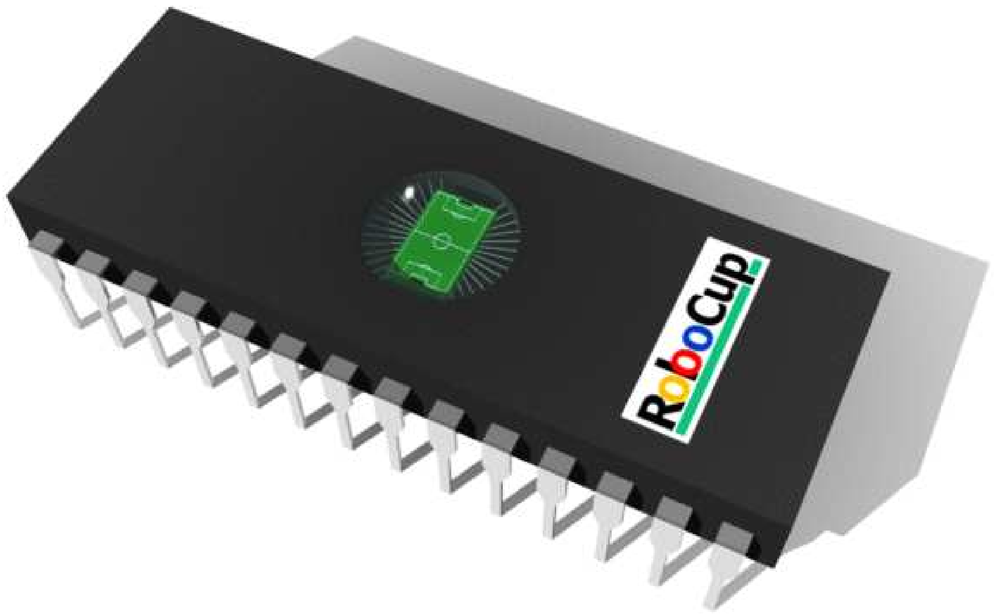
Maurice van der Pot & Nieke Roos

The Babylon Project 1999

The design and implementation of a RoboCup Simulator League soccer player



August 1999

Preface

Lying before you is the culmination of a period of four years at the Computing Science Department of the University of Nijmegen. Alas, we are moving on, to explore strange new worlds, to seek out new life and new civilisations, to boldly go where no one has gone before. Before our departure, however, we are leaving you this master thesis, resulting from our graduation project in the field of artificial intelligence. The project focuses on the RoboCup initiative and has been carried out under the supervision of dr. Th. E. Schouten.

In honour of the Babylon 5 TV−series, we have named our venture 'The Babylon Project 1999' (TBP1999 for short), after the challenging endeavour portrayed in it to establish and uphold the Babylon 5 space station that was built as a kind of United Nations, to help preserve peace in the galaxy (http://www.babylon5.com). The 1999 has been added to indicate the year this project will reach its completion.

Finally, we would like to thank our families, friends and everyone else who has contributed to the successful completion of this thesis. Special thanks go out to dr. Th. E. Schouten for supplying us with the opportunity to embark on the project in the first place and supporting us along the way. Also, we would like to express our gratitude to Patrick van Elk, Jeroen Kratz, Pim van Mun, Mark van Uden and Leon Versteegen for providing us with pleasant company and letting us give them the thrashing of their lifetime during the many games of table−tennis we have played.

Maurice van der Pot and Nieke Roos, July 1999.

Table of contents

1 Introduction

Especially in the last couple of years, research in the field of artificial intelligence (AI) has progressed enormously. IBM's Deep Blue defeating the human world chess champion Kasparov and the autonomous robotics system Sojourner exploring the surface of Mars are but a few examples of successful endeavours that have taken the AI discipline to a next level. Another project riding the wave alongside these challenges is the Robot Soccer World Cup, or 'RoboCup', initiative (http://www.robocup.org).

> *"RoboCup is an international initiative devoted to advancing the state of the art in AI and robotics. The particular goals of the project and potential research directions are numerous. The most ambitious and long range goal can be stated as: to build a team of robot soccer players, which can beat a human World Cup champion team." ([KITA1997])*

Every year the Robot Soccer World Cup is organised somewhere on the face of this earth. As the name already suggests, it is not a championship among human competitors. Instead, the competing teams, originating from all over the world, consist of so–called 'agents', players composed of hardware and/or software and controlled by AI. Currently, four main categories, or 'leagues', of agents are distinguished:

- Small Robot League
- Middle Size Robot League
- Sony Legged Robot League
- Simulator League

In the Small Robot League, the Middle Size Robot League and the Sony Legged Robot League real robots of different sizes, depending on the league, compete in real matches on real soccer pitches, contrary to the Simulator League, which is completely based on software.

In the Simulator League, one of the central parts is played by the soccer simulator, called 'Soccer Server'. The Soccer Server is responsible for controlling the (virtual) soccer pitch and all the (again virtual) objects that are on it. The RoboCup Simulator League soccer pitch is composed of a collection of markers or 'flags' (). Each flag has its own position on the pitch and its own identifier (e.g. the string "flag l t" indicates the top–left corner flag). As in real–life soccer, the moving objects on the simulated soccer pitch consist of one ball and a maximum of two teams of, at most, eleven players each, ten regular ones and one goalkeeper, and the movement of the ball is controlled by the behaviour of those players. The major difference with real life is that, in the simulator, each player is in fact a client connected to the Soccer Server. At regular intervals, every connected client receives sensory input from the server containing information about flags and moving objects that are in the player's line of sight. From this sensor information the client can construct an internal representation of the current state of affairs on the soccer pitch, which subsequently can be used to determine what actions are to be performed by the player under control. These actions are then communicated back to the server and the server will ensure the player acting accordingly. Also present in a match, apart from one ball and a maximum of twenty–two players, but invisible to the players, is a referee. Controlled by the Soccer Server, the referee responds to the situation in the match by changing the mode of play whenever necessary. For instance, when a team plays the ball out of bounds because it has crossed a sideline, the referee drops the ball on the position at which it crossed the sideline and awards a kick–in to the other team, changing the play mode accordingly. Besides controlling the mode of play, the referee also keeps track of the playtime, starting it at the beginning of the first and second half of a match and stopping it at half and full time. The judgements of the referee are broadcast to all the players by means of (virtual) auditory messages.

The objective of TBP1999 and the focus of this thesis are the design and implementation of the TBP1999 simulated soccer player, an instance of the aforementioned client controlling a player on the RoboCup Simulated League soccer pitch.

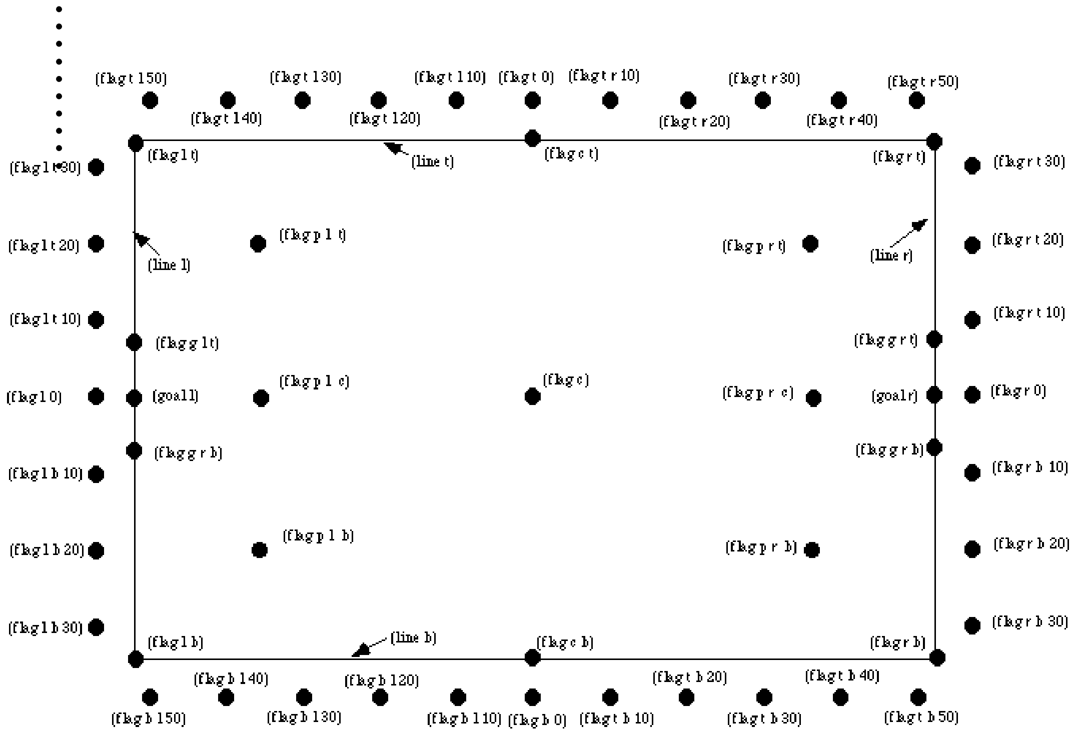


Figure 1: The RoboCup Simulator League soccer pitch (from [KUMM1999])

2 The design

In order to be able to start the implementation of the TBP1999 simulated soccer player, we have to have something upon which to base that implementation. Therefore, we have first drafted a design of the general structure of the player. This design is depicted in Figures 2, 3 and 4, using the OMT notation ([COOK1994]). Note that we have not completed the object–oriented modelling technique that is also described in [COOK1994], because the gap between the design in its current form and the implementation is sufficiently small to be bridged during that implementation stage. The main objective of the design, however, is not to guide us in the implementation, but to guide others in grasping its result. In other words: Our design is primarily meant to assist third parties in understanding our implementation.

According to the OMT notation, every separate type of entity that is part of the TBP1999 simulated soccer player is depicted in Figures 2, 3 and 4 as a rectangle. The name of each entity type is shown at the top of the rectangle. Lines connecting rectangles represent associations between entity types. An association can be viewed as being a 'has a'–relationship. At either end of a connection the role is displayed that the entity type at that end plays in the association. A connection ending in a black blob represents multiple associations, a bracketed number or range accompanying the blob indicates how many. A black blob without a number or a range in brackets represents zero or more associations. So when a particular entity type is connected to another entity type and the connection has a black blob at the end of the second one, the first entity type can be viewed as having zero or more instances of the second entity type. A connection ending in a diamond represents an aggregation, meaning that as long as the instance of the entity type at the end of the diamond exists, so does the instance of the entity type at the other end (but not the other way round). An oval attached to a particular connection represents an association type, meaning that the connection can be characterised by an instance of the entity type connected to the oval. So when an entity type is connected to an oval that, in turn, is attached to the connection between two other entity types, an instance of the first entity type can be viewed as being a property of that connection (e.g. an Employment entity type connected to an oval that is attached to a connection between a entity type Person and a Company entity type). Finally, a connection containing a triangle represents a type extension, or sub–type, relationship. A type extension relationship can be viewed as an 'is a'–relationship.

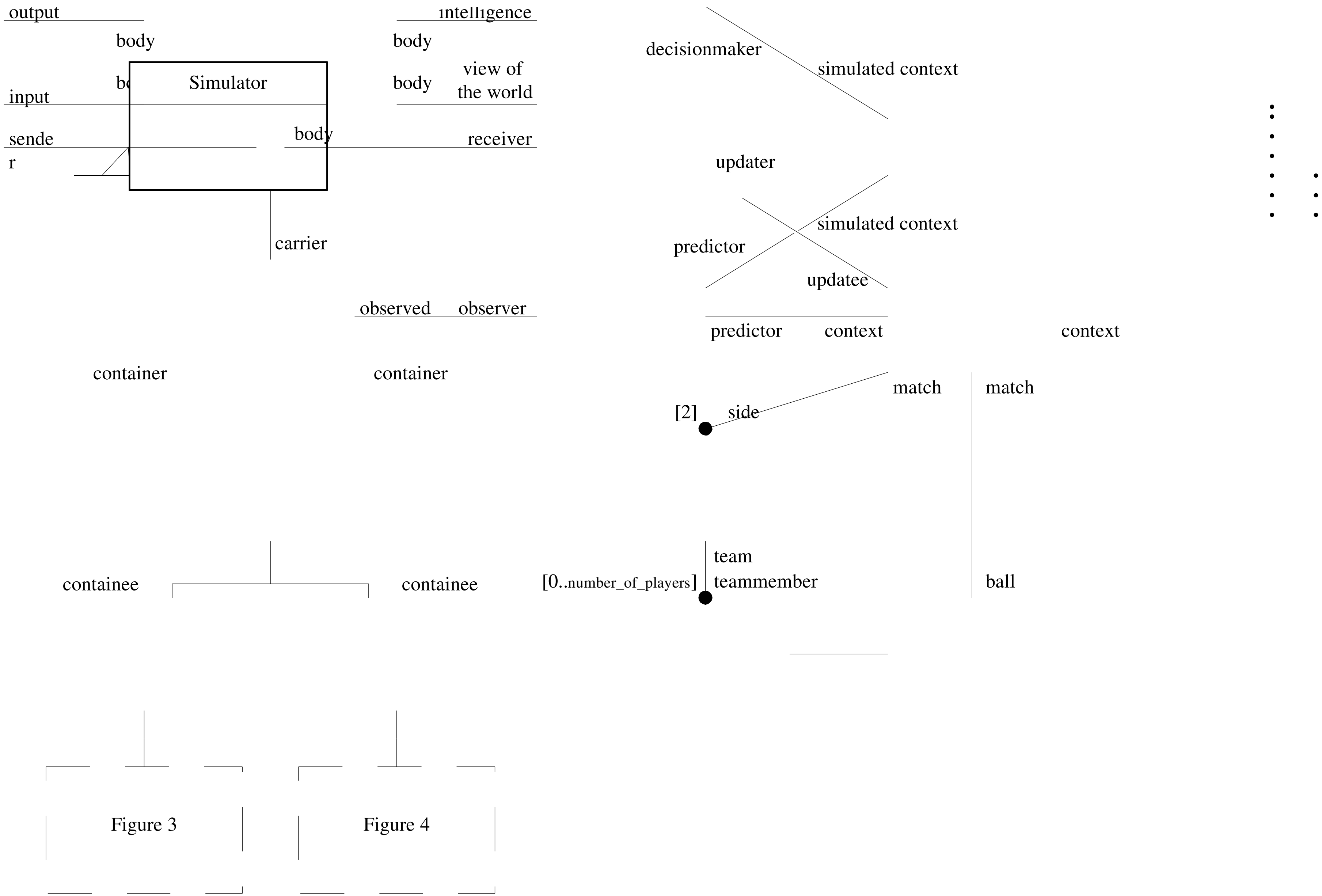


Figure 2: Design of the TBP1999 simulated soccer player

7

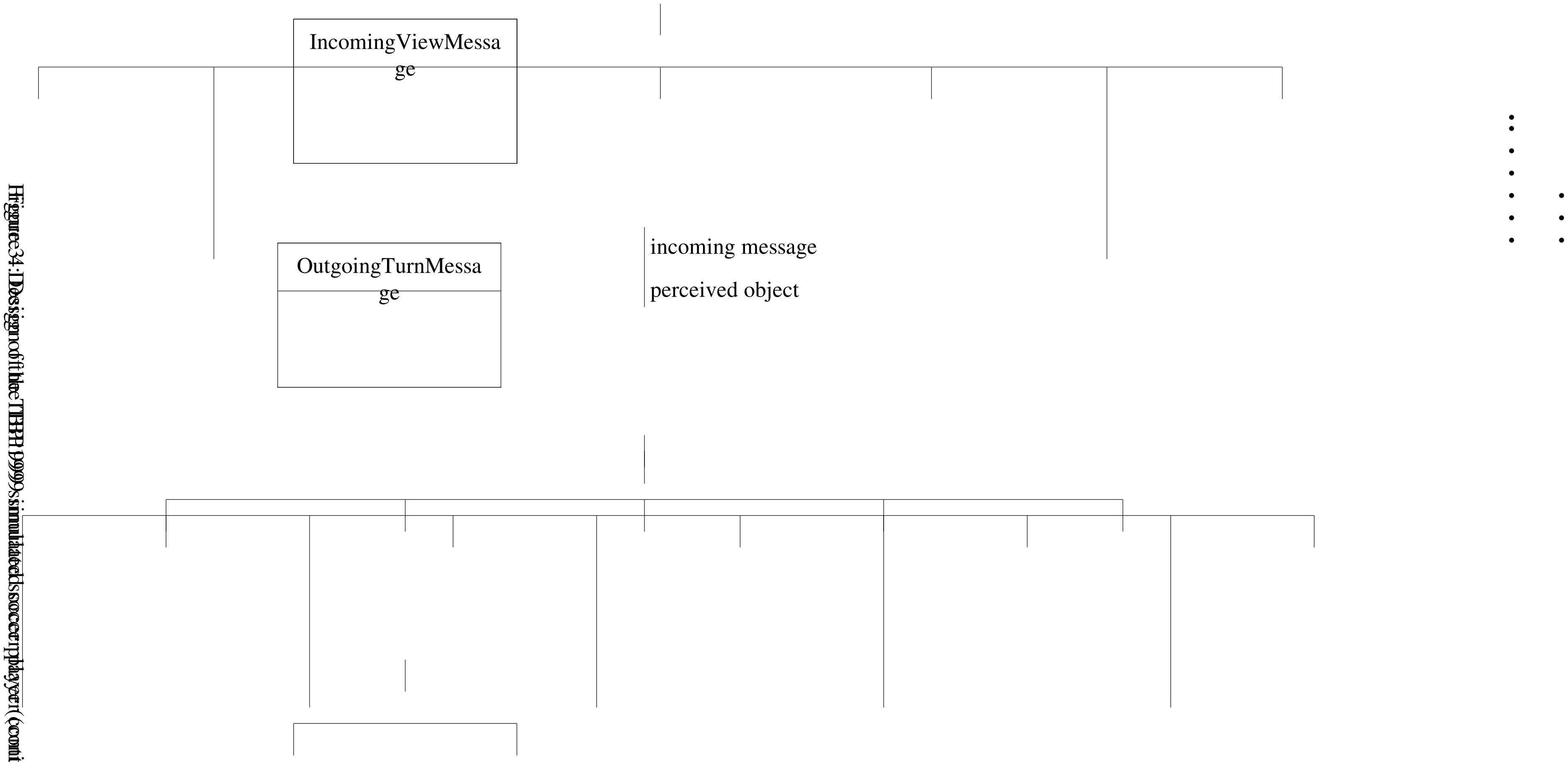


Figure 3: Design of the TBH99 simulated soccer player (continued)

8

3 The implementation

Implementing the design starts with choosing a programming language that is going to be used for the implementation. The outcome of this choice can subsequently be used to perform the actual act of implementation. Our choice of programming language will be discussed next, followed by a discussion of our implementation. We will conclude this section with a description of the coach program we have constructed to aid us in the testing of our implementation.

3.1 The programming language

Before choosing a programming language that we are going to use for implementation, we will have to come up with the criteria upon which we are going to base that choice. After we have tested the collection of possible programming languages against those criteria and have eliminated all except one and after we have completed the implementation of our soccer player using the language of our choosing, we can assess that choice by evaluating the problems encountered in using it.

3.1.1 The choice of programming language

To aid us in the decision which programming language to use for the implementation of our design of a RoboCup Simulator League soccer player we have drawn up the following set of criteria the eligible languages have to satisfy:

1. The programming paradigm to be used (e.g. imperative, functional or object–oriented) has to be suitable for the problem at hand.
2. The program written in the chosen programming language has to run on the platform used in the RoboCup competition as well as on the platform available to develop it. By ‘platform’ we mean the combination of operating system and underlying machine architecture. Since the SoccerServer only runs on the Unix operating system and Unix runs on a wide variety of different machine architectures, the programming language we decide to choose has to be architecture independent at the very least.
3. The programming language has to be sufficiently efficient. Because the problem at hand is highly complex, it is impossible to find the best solution by applying brute force. A larger amount of calculations per time unit therefore always yields a better solution.
4. Apart from being efficient at run–time, the program also has to be sufficiently fast at compile–time. We do not want to be forced to spend more time compiling than coding, to put it bluntly.
5. Because the program will most likely make use of one or more separate external modules, it will be an advantage to be able to load those modules at run–time. A programming language will be at a disadvantage if it forces one to recompile and/or re–link the entire program whenever another module is to be used.
6. The programming language must not be error–prone. Obviously, the program constructed has to contain as few errors as possible. However, the number of errors a program contains depends largely on the readability of its code and the readability of the code is dependent on the readability of the programming language. The easier it is to write unreadable code in a certain language, the bigger the chance to write erroneous code.
7. To minimise the effort and the time spent on learning, programming languages that we have already mastered and that we find comfortable to work with are preferable.

To restrict the number of eligible programming languages we start by taking a look at the nature of the problem at hand. A RoboCup match consists of a soccer pitch containing a ball and two competing teams comprised of a certain number of players. Each of these moving objects has particular properties (e.g. a position, a speed and a direction) and each moving object except for the ball is controlled by a separate program. Keeping track of the objects in its environment enables such a program to make a decision on what to do next. So given the fact that the program that is to be written has to deal with the object it controls and all the other objects on the field, we opt for a object–oriented programming language.

Looking at the last criterion, only a few object–oriented programming languages remain for us to choose from:

- Object Pascal
- C++
- Java

Because in the past we have had some troubles in using Object Pascal on a Unix operating system, the second criterion cuts matters down to a choice between C++ and Java.

As far as the third criterion is concerned, both C++ and Java seem to perform equally efficient, providing a Just In Time (JIT) compiler is used for compiling a Java program. We do not know whether or not it is possible to make use of a JIT−compiler during an actual RoboCup match. If this turns out to be impossible, C++ definitely is the winner on this one.

The garbage collector used by Java is another factor that renders C++ the more efficient language of the two. On the other hand, using such a garbage collector prevents the memory from becoming fragmented and reduces the chances of memory leaks occurring. However, it must be noted that memory leaks are the result of poor programming.

When it comes down to which of the two programming languages provides the fastest compiler, C++ certainly has the upper hand. We have not been able to retrieve any quantitative figures, though, on how much slower the Java compiler actually is, so we really cannot tell how big the victory is C++ has won here.

The fifth criterion speaks in favour of the Java programming language, for run−time switching between Java modules, called classes, requires almost no effort at all. Achieving the same in C++ is a lot more difficult and involves re−linking the executable. Although, essentially, Java classes and C++ object files are the same, C++ object files need to be linked together before execution is possible, whilst Java classes are not in need of such an extra step.

The C++ programming language is probably also more error−prone than Java, as C++ provides the programmer with full−scale pointer arithmetic. Pointers not only severely reduce the readability, but also the understandability of a particular program that uses them. As objects are called by reference, the Java programming language also makes an implicit use of pointers, but they cannot be explicitly used to perform calculations. Apart from the pointer possibilities of C++, both languages are quite equivalent. It is, however, quite impossible for a C++−programmer to renounce his/her right to use these possibilities, since pointers are practically embedded in the language.

The aforementioned considerations have led us to opt for the Java programming language (version 1.1.7b, the latest one containing a JIT−compiler). Especially criteria numbers five and six have made the scale turn towards Java. Another consideration, not mentioned before, that has played a part in our decision−making process, is the fact that creating of a RoboCup Simulator League soccer player in Java really contributes something new, as few Java players already exist.

3.1.2 General problems related to the choice of programming language

During the implementation of our design of the soccer player, we have come across some problems that are directly caused by our decision to use the Java programming language. These general problems will we discussed here, the language−independent problems we have encountered will be discussed in the next section.

Problem: The Java 1.1.7b compiler is not the fastest compiler on the face of this earth. In fact, we have probably spent more time compiling than doing something else.

Solution: There is not really a solution to this problem. We tried the latest Java release (version 2) combined with the Java HotSpot Performance Engine (http://java.sun.com/products/ hotspot/), but this did not turn out to be much of an improvement. Actually, the Java 2 platform somehow slowed matters down, so we decided to stick with the 1.1.7b version of the platform and live with its sluggishness.

Problem: Although the Java programming language does not support the explicit use of pointers, it does make use of the concept of calling objects by reference (implicit pointers). As a result of this changes made to an object within a method that has been passed that object through one of its arguments also apply outside the scope of that method. A much more tricky consequence is this: Assume a particular object, anObject, is assigned to another object, anotherObject of the same type ObjectType (i.e. ObjectType anotherObject = anObject). The concept of calling by reference causes subsequent alterations made to anotherObject also being made to anObject! This could have disastrous (or at least mildly unpleasant) effects when one is not aware of this and one thinks to have made a copy of anObject that can be safely manipulated without affecting the original.

Solution: To prevent unintentional alterations of objects we have made sure that a particular object can only be changed intentionally by forcing ourselves to have the notice “// Intended set (don’t remove this line)” precede every intentional call to a set–method. We have written a small program (see the Appendix) to check the TBP1999 simulated soccer player code for calls of set–methods that are not accompanied by the aforementioned message.

We have also taken some precautions that enable us to safely make use of a copy of a particular object without actually affecting that object. Whenever we need a copy of an object, we create that copy by calling a specific constructor of the object class that constructs an entirely new class instance from the object to be copied by making copies of all the member variables. This means that copying anObject to anotherObject is done by stating ObjectType anotherObject = new ObjectType(anObject). As a result of this, altering anotherObject does not affect anObject.

Problem: We noticed that the execution of a sequence of actions on the server did not exactly progress as planned. The intention was to have one action executed by the server every cycle. For some reason, however, some cycles remained deprived of the execution of an action. As a result of which, a discrepancy arose between the simulated world of the server and our simulation of that simulated world, on which we had based the plan. This resulted in actions taking place at the wrong time and making the whole plan fall apart.

Solution: Because the number of actions executed by the server was the same as the number of actions sent to it, it followed that the actions did not reach the server at the correct speed. The arrival rate of the messages turned out to be slightly smaller than the simulation cycle frequency of the server. This resulted in the client and the server steadily diverging to the extent of the server overtaking the client in the long run, even more than once, thus rendering one or more cycles action–less. On the other hand, a client running too fast would result in the server receiving two or more actions during the same cycle and executing only one of them, thus effectively skipping the others and making the whole plan fall apart again. Therefore the client should send exactly one action every server cycle. Ideally, this message–sending task should be periodic, executing once every server cycle. Java, however, does not support the concept of tasks, so we searched the Internet and came across an official future real–time extension to Java (http://www.sdct.itl.nist.gov/ ~carnahan/real–time/intro.html), which, because of its premature nature, was of no use to us, and an unofficial extension called JavaTime, containing a Task class (http://www–cad.eecs.berkeley.edu/~jimy/java/library/index.html). Unfortunately, this unofficial extension did not do the trick either, because a brief correspondence with the author of JavaTime revealed it to be implemented on top of Java threads. This meant that making use of the JavaTime extension would not solve the deficiencies we already encountered when using threads. Hence, we had to look for another way to provide for the timeliness in our client within the existing Java framework.

Our attempts to equip our client with the desired periodical behaviour whilst staying within the boundaries laid down by the current Java environment were seriously frustrated by two major drawbacks of that environment:

1. Java’s sleep−method “puts the currently executing thread to sleep for at least the specified number of milliseconds. ‘At least’ means there is no guarantee the thread will wake up in exactly the specified time.” ([ARNO1996]). Thus, it is quite impossible for the client to send a message at exact intervals using the sleep−method.
2. Java’s currentTimeMillis−method on the Windows 95/98 operating system makes use of a call to the operating system that uses a clock that is updated at a frequency of 18.2 Hz. This means that the value returned by currentTimeMillis (on Windows 95/98) will be updated every 55 milliseconds. Thus, it is also quite impossible to exactly create intervals that are not a multiple of 55 milliseconds using the currentTimeMillis−method in combination with the Windows 95/98 operating system. Note that the currentTimeMillis−method on a Unix operating system does not necessarily need to suffer from this flaw.

The second drawback can be circumvented by using Unix. This will not resolve the first drawback, though. Therefore, a solution had to be found that neither makes use of the sleep−method nor calls the currentTimeMillis−method, as we like to continue working on Windows 95/98.

Ultimately, we tackled the problem by attaching each client to its own external timer (see the Appendix) through a particular socket. The timer sends packets to that socket at a constant rate (i.e. a rate equalling the SoccerServer sending frequency) and the receipt of such a packet at the specified socket awakens the client (i.e. the receive−method returns). Next, the client can send what it has to send to the server, after which it goes back to sleep again by calling the receive−method on the timer socket. Because the timer is written in C, the aforementioned first drawback no longer holds, since C does contain functions that enable the client to sleep for an exact period of time. The second drawback has also been resolved, as the timer is run on Unix and Unix system clocks operate at a much higher frequency, thus rendering the timing much more accurate. Note that the timer code is an exact copy of SoccerServer code, thus ensuring an equal precision, and therefore equal message arrival rates, in both cases.

3.1.3 Evaluation of the choice of programming language

Looking at the set of the criteria we have drawn up to aid us in choosing a programming language and considering the problems resulting from this choice, we can conclude the following:

1. Criterion number 4 turns out not to hold for our choice. The Java 1.1.7b compiler really is too slow to work with in a reasonably comfortable way.
2. Criterion number 6 does not hold entirely either. Although Java does not explicitly support pointers, implicitly the programmer is confronted with them due to the ‘call by reference’−concept, bringing about the referencing troubles mentioned in the previous section.
3. We should have included a criterion stating some kind of real−time requirement. This omission has been directly responsible for the aforementioned problem concerning the lack of timeliness within the Java framework.

Pondering the advantages and the drawbacks of our opting for the Java programming language, it is difficult to say whether or not we would make the same choice again. Especially the absence of real−time features is a very serious disadvantage. On the other hand, Java definitely is one of the most aesthetic programming languages we have ever worked with and it certainly has some exquisite properties, such as the ability to dynamically load classes at run−time. Also, we believe that choosing Java to implement the TBP1999 simulated soccer player is innovating and contributing, as not many existing RoboCup competitors have made the same choice.

3.2 The classes

This section will provide an overview of the various classes our player is composed of. We have divided them into two categories: primary classes and auxiliary classes. Whenever we have stumbled on a problem whilst implementing a particular class, the description of that class is followed by the problem description and our solution.

3.2.1 The primary classes

This section will cover those classes that are primarily responsible for the correct operation of the TBP1999 simulated soccer player.

3.2.1.1 The Player class

The Player class is the primary class of our client to the SoccerServer. It contains the main–method that is executed when the Player class is run. The main–method parses the parameters the user can pass to the client program to customise the Player and, if one or more arguments are omitted, assigns default values to the corresponding parameters. The command to invoke the client program, the parameters the user can customise and the default values that are assigned to them, are shown in .

java TBP1999.Player [–<Parameter> <Value>]

Parameter	Value	Default
Host	Hostname	"poseidon"
Port	Port number	6000
Team	Team name	"TBP1999"
Type	Goalie or player	player
Mind	Mind class	"TBP1999.NeverMind"
Log	Name of file to log to	"TBP1999.log"

Table 1: Command line parameters

The host–parameter defines the name of the host on which the SoccerServer is running. The port number defines the number of the port on that host the SoccerServer is listening to, initially. The team–parameter contains the name of the team this client belongs to. The type–parameter defines what type of player this client will be. Possible values are “goalie”, indicating this client program is going to be the goalkeeper, and “player”, designating this client the role of a normal player (i.e. not a goalkeeper). The mind–parameter defines which mind is going to be used to control the player, for it is possible for the user to construct his/her own mind and hook it into the player, which will use it to generate its behaviour. We will return to this subject at a later stage. Finally, the log–parameter defines the name of the file certain important messages are logged to.

The main–method uses these parameters to invoke the constructor of the Player class. First, this constructor creates a log, which will be used to log certain important messages. After this an action history is created to keep track of the actions that are sent to the server. Next, it creates an instance of the NervousSystem class, which will be used to exchange messages between the different parts of the client. If the creation of the nervous system fails, the constructor of the Player class returns, for the nervous system is the key class that connects all major classes. If the creation is successful, the constructor creates the mind’s eye, an instance of the MindsEye class, that will be used to keep track of the situation on the pitch. Finally, the player’s mind is created. This is achieved by dynamically linking in the appropriate Mind class and then creating an instance of that class, which can then be used to determine the player’s behaviour. Both the instance of the MindsEye class and the instance of the Mind class are threads and are started after they have been created.

3.2.1.2 The NervousSystem class

The NervousSystem class is responsible for the communication from the server to the mind’s eye and back from the mind to the server. This communication between the client and the server consists of exchanging datagram packets through a socket, which is created in the constructor of this class. After this the actuators, responsible for sending messages to the server, are created. Next, the actuators’ init–method is invoked that sends an initial message to the server telling it the name of the team the connecting player belongs to and the version of the server the client has been built for. The init–method is also responsible for receiving the server’s reply confirming the connection. The server’s confirmation message also contains the number of the port through which the communication between this client and the server will be resumed. This port number is stored in the actuators, after which they are started. Finally, the sensors, responsible for receiving messages from the server, are created and started.

Problem: At first we erroneously assumed the version we had to send to the server was the version of our client program. The server's reply did not reveal any errors on our behalf, but when the server started sending information about the match the only visually oriented messages our client program received were objectless (See 0)–messages before kick–off and (See *time*)–messages after the kick–off.

Solution: First, we took a closer look at another Java–client we had obtained, but that seemed to initiate the connection in exactly the same way our client did, with one major difference: the visually oriented messages received by that client did contain objects in its visual range. Next, we examined the SoccerServer code to find out where these visually oriented messages were constructed and sent and why we only received messages containing just the time and nothing else. We discovered that the server assembled these messages based upon the version contained in the client's initial message. After we had changed the version to the server version our client has been built to support, the server provided our client with the proper visual information.

Problem: We, again erroneously, assumed the communication between server and client always takes place using the same port. Implementing this assumption resulted in the player correctly sending and receiving the initial messages, after which, however, each subsequent message sent to the server did not seem to arrive: the (Unix) SoccerServer did not show the expected results of the message sent, nor did it return or display an error message.

Solution: We repeated the same course of action on the Windows version of the SoccerServer, which resulted in the displaying of a command error each time we sent a command to it. Our commands apparently were getting through to the server, but in an incorrect way. First we thought there was something wrong with the message format we used. Looking at the other Java–client, however, did not reveal any message format dissimilarity. We did notice a slight difference in the way both clients handled the receipt of the initial message from the server. The other Java–client adjusted its communication port to the server upon receipt. It then occurred to us that to be able to distinguish between messages emanating from different players, the server had to assign different port numbers to them. So, the initial port number was just for sending the initial message, after which the client program had to set its port number to the one contained in the initial reply received from the server. After we had changed this, the server reacted in the proper way to the messages our client program sent it.

The nervous system also contains a facility that allows it to cope with action sequences called plans. The actions that make up a plan are generated by the mind and have to be distributed among several simulator cycles, because only one action may be sent per cycle. Every time the mind has assembled such a plan, it calls the nervous system's startPlan–method, after which the nervous system keeps track of the plan. The planExecuting–method returns true while the plan is executing and calling the abortPlan–method terminates the plan.

3.2.1.3 The Sensors class

Incoming messages from the server are received by the sensors. It is a thread that continuously accepts datagram packets until it is stopped. These packets are then parsed by the sensors' parseMsg–method, which looks at the kind of message that has been received and invokes the appropriate method to parse the rest of the message.

There are four kinds of message that can be sent by the server to the client program ():

1. Initial (init) messages. These are sent as a reply to the client sending an initial message. The init message from the server contains the side the player is playing from, a uniform number to identify a player within a team and the current mode of play (i.e. the state the match is in).
2. Hear messages. These are sent by the server as a reply to a client sending a say message and received by every client within a certain distance from the client making a statement. They are also sent when the referee has something to say, in

which case they are heard by every player. A hear message contains the time of the match, the sender of the message and, of course, the message itself.

3. See messages. The server sends these at regular intervals. Besides the time of the match, a see message also contains information on every object that is within the visual range of the player. This information is composed of the type of object that is observed, at what distance and in which direction it is perceived and changes in distance and direction since it was last seen. In case the observed object is another player information is also given about the direction that player is facing.
4. Sense body messages. These are sent as a reply to the client sending a sense body message. A sense body message consists of information on the internal state of the player itself. It contains the time of the match, the quality of the visual information the client is receiving, the angle within which objects can be observed by the client, the player's current stamina and speed and number of times it has kicked, dashed, turned and said anything so far.

After a server message has been parsed by the appropriate method according to its structure, an incoming message (i.e. an instance of the IncomingMessage class) is created from it. The kind of incoming message that is generated depends on the purpose of the server message.

There are three kinds of incoming message:

1. State messages. These are comprised of all init, sense body and referee server messages (transformed into IncomingInitMessages, IncomingSenseBodyMessages and IncomingRefereeMessages, respectively), because they convey information on the state of the match. Referee server messages can be recognised because their *sender* part reads referee. A fourth kind of state message is the IncomingSeeMessage. Every see server message is broken up into several IncomingSeeMessages in such a way that all the information on a particular object is contained within a separate IncomingSeeMessage. Possible IncomingSeeMessage object types are player (MsgPlayerObj), goal (MsgGoalObj), ball (MsgBallObj), flag (MsgFlagObj) and line (MsgLineObj) with parameters as shown in . In addition to the messages directly generated from a server message, there is a fifth state message, the IncomingViewMessage, which is generated before and after each see server message, indicating which IncomingSeeMessages have emanated from the same see server message.
2. Friendly messages. These consist of every hear server message that conveys visual information from team–mates (e.g. "Teammate Fred's position is this and that!"). Such a hear message can be recognised because its *message* part begins with "TiaFM" (This is a Friendly Message).
3. Other messages. These consist of hear server messages that do not have a *message* part beginning with "TiaFM" or a *sender* part reading referee. These messages are discarded unless there is some time left to use them and it is easy to do so (e.g. a message from the opponent can be retransmitted to confuse the opposing side).

These incoming messages are put in the corresponding message queue in the nervous system.

<table>
<tr><td>0.</td><td>(</td><td></td></tr>
<tr><td>1.</td><td>init side uNum playMode)</td><td></td></tr>
<tr><td></td><td>uNum</td><td>::= 1…11</td></tr>
<tr><td></td><td>playMode</td><td>::= before_kick_off</td></tr>
<tr><td></td><td></td><td>| time_over</td></tr>
<tr><td></td><td></td><td>| play_on</td></tr>
<tr><td></td><td></td><td>| kick_off_side</td></tr>
<tr><td></td><td></td><td>| kick_in_side</td></tr>
<tr><td></td><td></td><td>| free_kick_side</td></tr>
<tr><td></td><td></td><td>| corner_kick_side</td></tr>
<tr><td></td><td></td><td>| goal_kick_side</td></tr>
<tr><td></td><td></td><td>| goal_side</td></tr>
<tr><td></td><td>side</td><td>::= l|r</td></tr>
<tr><td>0.</td><td>(hear time sender message)</td><td></td></tr>
<tr><td></td><td>time</td><td>::= simulation cycle of the SoccerServer</td></tr>
<tr><td></td><td>sender</td><td>::= referee|self|direction</td></tr>
<tr><td></td><td>direction</td><td>::= −180…180 degrees</td></tr>
<tr><td></td><td>message</td><td>::= string containing a maximum of 512 characters</td></tr>
<tr><td>0.</td><td>(see time objInfo objInfo …)</td><td></td></tr>
<tr><td></td><td>time</td><td>::= simulation cycle of the SoccerServer</td></tr>
<tr><td></td><td>objInfo</td><td>::= (objName distance direction distChng dirChng faceDir)</td></tr>
<tr><td></td><td></td><td>| (objName distance direction distChng dirChng)</td></tr>
<tr><td></td><td></td><td>| (objName distance direction)</td></tr>
<tr><td></td><td></td><td>| (objName direction)</td></tr>
<tr><td></td><td>objName</td><td>::= (player teamname uNum)</td></tr>
<tr><td></td><td></td><td>| (goal side)</td></tr>
<tr><td></td><td></td><td>| (ball)</td></tr>
<tr><td></td><td></td><td>| (flag c)</td></tr>
<tr><td></td><td></td><td>| (flag [l|c|r] [t|b])</td></tr>
<tr><td></td><td></td><td>| (flag p [l|r] [t|c|b])</td></tr>
<tr><td></td><td></td><td>| (flag [t|b] [l|r] [10|20|30|40|50])</td></tr>
<tr><td></td><td></td><td>| (flag [l|r] [t|b] [10|20|30])</td></tr>
<tr><td></td><td></td><td>| (flag [l|r|t|b] 0)</td></tr>
<tr><td></td><td></td><td>| (line [l|r|t|b])</td></tr>
<tr><td></td><td>teamname</td><td>::= string</td></tr>
<tr><td></td><td>uNum</td><td>::= 1…11</td></tr>
<tr><td></td><td>side</td><td>::= l|r</td></tr>
<tr><td></td><td>distance</td><td>::= real number</td></tr>
<tr><td></td><td>direction</td><td>::= −180…180 degrees</td></tr>
<tr><td></td><td>distChng</td><td>::= real number</td></tr>
<tr><td></td><td>dirChng</td><td>::= −180…180 degrees</td></tr>
<tr><td></td><td>faceDir</td><td>::= −180…180 degrees</td></tr>
<tr><td>0.</td><td>(sense_body</td><td>time (view_mode [high|low] [narrow|normal|wide])
(stamina stamina effort) (speed amountOfSpeed) (kick kickCount)
(dash dashCount) (turn turnCount) (say sayCount))</td></tr>
<tr><td></td><td>time</td><td>::= simulation cycle of the SoccerServer</td></tr>
<tr><td></td><td>stamina</td><td>::= real number</td></tr>
<tr><td></td><td>effort</td><td>::= real number</td></tr>
<tr><td></td><td>amountOfSpeed</td><td>::= real number</td></tr>
<tr><td></td><td>kickCount</td><td>::= integer</td></tr>
<tr><td></td><td>dashCount</td><td>::= integer</td></tr>
<tr><td></td><td>turnCount</td><td>::= integer</td></tr>
<tr><td></td><td>sayCount</td><td>::= integer</td></tr>
</table>

Table 2: Sensor information

3.2.1.4 The Actuators class

The only task of the Actuators class is to retrieve outgoing messages (i.e. instances of the OutgoingMessage class) from the outgoing messages queue in the nervous system, wrap them in a datagram packet and send them on their way to the server.

There are ten kinds of outgoing message ():

1. Bye messages. These are sent by the client to inform the server that the client is about to close its connection to the server.
2. Catch messages. These are sent by a goalie to catch the ball in a certain direction.
3. Change view messages. These are sent by the client to change the view angle and quality of the player.
4. Dash messages. These are sent by the client to make the player run in the direction it is facing with a certain amount of power.
5. Init messages. These are sent by the client to inform the server of its presence. An init message contains the name of the team this client belongs to, the server version it has been built for and whether or not this client is the goalkeeper.
6. Kick messages. These are sent by the client to make the player kick the ball with certain power in a certain direction.
7. Move messages. These are sent by the client to make the player move to a certain position on the pitch. These messages are only allowed to be sent before the kick–off (i.e. in the before_kick_off–mode).
8. Say messages. These are sent by the client to pass a message (orally) to players in its neighbourhood.
9. Sense body messages. These are sent by the client to get information about the internal state of the player.
10. Turn messages. These are sent by the client to make the player turn in a certain direction.

1.	(bye)
2.	(catch direction) direction ::= −180…180 degrees
3.	(change_view angle quality) angle ::= narrow|normal|wide quality ::= high|low
4.	(dash power) power ::= −30…100
5.	(init teamname [(version verNum)] [(goalie)]) teamname ::= string verNum ::= number
6.	(kick power direction) power ::= −0…100 direction ::= −180…180 degrees
7.	(move X Y) X ::= −54…54 Y ::= −32…32
8.	(say message) message ::= string containing a maximum of 512 characters
9.	(sense_body)
10.	(turn moment) moment ::= −180…180 degrees

Table 3: Control commands

Problem: To avoid jamming the server with too many commands it is (strongly) recommended limiting the messages sent to the server every cycle.

Solution: We have divided the outgoing messages into four categories:

1. Change view messages (OutgoingChangeViewMessage class).
2. Say messages (OutgoingSayMessage class).
3. Sense body messages (OutgoingSenseBodyMessage class).
4. Action messages (the remaining OutgoingMessage classes).

Every cycle a maximum of one message from each group is sent to the server and the minimum time between two consecutive messages sent within the same cycle is RECV_STEP = 10 ms.

3.2.1.5 The MindsEye class

The mind's eye is a thread that keeps track of the match by getting state messages from the state messages queue and updating the information on the match (i.e. an instance of the MatchInfo class) according to the kind of state message (i.e. init, see or sense body). The mind's eye class is also responsible for sending in its constructor the initial message (i.e. an instance of the OutgoingInitMessage class) to inform the server of the arrival of a new player.

3.2.1.6 The MatchInfo class

The MatchInfo class holds the information on the match. It contains information about both teams (i.e. instances of the TeamInfo class and vectors containing information on unrecognisable players), about the ball (i.e. an instance of the FieldObjectInfo class), about the mode the play is in, about the score (i.e. an instance of the Point class) and about the time of the match (contained in an instance of the Clock class). This information is updated by the various update−methods as state messages are passed through by the mind's eye.

If the information of an incoming referee message has to be processed, the time, mode of play and score are updated according to the information contained in the message.

If the information of an incoming see message has to be processed, the horizontal and vertical displacement of the perceived object in relation to the perceiving player are calculated. If the observed object is another player, an attempt is made to determine its team−name and uniform number. If the attempt is a successful one, the information on the corresponding player (i.e. an instance of the PlayerInfo class) is updated accordingly. If the player is so far away that its team−name can be recognised but its number cannot, the observed information is added to the vector of unrecognisable team−mates or opponents, depending on which team−name has been perceived, in the form of an instance of the FieldObjectInfo class. If the player is even farther away and both the team−name and the number cannot be recognised, the observed information is added to the vector of unrecognisable players in the form of an instance of the FieldObjectInfo class. If the observed object is the ball, the information on the ball is updated accordingly. If the observed object is a goal, a line or a flag, the position of the player is determined and the information on the player is updated accordingly. Finally, the time of the match is synchronised with the time contained in the incoming see message.

If the information of an incoming sense body message has to be processed, the player's view, speed and stamina and the time of the match are updated according to the information contained in the message. Finally, the action history is updated with the stamina and the time.

If the information of an incoming initial message has to be processed, the player's uniform number, and direction, the side it is playing from and the mode of play are updated according to the information contained in the message.

Problem: How to calculate the position of the player from the information from incoming see messages?

Solution: The incoming see messages have been ordered in such a way that see messages containing information about lines come first, followed by see messages with information on flags, followed by the remaining see messages. The reason for this ordering is the fact that the information about lines is used first to calculate the direction the player is facing. This direction is then used, alongside the information about flags, to calculate the position of the player. If, after all the information on lines has been processed, the player's direction is still not known, the direction and position of the player are calculated from the information (i.e. direction and position) about three or more flags. That is to say, if that much information can be extracted from the incoming see messages. If not, the information on the player remains unchanged, as nothing new can be derived.

Problem: How to calculate the direction the player is facing from the information about lines from incoming see messages?

Solution: The angle contained in the information about a perceived line is the smallest angle from the player's face direction to that line (α_{line} in). Given this angle, the direction the player is facing (α_{player} in) can be calculated as shown in . Top, bottom, left and right indicate which line is perceived (from the point of view of the player).

Problem: How to calculate the position of the player from the information about one flag when the player's direction is known?

Solution: The information received about the flag contains a direction in which the flag is seen and the distance between the perceiving player and the flag. Basic trigonometry will yield the player's position. Note, however, that the distance at which the flag is perceived is unreliable, because it is quantised by the server to include noise, the magnitude of which increases as the flag is farther away. Because no other information is available at this time to determine the player's position more accurately, the current approximation is as good as it can get.

Problem: How to calculate the position of the player from the information about two or more flags when the player's direction is known?

Solution: As we have mentioned before, the distance between objects is unreliable. However, the directions in which objects are seen can be relied on, so we are only going to use those to calculate the player's position. The relative direction in which a flag is perceived is contained within the information about the flag. Because we know the direction the player is facing, we can calculate the absolute direction a particular flag is seen in. Because we also know which flag it is the player is seeing, for this information is also contained in the incoming see message, we know the position of the flag. Through this point, we can draw an imaginary line, making an angle with the X−axis equal to the absolute direction the flag is perceived in. The same can be done for a second flag. Calculating the intersection point of these imaginary lines gives us the position of the player.

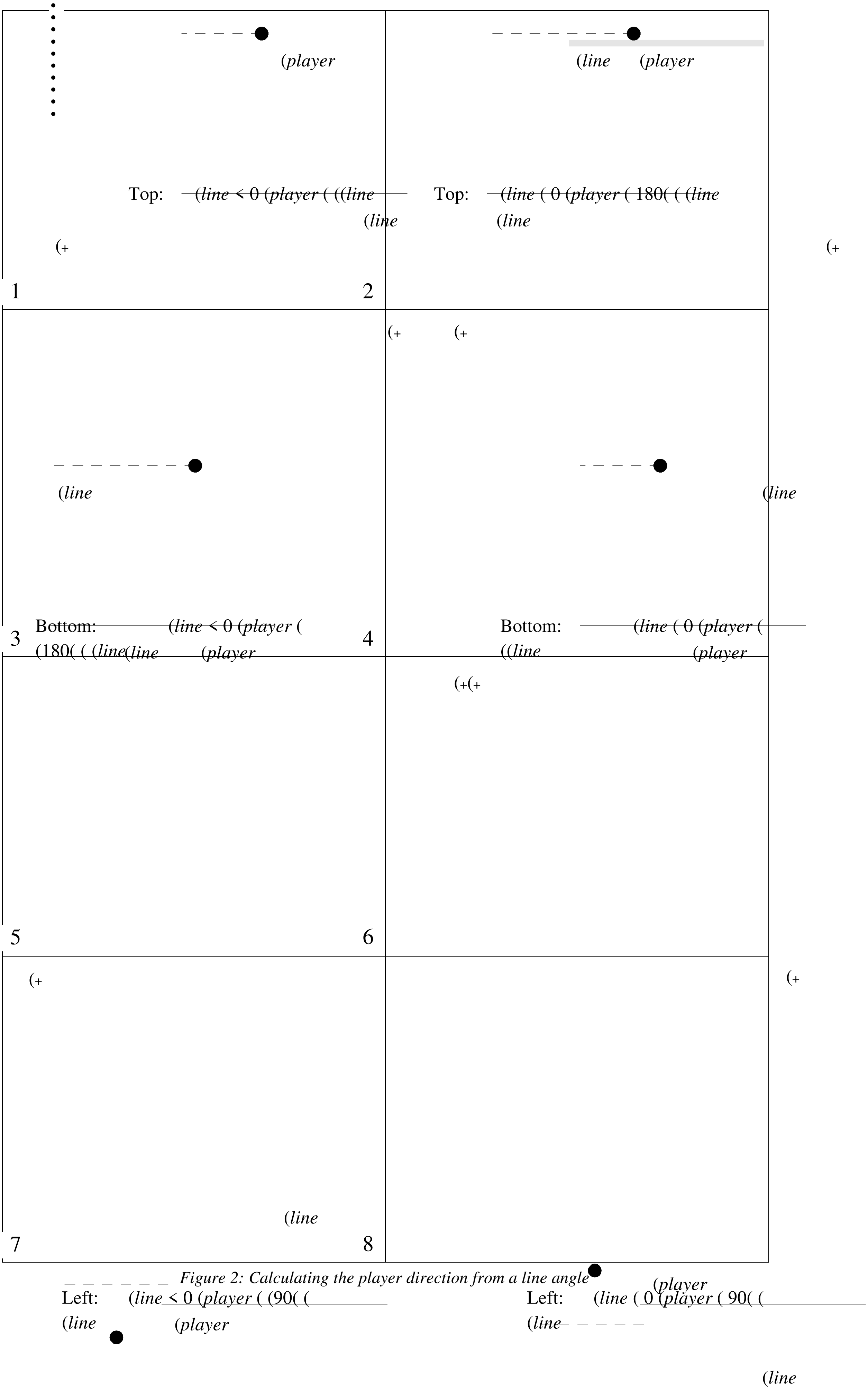


Figure 2: Calculating the player direction from a line angle

Problem: How to calculate the position of the player when the player's direction is not known?

Solution: It is possible to calculate the exact position of the perceiving player, given *only* the positions of three stationary objects and the directions in which they are perceived. This is done as follows.

Suppose the player, P, perceives three stationary objects, O_1, O_2 and O_3, in certain directions relative to P's direction. Then, the angles α, β and γ can be calculated (Figure 6).

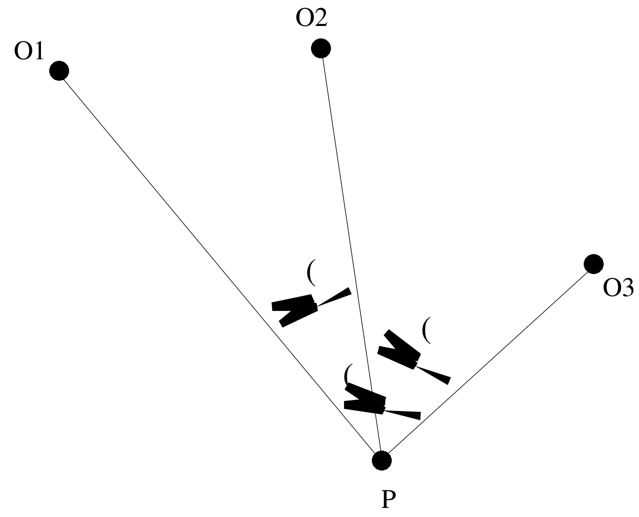
(*line*

(*player* (*player*

Right: (*line* < 0 (*player* (90((-(*line* Right: (*line* (0 (*player* ((90(((*line*

(*line*

*Figure SEQ Figure * ARABIC 6: A player P perceiving three stationary objects O_1, O_2 and O_3*



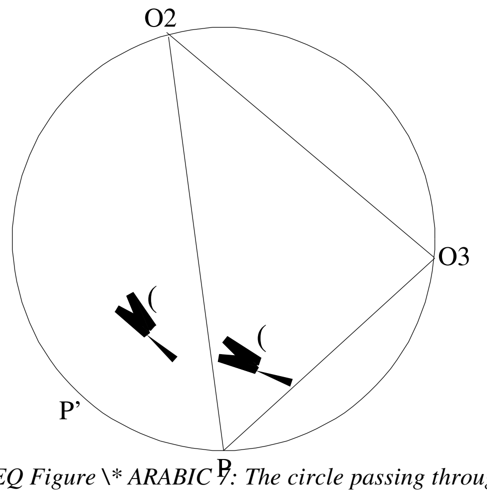


*Figure SEQ Figure * ARABIC 7: The circle passing through O_2, O_3, P and P'*

$$l(X) = X \tan\phi + b$$

$$b = \frac{Y_{O_2} + Y_{O_3}}{2} + \frac{(X_{O_2} + X_{O_3})(X_{O_3} - X_{O_2})}{2 \cdot (Y_{O_3} - Y_{O_2})}$$

$$X_{P'} = X_{O_2} - O_2P' \sin(\varphi - \frac{\gamma}{2})$$

$$Y_{P'} = l(X_{P'})$$

$$\varphi = \arctan\left|\frac{Y_{O_3} - Y_{O_2}}{X_{O_3} - X_{O_2}}\right|, \phi = \arctan\left|\frac{X_{O_3} - X_{O_2}}{Y_{O_3} - Y_{O_2}}\right|$$

Y

X

2

1

$$l(X) = X \tan\phi + b$$

$$b = \frac{Y_{O_2} + Y_{O_3}}{2} + \frac{(X_{O_2} + X_{O_3})(X_{O_3} - X_{O_2})}{2 \cdot (Y_{O_3} - Y_{O_2})}$$

$$X_{P'} = X_{O_2} + O_2P' \sin(\varphi - \frac{\gamma}{2})$$

$$Y_{P'} = l(X_{P'})$$

$$\varphi = \arctan\left|\frac{Y_{O_3} - Y_{O_2}}{X_{O_3} - X_{O_2}}\right|, \phi = \arctan\left|\frac{X_{O_3} - X_{O_2}}{Y_{O_3} - Y_{O_2}}\right|$$

3

4

Figure 3: Calculating the coordinates of P'

Let us now focus on the objects O_2 and O_3 and the angle γ (relative to each other) between them, as seen from P. Every point from which O_2 and O_3

have an angle γ (relative to each other) between them is located on the circle that passes through O_2, O_3, P and a point P’ that is the apex of the isosceles triangle having base O_2O_3 and an apex angle of γ degrees (Figure 7).

Our first objective is to calculate the coordinates of P’. To be able to complete that calculation, we have to distinguish between four situations, depending on the slope *a* of the perpendicular *l* to O_2O_3 (Figure 8.1: $0° \le a < 90°$; Figure 8.2: $90° \le a < 180°$; Figure 8.3: $180° \le a < 270°$; Figure 8.4: $270° \le a < 360°$). We will show how the coordinates of P’ are calculated for the situation depicted in Figure 8.1, similar calculations can be used in the other three situations. Looking at Figure 8.1 we can see that:

$$X_{P'} = X_{O_2} - O_2P' \cdot \sin(\varphi - \frac{\gamma}{2})$$

$$O_2P' = \frac{\frac{1}{2} O_2O_3}{\sin\frac{\gamma}{2}}$$

$$O_2O_3 = \sqrt{(X_{O_3} - X_{O_2})^2 + (Y_{O_3} - Y_{O_2})^2}$$

$$\varphi = \arctan\left|\frac{Y_{O_3} - Y_{O_2}}{X_{O_3} - X_{O_2}}\right|$$

We can also see that P’ is located on *l*. Therefore $Y_{P'} = l(X_{P'})$. Now we just need to derive the equation of *l*. Figure 8.1 shows that the slope of *l* equals tan ψ, where $\varnothing = \arctan\left|\frac{X_{O_3} - X_{O_2}}{Y_{O_3} - Y_{O_2}}\right|$. Knowing that $\left(\frac{X_{O_2} + X_{O_3}}{2}, \frac{Y_{O_2} + Y_{O_3}}{2}\right)$ is located on *l* gives us the equation, which, in turn, gives us the coordinates of P’.

Now we know the coordinates of P’, we can calculate the coordinates of the centre M_{23} of the circle circumscribing O_2O_3P' (Figure 9) and the radius of that circle. From Figure 9 it should be clear that M_{23} is both on line *l* and line *k*, the perpendicular to O_2P'.

Because the coordinates of O_2 and P’ are now known, we can derive the

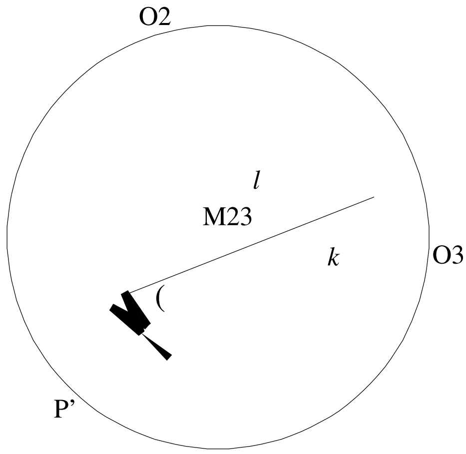


*Figure SEQ Figure * ARABIC 9: Constructing the centre of the circle passing through O_2, O_3 and P’*

equation *k*(*X*) for *k* in the same way as we have derived the equation for *l*. This gives us:

$$k(X) = -\frac{X_{P'} - X_{O_2}}{Y_{P'} - Y_{O_2}} X + \frac{Y_{O_2} + Y_{P'}}{2} + \frac{(X_{O_2} + X_{P'})(X_{P'} - X_{O_2})}{2 \cdot (Y_{P'} - Y_{O_2})}$$

Solving $k(X) = l(X)$ gives us $X_{M_{23}}$, the *X*–coordinate of M_{23}. Substituting

$X_{M_{23}}$ into the equation of l, gives us $Y_{M_{23}}$. It follows, then, that $r_{23} = \sqrt{(X_{M_{23}} - X_{O_2})^2 + (Y_{M_{23}} - Y_{O_2})^2}$, where r_{23} is the radius of the circumscribing circle of O_2O_3P'.

In a similar way we can calculate the coordinates of M_{12} and M_{13} (Figure 10) and the radii r_{12} and r_{13} of the circles they are the centres of. Knowing the centres and radii of two circles, we can calculate the intersection point(s) of those circles ([BOUR1997]). Because we know that P is on both circles, we know that P is one of those intersection points. So, if the two circles just have one intersection point, then that point must be P. If there are two intersection points, just take one of the two and check whether its distance to the centre of the third circle equals the radius of the third circle. If so, that point has to be P, because P is the only point located on all three circles. If not, the other of the two intersection points has to be P. This concludes our calculations, giving us the coordinates of P.

Problem: The origin of the coordinate system is always the centre of the pitch. The

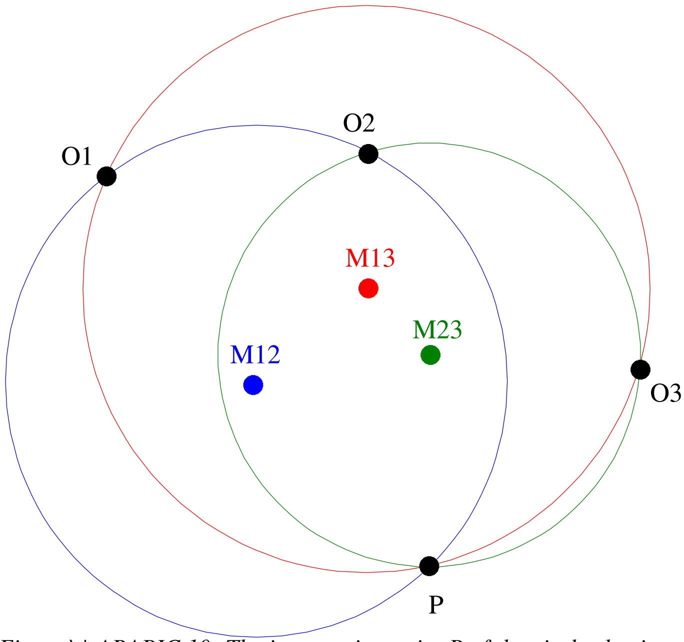


*Figure SEQ Figure * ARABIC 10: The intersection point P of the circles having centres M_{12}, M_{13} and M_{23}*

directions the positive axes of that system are pointing to, however, depend on the side the player is playing from. If the player is playing from left to right, the positive *X*–axis is directed towards the right (i.e. opposing) goal and the positive *Y*–axis is directed towards the bottom of the pitch. If the player is playing from right to left, the positive *X*–axis is directed towards the left (i.e. opposing, again) goal and the positive *Y*–axis is directed towards the top of the pitch.

Solution: The only fact that remains the same in both the situations, is the fact that the player is playing towards the opposing goal. So we decided to use a coordinate system relative to the side the player is playing from. This means that the positive *X*–axis is directed towards the opposing goal and the positive *Y*–axis is directed towards the right sideline, i.e. the sideline closest to the right post of the opposing goal. This decision also entailed that we had to convert the absolute coordinates to coordinates from the point of view of the player. Because it was possible to let our relative coordinate system coincide with one of the two absolute coordinate systems, we only had to convert the coordinates from the other system. We decided to let our relative coordinate system coincide with the absolute coordinate system that is used when the player plays from right to left. This meant that we had to write a method that converts the coordinates from the system that is used when the player plays from left to right to our relative coordinate system, i.e. left becomes right, right becomes left, top becomes bottom and bottom becomes top. The (polymorphic) method that does this is called fromMyPointOfView.

Problem: How to deal with objects that were in sight the last time but that are not anymore? Because only the information about objects that are in sight is updated, the information on objects that have disappeared out of sight is obsolete, yet still present. As a result of which, the player sees, and, consequently, possibly starts to pursue, things that aren't there ('ghost objects').

Solution: If the information of an incoming view message has to be processed and this view message is indicating the end of a sequence of incoming see messages belonging to each other, information about objects that should be in sight, according to that information (the angle the object is supposed to be seen in is within the player's view), but that are not, according to that last sequence of incoming see messages, is discarded.

Problem: How to deal with incoming referee messages?

Solution: Every hear message that has a sender part that reads referee is converted to an incoming referee message by the sensors, translating the message part of the original hear message to a message type (, the first nine referee messages correspond to the external modes of play, the play–modes used by the Soccer Server). The *Side* part of the message part of the hear message is changed to the point of view of the player, as mentioned before. Depending on the message type and, if applicable, the information it contains about the side it applies to, from the point of view of the player, a particular action is performed (). Sometimes this involves updating the internal mode of play (i.e. the play mode from the point of view of the player). contains all the possible internal modes of play.

Referee message	**Referee message type**
before_kick_off	BEFORE_KICK_OFF
time_over	TIME_OVER
play_on	PLAY_ON
kick_off_*Side*	KICK_OFF
kick_in_*Side*	KICK_IN
free_kick_*Side*	FREE_KICK
corner_kick_*Side*	CORNER_KICK
goal_kick_*Side*	GOAL_KICK
goal_ *Side*	GOAL
goalie_catch_ball_*Side*	GOALIE_CATCH_BALL
foul_ *Side*	FOUL
half_time	HALF_TIME
time_up	TIME_UP
time_up_without_a_team	TIME_UP_WITHOUT_A_TEAM
time_extended	TIME_EXTENDED
offside_ *Side*	OFFSIDE
drop_ball	DROP_BALL

Table 4: Referee messages

Referee message type	Side	Description	Action
BEFORE_ KICK_OFF	N.a.	Kick−off.	Set internal play mode to PM_BEFORE_KICK_OFF.
KICK_OFF	Ours	The player’s team is to kick off.	Set internal play mode to PM_OUR_KICK_OFF and start the clock.
	Theirs	The opposing team is to kick off.	Set internal play mode to PM_THEIR_KICK_OFF and start the clock.
KICK_IN	Ours	The player’s team is to kick in.	Set internal play mode to PM_OUR_KICK_IN.
	Theirs	The opposing team is to kick in.	Set internal play mode to PM_THEIR_KICK_IN.
CORNER_KICK	Ours	The player’s team is to take a corner kick.	Set internal play mode to PM_OUR_CORNER_KICK.
	Theirs	The opposing team is to take a corner kick.	Set internal play mode to PM_THEIR_CORNER_KICK.
GOAL_KICK	Ours	The player’s team is to take a goal kick.	Set internal play mode to PM_OUR_GOAL_KICK.
	Theirs	The opposing team is to take a goal kick.	Set internal play mode to PM_THEIR_GOAL_KICK.
GOALIE_CATCH_BALL	Ours	The player’s goalkeeper has caught the ball.	None, FREE_KICK to follow.
	Theirs	The opposing goalkeeper has caught the ball.	None, FREE_KICK to follow.

Referee message type	Side	Description	Action
FREE_KICK	Ours	The player's team is to take a free kick.	Set internal play mode to PM_OUR_FREE_KICK.
	Theirs	The opposing team is to take a free kick.	Set internal play mode to PM_THEIR_FREE_KICK.
DROP_BALL	N.a.	The referee has dropped the ball.	Reset ball position.
PLAY_ON	N.a.	Game on.	Set internal play mode to PM_PLAY_ON.
HALF_TIME	N.a.	Half time.	Stop the clock.
TIME_UP	N.a.	Game over.	None, TIME_OVER to follow.
TIME_UP_ WITHOUT_ A_TEAM	N.a.	One of the teams has failed to show up.	None, TIME_OVER to follow.
TIME_ EXTENDED	N.a.	Overtime.	Stop the clock.
TIME_OVER	N.a.	Game over (change of play mode).	Set internal play mode to PM_TIME_OVER and stop the clock.
FOUL	Ours	The player's team has committed a foul.	None, FREE_KICK to follow.
	Theirs	The opposing team has committed a foul.	None, FREE_KICK to follow.
GOAL	Ours	The player's team has just scored a goal.	Set internal play mode to PM_OUR_GOAL and update the score.
	Theirs	The opposing team has just scored a goal.	Set internal play mode to PM_THEIR_GOAL and update the score.
OFFSIDE	Ours	The player's team is offside.	None, FREE_KICK to follow.
	Theirs	The opposing team is offside.	None, FREE_KICK to follow.

Table 5: Dealing with incoming referee messages

PM_NONE
PM_BEFORE_KICK_OFF
PM_TIME_OVER
PM_PLAY_ON
PM_OUR_KICK_OFF
PM_THEIR_KICK_OFF
PM_OUR_KICK_IN
PM_THEIR_KICK_IN
PM_OUR_FREE_KICK
PM_THEIR_FREE_KICK
PM_OUR_CORNER_KICK
PM_THEIR_CORNER_KICK
PM_OUR_GOAL_KICK
PM_THEIR_GOAL_KICK
PM_OUR_GOAL
PM_THEIR_GOAL

Table 6: Internal modes of play

3.2.1.7 The TeamInfo class

Information on a team is recorded in the TeamInfo class. It contains the information about eleven players (i.e. instances of the PlayerInfo class).

3.2.1.8 The FieldObjectInfo class

Information about objects on the pitch is stored in the FieldObjectInfo class. It contains the direction the object is moving in and its current position and speed.

3.2.1.9 The PlayerInfo class

Information about a player is contained in the PlayerInfo class, which is a subclass of the FieldObjectInfo class. Besides the information stored in that class, it also contains the player's uniform number and stamina.

Problem: How to deal with the stamina, recovery and effort of a player?

Solution: The interaction of stamina, recovery and effort is described on page 38 in [KUMM1999]. We will restrict ourselves here to explaining how we have implemented what is said there. Because the player that has to be controlled is the only player, whose stamina, recovery and effort are of any relevance to us, our explanation will also be limited to this player.

Whenever we execute a dash and the stamina is within 100 of the recovery threshold (i.e. RECOVER_DEC_THR * STAMINA_MAX), the maximum power of that dash is limited to the difference between the current stamina and that recovery threshold. This is done in the getMaxDashPower–method of the PlayerInfo class. In addition to that, the actual power of the dash will be reduced further in such a way that, if the player is already moving, the resulting velocity will not exceed the maximum player speed (PLAYER_SPEED_MAX). This is also done in the getMaxDashPower–method of the PlayerInfo class.

We do not keep track of the player's effort, because we are ensuring it to remain valued 1.0 at all times. We accomplish this by simply not dashing too fast (i.e. limiting the maximum dash power whenever the stamina is within 100 of the recovery threshold, as mentioned before).

3.2.1.10 The Clock class

The time of the match is recorded by the Clock class. The clock is started at kick–off, synchronised on the arrival of (timed) messages from the server and stopped at the end of a half.

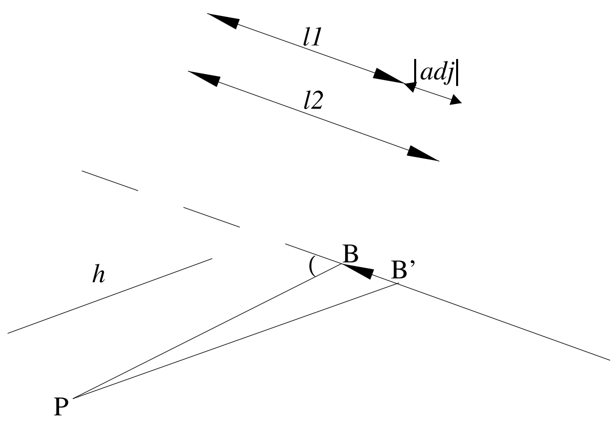
3.2.1.11 The Simulator class

The Simulator class contains functions that can be called to simulate the game in advance. These functions can be used to predict the position or the speed of a player or the ball after an arbitrary number of cycles and/or actions (e.g. dashes, kicks, et cetera). The future state of a movable object is calculated in the same way it is computed by the server.

Problem: The player and the ball cannot overlap (i.e. occupying (approximately) the same space at the same time). How does the simulator deal with this?

Solution: We have made use of the same collision−checking algorithm as is used by the server. As depicted in Figure 11, collision occurs when the current positions P of the player and B of the ball are less than PLAYER_RADIUS + BALL_RADIUS apart (when player and ball overlap, it is called a collision). If this is the case, the ball is moved backwards in such a way, that the distance between the position P of the player and the new position B' of the ball equals the aforementioned sum of radii.

The length of the vectorial adjustment *adj* that has to be made to the position of the ball to move it from B to B' can be calculated in the following fashion: Given the current positions P of the player and B of the ball and given the direction the ball is moving in, the angle θ can be calculated. This θ can then be used to calculate *h* and l_1. Next, *h* and the distance between P and B' (equal to PLAYER_RADIUS + BALL_RADIUS) are used to calculate l_2. Subtracting l_1 from l_2 gives us $|adj|$. Combining this with the direction the ball is moving in, we get the vectorial adjustment *adj*.



Note that, according to the server, collision occurs when player and ball are less than PLAYER_SIZE + BALL_SIZE apart. Taking a closer look at this, we discovered that what is meant by PLAYER_SIZE and BALL_SIZE actually amounts to the radius of the player and the ball, respectively. For clarity reasons, we have renamed PLAYER_SIZE to PLAYER_RADIUS

*Figure SEQ Figure * ARABIC 11: Collision detection*

and BALL_SIZE to BALL_RADIUS.

3.2.1.12 The ActionHistory class

The primary purpose of the ActionHistory class is to keep track of all the actions that have been dispatched ever since the first of the last two sense body messages has been received. This action history can be used to recover which actions have reached the server and which (up till now) have not. The information about dash actions is especially handy, because, together with the information on the stamina that was contained in the last sense body message, this can be used to make a more accurate estimation of the current stamina of the player. We do not use the stamina information from the latest incoming sense body message directly, because it is not up−to−date, as it is inevitable that, at the server side, the player's stamina has already been updated due to dash messages arriving whilst the sense body message was on its way back. If one knows what dash actions are still on their way to the server whenever a sense body message arrives, the stamina that results from simulating those dashes, beginning with the stamina contained within the latest sense body message, is very much closer to the real thing, assuming no dash gets lost on its way to the server.

Problem: How to recover which actions have reached the server and which (up till now) have not?

Solution: Every incoming sense body message contains information about the number of times the server has received a kick, a dash, a turn or a say (KICK_COUNT, DASH_COUNT, TURN_COUNT and SAY_COUNT, respectively). Note that, if one sends several messages of the same type in sequence and one of them gets lost, one can never know which one. Looking at the corresponding count, one can only know the number of messages that has been lost. This might be acceptable for non–dash actions, as non–dash actions do not affect the stamina. A dash, however, does affect the stamina, so to be able to make an accurate estimation of the player's current stamina one would like to know exactly which dash actions have been executed and which have not.

To acquire that knowledge, we have used the first two decimals of the player's stamina for a cunning encoding scheme. The reason why we only use the first two decimals of the player's stamina is the fact that the server truncates the value of the stamina contained within a sense body message in such a way that only those two decimals remain when the message arrives at the client. As a consequence, the maximum number of dashes that can be stored in the action history is 6, as 2^6 is the largest power of 2 that consists of two digits.

The encoding scheme we employ operates in the following fashion: The i^{th} dash in the action history is assigned the number 2^{i-1}, so the first action is numbered 1, the second 2, the third 4, and so on. Before sending that i^{th} dash to the server, its power is augmented with $2^{i-1} / 10^2$. For example, suppose that the fifth dash action in the history has an original power of, say, 60. Thanks to our encoding scheme this will be sent to the server as a dash having power 60.16. Because the server subtracts the power of an incoming dash from the player's stamina, the decimal part of that stamina can be inspected to find out which dashes have been executed. For example, suppose a sense body message arrives conveying a stamina having an all–zero decimal part and the stamina contained within the next sense body message has a decimal part that reads 81. This means that the server has subtracted 19, as a result of incoming dash actions. By rewriting 19 as the sum of 1, 2 and 16, taking only powers of 2 into consideration, it becomes clear that, the moment the last sense body message was sent, the first, second and fifth dash had already been executed by the server and the third and fourth dash had been lost.

Concerning the non–dash actions, we can only find out the number of those actions that has reached the server by inspecting the corresponding count. Therefore, we just go through the action history from front to back and every time we encounter a non–dash action we decrease the value of a variable indicating the maximum number of actions of that type that the list can still retain, according to the original count.

So now we know which (dash and non–dash) actions in the history the server has executed already. Knowing this, we can draw up a list of actions that are either on their way to the server but have not reached it yet or in the queue of outgoing action messages, ready to be sent. This list can then be used to simulate the current state of both player and ball.

Problem: How to make a more accurate estimation of the current stamina of the player, when one knows which dash actions are on their way to the server, but have not reached it yet?

Solution: Create an instance of the PlayerInfo class by combining the currently known player information and the information on the stamina contained within the last sense body message received. Use this newly generated player information to simulate (using the Simulator class) those actions that have been sent to but have not arrived at the server. The resulting player information contains the requested stamina.

3.2.1.13 The Mind class

The Mind class is responsible for generating the behaviour of the player. Therefore, the Mind class contains the AI of the TBP1999 simulated soccer player. Beforehand, we have identified three different ways to implement this AI:

1. Using genetic programming to evolve a player that exhibits the appropriate behaviour.
2. Neural networks that receive the input from the Soccer Server and generate the appropriate behaviour.
3. Hand–coding the responses to the Soccer Server input, or 'rules', to create the appropriate behaviour.

The first option has been rejected, because the work on the RoboCup entry that has already applied the concept of genetic programming has revealed the need of extreme computing power for an extended period of time (Sean Luke et al., [KUMM1999]). Of the two remaining options the approach of hand–coding rules is the most widely used in existing simulated soccer players. It has also been the most appealing approach, as we know from experience that neural networks require an enormous amount of tweaking and tuning to get anywhere near a desirable result. Therefore, we have decided to implement the AI of the TBP1999 simulated soccer player by hand–coding the player's responses to the Soccer Server into the player's mind.

For this purpose, the Mind class provides two abstract methods that can be implemented to accommodate the performing of one or more actions. The first abstract method, called thinkFirst, can be used to execute actions that have to be performed only once at the beginning of a match, e.g. moving the player to a particular position within the field. The second abstract method, called think, can be used to perform actions a particular fixed interval apart. The interval we use here is equal to the sum of the simulator step and the send step, i.e. the maximum time it takes the server to receive, process and give confirmation of the execution of the actions sent by the client. This means that the next time the think–method is executed the client's match information has been updated to reflect the current situation in the server, thus enabling the player to use the most up–to–date representation of the simulated environment. Every time the think–method is executed, a sense body message is sent to the server to get an update on the player's condition. If the server is jammed with too many commands the rate at which the sense body messages are sent can be reduced.

Because the thinkFirst– and think–methods are abstract, the entire Mind class is abstract, too. Therefore, to be able to run our player, a class must be created that extends the Mind class and that implements the two abstract method mentioned before. The NeverMind class is the default implementation that will be used, when the player hasn't been provided with an alternative.

Problem: Every method called from within the same thinkFirst– or think–method has to use the same information about the match (i.e. the instance of the MatchInfo class used by the mind should not be updated as long as the thinkFirst– or think–method is executing). Otherwise, it is possible for the mind to use not yet updated and already up–to–date information about the match simultaneously, thinking it belongs to one and the same situation, because an update has occurred between the subsequent fetching of information.

Solution: Synchronising the calls to the thinkFirst– and think–method by wrapping it in using Java's synchronized–statement ensures the instance of the MatchInfo class to be locked during the execution of the thinkFirst– or think–method. This effectively prevents other threads from accessing and updating the object in question in the mean time.

3.2.1.14 The NeverMind class

By default, an instance of the NeverMind class will be used to reason about the match and generate the appropriate actions. The behaviour exhibited by a player that makes use of a NeverMind is extremely rudimentary. While the internal mode of play equals PM_PLAY_ON and no other teammate has indicated to be in possession of the ball, the player turns as it searches for the whereabouts of the ball. When it has found them, it runs over there and starts kicking the ball towards the opponent's goal until a goal is scored. While the player thinks it is in possession of the ball it relays this notion to the other players on the RoboCup Simulated League soccer pitch.

Problem: How is the player able to accurately respond to the current state of affairs of a match?

Solution: Every time the think–method is executed, the current behaviour is abandoned (i.e. the current plan is aborted), for it is most likely that situations have changed since the time that behaviour was adopted. New behaviour is generated from the current match information about teammates and opponents and the information about player and ball simulated from the action history. Note that the minimum size of the newly generated plan, i.e. MIN_PLAN_SIZE = 5, is large enough to keep the actuators busy between subsequent invocations of the think–method.

Problem: How does the player know who is in possession of the ball?

Solution: Every time a particular player perceives the ball to be within kicking distance, that player broadcasts a friendly say message containing its player number, the current match time and the number of times this message has already been broadcast (the hop count, initially 0). Whenever the player receives such a message that it has not already received (i.e. the match time contained within this message differs from the match time contained within the last one) and the hop count is smaller than the HOP_LIMIT (i.e. the length of the diagonal of the RoboCup Simulated League soccer pitch divided by AUDIO_CUT_DIST, the maximum distance a spoken message can reach), it records the player having the number that is contained in the message to be in possession of the ball and re–broadcasts the original message with an increased hop count. After a particular time has elapsed (i.e. MAX_POSSESSION_ASSUMPTION_ TIME) without the arrival of a possession message, the record is reset to reflect that the opposing team is in possession of the ball (i.e. the record contains NOT_A_PLAYER, an invalid player number). By inspecting the record the player is able to determine if it itself is in possession (i.e. the record contains its own player number), if one of its teammates is in possession (i.e. the record contains another valid player number) or if the opposing team is in possession of the ball (i.e. the record contains the invalid player number).

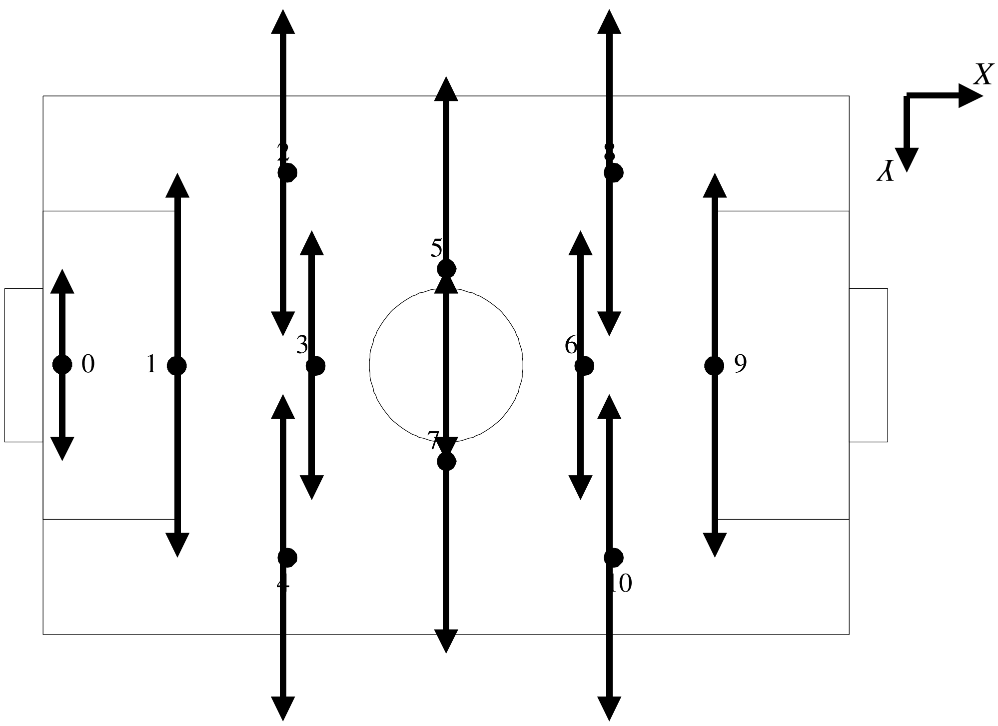
3.2.1.15 The MasterMind class

The MasterMind class is the mind that is ultimately going to be used for the TBP1999 simulated soccer player. It applies the same techniques as the NeverMind class to generate an accurate response to the environment and to deal with the possession of the ball. These techniques have been discussed in the previous section.

The main part of the MasterMind class consists of checking and acting upon the internal mode of play ().

Internal mode of play	Action
PM_BEFORE_KICK_OFF	If the player is not already at its default position, it moves over there and faces its default direction.
PM_TIME_OVER	Game over. The player quits.
PM_PLAY_ON, PM_OUR_KICK_OFF, PM_OUR_KICK_IN, PM_OUR_FREE_KICK, PM_OUR_CORNER_KICK, PM_OUR_GOAL_KICK	If the player does not perceive the ball, it turns until it does. If the whereabouts of the ball are known, the player checks who is in possession. If the player itself is in possession, the ball is within kicking distance, the player is near enough to the opposing goal and there is no one in the line of fire, the player takes a shot at goal. If the player is not near enough to the opposing goal or someone is in the way, it searches for a teammate to pass the ball to. If the player does not find one, it starts to dribble the ball. If the ball is out of range, the player runs to it. If the player itself does not possess the ball, but the ball is within kicking distance, the player considers itself to be in possession now and broadcasts this notion. Otherwise, the player looks to see if, of all the players in its team, it is closest to the ball. If it is, it tries to intercept it. If it is not, it assumes its default position.
PM_NONE, PM_THEIR_KICK_OFF, PM_THEIR_KICK_IN, PM_THEIR_FREE_KICK, PM_THEIR_CORNER_KICK, PM_THEIR_GOAL_KICK, PM_OUR_GOAL, PM_THEIR_GOAL	The player runs to its default position.

Table 7: Acting upon the internal mode of play

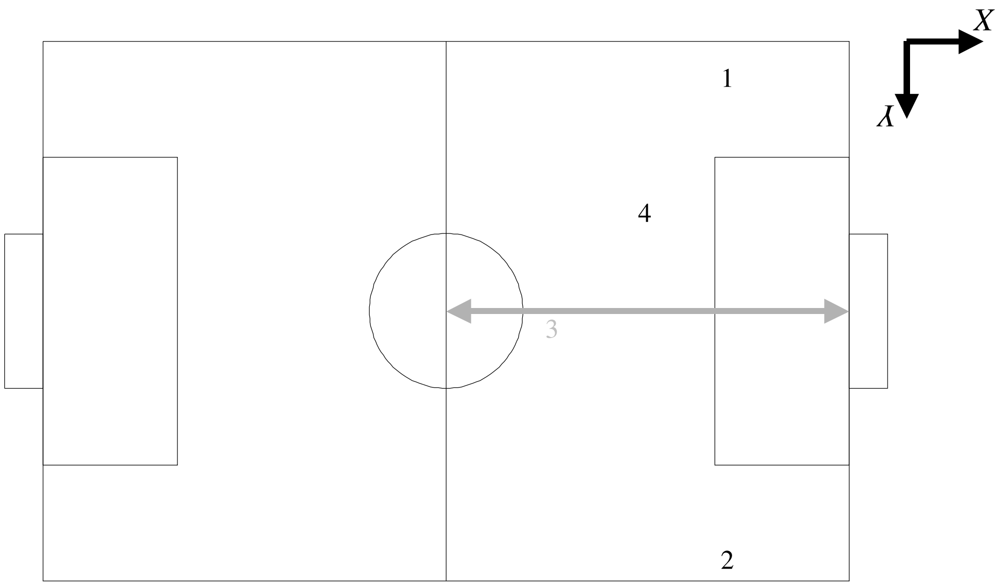


The default position of a player, mentioned in , depends on the formation of the team. The formation the TBP1999 team is depicted in Figure 12. Each bold line represents the optimal positional line of the player having the number accompanying the line. The optimal positional line of a player is the collection of positions on the field that have the highest fitness, i.e. are the most favourable places for a player to be. The farther a player is removed from its optimal positional line, the lower its positional fitness. The dot in the middle of such an optimal positional line represents that player's default position. So, the default position of the goalkeeper (player number 0) is directly in front of the centre of the goal on the right (providing the MasterMind class is used to control the goalkeeper). When the team has to be entirely on its own half (e.g. before the kick–off or after the scoring of a goal), the formation is 'shrunk' to fit on that half (i.e. the horizontal distance from each player to the default position of the goalkeeper is reduced by a factor of two).

Apart from assigning fitness to a player's position according to optimal positional lines, fitness can also be assigned to a player's position according to the pattern of play adopted by the team. We have defined a total of four patterns of play (Figure 13):

1. Launching an attack on the left wing (LEFT_WING_ATTACK).
2. Launching an attack on the right wing (RIGHT_WING_ATTACK).
3. Launching an attack through the middle (CENTER_ATTACK).

*Figure SEQ Figure * ARABIC 12: TBP1999 team formation*



4. Power play: Pushing the opposing team back into their penalty area (POWER_PLAY).

*Figure SEQ Figure * ARABIC 13: Patterns of play*

The position of the ball determines the pattern to be played. To this end we have divided the field into three horizontal sections and three vertical sections. The horizontal sections are (Figure 14):

1. The left flank, i.e. the section closest to the top sideline.
2. The central corridor, i.e. the middle section connecting both goals.
3. The right flank, i.e. the section closest to the bottom sideline.

The vertical sections are (Figure 15):

1. The home zone, i.e. the section closest to the player's own goal.
2. The mid–zone, i.e. the section surrounding the centre line.
3. The end zone, i.e. the section closest to the opposing goal.

If the ball is in the end zone, the team is going to execute a power play. If the ball is in the home zone, no pattern of play is adopted. If the ball is in the mid–zone and on the left flank, the team is going to launch an attack on the left wing. If the ball is in the mid–zone and in the central corridor, the team is going to launch an attack through the middle. If the ball is in the mid–zone and on the right flank, the team is going to launch an attack on the right wing.

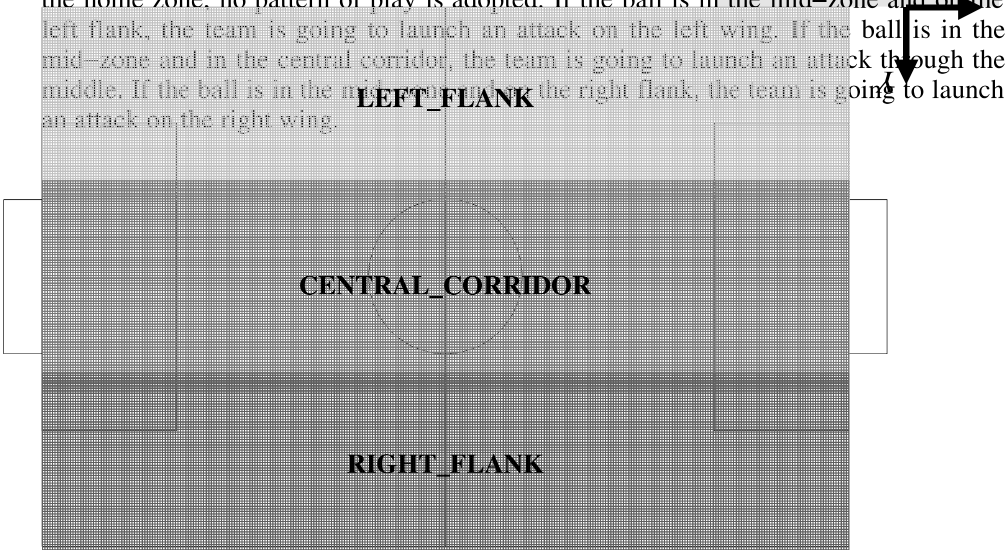


*Figure SEQ Figure * ARABIC 14: Horizontal sections*

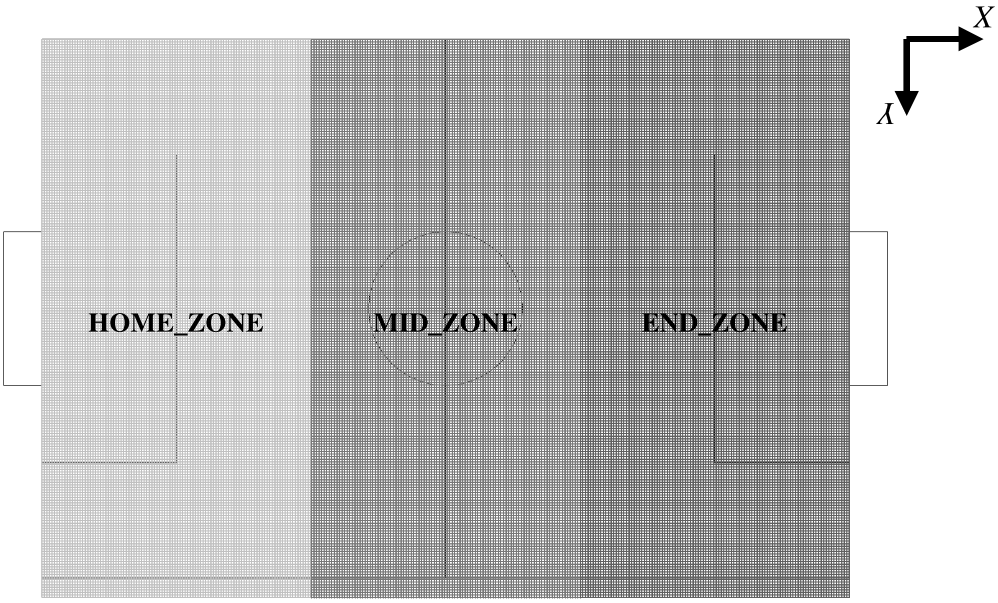


*Figure SEQ Figure * ARABIC 15: Vertical sections*

A particular pattern of play is composed of one or more lines having the highest fitness (i.e. being the most favourable for a player to stay). The farther a player is removed from the set of lines forming a pattern, the lower its patternwise fitness.

Combining its positional and patternwise fitness gives us the (overall) fitness of a particular player.

Problem: How does the player run to a particular target position?

Solution: First, the player turns towards the position to run to. Next, it starts simulating and executing dashes until the resulting simulated position equals the target position.

Problem: How does the player intercept the ball?
Solution: First, the player calculates the position of interception by simulating the movement of the ball until its resulting position can be reached at maximum speed. A particular position can be reached if the player is able to arrive at that position in about the same number of cycles it has taken the ball to get there. Next, it runs to the aforementioned calculated position of interception.

Problem: How does the player turn with the ball to a particular target direction?
Solution: First, the player determines which direction to turn to, i.e. clockwise or counterclockwise. If the ball trajectory in the direction that requires the fewest turns to reach the target direction is blocked by another player or crosses a line to go out of bounds, the player turns in the opposite direction. Next, the player starts kicking the ball in that direction until the target direction is roughly reached. In the mean time the player is seeing to it that it keeps the ball at a distance of about OPTIMAL_CONTROL_DISTANCE = 1.6, the distance at which the player is able to optimally handle the ball (the value of 1.6 has been taken from Carnegie Mellon University's implementation of a RoboCup Simulator League soccer player, CMUnited). Any movements on the player's part during execution of the turn are compensated for.

Problem: How does the player dribble to a particular target position?
Solution: First, the player calculates the required kicking power from the dashing power and dash–to–kick–ratio supplied. As a result of this the player is going to be able to continuously execute a number of dashes per kick equal to the aforementioned ratio. If the ball is within kicking distance, the player considers kicking the ball. First, it checks whether its body is in the way when the ball is kicked towards the target position. If the ball is going to bounce off its body, the player turns with the ball until it is not going to do so. Next, it simulates the kick and determines if the ball is going to end up closer to the target position with or without that kick (taking into account the inertia). If kicking the ball will result in it finishing up closer, the kick is executed. Otherwise, the player will perform a kick bringing the ball to a full stop. In any case, the player runs to the expected position of the ball. It will continue kicking (in case the ball is within kicking distance) and running until its position approximately equals the target position.

Problem: How does the player determine where to dribble to?
Solution: First, the player draws up a list of teammates and opponents in its immediate vicinity. Next, it sorts them according to the relative directions it is perceiving them in and it determines the largest gap between subsequent directions. The dribbling direction is then set in such a way that the dribbling trajectory runs right through the middle of that gap.

Problem: How does the player pass the ball to a particular target player?
Solution: The player kicks the ball towards the position the target player is at, using the given speed.

Problem: How does the player choose which player to pass to?
Solution: The player examines the position of every one of its teammates. If the ball cannot be intercepted by an opponent when it is kicked to the position of a particular teammate, the fitness of that teammate is recorded according to the fitness type supplied (i.e. positional, patternwise or overall). Ultimately, the player having the highest fitness value is chosen.

Problem: How does the player shoot at the goal?

Solution: First, the player kicks the ball to a position just inside the circle having a radius equalling the player's KICKABLE_AREA, the maximum distance within which the player can still kick the ball. It does this in such a way, that the trajectory from the position of placement to the goal is not blocked by its body. Next, it starts to kick the ball in such a way, that it follows the trajectory towards the goal. It continues to kick the ball until it its out of reach, thus giving it the accumulated velocity of multiple kicks.

Problem: Apart from being slow, as we have mentioned in section , the Java 1.1.7b compiler also makes use of an enormous amount of system resources. As a result of this, we have not been able to run more than one client on a single computer. Increasing the number of clients per machine results in a disorientation of some players, due to a lack of available resources, resulting in the client's inability to timely construct an internal representation from the server input. Consequently we have not been able to assemble a fully co–operating team of eleven players, let alone make them play against an opposing team. So we have been unable to completely test the built–in AI of the master mind. We have only been able to try out parts of it by creating an environment containing a limited (i.e. two or three) number of players.

Solution: Similar to the problem of Java's sluggishness discussed in section , there is not really a solution to this problem either. Although setting the Soccer Server to run at half its original speed by doubling the simulator and send steps (a tip from [KUMM1999]) initially enabled us to increase the number of clients per computer, enhancing the complexity of the mind has cancelled out the benefit in such a way that testing our eventual implementation would have required us to have at our disposal a couple of more powerful computers. We have, however, successfully tested the player's running, intercepting, turning, dribbling, passing and shooting skills separately and seeing as the behaviour generated by the master mind is a composition of these abilities, we assume that the composite behaviour will also measure up to our expectations.

3.2.1.16 The GoalieMind class

The GoalieMind class is the mind that is ultimately going to be used for the TBP1999 simulated goalkeeper. It applies the same techniques as the NeverMind and MasterMind classes to generate an accurate response to the environment and to deal with the possession of the ball. These techniques have been discussed in section .

If the ball is within the quarter of the RoboCup Simulator League soccer pitch that is closest to the TBP1999 goal, the GoalieMind class positions the goalkeeper a quarter of the way between the centre of that goal and the ball. If the ball is outside the aforementioned quarter of the field, the goalkeeper moves to the centre of the penalty area. If the ball is going to be within reach the next cycle, the goalkeeper will try to catch it and kick it away.

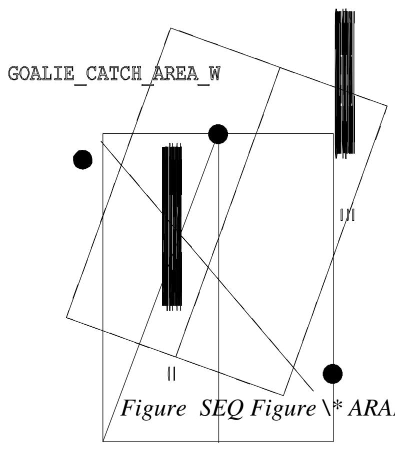


*Figure SEQ Figure * ARABIC 16: The goalkeeper's catchable area*

Problem: When is the goalkeeper able to catch the ball?

Solution: The goalkeeper can catch the ball when the ball is within its catchable area, a rectangular area GOALIE_CATCH_AREA_W wide and GOALIE_CATCH_
AREA_L long (Figure 16, the goalkeeper being the upperleft black dot having a line of sight that is perpendicular to the line representing the catchable area width).

Note that, if a ball is at a particular position that is not within the goalkeeper's catchable area, this does not necessarily mean that is it not catchable at all. It may well be that that position is within the catchable area if the goalkeeper turns. In Figure 16 on the left, the bottomright black dot is not within the goalkeeper's catchable area, on the right it is. A ball is completely uncatchable if its distance to the goalkeeper is greater than the MAXIMAL_ CATCH_DISTANCE.

3.2.2 The auxiliary classes

This section will cover those classes that support the aforementioned primary classes in operating correctly. The introduction of these auxiliary classes is purely a decision taken during implementation. Therefore, they do not appear in the TBP1999 simulated soccer player design depicted in Figures 2, 3 and 4.

3.2.2.1 The Location class

The Location class contains the functions used in the calculations covered in section .

3.2.2.2 The Logger class

The Logger class is used to log error, debug and/or normal messages to the screen and/or to a file. The messages are displayed in a three−column table, indicating the type of the message (i.e. ERROR, DEBUG or NORMAL), the method that has generated the message (e.g. Actuators::run, indicating the run−method of the Actuators−thread) and the actual content of the message, respectively.

3.2.2.3 The MessageQueue class

The MessageQueue class is responsible for storing the messages (i.e. instances of the Message class). A message can be entered into a message queue by invoking the message queue's putMessage−method. Getting a message from a message queue is done by calling the message queue's getMessage−method. If the message queue is empty, the getMessage−method does not return until another message is added to the queue by an invocation of the putMessage−method.

Problem: If a message queue is depleted, an invocation of the getMessage−method on that queue must not return until there is a message to return. In the meanwhile, the calling object must not perform a busy−waiting loop, but must go to sleep until it is awoken by the arrival of a new message. At first we did not have a separate MessageQueue class, but used the Java Vector class and its methods instead. As a result of this, the object containing the vector was made responsible for getting and putting a message and checking the message queue for emptiness. We assumed that the object getting the message was a thread that checked the message queue and, if it was empty, put itself to wait. On arrival of a new message the object containing the vector had to notify the waiting thread. This entailed two objects invoking each other's methods, which we deemed to be a problem.

Solution: What we are dealing with here, is nothing more and nothing less than the producer consumer problem, which has been solved by introducing the MessageQueue class with its synchronised getMessage− and putMessage−method ([ARNO1996]). The getMessage−method makes the calling object wait (sleep) while the message queue is empty and the putMessage−method notifies the waiting (sleeping) object on the arrival of a new message.

3.2.2.4 The CommonConstants class

lists the constants contained within the CommonConstants class and used throughout the other classes in alphabetical order.

Name	Type	Value	Description
AUDIO_CUT_DIST	double	50.0	The maximum distance a spoken message can reach.
BALL_DECAY	double	0.94	The percentage of ball speed remaining at the beginning of the next cycle.
BALL_RADIUS	double	0.085	Half the size of the ball.
BALL_SPEED_MAX	double	2.7	The maximum speed of the ball.
BOTTOM	int	3	The bottom side of the pitch.
CATCH_ANGLE_ ADJUSTMENT	double	$\arctan \frac{\text{GOALIE\_CATCHABLE\_AREA\_W}}{2 \cdot \text{GOALIE\_CATCHABLE\_AREA\_L}}$	The angle between the goalkeeper's line of sight and the line representing the MAXIMUM_CATCH_ DISTANCE (Figure 16).
CLIENT_TIMEOUT	int	3000	The maximum time in milliseconds the server is allowed to remain idle before the client closes the connection.
CLOCKWISE	int	1	In a clockwise direction.
COUNTERCLOCKWISE	int	−1	In a counterclockwise direction.
DASH_POWER_RATE	double	0.01	The multiplication factor used to convert the power of a dash to an actual speed.
DEFAULT_HOST	String	"poseidon"	The default host the client is trying to connect to.
DEFAULT_LOG	String	"TBP1999.log"	The default file to log to.
DEFAULT_MIND	String	"TBP1999.NeverMind"	The default mind used when no alternative is provided.
DEFAULT_PLAYERTYPE	boolean	PLAYER	By default, the client plays the part of a regular player.
DEFAULT_PORT	int	6000	The default port the client is trying to connect to on the default host.
DEFAULT_TIMER_HOST	String	"poseidon"	The default host this player's timer is running on.
DEFAULT_TIMER_PORT	int	7000	The default port the client is trying to connect to on the default timer host.
ENCODING_RANGE	double	1.0	The amount of stamina used for encoding purposes.
EPSILON	double	0.0001	The margin within which two doubles are called equal.
FIELD_EDGE	double	5.0	The distance between the border of the pitch and the flags outside the playing field.
FIELD_LENGTH	double	105.0	The length of the pitch.
FIELD_WIDTH	double	68.0	The width of the field.

Name	Type	Value	Description
FRIENDLY_ID	String	"TiaFM"	The string identifying a message from a teammate.
GOAL_WIDTH	double	14.64	The width of the goal.
GOALIE	boolean	true	The client plays the goalie part.
GOALIE_CATCHABLE_AREA_L	double	2.0	The length of the rectangular area within which the goalkeeper can catch the ball.
GOALIE_CATCHABLE_AREA_W	double	1.0	The width of the rectangular area within which the goalkeeper can catch the ball.
HOP_LIMIT	int	$\left\lceil \frac{\sqrt{\mathrm{FIELD\_LENGTH}^2 + \mathrm{FIELD\_WIDTH}^2}}{\mathrm{AUDIO\_CUT\_DIST}} \right\rceil$	The maximum number of times a spoken message can be relayed.
INERTIA_MOMENT	double	5.0	The effect the player's speed has on his/her capability to manoeuvre.
INITIAL_SEND_STEP	long	300	The initial value of the send step (in milliseconds).
INITIAL_SIMULATOR_STEP	long	200	The initial value of the simulator step (in milliseconds).
KICK_POWER_RATE	double	0.016	The multiplication factor used to convert the power of a kick to an actual speed.
KICKABLE_AREA	double	KICKABLE_MARGIN + BALL_RADIUS + PLAYER_RADIUS	The distance within which a player can kick the ball.
KICKABLE_MARGIN	double	1.0	The player's reach.
LEFT	int	0	The left side of the pitch.
MAX_DEVIATION	double	0.4	The maximum distance a perceived position deviates from the actual position.
MAX_DIR_DIFF	double	10.0	The maximum direction difference between player and target within which the player does not have to change direction.
MAX_PLAYER_DISTANCE	double	20.0	The distance within which the goalkeeper looks for players to pass the ball to.
MAX_POWER	double	100.0	The maximum power of a dash or a kick.
MAXIMUM_CATCH_DISTANCE	double	$\sqrt{\mathrm{GOALIE\_CATCHABLE\_AREA\_W}^2 + \mathrm{GOALIE\_CATCHABLE\_AREA\_L}^2}$	The maximum distance at which the goalkeeper is still able to catch the ball.
MIN_ANGLE	double	20.0	The minimum angle two flags have to be apart to be used in calculating the player's position.
MIN_PLAN_SIZE	int	5	The minimum number of actions in a plan to ensure smooth execution of that plan.

Name	Type	Value	Description
MIN_POWER	double	–30.0	The minimum power of a dash or a kick.
MINIMAL_CONTROL_ DISTANCE	double	0.97	The minimal distance between the player and the ball whilst turning with it.
NEWLINE	String	"\r\n"	The appearance of a transition to a new line on the system the player is running.
NOT_A_PLAYER	int	NUMBER_OF_ PLAYERS + 1	The identification for a player who is not part of any team.
NUMBER_OF_PLAYERS	int	11	The maximum number of players within one team.
OPTIMAL_CONTROL_ DISTANCE	double	1.6	The optimal distance between the player and the ball.
PENALTY_AREA_ LENGTH	double	16.0	The length of the penalty area.
PENALTY_AREA_ WIDTH	double	40.0	The width of the penalty area.
PLAYER	boolean	false	The client plays the part of a regular player.
PLAYER_DECAY	double	0.4	The percentage of player speed remaining at the beginning of the next cycle.
PLAYER_RADIUS	double	0.8	Half the size of a player.
PLAYER_SPEED_MAX	double	1.0	The maximum speed of a player.
PRIORITY_ACTUATORS	int	Thread.MAX_PRIORITY	The priority of the Actuators–thread.
PRIORITY_MIND	int	Thread.MAX_PRIORITY	The priority of the Mind–thread.
PRIORITY_MINDSEYE	int	Thread.MAX_PRIORITY	The priority of the MindsEye–thread.
PRIORITY_SENSORS	int	Thread.MAX_PRIORITY	The priority of the Sensors–thread.
RECOVER_DEC_THR	double	0.3	The percentage of the player's stamina below which the player's effort is affected.
RECOVER_THRESHOLD	double	RECOVER_DEC_THR * STAMINA_MAX	The amount of stamina below which the player's effort is affected.
RECV_STEP	long	10	The minimum time in milliseconds two messages have to be apart to ensure arrival of both.
RIGHT	int	1	The right side of the pitch.
SEPARATOR	String	"_"	The string used to separate the different values that are contained within a friendly message.
SERVER_VERSION	String	"4.21"	The version of the server that the client is trying to connect to.

Name	Type	Value	Description
STAMINA_INC	double	20.0	The amount the stamina is increased every cycle.
STAMINA_MAX	double	2000.0	The maximum stamina of a player.
TEAMNAME	String	"TBP1999"	The name of the connecting team.
TOP	int	2	The top side of the pitch.
TURN_DIR_DIFF	double	3.0	A player will cease turning with the ball when the direction the ball is differs this much from the target direction of the turn.
TURN_KICK_DISTANCE	double	0.2	The distance the ball is to travel each time it is kicked during a turn with ball.
UNKNOWN_ TEAMNAME	String	"unknown"	The name assigned to a team whose name cannot be recognised.

Table 8: Commonly used constants

3.2.2.5 The CommonFunctions class

The CommonFunctions class contains functions that are commonly made use of throughout the other classes.

3.2.2.6 The CommonVariables class

The CommonVariables class contains two commonly used variables:

1. Send step. The send step is the length in milliseconds of the interval for sending visual information to a player. In the standard view mode (normal, high) the visual information is sent every 300 milliseconds. Every time the view angle or the view quality is increased, the frequency is halved and every time they are decreased, the frequency is doubled.
2. Simulator step. The simulator step is the length of a simulator cycle. It has been made variable for synchronisation purposes between client and server.

3.2.2.7 The Circle class

A Circle object (i.e. an instance of the Circle class) contains two member variables, representing the centre and radius of the circle.

3.2.2.8 The Line class

A Line object (i.e. an instance of the Line class) contains three member variables, corresponding to a point on the line, a vector representing the orientation of the line and a value indicating whether or not the line is just a segment. In case the line is just a segment, the length of the orientation vector corresponds to the length of the segment. Otherwise, the aforementioned length is set to 1.

3.2.2.9 The Int class

The Int class wraps a value of the primitive type int in an object. Unlike the Integer class, it combines the arithmetic functionality of the primitive type with the possibilities of synchronisation of an object.

3.2.2.10 The PVector class

A PVector object (i.e. an instance of the PVector class) represents a positional vector, contrary to the Java Vector class, which represents a dynamic array. It contains four member variables: x and y, the Cartesian coordinates (i.e. the horizontal and the vertical component, respectively) of the vector, and r and theta, representing the same vector in polar coordinates. The PVector class ensures every one of these values to be up–to–date at all times, thus making it is possible to switch between the two representations whenever desired.

The PVector class also contains methods to manipulate (positional) vectors and to check whether or not a particular point is on a particular vector.

3.3 The coach

The coach program can be used to test our implementation of a RoboCup Simulator League soccer player. The coach is connected to the Soccer Server and it can be used to send commands to influence or query the state of affairs on the RoboCup Simulator League soccer pitch (and [KUMM1999]):

1. The change mode command can be used to set the server's mode of play.
2. The move command can be used to move a player or the ball to a particular position on the field and, optionally, change its absolute direction.
3. The check ball command can be used to retrieve the position of the ball.
4. The look command can be used to get information about:
   - The left and right goals.
   - The ball.
   - All active players.
5. The say message can be used to broadcast a message to all clients.
6. The ear message can be used to switch on or of the sending of auditory information to the coach. If the ear is turned on, the coach receives all auditory information from the server the same way as the players do. If it is turned off, the coach does not receive any auditory information.

1.	(change_mode playMode) playMode ::= one of the external modes of play (the first nine referee messages of)	
2.	(move object X Y [VDir]) object ::= (player teamname uNum) | (ball) X ::= −54…54 Y ::= −32…32 VDir ::= −180…180 degrees	
3.	(check_ball)	
4.	(look)	
5.	(say message) message ::= string containing a maximum of 512 characters	
6.	(ear mode) mode ::= on | off	

Table 9: Coach commands

The coach program has been written using the Microsoft® Visual C++® 6.0 programming environment instead of Java to achieve an easy–to–work–with interface in a reasonably short time. The program is composed of an edit box for specifying the name of the host the Soccer Server is running on, a button to actually set up the connection, an area displaying the commands send to and the responses received from the Soccer Server, a button to clear that area, another edit box to input the command to send to the server, a button to actually send the command, a drop down list box to select a mode of play, a section to load and execute scripts (i.e. sequences of commands to send to the server) and, finally, a button to quit the coach ().

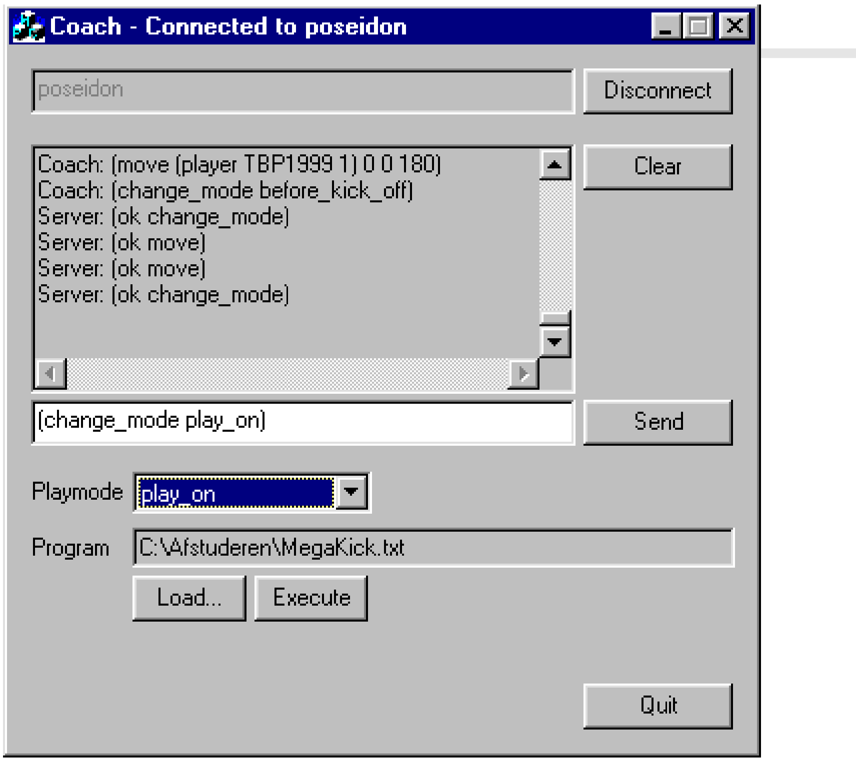
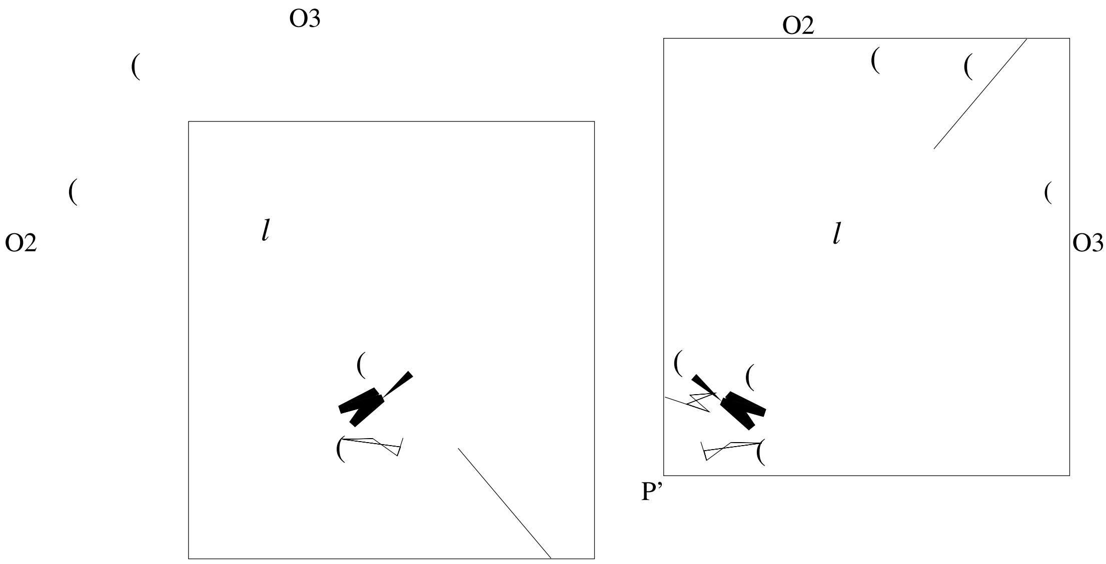


Figure 4: The coach

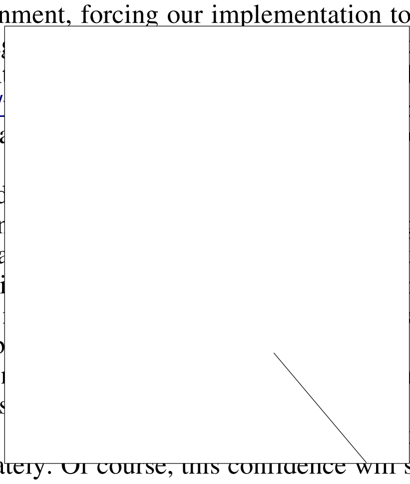


4 Conclusions and further research

Especially during the implementation of the TBP1999 simulated soccer player we have had to overcome a great magnitude of problems. Amongst other matters, this document addresses every encountered difficulty followed by the way we have tackled it. Contributing factor to nearly all of the aforementioned problems is the Soccer Server documentation or rather the lack thereof. Unfortunately, the official Soccer Server manual ([KUMM1999]) is of extremely poor quality. Many of the discussions contained in it are in serious need of an increase in depth and a great majority of the really important issues is not covered at all. Repeatedly we have been forced to look into the Soccer Server code to determine how certain issues, not covered by the manual, are resolved there. The other implementations of a RoboCup Simulator League soccer player have presented themselves to us as another vast source of inspiration. We hope that our implementation accompanied by this document will be of the same use to others in bridging the gap left unfilled by the manual.

Two other major problems that we have come across are directly caused by our decision to use the Java programming language to implement the TBP1999 simulated soccer player.

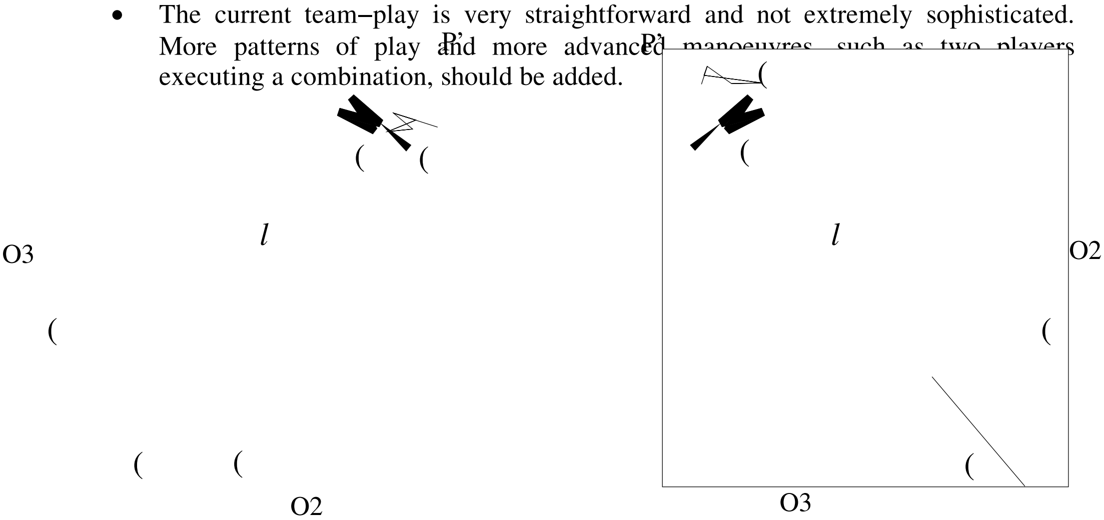
Firstly, the Java environment lacks the proper features to perform well in a real−time environment, forcing our implementation to being ‘polluted’ with timers written in C, a
langua
oned disadvantage. Hopefully, the efforts of
the in
l−time extensions for the Java platform
(http://
-time/intro.html) will lead to the changes
necess
ollution.



Second
an enormous amount of system resources,
resulti
client per computer. As we have had at our
dispos
r machines, the number of simultaneously
executi
reasonably testing our implementation as a
whole
eam, i.e. *eleven* players, and preferably also
(memb
had to settle for separately testing all the
differe
ntation. Seeing as the separate parts have
succes
em through, we are fairly confident that the
entire
osition of those parts, will also perform
adequately. Of course, this confidence will still have to be proven justified by putting to the test the implementation as a whole in the life−size surroundings of an actual match.

Finally, we would like to stress that there is always room for implementational improvement. Therefore, we will conclude this thesis by proposing a list of enhancements (in random order), the realisation of which we have not had time for:

- The Soccer Server provides a short distance sensor that senses objects that are near a particular player but out of sight. The use of this feature has been omitted from our implementation.
- Interrupting the current implementation of the player shooting the ball at the goal, results in erroneous behaviour. Therefore, we have prevented such an interruption from occurring by setting a boolean variable. This is not a pretty solution.
- The current team−play is very straightforward and not extremely sophisticated. More patterns of play and more advanced manoeuvres, such as two players executing a combination, should be added.



References

[ARNO1996] Ken Arnold and James Gosling. *The Java™ Programming Language*, Addison–Wesley, Reading, Massachusetts, 1996.

[BOUR1997] Paul Bourke. *Intersection of two circles*, http://www.mhri.edu.au/~pdb/geometry/2circle/ [July 7, 1999]

[COOK1994] Steve Cook and John Daniels. *Designing Object Systems – Object–Oriented Modelling with Syntropy*, Prentice Hall, New York, 1994.

[KITA1997] Hiroaki Kitano (Ed.). *RoboCup–97: Robot Soccer World Cup I*, Springer–Verlag, Berlin, 1998.

[KUMM1999] Johan Kummeneje et al.. *Soccerserver Manual Ver. 4 Rev. 02*, http://www.dsv.su.se/~johank/RoboCup/manual/ [July 7, 1999]

Appendix

This document is accompanied by a disk containing the TBP1999 simulated soccer player and accessories. displays the various sections the disk is divided into.

Directory	Contents
Software	The TBP1999 simulated soccer player software.
Source	The source of the software.
TBP1999	The source of the primary and auxiliary Java classes.
ommon	
sg	
Timer	The C−source of the timer.
Coach	The C++−source of the coach program.
CheckSet	The source of the program that checks the Java classes for the use of Set−methods.
Binaries	Compiled versions of the source.

TBP1999	The bytecode of the primary and auxiliary Java classes.
ommon	
sg	
Timer	The Unix executable of the timer.
Coach	The Windows executable of the coach program.
CheckSet	The MS–DOS executable of the program that checks the Java classes for the use of Set–methods.
Documentation	The TBP1999 simulated soccer player documentation.
Thesis	This document.
JavaDoc	HTML documentation generated by JavaDoc.

Table 10: Disk sectioning