player, only if the object seen is another player

2.1.3 The sensor inputs

As mentioned before there are three types of sensor inputs providing visual, aural and physical sensor data. Each of the sensors and what type of data they send to the agent will be discussed in this section.

The visual sensors

Each `send_step` the agent receives visual sensor data. In Table 2.2 the data given by the visual sensors are described. Some of the data is only available if the agent is close enough to the object, the quality of the data is also dependent on the distance to the object. For a detailed description of the exact details see the Soccer server manual [8].

The aural sensors

The agent will receive data from the aural sensor when it is available. Data is available when one of the players, the coaches or the referee says something

Table 2.3: Data given by the physical sensor.

Data	Description
ViewQuality	The current setting of the agent's *ViewQuality*, affects the amount and quality of the visual data sent to the agent
ViewWidth	The current setting of the agent's *ViewWidth*, affects the amount and quality of the visual data sent to the agent
Stamina	The current stamina of the agent
Effort	The current effort of the agent
AmountOfSpeed	The amount of the agent's current speed vector
HeadDirection	The relative direction of the agent's head
DashCount	The number of dashes made by the agent so far
KickCount	The number of kicks made by the agent so far
SayCount	The number of says made by the agent so far
TurnCount	The number of turns made by the agent so far
TurnNeckCount	The number of turn necks made by the agent so far

and the distance to the speaker is less than `audio_cut_dist`. The agent will hear all messages from itself, the referee and the coaches, but only one message from another teammate in two cycles. If more than one message is sent in the two cycles the first to arrive will be heard.

The physical sensors

Each `sense_body_step` the agent will receive physical sensor data. In Table 2.3 the data given by the physical sensor is described.

2.1.4 The available commands

This section will discuss what basic actions the agent can tell the server to perform. Table 2.4 contains a short description of the syntax of the commands, the range of the parameters and how often the actions can be sent. The detailed description of the effects of the actions below is taken from the Soccer server manual:

- `turn`: The turn moment must be between `minmoment` and `maxmoment` (-180 degrees and 180 degrees by default). However, there is a concept of inertia that makes it more difficult to turn when the agent is moving. The actual angle the player is turned is calculated as follows:

  $$ActualAngle = Moment/(1.0 + \texttt{inertia\_moment} \times player\_speed)$$

  (Note that player_speed is the amount of the player's velocity vector, and is therefore always positive.) `inertia_moment` is a parameter with default value 5.0. Therefore (with default values), when the player is at max speed (1.0), the maximum effective turn it can do is ± 30. However, because the agent can not dash and turn in the same cycle, the fastest a player can be going when executing a `turn` is `player_speed_max` $\times$ `player_decay`, which means the effective turn (with default values) is ± 60.

- `turn_neck`: Each client has a neck which can be turned independently of its body. The angle of the player's head is the viewing angle of the player. The `turn` command changes the angle of the player's body while `turn_neck` changes the angle of the player's head relative to its body. The maximum relative angle for the player's neck is 90 degrees to either side. Remember that the neck angle is relative to the player's body so if the client issues a `turn` command, the viewing angle changes even if no `turn_neck` command is issued.

  Also, `turn_neck` commands can be executed in the same cycle as `turn`, `dash`, and `kick` commands. `turn_neck` is not affected by momentum like `turn` is. The argument for a `turn_neck` command must be in the range $[-180, 180]$ and the resulting neck angle must be in $[-90, 90]$.

- `dash`: The `dash` is essentially a small push in the direction that the player's body is facing. It is *not* a sustained run. In order to have a sustained run, multiple `dash` commands must be sent. The power passed to the dash command is multiplied by `dash_power_rate` (default 0.006) and the effort (see Section 2.1.2) and applied in the direction that the player's body is facing. With negative power, the agent dashes backwards, but it consumes twice the stamina (see Section 2.1.2).

- `kick`: The `kick` is very similar to the dash except that it accelerates the ball instead of the player. If the player tries to kick when the ball is further than the `kickable_area` (which is equal to the `player_size` $+$`ball_size` $+$`kickable_margin`), there is no effect. The one important difference between dashes and kicks is how the kick power rate is figured. Let *dir_diff* be the absolute value of the angle of the ball relative to the direction the player's body is facing (if the ball is directly ahead, this would be 0). Let *dist_ball* be the distance from the center of the player to the center of the ball. Then the kick power rate is figured as follows:

  $$\texttt{kick\_power\_rate} \times \left(1 - \frac{.25 \times \mathit{dir\_diff}}{180} - \frac{.25 \times \left(\mathit{dist\_ball} - \texttt{player\_size} - \texttt{ball\_size}\right)}{\texttt{kickable\_margin}}\right)$$

  Basically, this means that the most powerful kick can be done when the ball is directly in front of the player and very close to it, and drops off as both distance and angle increase.

Table 2.4: The syntax of the basic actions the agent can perform [8].

Syntax	**One/turn?**
`(catch` *Direction*`)` *Direction* ::= `minmoment` ~ `maxmoment` degrees (default `-180` ~ `180`)	Yes
`(change_view` *Width Quality*`)` *Width* ::= narrow | normal | wide *Quality* ::= high | low	No
`(dash` *Power*`)` *Power* ::= `minpower` ~ `maxpower` (default `-100` ~ `100`)	Yes
`(kick` *Power Direction*`)` *Power* ::= `minpower` ~ `maxpower` (default `-100` ~ `100`) *Direction* ::= `minmoment` ~ `maxmoment` degrees (default `-180` ~ `180`)	Yes
`(move` *X Y*`)` *X* ::= `-54.5` ~ `54.5` *Y* ::= `-34` ~ `34` `move` only works in `before_kick_off` mode and 5 seconds after a goal is scored	Yes
`(say` *Message*`)` *Message* ::= a string of at most `say_msg_size` characters (default 512)	No
`(turn` *Moment*`)` *Moment* ::= `minmoment` ~ `maxmoment` degrees (default `-180` ~ `180`)	Yes
`(turn_neck` *Moment*`)` *Moment* ::= `minneckmoment` ~ `maxneckmoment` degrees (default `-180` ~ `180`) `turn_neck` is relative to to the direction of the body and the resulting angle must be between `minneckang` and `maxneckang` degrees (default `-90` ~ `90`). Can be invoked at the same cycle as a turn, dash, catch or kick.	Yes

2.1.5 The command cycle

The command cycle, i.e. the cycle within which the agent must reason, decide on action and send it to the server, is actually composed of four different cycles, running in parallel. The four cycles are:

- **The simulator cycle** controls the time in the simulation. The length is determined by the parameter `simulator_step`, its default value is 100 milliseconds.

- **The send physical sensor data cycle** controls when to send the current physical sensor data to the agent. The length is determined by `sense_body_step`, its default value is 100 milliseconds.

- **The send visual sensor data cycle** controls when to send the current visual sensor data to the agent. The length is determined by `send_step`, its default value is 150 milliseconds.

- **The receive commands from the clients cycle** controls when the server receives commands from the clients. The length is determined by `recv_step`, its default values is 10 milliseconds.

The four cycles are controlled by the same internal clock inside the server and due to the new implementation of the server they are updated quite accurately. But they are still asynchronous as shown in Figure 2.5. The

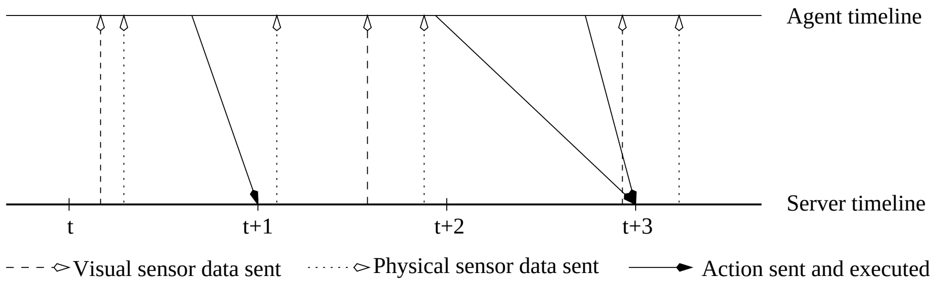

Figure 2.5: An example of what data the agent receives when, and also when commands sent by the agent will be executed.

example in the figure shows what data the agent receives and when. As shown, the data can arrive both earlier and later than expected.

All actions received by the server at the end of the simulator cycle will be used when updating the state of the simulation. The example shows the agent sending commands back to the server, but due to the delays in the network and the fact that the server takes some time before reading the messages received (between 0 and `recv_step` milliseconds) they are not always executed in the same simulation step as the agent sent them.

Since the agent neither knows when it will receive the next sensor input nor when the current simulator cycle will end it has to make a decision whether to wait for more information or to act on the current information. This decision will affect both how often the agent can act and how much information can be used when deciding what action to do. This balance is very central to every RoboCup agent and have great impact on the performance of the agent.

2.2 Educational aspects

Since 1997 the Department of Computer Science at Linköping university has been giving a course in AI-programming focusing on the problem of developing RoboCup teams. A paper describing the course has been written by Silvia Coradeschi and Jacek Malek [6]. They found that

> The use of a challenging and interesting task [RoboCup], and the incentive of having a tournament has made the [AI-programming] course quite successful, both in terms of enthusiasm of the students and of knowledge acquired.

Similar conclusions are also drawn by Russel and Norvig when they introduced the wumpus world in their introductory AI course [25].

The major problem with the AI-programming course, from the students' point of view, was that creating RoboCup agents requires a large amount of knowledge not related to AI, like real-time and process programming, and a large effort to overcome the initial problems, like communicating with the server, parsing the server messages, calculating the position of the player and sending simple commands back to the server. The most requested improvement was more help with these practical problems of developing a RoboCup agent [6].

This AI-programming course and the fact that there are no libraries developed for the purpose of helping students creating their own RoboCup teams has motivated and inspired me to do this master thesis. The educational aspects of the problem induces some important demands on the library, it should be:

- Simple enough to be easy to use;
- general, in order to allow different approaches to be implemented and tested;
- extendible, it should be possible to replace and extend the different parts of the library with as little difficulty as possible.

In order for a library, with the above properties, to be really useful from an educational point of view it has to take care of the basic problems that a RoboCup agent has to handle. Four essential problems I think should be addressed by a library are:

1. **Basic server communication:**
   The first practical problem when building RoboCup agents is to get the basic communication with the server working, i.e. the agent must be able to receive the data sent by the server then be able to use this data to decide on an action and finally to be able to send that action, with the correct parameters, back to the server.

2. **Timing:**
   As described in the section about the different cycles of the server there are a lot of things to keep in mind when handling the timing. Together they make the problem of keeping track of the current simulation step and send commands in time for them to be executed in the current step, but at the same time use the best available information when making the decision, very hard. It is also a major source of confusion when first encountered. Therefore it is very important that students do not have to worry about these low-level problems.

3. **World modeling:**
   The next problem is to keep track of the state of the simulation based on the sensory data received from the server. Since the soccer environment is not very hard to represent, the main problem is to use the

noisy and incomplete sensor data to draw correct conclusions about the objects in the world. There is also the problem of making predictions about future states of the server, but that is not so important as having a model of the current world which is as complete, consistent and updated as possible.

4. **Support for decision making:**
   The fourth problem is, based on the current world model, to decide on what action to do. This is of course the job of the agent designer not the library designer but to assist the agent programmer it is very good to have some support for the decision making like notifications when new information arrive from the server and when a new cycle is started. Since most decision makers used in RoboCup use some sorts of rules it is practical to have some predefined predicates, like is the ball on our half, are we playing on the left side of the field and so on, which can be used to trigger rules. With decision making I mean the very general meaning of the concept from a RoboCup point of view, to select some actions to do based on the current world model. It is also very convenient to have some higher level of abstraction when it comes to actions. Therefore intermediate skills are needed, which are based on the basic actions the agent can do. For example turn to absolute direction D, or kick the ball to position (X, Y).

2.3 Related work

This section describes some of the related work to this report. The first section will discuss the concept of an agent. The second section will define what an agent architecture is and what kind of architectures exist. The third, and final, section describes some of the RoboCup libraries available today.

2.3.1 What is an agent?

The are many answers to that question, almost as many as there are agent researchers. The definition that is most appropriate for this work is the definition of Russel and Norvig, they define an agent as "anything that can be viewed as *perceiving* its environment through *sensors* and acting upon that environment through *effectors*" [26]. Even though it is a very simple definition of an agent, compared to most other definitions (see [11] for a collection of agent definitions). It is general enough to cover most agents used in RoboCup, since all RoboCup agents decide upon sensor information from the soccer server what effectors to use by sending commands back to the server.

2.3.2 Agent architectures

Since there are many definitions of what an agent is, there are also many definitions of what an agent architecture is. For example Pattie Maes defines an agent architecture as: "[A] particular methodology for building [agents]. It specifies how ... the agent can be decomposed into the construction of a set of component modules and how these modules should be made to interact. The total set of modules and their interactions has to provide an answer to the question of how the sensor data and the current internal state of the agent determine the actions ... and future internal state of the agent. An architecture encompasses techniques and algorithms that support this methodology." [16]

Over the past decade a number of different approaches to the problem of finding good functional decompositions of agents has lead to at least four major types of agent architectures: reactive, deliberative, hybrid and interacting architectures [17].

Reactive architectures

The reactive, behavior-based or situated, architectures are strongly influenced by behaviorist psychology. The goal is to achieve robust behavior instead of correct or optimal behavior. They do this by using simple action rules based on the current situation with little or no explicit information [17].

An example of a behavior-based architecture used in the RoboCup domain is the architecture used by the Headless Chickens [7, 27].

Deliberative architectures

Agents based on Simon and Newell's physical symbol system hypothesis [19], the assumption that agents maintain an internal representation of their world, and that there is an explicit mental state which can be modified by some form of symbolic reasoning are called deliberative agents [17].

One example of a deliberative architecture is the Belief, Desire, Intention architecture (BDI) developed by Bratman et al [3]. BDI is used in the RoboCup domain by for example AT Humboldt [4, 5].

Hybrid reactive-deliberative architectures

Since most agents need to be both reactive and able to use deliberation in their decision processes the hybrid, layered, architecure has emerged. It usually combines reactive and deliberative layers in some way to create a hybrid agent architecture [17]. The major problem is how to combine the different layers into a single unit. Wooldridge and Jennings even argue that hybrid architectures are very *ad-hoc* since it is not clear how to reason about them and what the underlying theory is [36].

An example of hybrid architecture used in RoboCup is the architecture used by FCFoo [12].

Interacting architectures

Architectures which mainly deal with coordination and cooperation among distributed intelligent agents are called interacting architectures [18].

No example of interacting architectures have be found in the RoboCup domain.

2.3.3 Other RoboCup libraries

Previous attempts to build RoboCup libraries have usually been a team releasing parts of their code for others to use. The problem with these releases are usually that they come with little or no documentation and the code is tightly connected to the structure of the team. Therefore there is a need for a well documented, well structured and generic library for people to use. The structure of the library should help and encourage different developers to share their code. The benefit is that new teams do not have to start from scratch but can instead build on previous teams experience.

The rest of the section will be used to describe two existing libraries, libsclient and RoboLog, and also the team CMUnited which have been a great source of ideas and inspiration.

Libsclient

Libsclient is a library of C routines for basic RoboCup client functionality first developed by Itsuki Noda at ETL, Japan, and then extended by Yaser-Al Onaizan, Gal Kaminka, Jafar Adibi, and the other members of the ISIS team at the University of Southern California/Information Sciences Institute, USA.

This library contains an interface to the server, a parser, an algorithm for calculating the position of the agent and some very basic algorithms for calculating the absolute position of an object and for calculating the turn's and dash's the agent has to make in order to move to a certain position.

Unfortunately this library is not very suitable for educational use since it is very basic and does not really handle any of the low-level problems of the server. For example the basic network services only takes care of sending and receiving messages on request by the agent. It does not address any of the problems with synchronization with the server or making sure that the agent tries to send commands each cycle and base them on the best available data, which is one of the most important tasks for any library useful for educational purposes. But it is useful for scientists who want to do their own thing, but do not want to do the parsing and need a better interface to the communication with the server. Another problem is that the libsclient is no longer supported, and it is not up to date with the current server.

RoboLog

RoboLog is an ECLiPSe-Prolog library built on top of a C++ library developed by Oliver Obst at the university of Koblenz-Landau. The C++ library was developed to make the server functions accessible from Prolog and to build a database with the data from the soccer server. The interface to the database is somewhat awkward and not very elegant since all the data is stored in one large structure. A more serious problem with this world model is that no inference about the objects are made from old knowledge, instead the new sensor data is simply stored in the database and the client programmer has to do all the inference.

Apart from the world model RoboLog is an advanced library with many nice features and algorithms for computing the position of the agent, do geometric calculations and support the development of advanced skills. Even though it does not have any abstractions of the basic server commands, like turn to absolute direction D or kick the ball to the point (X, Y). It does take care of most of the low-level tasks like sending and receiving data from the server and the timing of the commands, but in a rather simple manner. It makes sure that the agent will not send more than one catch, dash, move or turn each cycle, but it does not give the agent any hints of when it has to send the commands or when it is getting too late to send a command the current cycle.

Another serious problem, from my point of view, is the fact that it is very hard to replace parts of the RoboLog code with new, without having to change other, irrelevant, parts of the code, mostly because of the design of the world model. This together with the complexity of the library mainly due to lack of a clear general architecture makes it quite hard to understand and exchanging parts of the code to try different approaches is cumbersome. Finally the threshold before productivity is not low enough for it to be useful in an educational setting. The benefits from the library is the possibility to program agents in Prolog, and the large amount of features available when the initial obstacles are overcome.

More information about RoboLog can be found on their web site [23].

CMUnited

CMUnited is the most successful team in the history of RoboCup so far, it has won the last two competitions, 1998 in Paris and 1999 in Stockholm. It is developed by Peter Stone, Manuela Veloso and Patrick Riley from Carnegie

Mellon University, USA. The team and its development is described in a series of articles, among them [31, 33, 34], and in Peter Stone's PhD-thesis [30].

To help the rest of the RoboCup community they have released parts of their team. The released code does the basic server communication, most of their world modeling and their basic actions and skills. Their implementation is very advanced and they have one of the most accurate world models available. The main problem is the very tight connection between the parts, it is almost impossible to replace parts of it. The code is also very complex, and it is hard to get a grasp of what it really does. The documentation is very good and it is easy to get an overall picture of how their team and the major algorithms works.

Even though their code is not very useful from a student perspective their release is very important since others can use it as a source of inspiration and ideas and are also allowed to take the parts of the code they find use for.

Chapter 3

RoboSoc

This chapter describes the RoboSoc system for developing RoboCup agents. It discusses how the system works, its design and the reason behind the design. It is not a manual of how to use RoboSoc, a limited user's guide can be found in Appendix A.

Since RoboSoc is object-oriented, and I assume the reader have some basic knowledge about the object-oriented paradigm, I will sometimes refer to objects, classes and methods when I talk about different parts or features of RoboSoc.

Everything described in this chapter is actually implemented and has been used in a course on AI-programming at the computer and information science department at Linköping university in the fall of 1999 [1]. The implementation is done in C++ on a UNIX platform.

3.1 Overview

The design goal of RoboSoc is to create a system for developing RoboCup agents especially, but not only, for students which is as general, open, and easy to use as possible and that encourages and simplifies the modification, extension and sharing of RoboCup agents, and parts of them.

As stated in the introduction this is a very open problem and some assumptions about what the user wants from the system have to be made. I assume the user wants:

1. accurate, complete and consistent data, in that order of importance;

2. use as much time as possible for the decision making;

3. rather act on incomplete information than not act at all;

4. soccer objects the agent can manipulate, like representations of the ball and the players.

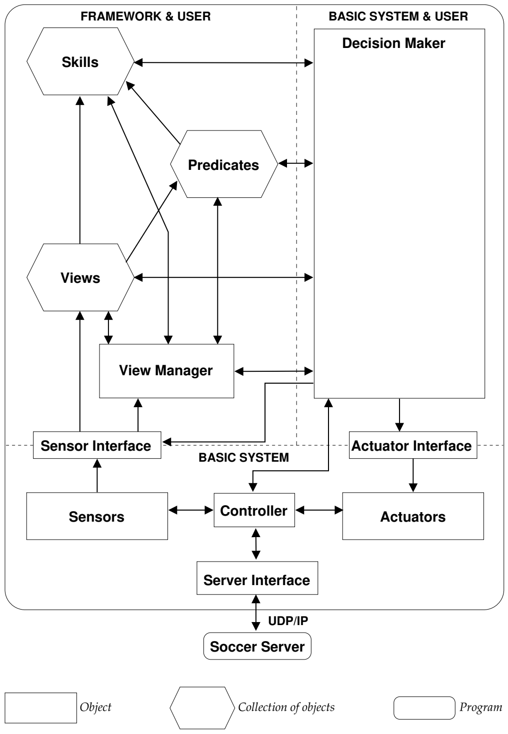

Figure 3.1: The RoboSoc architecture.

The design goal together with the four assumptions above and the four essential problems discussed in section 2.2 has lead to a system consisting of three major parts:

- a library of utility classes, like geometric objects, game objects, and some basic building blocks to be used with the framework;
- a basic system for the timing and the interaction with the server; and
- a framework for information processing, skills and predicates.

The system architecture is shown in Figure 3.1 and includes the basic system and the framework. The figure consists of three different kinds of boxes representing different kinds of units. The rectangular boxes represent objects (classes), the hexagonal boxes collections of objects (from the same base class) and the boxes with the rounded edges separate independent programs. The arrows show how data and information flows between the different subparts. An arrow does not always implicate a dependency between the units. In fact, the library is not dependent on any part of the system, the basic system is dependent on the library, and the framework is dependent on the basic system and therefore also on the library. This means that a user can either use the library, the basic system and the library or the whole system when creating a team. Most of the implementation of the framework and the decision maker is supposed to be done by the user, with support from the framework and the library, while the basic system is not supposed to be changed be the user, unless it is really necessary.

The following sections of this chapter describe the functionality and design of the three majors parts.

3.2 The library

The RoboSoc library is the foundation for the rest of the system. It provides the necessary types, classes and utilities needed by the other parts of the system. There are classes for representing geometric objects, there are classes for representing the objects found in the soccer environment and there is a class for each of the commands that can be sent to the soccer server.

3.2.1 Data types

To make the system more machine independent there is a set of basic types for representing integers and floating point numbers. They are named after the type they represent and the number of bits they can store. There are also types defined to represent simple objects in the RoboCup simulation, like a command name, a marker name and so on.

Modifiers

Since most values a RoboCup agent has to handle are based on uncertain observations, or estimations there is a need for representing the confidence of the correctness of a value. Since the agent is not always aware of the state of an object in the world, like the velocity of the ball, there is also a need for representing unknown values. To support the reasoning with observation times, uncertainty and unknown values I have implemented a system with modifiers where a basic type can be modified to handle one of the features.

3.2.2 Utilities

The library provides some basic mathematic utilities like calculating absolute values, convert radians to degrees and so on, but also a class for representing a communication socket and a class for representing a real-time timer.

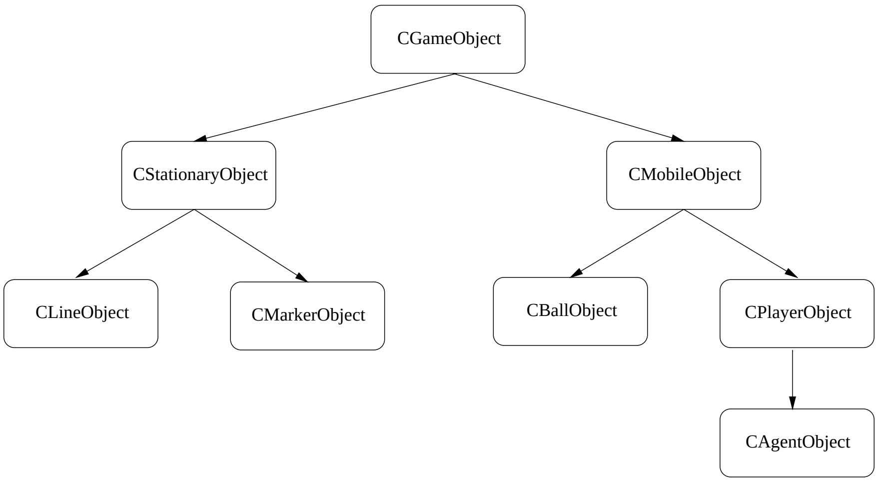

Figure 3.2: The object hierarchy of the game objects in the RoboSoc library.

3.2.3 Geometric objects

The geometric objects included in the library are classes for representing angles, both in degrees and in radians, vectors, point, lines and rectangles. The reason for having two types of angles is that you do not want to confuse the user by making it unclear what unit the angles are in.

Modifiers

The same argument about uncertainty and unknown values is valid for geometric objects therefore there are modifiers for them too. Unfortunately the modifiers described above only work on basic types like integers and floats, therefore there exist specialized modifiers for the geometric objects.

3.2.4 Game objects

The game objects represent the objects introduced by the soccer environment. They include a ball object, a player object and an agent object. The object hierarchy is shown in Figure 3.2.

This hierarchy looks almost the same as the object hierarchy used by CMUnited [34], but in fact they are not very similar because my objects have no reference to the current world model of the agent, as the CMUnited objects have. The main reason is that I want the library to be stand-alone and not dependent on anything else, especially not the world model which is usually very specific for each agent design.

Since each game object represents an object at a certain moment in time, and they do not have any connection to the rest of the world model, they can also be used by a prediction system to create multiple, possible, versions of the object. This is not possible with the objects used by CMUnited. If the user desires, she can extend the basic game objects by connecting them to the world model of the agent. The benefit is that you can make sure that the state of the object is valid according to the world model and the simulation, for example by checking the speed and see if it is within the limits of the object, but also to make special, world or simulation dependent, update methods directly in the game object.

There is also a template for representing a collection of game objects called ObjectCollection used in the views to represent the history on an object. It can be instantiated with any class derived from the CGameObject class.

3.2.5 Command objects

The command objects are designed to represent the commands that can be sent to the Soccer server. The version of the command objects included in the library are very simple since they are completely separated from the world model of the agent, for the same reason as the game objects. This unfortunately implies that no checks can be made about the validity of the commands sent to the server. But the objects can be extended to support more advanced functionality like making sure the arguments are legal according to the simulation and to assist the updating of the world model.

The available commands and their arguments are shown in Table 3.1 in section 3.3.5.

3.3 The basic system

The purpose of the basic system is to take care of the basic server communication and the timing as described in section 2.2. To accomplish this the task is to receive the messages from the server as soon as they arrive and send them to the sensors which parse the messages and store the raw data. Concurrently the controller keeps track of the current game time and when it is time to send commands back to the server. When the time comes the controller gets the next command from the actuators, which are responsible for queuing the commands from the decision maker, and sends it to the server for execution. The type of data and how it flows through the basic system is shown in Figure 3.3.

It is possible to use only the basic system, without the information processing and skills framework on top of it. The only difference is that the user must use simpler versions of the sensor and actuator interfaces, all the functionality of the basic system are still available to the user.

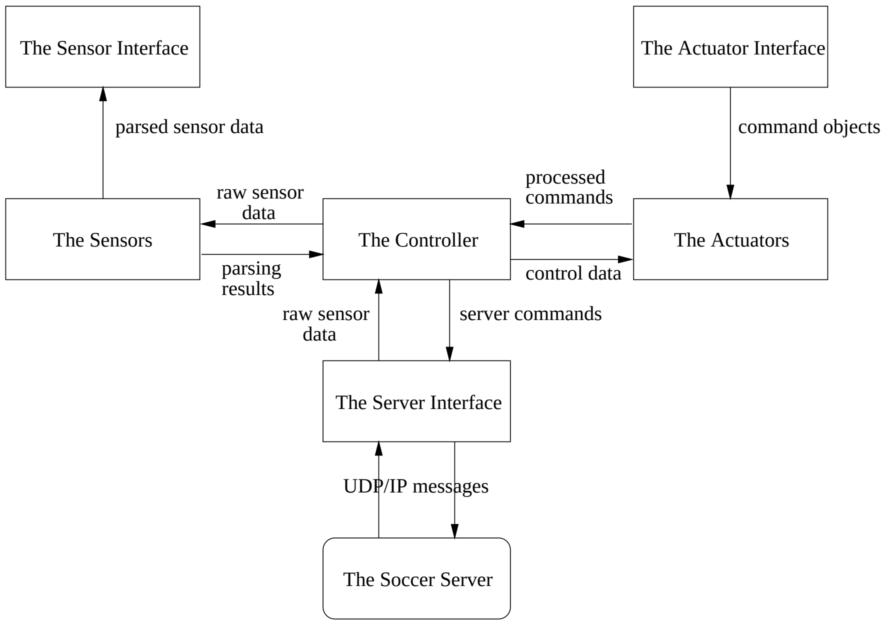

Figure 3.3: The data flow in the basic system of RoboSoc.

The basic system consists of the server interface, the controller, the sensors, the sensor interface, the actuators, the actuator interface and the decision maker. The rest of this section is devoted to describe these different subparts in detail.

3.3.1 The server interface

The server interface takes care of the lowest level of communication with the server by providing an interface to the socket implementing the actual server communication. The controller uses the interface to connect to the server when the agent is started, receive the server messages when they arrive and send the commands back to the server.

3.3.2 The controller

One of the most important units in RoboSoc is the controller. It is responsible for the timing of the agent and the synchronization with the server. Its task is to keep track of the current game time, receive messages as soon as they arrive and control when to act. It should also dispatch the messages from the server to the sensors and guide the decision maker by generating events.

To complete its task the controller uses an internal clock, an IO handler and three control algorithms. The internal clock will generate an interrupt every `recv_step` (usually 10) milliseconds and the IO handler an interrupt as soon as data arrive to the socket connecting the agent to the Soccer server. The first algorithm is used to update the agent after a tick of the internal clock, the second algorithm is used to update the agent when sensor data is received and the third algorithm is used to update the current time of the agent based on the last known server time. These three algorithms define the working cycle of the agent since they generate the events which guide the decision maker.

The only assumption about the server the algorithms make is that the agent receives physical sensor data from a `sense_body` message almost every cycle either at the beginning of the new cycle or at the end of the old cycle. This assumption makes it possible to assume a new cycle has started when the agent receive a `sense_body` message and it can also assume it has almost 100 milliseconds before its command for this cycle has to be at the server. Before discussing the algorithms in detail the RoboSoc concept of time needs to be defined.

Time in RoboSoc

The time concept used in RoboSoc is based on the CMUnited concept of time [8]. They represent time as a tuple of the last known server time and the number of cycles since the clock was stopped. If the clock is not stopped then the second value is 0. The reason behind this concept is that the agent wants to reason about time even when the game clock is stopped and the time reported by the server does not change. This makes it possible to use the same internal work cycle even when the game is stopped, which would not be possible otherwise. Whether the game clock is stopped or not is depending on what state the game is in. The state of the game is only changed after a call of the referee, therefore the starting and stopping of the clock is done in the sensor interface after the sensors have received a message from the referee.

Algorithm for updating the agent after a tick of the internal clock

The internal clock is used to time the acting of the agent and to predict if the agent is missing messages from the server. Therefore the algorithm has to check if a new cycle ought to have begun and in that case update the state of the agent with this prediction, but since the controller does not want to assume that a new cycle actually has begun it will generate an event which allows the decision maker to react to the situation. The decision maker has two possibilities, either to assume that a new cycle has begun or to assume that a new cycle has not begun. When this is done the algorithm will check to see if a periodic command is waiting to be sent. Otherwise, it checks if the last time for sending a periodic command this cycle is approaching, then it generates an event for the decision maker to handle. The reason for this is the second assumption about the user given above. Then the algorithm checks if there are some other commands waiting to be sent.

Algorithm 3.1 Update the agent after a tick of the internal clock.

```
begin
    Generate a BeforeTick event
    Update counters
    if ( ticks > ticks between cycles ) then begin
        Assume new cycle
        Update current time
    end
    if ( assumed new cycle ) then begin
        Generate an EstimatedNewCycle event
        if ( the decision maker forced a new cycle ) then
            Generate a NewCycle event
    end
    if ( periodic actions waiting and current time > last action time ) then
        Send next periodic action
    else begin
        if ( ticks > ticks before command warning ) then
            Generate a CommandWarning event
        if ( immediate actions waiting ) then
            Send next immediate action
    end
    Generate an AfterTick event
end
```

Algorithm for updating the agent when receiving sensor data

When sensor data is received it is first parsed by the sensors, then a suitable event is generated based on what type of sensor was used in parsing the message.

Algorithm 3.2 Update the agent when receiving sensor data.

```
begin
    Generate a BeforeSensorData event
    while ( sensor data waiting to be received ) do begin
        Receive the sensor data
        Let the sensors analyze the sensor data received
        switch ( last sensor type ) begin
            case Physical sensor: Generate a PhysicalSensorData event
            case Visual sensor: Generate a VisualSensorData event
            case Aural sensor: Generate an AuralSensorData event
            case Init sensor: Generate an Init event
            case Sensor error: Generate a SensorError event
        end
    end
    Generate an AfterSensorData event
end
```

Algorithm for updating the current time after receiving the current server time

Assumptions made about the current time of the agent:

- If the clock is not stopped then the time received with the sensor data is assumed to be correct.

- If the agent thinks the clock is stopped but the time received from the sensor data is different from the current time then assume the agent is wrong and the clock is started again.

- If the clock is stopped then the agent assumes a new cycle starts with a `sense_body` message, and therefore updates the time.

- A new cycle starts when the current time is greater than the time of the last cycle.

Based on the assumptions above the following algorithm was developed, the only addition is the number of ticks of the internal clock that are counted from a visual or a physical sensor event. They are used to predict how much time is remaining before the next visual sensor data and how much time the agent has left in this cycle.

Algorithm 3.3 Update the current time based on current server time.

```
begin
    if ( not clock stopped ) then
        Reset estimated time
    else if ( last known server not equal new server time ) then
        Assume the clock is running
    Update last known server time with the new server time
    Update the current time with the new server time
    switch ( last sensor type ) begin
    case Physical sensor:
        if ( clock is stopped and last sense time is equal to current server time ) then
            Update estimated time since clock was stopped
        Reset ticks since last physical sensor data
    case Visual sensor: Reset ticks since last visual sensor data
    end
end
```

3.3.3 The sensors

The purpose of the sensors are to parse the messages sent by the server and store the intermediate result for further processing either by some user-defined unit or, if the framework is used, by the views. The parser is an adapted version of the parser used in CMUnited-98 [34].

3.3.4 The sensor interface

To hide the implementation of the sensors there is a sensor interface. There are actually two different sensor interfaces, the first, called the basic sensor interface, should be used when only using the basic system and the second, called the sensor interface, should be used when the information processing framework is used.

The basic sensor interface contains methods for accessing all the raw data produced by the sensors. If the information processing framework is used then the sensor interface is also responsible for telling the views when new data is available from the sensors. How this is done is described in section 3.4.2.

3.3.5 The actuators

The actuators are designed to provide an interface between the agent and the server when it comes to acting. It takes special command objects, there is one class for each type of command available to the agent, as input and converts them to something executable by the server. The actuators are called by the controller when it is time to send a command. The commands available are described in Table 3.1.

Since there are two basic types of commands, those that can be executed immediately (immediate commands) and those that only one can be executed every cycle (periodic commands), the actuators maintain two separate queues, one for each type of command.

3.3.6 The actuator interface

To hide the implementation of the actuators, in the same way as with the sensors, there is an actuator interface the agent should use to send commands to the actuators.

3.3.7 The decision maker

The decision maker is the unit that implements the decision making process of the agent. This is the unit the agent designer will extend to program the behavior of the agent.

Since the second assumption about the user is that she wants to use as much time as possible to do the decision making, the main loop of the agent is inside the decision maker. To help the user, the controller generates events when important things happen, like new sensor data arrives or when it is urgent to decide on a command for this cycle (because of the third assumption about the user) before it is too late. The decision maker also has limited possibilities to give feedback to the controller by changing some of its parameters like how late in the cycle the decision maker wants the command warning event.

The events the decision maker has to handle are described in Table 3.2. Since the events are generated from the critical section of the controller's signal handler the decision maker should not use too much time in the event handlers, since there will be no other interrupts while in the critical section. Otherwise it can lead to serious problems with the synchronization with the server and the the estimation of the current time.

Table 3.1: The actuator commands available to the agent.

Command	**Description**	**Periodic**
`Bye`()	Disconnect the agent	No
`Catch`(*direction*) *direction* ∈[`minmoment` : `maxmoment`]	Tries to catch the ball in direction *direction*	Yes
`ChangeView`(*view_width*, *view_quality*) *view_width* ∈{`narrow`, `normal`, `wide`} *view_quality* ∈{`normal`, `high`}	Change the current view width and view quality of the agent	No
`Dash`(*power*) *power* ∈[`minpower` : `maxpower`]	Accelerates the agent based on the *power*	Yes
`Init`(*team_name*, *version*, *goalie*) *team_name* a string *version* a string *goalie* true or false	Connect initialize the agent	No
`Kick`(*power*, *direction*) *power* ∈[`minpower` : `maxpower`] *direction* ∈[`minmoment` : `maxmoment`]	Tries to kick the ball in *direction* with *power*	Yes
`Move`(x,y) x ∈[$-52.5 : 52.5$] y ∈[$-34 : 34$]	Moves the agent to absolute position (x, y) if it is allowed to use `move`	Yes
`Say`(*message*) *message* a string of max `msg_say_size` characters	The player screams out *message* loud and clear	No
`Turn`(*moment*) *moment* ∈[`minmoment` : `maxmoment`]	Turn the agent *moment* degrees to the right	Yes
`TurnNeck`(*moment*) *moment* ∈[`minneckmoment` : `maxneckmoment`]	Turn the neck of the agent *moment* degrees to the right	No

Table 3.2: The events available to the decision maker.

Event	**Generated**
BeforeTick	Before a tick of the internal clock
AfterTick	After a tick of the internal clock
BeforeSensorData	Before receiving sensor data
AfterSensorData	After receiving sensor data
ActuatorSensorData	After receiving actuator feedback
AuralSensorData	After receiving aural sensor data
PhysicalSensorData	After receiving physical sensor data
VisualSensorData	After receiving visual sensor data
Init	After receiving init data
SensorError	After sensor errors
NewCycle	When a new cycle has started
EstimatedNewCycle	When the controller estimates that a new cycle has started
CommandWarning	When it is time to send a command before it is too late
DelayedActions	When a sense_body indicates that more actions were performed than was sent the previous cycle
MissingActions	When a sense_body indicates that less actions were performed than was sent the previous cycle

It should be possible to use any type of agent architecture with this decision maker, whether it is a reactive, deliberative or hybrid architecture. The deliberate agent can use only the main loop to guide the behavior of the agent, the reactive agent can use only the events and finally the hybrid agent can use both the events and the main loop to control the behavior of the agent.

3.4 The framework

On top of the basic system it is possible to use the framework designed to support and modularize the world modeling and the support for decision making discussed in section 2.2. The purpose is to provide building blocks for the user, who either can use the existing blocks as they are or extend them with new functionality. It is also possible to create completely new blocks. There are three kinds of building blocks, views, skills and predicates. They each encapsulate an important concept used in the information processing and the decision making. A view is a way of looking at and extracting information, focusing on some special property or object, from a collection of data which is dependent either on the current time or on the history of the agent. A skill is a complex action, combined of several primitive actions, that will take an agent towards a certain goal state. A predicate works like a predicate defined in a three-valued logic. It is used to answer yes or no questions about the state of the world. They are supposed to be used as conditions for rules used in either the decision maker or in the skills.

This section discusses the only concrete unit in the framework, the view manager, and the three different building blocks, or concepts, that it tries to capture. The following sections describes instances of the different concepts, the predefined building blocks available in more detail and how they are used to supply an agent with a basic world model.

3.4.1 The view manager

The purpose of the view manager is to store and keep track of all the views in the system and to make sure they are updated when new information arrives. The view manager is not responsible for the creation of the views. This is done by the unit where they are needed, when they are needed. There are two ideas behind this. The first idea is that it should be possible to have hundreds of views in the system but they should only use resources when they are actually used and it should not be necessary to know this at compile time. The second idea is that the actual views in the system do not need to be known by the framework or the basic system, they only need to be known to the users of the views. To support the first idea the view manager keeps a reference counter for each view it stores. When a view is needed, a request for the pointer to this view is sent to the view manager.

If the view is stored, then the pointer is returned and the reference count is increased. If the view is not stored then a null-pointer is returned and the user has to create the view and send it to the view manager which stores it and sets the reference counter to 1.

When a view is no longer needed a release view message is sent to the view manager, which will decrease the reference counter. If the reference counter reaches zero then the view is removed from the system, unless the persistent flag is set. The persistent flag is used to prevent a view from being removed from the system every time its reference counter reach zero. The reason for this feature is if a view which continuously keeps track of historic data is not used continuously or if you have a view which takes a lot of resources when it is started but not very much resource when it is running.

When the view manager gets a request to update the views after new sensor data have been received it will go through the current list of views in its storage and send an update message to each of them. Since each view has the possibility to request that another view should update itself, a counter is used to prevent the view to be updated twice with the same information. It is also possible for the views to detect circular dependencies, which can not be handled with the current design.

3.4.2 The views

The view is the basic component of the information processing in RoboSoc. With information processing I mean transforming the data from the sensors together with information from other views to more detailed or specialized information and storing it. This information is used to build the agent's world model. Each view should (but does not have to) be specialized in some area, like modeling the ball, a certain skill or some other subject needed by the decision maker, a skill or another part of the agent.

An important issue is to minimize the redundancy of information. To make this possible each view can have links to other views containing already processed and stored information. To solve the problem with dependencies each update contains a unique number so that the view knows if it has been updated already, this also makes it possible to discover circular dependencies, as described above.

The views are, like the decision maker, controlled by events. The sensor interface will generate events whenever new sensor data is available or something else occurred that the views should know about, like a new cycle started. All the events are described in Table 3.3.

Table 3.3: The events the views needs to handle.

Event	Generated
NewCycle	After a new cycle has started
UpdateAfterInit	After the agent has been initiated
UpdateAfterSee	After visual sensor data has been received
UpdateAfterHear	After aural sensor data has been received
UpdateAfterSense	After physical sensor data has been received
UpdateAfterCommand	After feedback from the actuators has been received

Information processing

To provide the basic information processing needed by most RoboCup agents, RoboSoc comes with a set of views that will build up a world model consisting of a model of the ball, a model of the agent, a model of the other players and a model of the markers on the soccer field. Each of these models are put in its own view. There are also three more views: the parameter view which keeps track of the server parameters and optional client parameters; the command view which is used to calculate the total effects of the actions executed by the agent; and the game view which keeps track of the current game. Each of the views mentioned above are described in the following sections.

In Figure 3.4 the dependency between the views are shown. To overcome the problem with circular references, some of the views use the previous cycle's data instead of the current data. This is possible since each of the views keep historic data from the last cycles' in case a need for them are found. They could for example be used for calculating a trajectory of an object based on the previous observations of it.

The parameter view

The parameter view keeps track of all the Soccer server parameters, described in section 2.1.1, and a few other important, user defined, parameters like the team name and whether the agent is the goalie or not. It is also possible for the user to define her own parameters and let the parameter view, or rather an extended version of the parameter view, take care of them. The values for the parameters can either be supplied from text files containing them or from the command line. If no value for a parameter is supplied a default value is used. It is possible for the user to change the parameters during run-time, even though it is not likely to be necessary. Since the parameter view only contains parameters that are independent of the rest of the agent nothing is done in the event handlers.

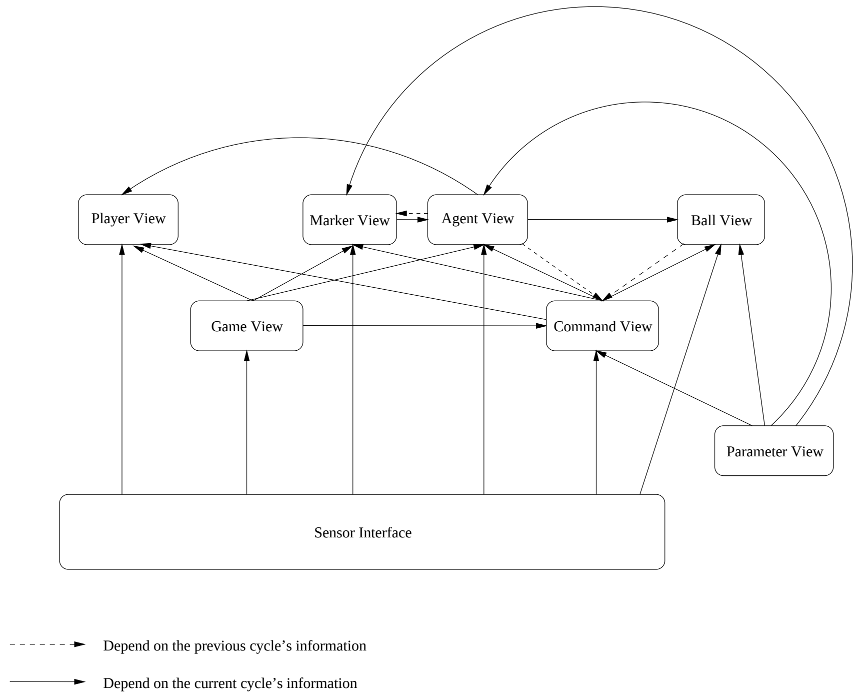

Figure 3.4: Graph showing the dependency between the views.

The command view

The command view calculates the effect of the commands executed the during current cycle. It will give the total movement vector of the agent and the ball for the current cycle. It will also give the total turn angle of the agent and its neck. The available methods are described in Table 3.4.

The game view

The game view stores information about the status of the game, like the play mode, the score and the current time. The complete set of methods is shown in Table 3.5.

Table 3.4: Functionality provided by the command view.

Name	**Return type**	**Description**
GetAgentMovement	A vector	Get the previous cycle's total agent movement vector
GetBallMovement	A vector	Get the previous cycle's total ball movement vector
GetActualTurn	An angle	Get the previous cycle's total agent turn body angle
GetActualTurnNeck	An angle	Get the previous cycle's total agent turn neck angle
GetAgentAcceleration	A vector	Get the previous cycle's total agent acceleration vector
GetBallAcceleration	A vector	Get the previous cycle's total ball acceleration vector
GetActualDashPower	A number	Get the dash power used the previous cycle

Table 3.5: Functionality provided by the game view.

Name	Return type	Description
GetCurrentTime	A time object	Get the current time according to the agent
GetPlayMode(hist)	A play mode	Get the play mode for hist cycles ago
GetOurSide	A side	Get the side of the field the agent is playing on
GetTheirSide	A side	Get the side of the field the agent is not playing on
GetOurScore	A number	Get the score for the agent's team
GetTheirScore	A number	Get the score for the other team
GetOurTeamName	A string	Get the name of the agent's team
GetTheirTeamName	A string	Get the name of the other team

Table 3.6: Functionality provided by the marker view.

Name	Return type	Description
GetNumberOfSeenLines	A number	Get the number of lines seen in the last see message
GetClosestSeenLine	A line object	Get the closest line last seen
GetLine(line, hist)	A line object	Get the line *line* as seen *hist* cycles ago
GetLine{L,R,T,B}(hist)	A line object	Get the line object as seen *hist* cycles ago
GetClosestSeenMarker	A marker object	Get the closest marker seen in the last see message
GetMarker(marker, hist)	A marker object	Get the marker *marker* as seen *hist* cycles ago
GetClosestMarkerTo(point, hist)	A marker object	Get the closest marker to *point* as seen *hist* cycles ago
GetClosestGoalTo(point, hist)	A marker object	Get the closest goal to *point* as seen *hist* cycles ago
GetTheirGoal(hist)	A marker object	Get the opponents' goal as seen *hist* cycles ago
GetOurGoal(hist)	A marker object	Get the agent's team's goal as seen *hist* cycles ago
GetGoal{L,R}(hist)	A marker object	Get the goal object as seen *hist* cycles ago
GetFlag*Name*(hist)	A marker object	Get the marker object as seen *hist* cycles ago

The marker view

The marker view stores information about all the markers on the field. It will give absolute position, relative distance and direction to each marker. In Table 3.6 all the methods are described.

The agent view

The agent view keeps track of the agents status, i.e. its absolute position, stamina, effort, recover, absolute face direction, body and neck directions. In Table 3.7 all the methods are described.

Table 3.7: Functionality provided by the agent view.

Name	**Return type**	**Description**
GetAgentObject(hist)	An agent object	Get the object representing the agent *hist* cycles ago
GetAgentPosition(hist)	A point	Get an estimation of the position of the agent *hist* cycles ago
GetFaceDirection(hist)	An angle	Get an estimation of the facing direction of the agent *hist* cycles ago
GetBodyDirection(hist)	An angle	Get an estimation of the body direction of the agent *hist* cycles ago
GetNeckDirection(hist)	An angle	Get an estimation of the neck direction of the agent *hist* cycles ago
GetEffort(hist)	A number	Get an estimation of the effort of the agent *hist* cycles ago
GetRecover(hist)	A number	Get an estimation of the recover of the agent *hist* cycles ago
GetStamina(hist)	A number	Get an estimation of the stamina of the agent *hist* cycles ago
IsGoalie	A boolean	Return true if the agent is the goalie otherwise false
GetUniformNumber	A number	Get the uniform number of the agent
GetSpeed(hist)	A number	Get an estimation of the speed of the agent *hist* cycles ago
GetViewWidth(hist)	A view width	Get the view width of the agent *hist* cycles ago
GetViewQuality(hist)	A view quality	Get the view quality of the agent *hist* cycles ago
GetViewAngle(hist)	An angle	Get the width of view cone of the agent *hist* cycles ago
GetDashes(hist)	A number	Get the number of dashes the agent has made up to *hist* cycles ago
GetKicks(hist)	A number	Get number of kicks the agent has made up to *hist* cycles ago
GetSays(hist)	A number	Get the number of says the agent has made up to *hist* cycles ago
GetTurns(hist)	A number	Get the number of turns the agent has made up to *hist* cycles ago
GetTurnNecks(hist)	A number	Get the number of turn necks the agent has made up to *hist* cycles ago

Table 3.8: Functionality provided by the ball view.

Name	Return type	Description
GetBallObject(hist)	A ball object	Get the object representing the ball *hist* cycles ago
GetBallDistance(hist)	A float	Get the distance to the ball *hist* cycles ago
GetRelativeBallVector(hist)	A vector	Get the relative vector to the ball *hist* cycles ago
GetRelativeBallDirection(hist)	An angle	Get the relative direction to the ball *hist* cycles ago
GetAbsoluteBallVector(hist)	A vector	Get the absolute vector to the ball *hist* cycles ago
GetAbsoluteBallDirection(hist)	An angle	Get the absolute direction to the ball *hist* cycles ago
GetBallPosition(hist)	A point	Get the absolute position of the ball *hist* cycles ago
GetBallDistChanged(hist)	A float	Get the distance change of the ball *hist* cycles ago
GetBallDirChanged(hist)	An angle	Get the direction change of the ball *hist* cycles ago
GetBallAbsoluteSpeedVector(hist)	A vector	Get the absolute speed vector of the ball *hist* cycles ago
GetBallRelativeSpeedVector(hist)	A vector	Get the relative speed vector of the ball *hist* cycles ago
GetBallSpeed(hist)	A float	Get the speed of the ball *hist* cycles ago
GetBallAbsoluteSpeedDirection(hist)	An angle	Get the absolute direction of the speed of the ball *hist* cycles ago
GetBallRelativeSpeedDirection(hist)	An angle	Get the relative direction of the speed of the ball *hist* cycles ago

The ball view

The ball view keeps track of the ball, its absolute position, relative distance and direction and a very rough estimation of its speed. In Table 3.8 all the methods are described. All relative angles are relative to the facing direction of the agent, taken from the agent view.

The player view

The player view keeps track of the other players. It will give access to three structures: teammates, opponents and unknown players. Each will give information about the absolute position, relative distance and direction, body direction, face direction and uniform number (not available for the unknown players). In Table 3.9 all the methods are described. All relative angles are relative to the facing direction of the agent, taken from the agent view.

Table 3.9: Functionality provided by the player view.

Name	Return type	Description
GetClosestPlayer(hist)	A pointer to a player object	Get the closest seen player *hist* cycles ago
GetClosestOpponent(hist)	A pointer to a player object	Get the closest seen opponent seen *hist* cycles ago
GetClosestTeammate(hist)	A pointer to a player object	Get the closest seen teammate seen *hist* cycles ago
GetOpponents(hist)	A vector of player objects	Get the opponents seen *hist* cycles ago
GetTeammates(hist)	A vector of player objects	Get the teammates seen *hist* cycles ago
GetUnknownPlayers(hist)	A vector of player objects	Get the unknown players seen *hist* cycles ago

Table 3.10: The truth-tables used by the predicates.

	$\neg$	$\wedge$	**t**	**f**	**u**	$\vee$	**t**	**f**	**u**
t	f	**t**	t	f	u	**t**	t	t	t
f	t	**f**	f	f	f	**f**	t	f	u
u	u	**u**	u	f	u	**u**	t	u	u

3.4.3 The predicates

The purpose of the predicates is to encapsulate the concept of a predicate, in a three-valued logic definition of the concept. In other words one has a static feature the predicate should describe, like "is the ball on our half of the field?", with or without parameters, that can either be true, false or unknown. The truth-tables used comes from Kleene [15] and are shown in Table 3.10. The predicates are not automatically updated when new information is available, but they can use information stored in the views and thereby get access to historic data.

3.4.4 The skills

The skills encapsulate the concept of acting, or rather short term specialized planning, or to use the current world model as provided by the views and the predicates to derive a sequence of primitive actions that will make the agent perform a certain task or take it towards an intended goal state.

Every skill should do two things, first of all it should be able to determine if it is applicable at a given moment or if the goal state is already reached or some of its preconditions are not fulfilled. It should also be able to generate a plan, a sequence of primitive actions, consisting of at least one primitive action. The primitive actions are the actions the server can execute, described in section 3.3.5.

To support the decision maker each skill has the possibility to set its own persistence flag, to tell the decision maker that it wants to be called again. This can be used if a skill wants to do a sequence of actions but is only allowed to return one primitive action each time it is called. Which is the case with all the basic skills provided. The reason is efficiency and the fact that the world is highly dynamic and changes significantly from cycle to cycle.

The following skills are currently implemented in RoboSoc. They are not really intended to be used in a real team, but rather act as examples of what skills can look like.

CatchBall

Purpose: To catch the ball
Arguments: None
Preconditions: The ball is catchable and the agent is the goalie
Algorithm: Catch in the direction of the ball
Limitations: It does not take the movement of the ball or the agent into account

FindBall

Purpose: To find the ball
Arguments: None
Preconditions: None
Algorithm: `if` the direction to the ball is known
`then` turn towards the ball
`else` turn 45 degrees to the right
Limitations: It does not take the time between the visual sensor data into account, it does not use previous knowledge of the ball

InterceptBall

Purpose: To intercept the ball
Arguments: None
Preconditions: The distance and direction to the ball is known
Algorithm: `if` the direction to the ball is greater than 5 degrees
`then` turn towards the ball
`else` dash with 50% of the maximum power
Limitations: It does not take obstacles or the movement of the ball into account

MoveTo

Purpose: To move to the absolute position (X, Y)
Arguments: X and Y
Preconditions: The position of the agent is known
Algorithm: `if` allowed to use the move command

```
then Move(X, Y)
else if the direction to the point(X,Y) is greater than 5 degrees
     then turn towards the point
     else dash with 50% of the maximum power
```

Limitations: It does not take obstacles into account

Score

Purpose: To kick the ball into the opponents' goal
Arguments: None
Preconditions: The ball is kickable
Algorithm: Kick the ball as hard as possible towards the center of the opponents' goal
Limitations: It does not take obstacles nor the agent's body into account when kicking

TrackBall

Purpose: Follow the motion of the ball
Arguments: None
Preconditions: The direction to the ball is known
Algorithm: Turn towards the ball
Limitations: It does not take motion of the ball into account

3.5 Educational value

As stated previously in this chapter the RoboSoc system have been used in the AI programming course given by the department of computer and information science at Linköping university. For more information about the course, look at its web page [1]. The main goal of the course is for the students to create their own RoboCup teams and then compete against each other at the end. Previous years the server parameters were adjusted to remove some problems regarding the timing of the agent to make it easier to make good teams. The first benefit of the RoboSoc system was that students could use the standard parameters and still get better teams, since RoboSoc takes care of the timing of the agent. Another benefit was that the students could focus on the problem of developing the skills and the decision making for their agents instead of having to implement the server communication and the processing and storing of information sent by the server.

Most students felt that it was easy to get started when using RoboSoc and that it was easy to implement their own ideas. The major problem reported was the implementation of the modifiers, which is not very user friendly at the moment. The problem has to do with the way C++ handles types and type conversions.

Chapter 4

Conclusions

4.1 Summary

This report discusses RoboSoc, a system for developing RoboCup agents suitable for educational use. It is designed to be as general, open, and easy to use as possible and to encourage and simplify the modification, extension and sharing of RoboCup agents, and parts of them. To do this I assumed four requirements from the user: she wants the best possible data, use as much time as possible for the decision making, rather act on incomplete information than not act at all, and she wants to manipulate the objects found in the soccer environment.

In order for the system to be useful from a student's point of view I stated four essential problems the system has to solve or at least support. It should take care of the basic interactions with the Soccer server, do the timing, have support for different world models, and have support for decision making.

The resulting system consists of three parts, the library, the basic system and the framework. The library consists of basic objects and utilities used by the rest of the system, and is not dependent on any other part of the system. The basic system takes care of the interactions with the server, like sending and receiving data. It is also responsible for the timing and most of the decision making support by generating events when new things happen. The basic system is only depending on the library and can be used without the framework. The framework defines three concepts, used for world modeling and decision support, views, predicates and skills. The views are specialized information processing units responsible for a specific part of the world model, like modeling the ball or the agent. They are also controlled by events generated by the basic system. The predicates can be used either by the decision maker or the skills to test the state of the world. They work like predicates in a three-valued logic, and can either be true, false or unknown. The skills are specialized, short-term planners which generate plans for what actions the agent should take in order to reach a desired goal state. The framework is depending on both the library and the basic system.

In [17] Jörg Müller argue that an agent needs five basic capabilities to cope with difficult tasks. The agent needs to be:

- **Reactive:** it should react timely and appropriately to changes in the environment, even unforeseen changes.
- **Deliberative:** it should be able to perform tasks in a goal-directed manner.
- **Efficient:** it should be able to solve its tasks efficiently by using hard-coded procedures in routine situations.
- **Interactive:** it should be able to interact with other agents.
- **Adaptive:** it should be able to adapt to a changing environment and to cope with unforeseen events.

Since the computational resources available to the agent is limited a central task is to "define a control architecture for resource-bounded agents, which allows the designer of an agent-based system to integrate the requirements mentioned above, and to define the trade-offs between them in a way that is adequate for the application domain under consideration" [17]. RoboSoc is such a control architecture. Since it generates events when the environment changes, the agent programmerer has the opportunity to interrupt the current deliberative decision process and react to the event. At the same time the implementation is efficient enough and the skill framework provides a way to define hard-coded procedures. Other types of reactive procedures, like RAPs [10] or CONTAP [9], can also be incorporated by the user without too much trouble. There is currently only very limited support for adaptation consisting of the history of the world from the last cycles provided by the information processing framework which can be used for machine learning or other adaptation mechanisms. The only thing that is not explicitly supported is the interactive capability, but at the same time there are no major obstacles if the user wants to add communication capabilities to their agents.

The contributions of this work is mainly an architecture and an infrastructure which makes it easier to develop RoboCup agents that is actually implemented and working. It takes care of all the low-level details and let the user focus on the more interesting AI-parts of the agent. It also gives the users the possibility to share different parts of their agents, by providing a framework for the most common concepts used in RoboCup.

Since more and more educational institutes are starting to use RoboCup as part of their curriculum I think this system can play an important role in the promotion of RoboCup, and AI, by making it more accessible to everyone.

4.2 Future work

There is a vast number of things that would improve RoboSoc. The following paragraphs each discuss one area where improvements are possible.

Improve the existing basic system

The most needed improvement to the basic system is to make it a distributed application with different units running in separate threads. The benefit would be better performance and a nicer computational model. At the same time making it less machine dependent would make it easier to transfer it to other platforms. Currently RoboSoc will run on most unix flavors with minor changes, but all operating systems that support sockets and some type of signals should be able to run it after modifications.

Improve the existing framework

The most obvious improvement is to improve the views, skills, and predicates that are included. They could be made to deliver more accurate information and be made to do more advanced geometric calculations.

New views, skills and predicates are always welcome and I hope that users of RoboSoc will contribute with theirs to advance the state of the art of RoboCup teams. This could help move RoboCup from the hacking stage to the stage where all teams have almost the same level of basic functionality when it comes to individual players. Then more scientific methods becomes more and more important. One will need a team that has models of the opponents and uses team tactics and plays to be able to win. Today one can win most games by only having good individual players.

Another interesting experiment would be to connect existing specialized software to RoboSoc to take care of certain tasks like planning or prediction.

Other frameworks

The three frameworks that exist today are the most obvious, and necessary, ones for creating a functional RoboCup agent. But adding other frameworks is possible. I can think of at least four other useful frameworks: a role framework, a communication framework, a coach framework, and a machine learning framework. The role framework is probably the most wanted one since most teams use some sort of roles to divide the tasks between the agents today. Otherwise it is interesting to wait and see where the development of RoboCup is going before taking a concept on as one of general interest.

It is also possible to create completely separate frameworks, that do not use the current ones, for example support for genetic programming.

Development tools and debugging

One direction of development which could be more fruitful than to add more frameworks is to provide a set of development tools that can help the developers with the creation of the agents. Tools that are of interest are for example the layered disclosure tool provided by CMUnited [32], and tools that graphically show what the agent currently believes in.

Regarding debugging I see two possible types of debugging. The first type is the retrospective analysis like the logplayer from CMU, where you first store data from a game and then watch the game and inspect the actions of the agents afterwards. The second type is the interactive analysis and maybe even pro active analysis. The interactive analysis is when the current state of the agent can be inspected during a game, and where you can step through the execution of the agent and the whole game. The major problem for the interactive analysis is the support for the step feature within the current Soccer server. The whole area of developing support for debugging of real-time, multi-agent systems is very interesting.

A more realistic debug feature could be view-servers, used by RoboLog [23] and in some sense also the Headless chickens from Linköping [7]. A view-server is a server you connect a, running, agent to which then use the data sent to it by the agents to visualize the current state of the agent, or use it for inspection or debugging purposes.

I hope to be able to continue working on RoboSoc and add some of the improvements discussed above. The next event will be to create a team with RoboSoc that will compete in the Swedish and European championships in May.

Appendix A

The user's manual

A.1 Introduction

This is a preliminary version of the user's and the reference manual for the RoboSoc system for developing RoboCup agents. It consists of four sections. This first section contains a short introduction to the software and the following three sections described each of the three modules that make up the complete RoboSoc system.

For the latest information about the current release of RoboSoc, look at the `README` file included in the latest release. For information about how to install the software look in the `INSTALL` file included in the release. To download the latest version of RoboSoc, including the latest version of the manual, or to get more information about the development of it look at the RoboSoc web-site [24].

A problem with the current release is that RoboSoc uses namespaces, because of a nameclash between the Unix socket management and the standard template library. Therefore your compiler needs to support namespaces, for example gcc 2.95 and CC 5.0 does.

A.1.1 Overview of the software package

The RoboSoc system consists of three different parts, the library, the basic system and the framework. You can either use only the library, the library and the basic system, or the whole system. Therefore there are three software packages available the RoboSoc library, the RoboSoc basic system, and the RoboSoc framework package. Each of them contains all the files needed to use the software. Each of the three packages are described in more detail in the following sections.

A.1.2 General parts

There are two files included in each of the packages, the compiler directives and the types for the package. The compiler directives are flags that are set to control the compilation of the package. The types are those types that are used by the package. There is also an extra set of compiler directives and types in the library package for machine dependent compiler directives and types. What directives and types included in each package are described in the section describing that package.

A.2 The library

This section will describe the functionality and the use of the library module of the RoboSoc system.

A.2.1 How to use the library?

The library is not a single unit but rather a collection of classes. To use the library you therefore have to include the header file for the wanted class. The names of the include files are usually the name of the class with the extension `.h`. The definitions of the classes are in a file with the same name as the include file, but with the extension `.cc`.

A.2.2 Compiler directives

The following directives can be defined in `library_compiler_directives.h`:
`_TRIG_IN_DEG_`: If trigonometry functions should return angles in degrees.
`_TRIG_IN_RAD_`: If trigonometry functions should return angles in radians.
`_USE_EXCEPTIONS_`: If exceptions should be used.
`_USE_DEBUG_`: If the debug features should be used.

Only one of the first two directives can be defined, otherwise the compilation will fail.

A.2.3 Data types

RoboSoc contains many different data types used to encapsulate both machine dependent and implementation dependent data types. The machine dependent data types are shown in Table A.1, and they are all defined in `machine_dependent_types.h`. The definition of these must be changed to match the size of the data types on the local system.

There are also three basic number types that have an unspecified maximum size, but a specified minimum size, since they are used in such a way that the size of the type mainly affects the precision of the calculation, not the range of the values. The three types are shown in Table A.2.

Table A.1: The basic, machine dependent, data types available in RoboSoc.

Type	Signed	Size (bytes)	Description
t_uint16	No	2	Unsigned 16-bit integers
t_uint32	No	4	Unsigned 32-bit integers
t_uint64	No	8	Unsigned 64-bit integers
t_int16	Yes	2	Signed 16-bit integers
t_int32	Yes	4	Signed 32-bit integers
t_int64	Yes	8	Signed 64-bit integers
t_float32	Yes	4	Signed 32-bit floats
t_float64	Yes	8	Signed 64-bit floats

Table A.2: The basic, precision dependent, data types available in RoboSoc.

Type	Signed	Min size (bytes)	Description
t_uint	No	2	Unsigned integers with at least 16 bits
t_int	Yes	4	Signed integers with at least 32 bits
t_float	Yes	4	Signed floats with at least 32 bits

RoboCup types

The RoboCup specific enumeration types are shown in Table A.3 and are all defined in `library_types.h`.

RoboSoc templates

RoboSoc contains a few templates that are used to represent angles, both in radians (AngleRad<*type*>) and in degrees (AngleDeg<*type*>), and coordinates, both vectors (CoordVector<*type*>) and points (CoordPoint<*type*>). The types in the templates are used to represent basic values, like the angle or the x and y coordinate.

There is also a template for representing collections of objects called ObjectCollection<*object-type*>, the object-type is the type of the object in the collection.

RoboSoc types

The RoboSoc specific types are shown in Table A.4 and are all defined in `library_types.h`. There are also some named instantiations of templates defined in `library_types.h`, shown in Table A.5.

Modifiers

There are three types of modifiers. Each with its own feature. The three modifiers are with unknown, with confidence and with observation time.

Table A.3: Data types for RoboCup concepts available in RoboSoc.

Type	**Values**	**Description**
t_view_width	VW_Normal, VW_Wide, VW_Narrow, VW_Unknown	A type for representing the concept of view width
t_view_quality	VQ_High, VQ_Low, VQ_Unknown	A type for representing the concept of view quality
t_cmd_type	CMD_Unknown, CMD_Bye, CMD_Catch, CMD_ChangeView, CMD_Dash, CMD_Init, CMD_Kick, CMD_Move, CMD_Say, CMD_SenseBody, CMD_Turn, CMD_TurnNeck	A type for representing the available server commands
t_side_line	SL_Left, SL_Right, SL_Top, SL_Bottom, SL_Unknown	A type for representing the side lines
t_marker	Goal_L, Goal_R, Flag_C, Flag_CT, Flag_CB, Flag_LT, Flag_LB, Flag_RT, Flag_RB, Flag_PLT, Flag_PLC, Flag_PLB, Flag_PRT, Flag_PRC, Flag_PRB, Flag_GLT, Flag_GLB, Flag_GRT, Flag_GRB, Flag_TL50, Flag_TL40, Flag_TL30, Flag_TL20, Flag_TL10, Flag_T0, Flag_TR10, Flag_TR20, Flag_TR30, Flag_TR40, Flag_TR50, Flag_BL50, Flag_BL40, Flag_BL30, Flag_BL20, Flag_BL10, Flag_B0, Flag_BR10, Flag_BR20, Flag_BR30, Flag_BR40, Flag_BR50, Flag_LT30, Flag_LT20, Flag_LT10, Flag_L0, Flag_LB10, Flag_LB20, Flag_LB30, Flag_RT30, Flag_RT20, Flag_RT10, Flag_R0, Flag_RB10, Flag_RB20, Flag_RB30, Unknown_Marker, Unknown_Goal, Unknown_Flag	A type for representing the markers

Table A.4: Data types for RoboSoc concepts available in RoboSoc.

Type	Values	Description
t_side	S_Unknown, S_Left, S_Right	A type for representing the sides of the field
t_team	T_Unknown, T_Our_Team, T_Their_Team	A type for representing the teams
t_play_mode	PM_Unknown, PM_Before_Kick_Off, PM_Time_Over, PM_Play_On, PM_Drop_Ball, PM_Offside_Kick, PM_Our_Offside_Kick, PM_Their_Offside_Kick, PM_Half_Time, PM_Time_Up, PM_Extended_Time, PM_Kick_Off, PM_Our_Kick_Off, PM_Their_Kick_Off, PM_Kick_In, PM_Our_Kick_In, PM_Their_Kick_In, PM_Free_Kick, PM_Our_Free_Kick, PM_Their_Free_Kick, PM_Corner_Kick, PM_Our_Corner_Kick, PM_Their_Corner_Kick, PM_Goal_Kick, PM_Our_Goal_Kick, PM_Their_Goal_Kick, PM_Goal, PM_Our_Goal, PM_Their_Goal, PM_Goalie_Got_Ball, PM_Our_Goalie_Got_Ball, PM_Their_Goalie_Got_Ball	A type for representing the possible game modes

Table A.5: Named instatiations of templates available in RoboSoc.

Type	**Instatition**
t_uint16_u	WithUnknown<t_uint16>
t_int32_u	WithUnknown<t_int32>
t_float32_u	WithUnknown<t_float32>
t_float64_u	WithUnknown<t_float64>
t_int_unknown	WithUnknown<t_int>
t_uint_unknown	WithUnknown<t_uint>
t_float_unknown	WithUnknown<t_float>
t_float_uco	WithUCO<t_float, CTimeUnknown>
CAngleDeg	AngleDeg<t_float>
CAngleDegUnknown	AngleDegUnknown<t_float>
CAngleDegUCO	AngleDegUCO<t_float, CTimeUnknown>
CPoint	CoordPoint<t_float>
CPointUnknown	CoordPointUnknown<t_float>
CPointUC	CoordPointUC<t_float>
CPointUCO	CoordPointUCO<t_float, CTimeUnknown>
CVector	CoordVector<t_float>
CVectorUnknown	CoordVectorUnknown<t_float>
CVectorUC	CoordVectorUC<t_float>
CVectorUCO	CoordVectorUCO<t_float, CTimeUnknown>
CVectorCollection	ObjectCollection<CVector>
CAngleDegCollection	ObjectCollection<CAngleDeg>
FloatCollection	ObjectCollection<t_float>

They add the feature of unknown values, a confidence factor and an observation time respectively.

They are implemented as templates but are only tested for basic C++ types and other modifiers instantiated with basic types. The basic idea is to create a new class with an attribute to store a value of the type the template was instatitated with and extra attributes for the feature. The class then implements all the basic arithmetic operators and also some special operators for that modifier. There is also a method called GetValue() which returns the value of the variable, but without the modification. Therefore it will discard information stored in the variable. In some cases it might not even work, if you try to take the value of an unknown variable then it will throw an UnknownValueException. The special operators are described below.

With unknown The "with unknown" modifier makes it possible for a value to be unknown. The special methods available are MakeUnknown(), which makes the variable unknown and return a reference to that variable, and IsUnknown() which returns true if the value is unknown or false otherwise.

With confidence The "with confidence" modifier makes it possible to add a confidence factor to the value. The special methods available are GetConfidence(), which returns the current confidence value, and SetConfidence(CConfidence) which sets the current confidence value.

There is a compiler directive called `_PESSIMISTIC_OBSERVATIONS_` which controlls how confidence values are updated when doing arithmetic with modified values. If it is set then the lowest confidence value is taken, otherwise the highest.

With observation time The "with observation time" modifier makes it possible to add a timestamp to the value. The special methods are GetObservationTime(), which returns the observation time, SetObservationTime(Time), which sets the current observation time, and GetCyclesSince(Time) which returns the number of cycles between observation time and time.

There is a compiler directive called `_PESSIMISTIC_OBSERVATIONS_` which controls how observation times are updated when doing arithmetic with modified values. If it is set then the earliest observation time value is taken, otherwise the latest.

The standard modifiers (WithUnknown<*type*>, WithConfidence<*type*>, and WithObservationTime<*type*, *time*>) does not work with the templates for the angles and coordinates there are special versions of the modifiers available, called AngleDegUnknown<*type*>, for unknown, AngleDegUC<*type*> for both unknown and confidence, and AngleDegUCO<*type*> for unknown, confidence and observation time and similar for AngleRad, CoordVector and CoordPoint.

A.3 The basic system

This section describes the basic system of RoboSoc and how to use it.

A.3.1 How to create an agent?

To create an agent using the RoboSoc basic system, without the framework, you have to create a main function where you start the different parts. In Example A.1 you can see how the code for a basic agent might look like. The function `rs_debug` is used for debugging, and is only needed if the compiler directive `_USE_DEBUG_` defined in `library_compiler_directives.h`. The constructors for the different units are described in the following sections.

The initialization of the agent is done in the CDecision class, and since the base class CDecision does not send any init message to the server you have to write your own decision class. How this is done is described in section A.3.9.

A.3.2 How to use the basic system with the framework?

The only difference is that you have to use the class CSensorInterface instead of CBasicSensorInterface when you start the sensor interface in Example A.1. You also need to start the view manager and use a different decision class, but that is discussed in section A.4.

A.3.3 The controller

The constructor is `CController(CServerInterface* const, CSensors* const, CActuators* const, CDecision* const, const t_uint16 cycle_length, const t_uint16 recv_step, const t_uint16 sense_step, const t_uint16 visual_step)` where *CServerInterface, CSensors, CActuators, CDecsion* is pointers to the current server interface, sensors, actuators and decision objects. The *cycle_length* it the length of a cycle in milliseconds, *recv_step* is the interval between the server polls the sockets, *sense_step* the interval between `sense_body` messages, and *visual_step* the interval between `see` messages. recv_step, sense_step, and visual_step can be found in `server.conf`.

A.3.4 The sensors

The constructor is `CSensors(CBasicSensorInterface*)`, where *CBasicSensorInterface* is a pointer to the sensor interface object.

A.3.5 The sensor interface

The constructor is `CBasicSensorInterface(string& my_team_name)` where *my_team_name* is the name of the team. The string is used to infere which team a player belongs to.

A.3.6 The actuators

The constructor is `CActuators(CActuatorInterface*)`, where *CActuatorInterface* is a pointer to the actuator interface object.

A.3.7 The actuator interface

The constructor is `CActuatorInterface()`.

A.3.8 The decision maker

The constructor is `CDecision(CActuatorInterface* const actuator_interface, CBasicSensorInterface* const sensor_interface)`, where *CActuatorInterface* is a pointer to the actuator interface object and *CBasicSensorInterface* is a pointer to the sensor interface object.

Table A.6: The events available to the decision maker.

Event	**Generated**
BeforeTick	Before a tick of the internal clock
AfterTick	After a tick of the internal clock
BeforeSensorData	Before starting to receive sensor data
AfterSensorData	After receiving sensor data
ActuatorSensorData	After receiving actuator feedback
AuralSensorData	After receiving aural sensor data
PhysicalSensorData	After receiving physical sensor data (`sense_body`)
VisualSensorData	After receiving visual sensor data
Init	After receiving init data
SensorError	After sensor errors
NewCycle	When a new cycle has started
EstimatedNewCycle	When the controller estimates that a new cycle has started
CommandWarning	When it is time to send a command before it is too late
DelayedActions	When a sense_body indicates that more actions were performed than was sent the previous cycle
MissingActions	When a sense_body indicates that less actions were performed than was sent the previous cycle

A.3.9 How to do the decision making?

To add the decision making you have to the following. Create a new class that inherits from CDecision. In that class you have to start the main loop of the program, for an example see Example A.2. Then you have to implement some of the event handlers that the controller will call. The event handlers are described in Table A.6. For an example implementation of a basic agent look at the basic decision class in Example A.3.

A.3.10 How to get the sensor information?

To get the sensor information you have to call the methods in CBasicSensorInterface. They are described in Table A.7. Since all the values in the basic sensor interface is reset at the begining of a cycle, all the data given by is from the current cycle.

A.3.11 How to send commands to the server?

To send a command to the server you have to create a new pointer to the wanted command object class, for example `new CDashCommand(75)` to create a new command `(dash 75)`, then send it to server by using the methods available in the actuator interface described in Table A.8.

Table A.7: Public methods in class CBasicSensorInterface.

Name	Description
NewCycle()	Called by the controller when a new cycle has started
ForceNewCycle()	Called by the controller when a new cycle was forced to start by the user
CTime GetCurrentTime()	Get the current time
ForEachMarker(void (*fn)(const marker_object))	Apply the function *fn* on each marker
t_uint GetNumberOfSeenMarkers()	Get the number of markers in the last `see` message
ForEachLine(void (*fn)(const line_object))	Apply the function *fn* on each line
t_uint GetNumberOfSeenLines()	Get the number of lines in the last `see` message
ForEachPlayer(void (*fn)(const player_object))	Apply the function *fn* on each player
t_uint GetNumberOfSeenPlayers()	Get number of players in the last `see` message
bool SawBall()	Return true if the ball was seen in the last `see` message
t_float_unknown GetBallDistance()	Return the distance to the ball
CAngleDegUnknown GetBallDirection()	Return the direction to the ball
t_float_unknown GetBallDistanceChange()	Return the distance change for the ball
t_float_unknown GetBallDirectionChange()	Return the direction change for the ball
t_view_quality GetViewQuality()	Return the view quality
t_view_width GetViewWidth()	Return the view width
t_float GetStamina()	Return the stamina
t_float GetEffort()	Return the effort
t_float GetSpeed()	Return the speed
CAngleDeg GetHeadDirection()	Return the direction of the head
t_uint GetKicks()	Return the number of kicks made
t_uint GetDashes()	Return the number of dashes made
t_uint GetTurns()	Return the number of turns made
t_uint GetSays()	Return the number of says made
t_uint GetTurnNecks()	Return the number of turn necks made
string& GetOurTeamName()	Return our team name
string& GetTheirTeamName()	Return the other teams name
t_side GetOurSide()	Get the side of our team
t_side GetTheirSide()	Get the side of the other team
t_uint GetUniformNumber()	Get my uniform number
t_play_mode GetPlayMode()	Get the play mode
t_uint GetOurScore()	Get our score
t_uint GetTheirScore()	Get the score of the other team
string GetCoachMessage()	Get the message from the coach
string GetPlayerMessage()	Get the message from another player
CCommand* GetLastCommand()	Get the last command sent to the server

Table A.8: Public methods in class CActuatorInterface.

Name	Description
Init(CActuators*)	Tells the interface what actuators to use called by the actuators
bool AddPlan(CCommand*)	Add a command to the send queue

A.4 The framework

A.4.1 How to create an agent with the framework?

To create an agent using the RoboSoc framework you have to create a main function where you start the different parts. In Example A.4 you can see how the code for a basic agent, using the framework, might look like.
The function `rs_debug` is used for debugging, and is only needed if the compiler directive _USE_DEBUG_ is defined in `library_compiler_directives.h`. The constructors for the different units were described in the last sections.

The initialization of the agent is done in the CDecision class, and since the base class CDecision does not send any init message to the server you have to write your own decision class. How this is done see section A.3.9.

A.4.2 The views

The view is the basic component of the information processing in RoboSoc. With information processing we mean transforming the data from the sensors together with information from other views to more detailed or specialized information and storing it. Each view should (but does not have to) be specialized in some area, like the ball, a certain skill or some other subject needed by the decision process or other parts of RoboSoc.

An important issue is to minimize the redundancy of information. To make this possible each view can have links to other views containing already processed and stored information. To solve the problem with dependencies each update contains a unique number so that the view knows if it has been updated already, this also makes it possible to discover circular dependencies.

How do they work?

The purpose of the views is to process and store information. To make this possible they need to know when new information has arrived and they need to be able to access the raw data from the server and information stored in other views. The data from the sensor interface is available from the class variable CView::sensorInterface. (To make this work you need to assign the address of the sensor interface object to the CView::sensorInterface variable.)

Other views are reachable through the view manager pointer, viewManager, stored in each view.

In order to know when new information is available the views have a method for each type of update in the system. They are:

- **NewCycle** which is called by the view manager when a new server cycle is about to begin. It's called before the UpdateBeforeNewCycle() method of the sensor interface is called. This is the last chance to use any data stored in the sensor interface, afterwards it will be reset.

- **UpdateAfterInit** is called when the init messsage has arrived from the server.

- **UpdateAfterSee** is called when new see information has arrived from the server.

- **UpdateAfterHear** is called when new audio information has arrived from the server.

- **UpdateAfterSense** is called when a new sense body has arrived from the server.

- **UpdateAfterCommand** is called when a command was sent to the server.

What kind of data you can get from the sensor interface and when it is available is described in the previous section.

How are the views updated?

When new information arrives to the sensor interface it will tell the view manager that it is time to update after XXX (which is the type of the information). The view manager will then generate an unique update number and iterate over all the stored views and for each one of them call their UpdateAfterXXX() method with the update number as the argument. This method will check to see if the view has already been updated or not. If it has not been updated then it will call the method MyUpdateAfterXXX(). If the view has been updated nothing happens, if the view detects a circular dependency it will abort the program.

If your view needs updated information from some other view then you need to call its UpdateAfterXXX()-method with the same update number you got, which is stored in the view as currentUpdateNumber.

The update for a new cycle is somewhat different. The NewCycle()-method of a view is called before the sensor interface updates itself for a new cycle, all the other update methods are called after the sensor interface has been updated. When the sensor interface updates itself it will erase all data stored in it. Therefore this is the last chance for the view to use the data stored in the sensor interface for the current server cycle. It is also time for the view to make it ready for a new cycle.

How do I use the views?

First of all you need the wanted view object, or a pointer to it. Then you can call it's public methods and access its public attributes.

To get a pointer to a view object you need to ask the view manager if the view is stored there. This is done by calling the GetView-method. The argument to GetView is the id-number of the view you need. A symbolic constant for this value can usually be found in view_data.h. The GetView-method will return either a NULL-pointer if the view doesn't exist or a pointer to the object if it exist. The problem is that the pointer is of the type CView*, and not the correct view-pointer you need. Therefore you have to use the static_cast template to convert the pointer from one type to another. Eg. to convert v_ptr from a CView* to a CBallView* you do: $v\_ptr = static\_cast < CBallView* > (v\_ptr)$.

How do I create a view object?

First of all, create an instance of the view you need. Then you need to store the view in the view manager. This is done by calling the view manager's AddView()-method with the address of the view and a persistence flag. If the persistence flag is set the view will not be destroyed when no one is using the view, otherwise it will be destroyed when no one is using it. With no one using the view I mean that there are no pointers to the view active in the system. An active pointer to a view is given when you ask the view manager for a view with the GetView()-method and they are deactivated with a call to the ReleaseView()-method. The reason for storing the viewe in the view manager is to make them available to the rest of the system.

How do I extend them?

The basic rule is to create new views which use information from the view you want to extend instead of changing the code of the existing view. If you really would like to extend an existing view, inherit from the view you want to extend and then add the things you miss and redefine the necessary methods.

How do I write my own views?

The best way to add new functionality regarding information processing in RoboSoc is to create a new view. To create a new view all you have to do is to inherit from CView (or any other view that you want to use as your starting point) and write your own update methods. I.e. you need to define what the view should do when new information has arrived and what to do when a new cycle begin. If you need to store data (which is very likely) you will need to create the necessary data structures and initialize them in the constructor, destroy them in the destructor and supply methods to extract and use the information stored in them. If you need access to other views, the best way to do this is to store pointers to those views in the object and initialize them in the constructor. Then you'll know that the views you want are always accessible.

For examples and hints, look at the code for the views that are included in the RoboSoc-package.

What views are included in RoboSoc?

Currently seven views are supplied with RoboSoc version 1.5.0. I hope to be able to add more views as the users of RoboSoc get their views working and sending them to me.
The views are:

- **ParameterView**: stores all the parameters from server.conf. Use them to make your agent independent of the actual server configuration. It also contains methods for parsing the command line. Available options are `-f` *config-file* to read a config file, `-goalie` *true* to start a player as the goalie, and `-team_name` *team-name* to set the team name of the agent.

- **GameView**: which stores information about the status of the game (play mode, score and current time).

- **CommandView**: calculates the effect of the commands executed the current cycle. It will give you the total movement vector of the agent and the ball for the current cycle. You will also get the total turn angle of the agent and its neck.

- **MarkerView**: stores information about all the markers on the field. It will give you absolute position, relative distance and direction to the markers.

- **AgentView**: keeps track of the agents status (absolute position, stamina, effort, recover, absolute face direction, body and neck directions)

- **BallView**: keeps track of the ball (absolute position, relative distance and direction and a very rough estimation of the speed) item **PlayerView**: keeps track of the other players. You will get access to three structures: teammates, opponents and unknown players. Each will give information about absolute position, relative distance and direction, body, face direction and uniform number (if available).

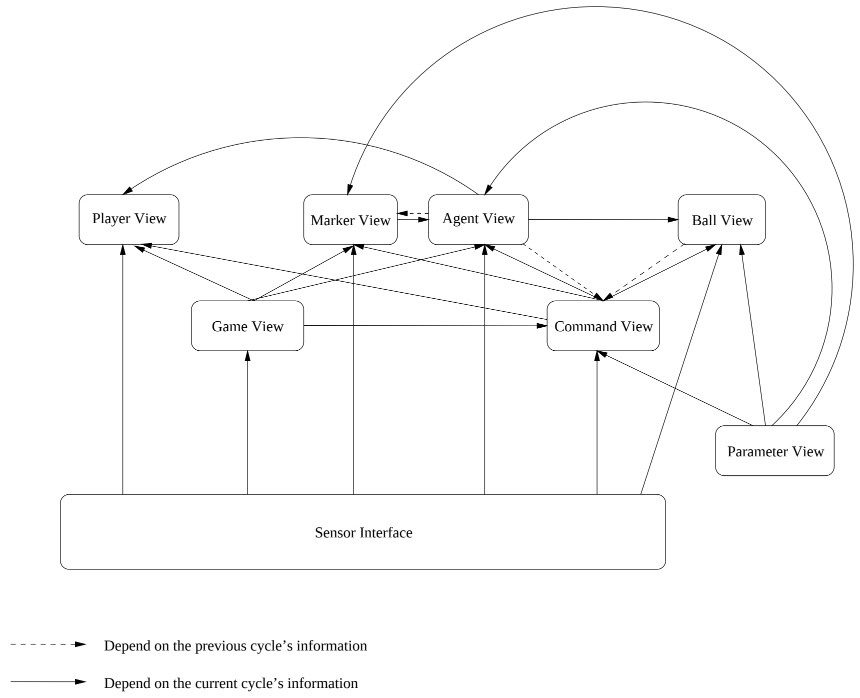

Figure A.1: The dependencies between the views.

How the views depend on each other can be seen in Figure A.1. A more detailed description of these views is available in the section 3.4.2.

A.4.3 How to use the predicates

Create an instance of the predicate you want. Call the Evaluate() method, which will return either true or false.

A.4.4 How to create your own predicates

Create a new class that inherits from the CPredicate class. Implement the bool Evaluate() method.

A.4.5 How to extend existing predicates

Create a new class that inherits from the predicate you want to extend. Call its Evaluate() method if its functionality is useful, otherwise rewrite it from scratch.

A.4.6 How to use the skills

Create an instance of the skill you want. To check whether the skill is applicable call its Applicable() method which will return true if it is applicable otherwise false. To get the next command generated by the skill call its GeneratePlan() method. If the skill wants to be called again the next cycle it will set the persistence flag of the skill. This flag is checked with the Persistent() method.

A.4.7 How to create your own skills

Create a new class that inherits from the CSkill class. Implement the bool Applicable(), bool Persitent() and CCommand* GeneratePlan() methods.

A.4.8 How to extend existing skills

Create a new class that either inherits from the skill you want to extend and then use its Applicable, Persistent and GeneratePlan methods to create the new skill or inherit from some other skill (or CSkill) and create a local instance of the skill you want to extend and calls its Applicable, Persistent and GeneratePlan methods to implement the class.

Example A.1 A basic agent using only the basic system.

```
#include "CController.h"
#include "CActuatorInterface.h"
#include "CBasicSensorInterface.h"
#include "CActuators.h"
#include "CSensors.h"
#include "CDecision.h"

extern void rs_debug(const std::string& str)
{
  std::cerr << "Error: " << str << std::endl;
}

int main(int argc, char** argv)
{
  std::cout << "Starting actuator interface...\n";
  CActuatorInterface actuator_interface;
  std::cout << "Starting actuator...\n";
  CActuators actuators(&actuator_interface);

  std::cout << "Starting sensor interface...\n";
  CBasicSensorInterface sensor_interface("RoboSoc");
  std::cout << "Starting sensors...\n";
  CSensors sensors(&sensor_interface);

  std::cout << "Starting server interface...\n";
  CServerInterface server_interface("localhost", 6000, 2048);

  std::cout << "Starting decision...\n";
  CDecision decision(&actuator_interface, &sensor_interface);

  std::cout << "Starting controller...\n";
  CController controller(&server_interface, &sensors, &actuators,
                         &decision, 100, 10, 100, 150);

  if ( !controller.Init() ) {
    std::cout << "Error in Init!\nExit...\n";
    return 1;
  }

  controller.Start();

  return 0;
}
```

Example A.2 An example of a main loop.

```
void CBasicDecision::Start()
{
  while ( serverAlive )
    if ( pause() == -1 && errno != EINTR )
      std::cerr << "Something went wrong in pause!\n";
}
```

Example A.3 An example of a basic decision maker.

```
CCommand* CBasicDecision::InitAgent() {
  return new CInitCommand(''RoboSoc'', 5, false);
}

void CBasicDecision::Decide() {
  if ( !ballDir.IsUnknown() )
    ballDir -= lastTurn;
  lastTurn = 0;

  if ( sensorInterface->SawBall() ) {
    ballDir = sensorInterface->GetBallDirection();
    ballDist = sensorInterface->GetBallDistance();
    counter = 0;
  } else {
    counter++;
    if ( counter > 4 ) {
      ballDir.MakeUnknown();
      ballDist.MakeUnknown();
      counter = 0;
    }
  }

  if ( ballDir.IsUnknown() ) {
    actuatorInterface->AddPlan(new CTurnCommand(CAngleDeg(30)));
  } else if ( ballDist < 1 ) {
    actuatorInterface->AddPlan(new CKickCommand(100, CAngleDeg(0)));
  } else if ( Abs(ballDir) < 5 ) {
    actuatorInterface->AddPlan(new CDashCommand(50));
  } else {
    actuatorInterface->AddPlan(new CTurnCommand(ballDir.GetDeg()));
  }
}

void CBasicDecision::OnActuatorSensorData() {
  const CCommand* const cmd = sensorInterface->GetLastCommand();
  if ( cmd != NULL && cmd->GetType() == CMD_Turn ) {
    lastTurn = static_cast<const CTurnCommand*>(cmd)->GetAngle();
  }
}

void CBasicDecision::OnInit() {
  actuatorInterface->AddPlan(new CMoveCommand(CPoint(-10, -10)));
}

void CBasicDecision::OnCommandWarning() {
  Decide();
}
```

Example A.4 An example of a basic agent using the framework.

```
int main(int argc, char** argv) {
  CViewManager view_manager;
  CParameterView* parameterView =
     new CParameterView(PARAMETER_VIEW_ID, &view_manager,
                        argc, argv);
  view_manager.AddView(parameterView, true);

  CActuatorInterface actuator_interface;
  CActuators actuators(&actuator_interface);
  CSensorInterface sensor_interface(parameterView->CP_team_name,
                                    &view_manager);
  CSensors sensors(&sensor_interface);

  CGameView* gameView = new CGameView(GAME_VIEW_ID, &view_manager);
  view_manager.AddView(gameView, true);
  CCommandView* commandView =
     new CCommandView(COMMAND_VIEW_ID, &view_manager);
  view_manager.AddView(commandView, true);
  CMarkerView* markerView =
     new CMarkerView(MARKER_VIEW_ID, &view_manager, parameterView);
  view_manager.AddView(markerView, true);
  CAgentView* agentView =
     new CAgentView(AGENT_VIEW_ID, &view_manager, parameterView);
  view_manager.AddView(agentView, true);
  CBallView* ballView = new CBallView(BALL_VIEW_ID, &view_manager);
  view_manager.AddView(ballView, true);
  CPlayerView* playerView =
     new CPlayerView(PLAYER_VIEW_ID, &view_manager);
  view_manager.AddView(playerView, true);

  CServerInterface server_interface(parameterView->SP_host,
                                    parameterView->SP_port,
                                    parameterView->SP_buffer_size);

  CMyDecision decision(&actuator_interface, &sensor_interface,
                       &view_manager);

  CController controller(&server_interface, &sensors, &actuators,
                         &decision,
                         parameterView->SP_simulator_step,
                         parameterView->SP_recv_step,
                         parameterView->SP_sense_body_step,
                         parameterView->SP_send_step);

  controller.Start();
  return 0;
}
```

Bibliography

[1] AI-Programming course web-page. http://www.ida.liu.se/~TDDA14/, March 2000.

[2] Minoru Asada and Hiroaki Kitano, editors. *RoboCup-98: Robot Soccer World Cup II*. Springer Verlag, Berlin, 1999.

[3] M. E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987.

[4] Hans-Dieter Burkhard, Markus Hannebauer, and Jan Wendler. AT Humboldt - development, practice and theory. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, pages 357–372. Springer Verlag, Berlin, 1998.

[5] Hans-Dieter Burkhard, Jan Wendler, Pascal Gugenberger, Kay Schröter, and Ralf Kühnel. AT Humboldt in RoboCup-98. In Minoru Asada and Hiroaki Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*. Springer Verlag, Berlin, 1999.

[6] Silvia Coradeschi and Jacek Malek. How to make a challenging AI course enjoyable using the RoboCup soccer simulation system. 1999.

[7] Silvia Coradeschi and Paul Scerri. A User Oriented System for Developing Behavior Based Agents. In Minoru Asada and Hiroaki Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*. Springer Verlag, Berlin, 1999.

[8] Emiel Corten, Klaus Dorer, Fredrik Heintz, Kostas Kostiadis, Johan Kummeneje, Helmut Myritz, Itsuki Noda, Jukka Riekki, Patrick Riley, Peter Stone, and Tralvex Yeap. *Soccerserver Manual Ver. 5.1*, 1999.

[9] Patrick Doherty. The WITAS Integrated Software System Architecture. *Linköping Electronic Articles in Computer and Information Science*, Vol 4(1999): no 17, December 1999. http://www.ep.liu.se/ea/cis/1999/017/.

[10] James R. Firby. Building symbolic primitives with continuous control routines. In J. Hendler, editor, *Proceedings of the 1st International Conference on Artificial Intelligence Planning Systems (AIPS-92)*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[11] Stanley Franklin and Arthur Graesser. Is it an agent, or just a program? In *Intelligent Agents III*, pages 21–36.

[12] Fredrik Heintz. FCFoo a short description. In *Robocup 1999 Team Description: Simulation League*. Linköping Electronic Press, 1999.

[13] Hiroaki Kitano, editor. *RoboCup-97: Robot Soccer World Cup I*. Springer Verlag, Berlin, 1998.

[14] Hiroaki Kitano, Milind Tambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda, and Minoru Asada. The RoboCup synthetic agent challenge 97. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 24–29, San Francisco, CA, 1997. Morgan Kaufmann.

[15] S. C. Kleene. *Introduction to Metamathematics*. Princeton N.J., 1952.

[16] Pattie Maes. The agent network architecture (ANA). *SIGART Bulletin*, 2(4):115–120, 1991.

[17] Jörg P. Müller. *The Design of Autonomous Agents - A Layered Approach*. Springer-Verlag, Heidelberg, 1996.

[18] Jörg P. Müller. The right agent (architecture) to do the right thing. In Jörg P. Müller, Munindar P. Singh, and Anand S. Rao, editors, *Intelligent Agents V - Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*. Springer-Verlag, Heidelberg, 1999.

[19] A. Newell and H. A. Simon. Computer science as empirical enquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.

[20] Itsuki Noda, Hitoshi Matsubara, Kazuo Hiraki, and Ian Frank. Soccer server: A tool for research on multiagent systems. *Applied Artificial Intelligence*, 12:233–250, 1998.

[21] The Robot World Cup Initiative web-page. http://www.robocup.org, March 2000.

[22] The RoboCup Simulator Team Repository. http://medialab.di.unipi.it/Project/Robocup/pub/, March 2000.

[23] The RoboLog Soccer Agent Libraries web-page. http://www.uni-koblenz.de/ag-ki/ROBOCUP/ROBOLOG/, March 2000.

[24] The RoboSoc web site. http://www.ida.liu.se/~frehe/RoboCup/RoboSoc/, March 2000.

[25] Stuart Russel and Peter Norvig. A Modern, Agent-Oriented Approach to Introductory Artificial Intelligence. 1995.

[26] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall International, 1995.

[27] Paul Scerri, Johan Ydren, Tobias Wiren, Mikael Lönneberg, and Per-Erik Nilsson. Headless Chickens III. In Manuela Veloso, Enrico Pagello, and Hiroaki Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*. Springer Verlag, Berlin, 2000.

[28] The Soccer Server web-page. http://ci.etl.go.jp/~noda/soccer/server, March 2000.

[29] Richard W. Stevens. *UNIX network programming Vol 1*. Prentice-Hall PTR, Upper Saddle River, NJ, 1998.

[30] Peter Stone. *Layered Learning in Multi-Agent Systems*. PhD thesis, Carnegie Mellon University, 1998.

[31] Peter Stone, Patrick Riley, and Manuela Veloso. The CMUnited-99 champion simulator team. In Manuela Veloso, Enrico Pagello, and Hiroaki Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*. Springer Verlag, Berlin, 2000.

[32] Peter Stone, Patrick Riley, and Manuela Veloso. Layered Disclosure: Why is the agent doing what it's doing? In *Proceedings of the Fourth International Conference on Autonomous Agents*, 2000.

[33] Peter Stone and Manuela Veloso. The cmunited-97 simulator team. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*. Springer Verlag, Berlin, 1998.

[34] Peter Stone, Manuela Veloso, and Patrick Riley. The CMUnited-98 Champion Simulator Team. In Minoru Asada and Hiroaki Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*. Springer Verlag, Berlin, 1999.

[35] Manuela Veloso, Enrico Pagello, and Hiroaki Kitano, editors. *RoboCup-99: Robot Soccer World Cup III*. Springer Verlag, Berlin, 2000.

[36] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[37] Michael Wooldridge and Nicholas R. Jennings. Pitfalls of Agent-Oriented Development. 1998.