The RoboCup Soccer Simulator Users Manual

The RoboCup Soccer Simulator Maintenance Committee

Mar 25, 2024

CONTENTS

last update: Mar 25, 2024

CHAPTER

ONE

INTRODUCTION

We are in the early days of RoboCup [Kitano95IJCAI], with half a century to go before we can "...build a team of robot soccer players, which can beat a human world cup champion team" [RoboCup97]. The challenge posed by the goal is enormous and inspires hundreds of researchers yearly throughout the world to engage themselves and their students in RoboCup. RoboCup has been used as a research challenge in parallel with a usage for educational purposes, and to stimulate the interest of the public for robotics and artificial intelligence(AI). Each year since 1997, researchers from different countries have gathered to play the world cup. The event has drawn an increasing amount of interest from the public, as robotics is still not commonplace.

The intention of this manual[1] is to guide the developers of simulated league teams in the beginning steps, and also serve as a reference manual for the experienced users.

1.1 Background

Mackworth [Mackworth93] introduced the idea of using soccer-playing robots in research. Unfortunately, the idea did not get the proper response until the idea was further developed and adapted by Kitano, Asada, and Kuniyoshi, when proposing a Japanese research programme, called Robot J-League[2]. During the autumn of 1993, several American researchers took interest in the Robot J-League, and it thereafter changed name to the Robot World Cup Initiative or RoboCup for short. RoboCup is sometimes referred to as the RoboCup challenge or the RoboCup domain.

In 1995, Kitano et al. [Kitano95IJCAI] proposed the first Robot World Cup Soccer Games and Conferences to take place in 1997. The aim of RoboCup was to present a new standard problem for AI and robotics, somewhat jokingly described as the life of AI after Deep Blue[3] . RoboCup differs from previous research in AI by focusing on a distributed solution instead of a centralised solution, and by challenging researchers from not only traditionally AI-related fields, but also researchers in the areas of robotics, sociology, real-time mission critical systems, etc.

To co-ordinate the efforts of all researchers, the RoboCup Federation was formed. The goal of RoboCup Federation is to promote RoboCup, for example by annually arranging the world cup tournament. Members of the RoboCup Federation are all active researchers in the field, and represent a number of universities and major companies. As the body of researchers is quite large and widespread, local committees are formed to promote RoboCup-related events in their geographical area.

[1] Parts of this chapter is taken directly from [Kummeneje01PhL]
[2] The J-League is the professional soccer league in Japan.
[3] In reference to Deep Blue and its games with Kasparov, see http://www.chess.ibm.com.

1.2 The Goals of RoboCup

The RoboCup Federation has set goals and a timetable for the research. Setting goals and a timetable are means of pushing the state-of-the-art further, in conjunction with formalised test-beds. In resemblance with John F. Kennedy's national goal of "landing a man on the moon and returning him safely to earth" ([4], p. 8276), the main accomplishment was not to land a man on the moon and returning him safely, but the overall technological advancement. Therefore, the most important goal of RoboCup is to advance the overall technological level of society, and as a more pragmatic goal to achieve the following:

> By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win the soccer game, comply with the official rule of the FIFA 4 , against the winner of the most recent World Cup.

There will be several technological advancements, even if the goal of the robotic soccer team is not reached, starting with Team-Partitioned, Opaque-Transition Reinforcement Learning (TPOT-RL) [Stone98] which has found application in the domain of packet routing in computer networks. TPOT-RL is a distributed learning method in domains where "agents have limited information about environmental state transitions" ([Stone98], p. 22).

In most RoboCup leagues, the teams consist of either robots or programs that cooperate in order to defeat the opponent team. RoboCup Rescue and the commentator exhibition diverge from the other RoboCup leagues. The goal of defeating an opponent would raise ethical issues in RoboCup Rescue, since we cannot assign comparable utilities to human lives and buildings. Hence, the focus in RoboCup Rescue is on the co-operative efforts between heterogeneous agents. In the commentator exhibition, the goal is to observe and comment.

Besides the commentator exhibition and RoboCup Rescue, the main body of the RoboCup challenge consists of several leagues for soccer playing. However, as this manual is about the simulated league we will only focus on it.

1.2.1 Simulated League

The RoboCup simulator league is based on the RoboCup simulator called the soccer server [Noda97RoboCup97], a physical soccer simulation system. All games are visualised by displaying the field of the simulator by the soccer monitor on a computer screen. The soccer server is written to support competition among multiple virtual soccer players in an uncertain multi-agent environment, with real-time demands as well as semi-structured conditions. One of the advantages of the soccer server is the abstraction made, which relieves the researchers from having to handle robot problems such as object recognition [9], communications, and hardware issues, e.g., how to make a robot move. The abstraction enables researchers to focus on higher level concepts such as cooperation and learning.

Since the soccer server provides a challenging environment, i.e., the intentions of the players cannot mechanically be deduced, there is a need for a referee when playing a match. The included artificial referee is only partially implemented and can detect trivial situations, e.g., when a team scores. However, there are several hard-to-detect situations in the soccer server, e.g., deadlocks, which brings the need for a human referee. All participating teams are also obliged to play according to a gentlemen's agreement, e.g., not to use loopholes.

Since the first version of the soccer server was completed in 1995, there have been four world cups and one pre-world cup event, not to mention all other RoboCup-related events. The 1996 pre-RoboCup event [PreRoboCup96] was held in Osaka, with only seven entrants in the competition which ended with a Japanese victory by the team Ogalets from Tokyo University. In Nagoya the following year, the first formal competition was held in conjunction with the IJCAI'97 conference. The competition had 29 teams participating, and the winner was AT Humboldt [Burkhard97]. The RoboCup world cup of 1998 was played in conjunction with the human world cup in Paris, and the winning team was CMUnited98 [CMUnited98]. During the world cup, media was heavily covering the event, as it was public in a museum in the suburbs of Paris. The year after, the world cup was held in conjunction with IJCAI'99 in Stockholm, and the winners (once again) were CMUnited99 [CMUnited99]. An unchanged version of the champion team must participate, as a benchmark, in the next world cup. The benchmarking teams have always been able to win their group, but only in 2000 did the benchmark team advance further than the first game after group play.

1.2.2 What is the Soccerserver

Soccer Server is a system that enables autonomous agents consisting of programs written in various languages to play a match of soccer (association football) against each other.

A match is carried out in a client/server style: A server, soccerserver, provides a virtual field and simulates all movements of a ball and players. Each client controls movements of one player. Communication between the server and each client is done via UDP/IP sockets. Therefore users can use any kind of programing systems that have UDP/IP facilities.

The soccerserver consists of 2 programs, soccerserver and soccermonitor. Soccer Server is a server program that simulates movements of a ball and players, communicates with clients, and controls a game according to rules. Soccermonitor is a program that displays the virtual field from the soccerserver on the monitor using the X window system. A number of soccermonitor programs can connect with one soccerserver, so we can display field-windows on multiple displays.

A client connects with soccerserver by an UDP socket. Using the socket, the client sends commands to control a player of the client and receives information from sensors of the player. In other words, a client program is a brain of the player: The client receives visual and auditory sensor information from the server, and sends control-commands to the server.

Each client can control only one player 56 . So a team consists of the same number of clients as players. Communications between the clients must be done via soccerserver using say and hear protocols. (See section *Player Command Protocol*.) One of the purposes of soccerserver is evaluation of multi-agent systems, in which efficiency of communication between agents is one of the criteria. Users must realize control of multiple clients by such restricted communication.

1.3 History

In this section we will first describe the history of the soccerserver and thereafter the history of the RoboCup Simulation League. To end the section we will also describe the history of the manual effort.

1.3.1 History of the Soccer Server

The first, preliminary, original system of soccerserver was written in September of 1993 by Itsuki Noda, ETL. This system was built as a library module for demonstration of a programming language called MWP, a kind of Prolog system that has multi-threads and high level program manipulation. The module was a closed system and displayed a field on a character display, that is VT100.

The first version (version 0) of the client-server style server was written in July of 1994 on a LISP system. The server shows the field on an X window system, but each player was shown in an alphabet character. It used the TCP/IP protocol for connections with clients. This LISP version of soccerserver became the original style of the current soccerserver. Therefore, the current soccerserver uses S-expressions for the protocol between clients and the server.

The LISP version of soccerserver was re-written in C++ in August of 1995 (version 1). This version was announced at the IJCAI workshop on Entertainment and AI/Alife held in Montreal, Canada, August 1995.

The development of version 2 started January of 1996 in order to provide the official server of preRoboCup-96 held at Osaka, Japan, November 1996. From this version, the system is divided into two modules, soccerserver and soccerdisplay (currently, soccermonitor). Moreover, the feature of coach mode was introduced into the system. These two features enabled researchers on machine learning to execute games automatically. Peter Stone at Carnegie Mellon University joined the decision-making process for the development of the soccerserver at this stage. For example, he created the configuration files that were used at preRoboCup-96.

After preRoboCup-96, the development of the official server for the first RoboCup, RoboCup-97 held at Nagoya, Japan, August 1997, started immediately, and the version 3 was announced in February of 1997. Simon Ch'ng at RMIT joined decisions of regulations of soccerserver from this stage. The following features were added into the new version:

- logplayer
- information about movement of seen objects in visual information
- capacity of hearing messages

The development of version 4 started after RoboCup-97, and announced November 1997. From this version, the regulations are discussed on the mailing list organized by Gal Kaminka. As a result, many contributers joined the development. Version 4 had the following new features:

- more realistic stamina model
- goalie
- handling offside rule
- disabling players for evaluation
- facing direction of players in visual information
- sense body command

Version 4 was used in JapanOpen 98, RoboCup98 and Pacific Rim Series 98.

Version 5 was used in JapanOpen 99, and will also be used in RoboCup99 in Stockholm during the summer of 1999.

In Melbourne 2000, version 6 was used, and for the world cup in 2001 version 7 will be used.

1.3.2 History of the RoboCup Simulation League

The RoboCup simulation league has had 26 main official events: Starting with a preRoboCup96 in 1996 event, from 1997 onward an official world championship tournament was held each year (from RoboCup97 to RoboCup 2021). Research results have been reported extensively in the proceedings of the workshops and conferences associated with these competitions. In this section, we focus mainly on the competitions themselves.

preRoboCup96

preRoboCup96 was the first robotic soccer competition of any sort. It was held on November 5–7, 1996 in Osaka, Japan [5]. In conjunction with the IROS-96 conference, preRoboCup96 was meant as an informal, small-scale competition to test the RoboCup soccerserver in preparation for RoboCup97. 5 of the 7 entrants were from the Tokyo region. The other 2 were from Ch'ng at RMIT and Stone and Veloso from CMU. The winning teams were entered by:

1. Ogawara (Tokyo University)
2. Sekine (Tokyo Institute of Technology)
3. Inoue (Waseda University)
4. Stone and Veloso (Carnegie Mellon University)

In this tournament, team strategies were generally quite straightforward. Most of the teams kept players in fixed locations, only moving them towards the ball when it was nearby.

RoboCup97

The RoboCup97 simulator competition was the first formal simulated robotic soccer competition. It was held on August 23–29, 1997 in Nagoya, Japan in conjunction with the IJCAI-97 conference [6]. With 29 teams entering from all around the world, it was a very successful tournament. The winning teams were entered by:

1. Burkhard et al. (Humboldt University)
2. Andou (Tokyo Institute of Technology)
3. Tambe et al. (ISI/University of Southern California)
4. Stone and Veloso (Carnegie Mellon University)

In this competition, the champion team exhibited clearly superior low-level skills. One of its main advantages in this regard was its ability to kick the ball harder than any other team. Its players did so by kicking the ball around themselves, continually increasing its velocity so that it ended up moving towards the goal faster than was imagined possible. Since the soccerserver did not (at that time) enforce a maximum ball speed, a property that was changed immediately after the competition, the ball could move arbitrarily fast, making it almost impossible to stop. With this advantage at the low-level behavior level, no team, regardless of how strategically sophisticated, was able to defeat the eventual champion.

At RoboCup97, the RoboCup scientific challenge award was introduced. Its purpose is to recognize scientific research results regardless of performance in the competitions. The 1997 award went to Sean Luke [10] of the University of Maryland "for demonstrating the utility of evolutionary approach by co-evolving soccer teams in the simulator league."

RoboCup98

The second international RoboCup championship, RoboCup-98, was held on July 2–9, 1998 in Paris, France [1]. It was held in conjunction with the ICMAS-98 conference.

The winning teams were entered by:

1. Stone et al. (Carnegie Mellon University)
2. Burkhard et al. (Humboldt University)
3. Corten and Rondema (University of Amsterdam)
4. Tambe et al. (ISI/University of Southern California)

Unlike in the previous year's competition, there was no team that exhibited a clear superiority in terms of low-level agent skills. Games among the top three teams were all quite closely contested with the differences being most noticeable at the strategic, team levels.

One interesting result at this competition was that the previous year's champion team competed with minimal modifications and finished roughly in the middle of the final standings. Thus, there was evidence that as a whole, the field of entries was much stronger than during the previous year: roughly half the teams could beat the previous champion.

The 1998 scientific challenge award was shared by Electro Technical Laboratory (ETL), Sony Computer Science Laboratories, Inc., and German Research Center for Artificial Intelligence GmbH (DFKI) for "development of fully automatic commentator systems for RoboCup simulator league."

To encourage the transfer of results from RoboCup to the scientific community at large, RoboCup98 was the first to host the Multi-Agent Scientific Evaluation Session. 13 different teams participated in the session, in which their adaptability to loss of team-members was evaluated comparatively. Each team was played against the same fixed opponent (the 1997 winner, AT Humboldt'97) four half-games under official RoboCup rules. The first half-game (phase A) served as a base-line. In the other three half- games (phases B-D), 3 players were disabled incrementally: A randomly chosen player, a player chosen by the representative of the fixed opponent to maximize "damage" to the evaluated team, and the goalie. The idea is that a more adaptive team would be able to respond better to these.

Very early on, even during the session itself, it became clear that while in fact most participants agreed intuitively with the evaluation protocol, it wasn't clear how to quantitatively, or even qualitatively, analyse the data. The most obvious measure of the goal-difference at the end of each half may not be sufficient: some teams seem to do better with less players, some do worse. Performance, as measured by the goal-difference, really varied not only from team to team, but also for the same team between phases. The evaluation methodology itself and analysis of the results became open research problems in themselves. To facilitate this line of research, the data from the evaluation was made public at: http://www.isi.edu/~galk/Eval/

RoboCup99

The third international RoboCup championship, RoboCup-99, was held in late July and early August, 1999 in Stockholm, Sweden [3]. It was held in conjunction with the IJCAI-99 conference.

RoboCup2000

The fourth international RoboCup championship, RoboCup 2000, was held in early September, 2000 in Melbourne, Australia [16]. It was held in conjunction with the PRICAI-2000 conference.

RoboCup 2004

The eigth international RoboCup championship, RoboCup 2004, was held in Lisbon, Portugal. It was accompanied by the RoboCup 2004 Symposium, held at the Instituto Superior Tecnico and was co-located with the 5th IFAC/EURON International Symposium on Intelligent Autonomous Vehicles (IAV 2004).

The main novelty in the Soccer Simulation League in 2004 was the introduction of the 3D soccer simulator, where players are spheres in a three-dimensional environment with a full physical model. This sub-competition was the spawning point for the Soccer Simulation 3D League in later years.

The winning teams in the Soccer Simulation 2D competition, for which 24 teams were qualified, were:

1. STEP (ElectroPult Plant Company, Russia)
2. Brainstormers (University of Osnabrueck, Germany)
3. Mersad (Allameh Helli High School, Iran)

The winning teams in the coach competition were:

1. MRL (Azad University of Qazvin, Iran)
2. FC Portugal (Universities of Porto and Aveiro, Portugal)
3. Caspian (Iran University of Science and Technology, Iran)

The winning teams in the 3D competition were:

1. Aria (Amirkabir University of Technology, Iran)
2. AT-Humboldt (Humboldt University Berlin, Germany)
3. UTUtd 2004 (University of Tehran, Iran)

RoboCup 2005

The tenth international RoboCup championship, RoboCup 2005, was held in July 2005 in Osaka, Japan [16]. It was accompanied by the RoboCup Symposium. Since, for the first time, the 3D sub-league of soccer simulation had its own tournament, the number of teams that were maximally allowed to qualify for the Soccer Simulation 2D competitions at RoboCup 2005 was reduced to 16 (though a 17th team was permitted for reasons of the qualifying procedure).

The winning teams in the Soccer Simulation 2D competition were:

1. Brainstormers (University of Osnabrueck, Germany)
2. WrightEagle (University of Science and Technology of China, China)
3. TokyoTech SFC (Tokyo Institute of Technology, Japan)

An interesting observation, quite similar to the related remark for RoboCup98, could be made in this year: Last year's champion (STEP, Russia) entered the competition without any modifications made to their team and finished the tournament on rank 4.

1.3.3 History of the Soccer Manual Effort

The first versions of the manual were written by Itsuki Noda, while developing the soccerserver, and around version 3.00 there were several requests on an updated manual, to better correspond to the server as well as enable newcomers to more easily participate in the RoboCup World Cup Initiative. In the fall of 1998 Peter Stone initiated the Soccer Manual Effort, over which Johan Kummeneje took responsibility to organize and as a result the Soccer Server Manual version 4.0 was released on the 1st of November 1998.

In 1999, the manual for the soccerserver version 5.0 was released. Unfortunately the manual lost part of its pace, and there was no release of the manual for soccerserver version 6.0.

Since 1999, the soccerserver has changed major version to 7 and is continuously developed. Therefore the Soccer Manual Effort has developed a new version, which resulted in a PDF version of the Soccer Manual (available on Sourceforge) that has been the main reference document for many years.

In 2009 and 2010 (soccerserver versions 12 and 14), significant changes were introduced to the way the soccerserver simulates soccer, including a changed tackle model and a sideward dash model to mention just a few. The corresponding changes of those times were, unfortunately, not incorporated into the existing soccerserver manual, but were reflected only in the NEWS text file as part of the soccerserver software package.

In 2019, a joint effort was started to migrate the existing Latex-based soccerserver manual to the Github-hosted version that is based on reStructured text and that you are reading here.

1.4 About This Manual

This manual is the joint effort of the authors from a diverse range of universities and organizations, which build upon the original work of Itsuki Noda. If there are errors, inconsistencies, or oddities, please notify johank@dsv.su.se or fruit@uni-koblenz.de with the location of the error and a suggestion of how it should be corrected.

We are always looking for anyone who has an idea on how to improve the manual, as well as proofread or (re)write a section of the manual. If you have any ideas, or feel that you can contribute with anything to the SoccerServer Manual Effort. .. please mail johank@dsv.su.se or fruit@uni-koblenz.de.

1.5 Reader's Guide to the Manual

The thesis is written for a wide range of readers, and therefore the chapters are not equally important to all readers. We shortly describe the remaining chapters to give an overview of the thesis.

Chapter 2 introduces the concepts of the simulated league and will help the newcomer to get to terms with the different parts.

Chapter 3 helps the beginners to start compiling and running the software.

Chapter 4 describes the soccerserver.

Chapter 5 describes the soccermonitor.

Chapter 6 describes the soccerclient and how to create one.

Chapter 7 describes the coachclient.

Chapter 8 suggests some further reading.

CHAPTER

TWO

OVERVIEW

2.1 Getting Started

This section is designed to give you a quick introduction to the main components of the RoboCup simulator. For each of these components you will find detailed information (i.e. configuration parameters, run-time options, etc.) later on in this manual.

2.1.1 The Server

The server is a system that enables various teams to compete in a game of soccer. Since the match is carried out in a client-server style, there are no restrictions as to how teams are built. The only requirement is that the tools used to develop a team support client-server communication via UDP/IP. This is due to the fact that all communication between the server and each client is done via UDP/IP sockets. Each client is a separate process and connects to the server through a specified port. After a player connects to the server, all messages are transferred through this port. A team can have up to 12 clients, i.e. 11 players (10 fielders + 1 goalie) and a coach. The players send requests to the server regarding the actions they want to perform (e.g. kick the ball, turn, run, etc.). The server receives those messages, handles the requests, and updates the environment accordingly. In addition, the server provides all players with sensory information (e.g. visual data regarding the position of objects on the field, or data about the player's ressources like stamina or speed). It is important to mention that the server is a real-time system working with discrete time intervals (or cycles). Each cycle has a specified duration, and actions that need to be executed in a given cycle, must arrive at the server during the right interval. Therefore, slow performance of a player that results in missing action opportunities has a major impact on the performance of the team as a whole. A detailed description of the server can be found in Chapter *Soccer Server*.

2.1.2 The Monitor

The Soccer Monitor is a visualisation tool that allows people to see what is happening within the server during a game. Currently the monitor comes in two flavors, the `rcssmonitor` and the `rcssmonitor_classic`. The information shown on both monitors include the score, team names, and the positions of all the players and the ball. They also provide simple interfaces to the server. For example, when both teams have connected, the "Kick-Off" button on the monitor allows a human referee to start the game. The `rcssmonitor`, which is based on the `frameview` by Artur~Merke, extends the functionality of the classic monitor by several features.

- It is possible to zoom into areas of the field. This is especially useful for debugging purposes.
- The current positions and velocities of all players and the ball can be printed to the console at any time.
- A variety of information can be shown on the monitor, e.g. a player's view cone, stamina or (in the case of heterogeneous players) player type.
- Players and the ball can be moved around with the mouse.

As you will discover later on, to run a game on the server, a monitor is not required. However, if needed, a number of monitors can be connected to the server at the same time (for example if you want to show the same game at different terminals). For further details on the monitor please have a look at Chapter *Soccer Monitor*.

2.1.3 The Logplayer

The logplayer can be thought of as a video player. It is a tool that is used to replay matches. When running the server, certain options can be used that will cause the server to store all the data for a given match on the hard drive. (Pretty much like pressing the record button on your video). Then, the program `rcsslogplayer` combined with a monitor can be used to replay that game as many times as needed. This is quite useful for doing team analysis and discovering the strong or weak points of a team. Much like a video player, the logplayer is equipped with play, stop, fast forward and rewind buttons. Also the logplayer allows you to jump to a particular cycle in a game (for example if you only want to see the goals). Finally the logplayer allows you to edit existing recordings, i.e. you can save interesting scenes from a match (or several matches) to another logfile and thus create a presentation easily.

The logplayer can be controlled via a small GUI or a command line interface. In addition commands can be read from a file, which adds limited scripting capabilities to the logplayer.

2.1.4 The Demo Client

Bundled with the RoboCup Soccer Simulator is a program called `rcssclient`, which implements a very primitive textbased client for the simulation. The purpose of this program is to give you a first idea of how the whole affair works.

When `rcssclient` is started, it connects to the server. You are presented with a simple ncurses-based interface. You can then enter commands that are executed by the server. Any information that is received by the client will be shown in a different section of the screen according to its type (visual, sense body or other). By entering commands and see what happens you can get a first idea of the way things work in the simulation. Even if you are not a newbie any more, the program is handy for simple tests, e.g. getting a grip on new commands added to the simulation.

2.2 The Rules of the Game

During a game, a number of rules are enforced either by the automated referee within the server, or by a human referee. The aim of this section is to describe how these rules work, and how they affect the game.

2.2.1 Rules Judged by the Automated Referee

Kick-Off

Just before a kick off (either before a half time starts, or after a goal), all players must be in their own half. To allow for this to happen, after a goal is scored, the referee suspends the match for an interval of 5 seconds. During this interval, players can use the **move** command to teleport to a position within its own side, rather than run to this position, which is much slower and consumes stamina. If a player remains in the opponent half after the 5-second interval has expired or tries to teleport there during the interval, the referee moves the player to a random position within their own half.

Goal

When a team scores, the referee performs a number of tasks. Initially, it announces the goal by broadcasting a message to all players. It also updates the score, moves the ball to the centre mark, and changes the play-mode to kick_off_x (where x is either left or right). Finally, it suspends the match for 5 seconds allowing players to move back to their own half (as described above in the "Kick-Off" section).

Out of Field

When the ball goes out of the field, the referee moves the ball to a proper position (a touchline, corner or goal-area) and changes the play-mode to kick_in, corner_kick, or goal_kick. In the case of a corner kick, the referee places the ball at (1m, 1m) inside the appropriate corner of the field.

Player Clearance

When the play-mode is kick_off, free_kick, kick_in, or corner_kick, the referee removes all defending players located within a circle centred on the ball. The radius of this circle is a parameter within the server (normally 9.15 meters). The removed players are placed on the perimeter of that circle. When the play-mode is offside, all offending players are moved back to a non-offside position. Offending players in this case are all players in the offside area and all players inside a circle with radius 9.15 meters from the ball. When the play-mode is goal_kick, all offending players are moved outside the penalty area. The offending players cannot re-enter the penalty area while the goal kick takes place. The play-mode changes to play_on immediately after the ball goes outside the penalty area.

Play-Mode Control

When the play-mode is kick_off, free_kick, kick_in, or corner_kick, the referee changes the play-mode to play_on immediately after the ball starts moving through a **kick** command.

Offside

A player is marked offside, if it is - in the opponent half of the field, - closer to the opponent goal than at least two defending players, - closer to the opponent goal than the ball,\ - closer to the ball than 2.5 meters (this can be changed with the server parameter **server::offside_active_area_size**).

Backpasses

Just like in real soccer games, the goalie is not allowed to catch a ball that was passed to him by a teammate. If this happens, the referee calls a **back_pass_l** or **back_pass_r** and assigns a free kick to the opposing team. As such a back pass can only happen within the penalty area, the ball is placed on the corner of the penalty area that is closest to the position the goalie tried to catch. Note, that it is perfectly legal to pass the ball to the goalie if the goalie does not try to catch the ball.

Free Kick Faults

When taking a free kick, corner kick, goalie free kick, or kick in, a player is not allowed to pass the ball to itself. If a player kicks the ball again after performing one of those free kicks, the referee calls a **free_kick_fault_l** or **free_kick_fault_r** and the oppsing team is awarded a free_kick.

As a player may have to kick the ball more than once in order to accelerate it to the desired speed, a free kick fault is only called if the player taking the free kick

1. is the first player to kick the ball again, and
2. the player has moved (= dashed) between the kicks.

So issuing command sequences like **kick–kick–dash** or **kick–turn–kick** is perfectly legal. The sequence **kick–dash–kick**, on the other hand, results in a free kick fault.

Half-Time and Time-Up

The referee suspends the match when the first or the second half finishes. The default length for each half is 3000 simulation cycles (about 5 minutes). If the match is drawn after the second half, the match is extended. Extra time continues until a goal is scored. The team that scores the first goal in extra time wins the game. This is also known as the "golden goal" rule or "sudden death".

2.2.2 Rules Judged by the Human Referee

Fouls like "obstruction" are difficult to judge automatically because they concern players' intentions. To resolve such situations, the server provides an interface for human-intervention. This way, a human-referee can suspend the match and give free kicks to either of the teams. The following are the guidelines that were agreed prior to the RoboCup 2000 competition, but they have been used since then.

- Surrounding the ball
- Blocking the goal with too many players
- Not putting the ball into play after a given number of cycles. By now this rule is handled by the automatic referee, as well. If a team fails to put the ball back into play for **servr::drop_ball_time** cycles, a drop_ball is issued by the referee. However, if a team repeatedly fails to put the ball into play, the human referee may drop the ball prematurely.
- Intentionally blocking the movement of other players
- Abusing the goalie **catch** command (the goalie may not repeatedly kick and catch the ball, as this provides a safe way to move the ball anywhere within the penalty area).
- Flooding the Server with Messages: A player should not send more than 3 or 4 commands per simulation cycle to the soccer server. Abuse may be checked if the server is jammed, or upon request after a game.
- Inappropriate Behaviour: If a player is observed to interfere with the match in an inappropriate way, the human-referee can suspend the match and give a free kick to the opposite team.

CHAPTER

THREE

GETTING STARTED

This section contains all the information necessary to get the RoboCup Soccer Simulator source files and to install the software. Since you are reading this manual, you probably already know where to find the RoboCup Soccer Simulator related software and documentation. But we will tell you just in case. :-)

3.1 The Homepage

The official homepage of the RoboCup Soccer Simulator can be found at https://rcsoccersim.github.io/ . This page contains (links to) useful information about RoboCup in general and the RoboCup Soccer Simulator.

3.2 Getting and installing the server

The procedure shown was performed on a computer running Linux. We recommend using a stable Linux environment for running the simulation server and monitor. Get tar.gz files for the version you are after from the Simulator's code repository:

- rcssserver performs the actual simulation.
- rcssmonitor allows you to watch game in progress.
- rcsslogplayer allows you to replay logs (*.rcg files) created by rcssserver.

Unless there is a particular reason, we recommend using the latest released version. If you have downloaded rcssserver-x.x.x.tar.gz, then first extract the source files by running:

```
$ tar zxvf rcssserver-x.x.x.tar.gz
```

directory to **rcssserver-x.x.x**. This directory contains the following files:

```
$ cd rcssserver-x.x.x
$ ls
AUTHORS         Makefile.in   compile         configure     m4
CMakeLists.txt  NEWS          config.guess    configure.ac  missing
COPYING.LESSER  README.md     config.h.cmake  depcomp       rcss
ChangeLog       acinclude.m4  config.h.in     install-sh    src
Makefile.am     aclocal.m4    config.sub      ltmain.sh     ylwrap
```

Always read the **README.md** file first:

```
$ more README.md
```

3.3 Quick Start

From the rcssserver-* directory execute:

```
$ ./configure
$ make
```

This will build the necessary binaries to get you up and running. The monitor and logplayer can be built with same procedure.

rcssserver-*/src/rcssserver is the binary for the simulator server. The simulator server manages the actual simulation and communicates with client programs that control the simulated robots. A sample client can be found at **rcssserver-*/src/rcssclient**.

To see what is actually happening in the simulator, you will need to start a simulator monitor, which can be found at **rcssmonitor-*/src/rcssmonitor**.

To playback games that you have recorded or downloaded, you will need to start the log player, **rcsslogplayer-*/src/rcsslogplayer**. The log player will control what part of the game you see, but you will need to start a monitor (like rcssmonitor) to see the actual playback.

3.4 Full installation

3.4.1 Configuring

Before you can build the RoboCup Soccer Simulator you will need to run the **configure** script located in the root of the distribution directory.

The default configuration will set up to install the simulator components in the following locations:

- **/usr/local/bin**
  for the executables
- **/usr/local/include**
  for the headers
- **/usr/local/lib**
  for the libraries

You may need administrator privileges to install the simulator into the default location. This location can be modified by using configure's `--prefix=DIR` and related options. See **configure –help** for an overview over the available options.

There are a number of features specific to the package. Some of them are enabled by default. If you want to enable a feature, use the option `--enable-FEATURE[=yes]`.

Disabling a feature can be done by using either `--disable-FEATURE` or `--enable-FEATURE=no`.

3.4.2 Building

Once you have successfully configured the simulator, simply run `make` to build the sources.

3.4.3 Installing

When you have completed building the simulator, its components can be installed into their default locations or the locations specified during configuration by running `make install`. Depending on where you are installing the simulator, you may need special permissions.

3.4.4 Uninstalling

The simulator can also be easily removed by entering the distribution directory and running `make uninstall`. This will remove all the files that where installed, but not any directories that were created during the installation process.

3.5 Using the Simulator

To start the server either type:

```
./rcssserver
```

from the directory containing the executable or:

```
rcssserver
```

if you installed the executables in your PATH.

rcssserver will look in your home directory for the configuration files:

- .rcssserver/server.conf
- .rcssserver/player.conf
- .rcssserver/CSVSaver.conf
- .rcssserver-landmark.xml

If .conf files do not exist, they will be created and populated with default values.

You can include additional configuration files by using the `include=FILE` option to Com{rcssserver}.

You can then see what's happening in the simulator by using **./rcssmonitor** or **rcssmonitor** as above.

If you installed the executables in your PATH, you can start both the server and the monitor by using the **rcsoccersim** script which would have also been installed in your PATH. This script will start the server and the monitor and automatically stop the server when you close the monitor.

In order to actually start a match on the simulation server, the user must connect some clients to the server (maximum of 11 per side plus coaches). When these clients are ready, the user can click the **Kick Off** button on the monitor to start the game. It is likely that you have not yet programmed your own clients, in which case, you can read section ??? for instructions how to set up a whole match with the available teams that other RoboCuppers have contributed.

Also, there is a sample client **rcssclient** included with every distribution of the server. .. It has either an ncurses interface or a .. command line interface (CLI) if ncurses is not available, or it it's .. started with the Com{-nogui} option.

Running **rcssclient** attempts to connect to the server using default parameters (host=localhost, port=6000). Of course, these server parameters can be changed using the arguments that the server accepts when it is started. When the client is started, you need to initialise its connection to the server. This is done by manually typing in an init command and hitting enter. So, to initialise the connection:

```
(init MyTeam (version 15))
```

You will notice that one of the two teams is now named "MyTeam" and one of the players that are standing by the side-line is active. This player corresponds to the client you've just initialised. Also, notice the information that the client writes on the terminal. This is what the client receives from the server.

In the following text (which has line breaks added for clarity), the first eleven lines correspond to the initialisation[1] and the other data is the sensor information that the server sends to this client:

```
(init MyTeam (version 15))
(init l 2 before_kick_off)
(server_param (catch_ban_cycle 5)(clang_advice_win 1)
  (clang_define_win 1)(clang_del_win 1)(clang_info_win 1)
  (clang_mess_delay 50)(clang_mess_per_cycle 1)
  (clang_meta_win 1)(clang_rule_win 1)(clang_win_size 300)
  (coach_port 6001)(connect_wait 300)(drop_ball_time 0)
  (freeform_send_period 20)(freeform_wait_period 600)
  (game_log_compression 0)(game_log_version 3)
  (game_over_wait 100)(goalie_max_moves 2)(half_time -10)
  (hear_decay 1)(hear_inc 1)(hear_max 1)(keepaway_start -1)
  (kick_off_wait 100)(max_goal_kicks 3)(olcoach_port 6002)
  (point_to_ban 5)(point_to_duration 20)(port 6000)
  (recv_step 10)(say_coach_cnt_max 128)
  (say_coach_msg_size 128)(say_msg_size 10)
  (send_step 150)(send_vi_step 100)(sense_body_step 100)
  (simulator_step 100)(slow_down_factor 1)(start_goal_l 0)
  (start_goal_r 0)(synch_micro_sleep 1)(synch_offset 60)
  (tackle_cycles 10)(text_log_compression 0)
  (game_log_dir "/home/thoward/data")
  (game_log_fixed_name "rcssserver")keepaway_log_dir "./")
  (keepaway_log_fixed_name "rcssserver")
  (landmark_file "~/.rcssserver-landmark.xml")
  (log_date_format "%Y%m%d%H%M-")(team_l_start "")
  (team_r_start "")(text_log_dir "/home/thoward/data")
  (text_log_fixed_name "rcssserver")(coach 0)
  (coach_w_referee 1)(old_coach_hear 0)(wind_none 0)
  (wind_random 0)(auto_mode 0)(back_passes 1)
  (forbid_kick_off_offside 1)(free_kick_faults 1)
  (fullstate_l 0)(fullstate_r 0)(game_log_dated 1)
  (game_log_fixed 1)(game_logging 1)(keepaway 0)
  (keepaway_log_dated 1)(keepaway_log_fixed 0)
  (keepaway_logging 1)(log_times 0)(profile 0)
  (proper_goal_kicks 0)(record_messages 0)(send_comms 0)
  (synch_mode 0)(team_actuator_noise 0)(text_log_dated 1)
  (text_log_fixed 1)(text_logging 1)(use_offside 1)
  (verbose 0)(audio_cut_dist 50)(ball_accel_max 2.7)
  (ball_decay 0.94)(ball_rand 0.05)(ball_size 0.085)
```

(continues on next page)

[1] The response from the server means that the client plays for the left side, has the number one and the play mode is before_kick_off. The other lines correspond the the current server parameters and player types.

(continued from previous page)

```
  (ball_speed_max 2.7)(ball_weight 0.2)(catch_probability 1)
  (catchable_area_l 2)(catchable_area_w 1)(ckick_margin 1)
  (control_radius 2)(dash_power_rate 0.006)(effort_dec 0.005)
  (effort_dec_thr 0.3)(effort_inc 0.01)(effort_inc_thr 0.6)
  (effort_init 0)(effort_min 0.6)(goal_width 14.02)
  (inertia_moment 5)(keepaway_length 20)(keepaway_width 20)
  (kick_power_rate 0.027)(kick_rand 0)(kick_rand_factor_l 1)
  (kick_rand_factor_r 1)(kickable_margin 0.7)(maxmoment 180)
  (maxneckang 90)(maxneckmoment 180)(maxpower 100)
  (minmoment -180)(minneckang -90)(minneckmoment -180)
  (minpower -100)(offside_active_area_size 2.5)
  (offside_kick_margin 9.15)(player_accel_max 1)
  (player_decay 0.4)(player_rand 0.1)(player_size 0.3)
  (player_speed_max 1)(player_weight 60)(prand_factor_l 1)
  (prand_factor_r 1)(quantize_step 0.1)(quantize_step_l 0.01)
  (recover_dec 0.002)(recover_dec_thr 0.3)(recover_min 0.5)
  (slowness_on_top_for_left_team 1)
  (slowness_on_top_for_right_team 1)(stamina_inc_max 45)
  (stamina_max 4000)(stopped_ball_vel 0.01)
  (tackle_back_dist 0.5)(tackle_dist 2.5)(tackle_exponent 6)
  (tackle_power_rate 0.027)(tackle_width 1.25)
  (visible_angle 90)(visible_distance 3)(wind_ang 0)
  (wind_dir 0)(wind_force 0)(wind_rand 0))
(player_param (player_types 7)(pt_max 3)(random_seed -1)
  (subs_max 3)(dash_power_rate_delta_max 0)
  (dash_power_rate_delta_min 0)
  (effort_max_delta_factor -0.002)
  (effort_min_delta_factor -0.002)
  (extra_stamina_delta_max 100)
  (extra_stamina_delta_min 0)
  (inertia_moment_delta_factor 25)
  (kick_rand_delta_factor 0.5)
  (kickable_margin_delta_max 0.2)
  (kickable_margin_delta_min 0)
  (new_dash_power_rate_delta_max 0.002)
  (new_dash_power_rate_delta_min 0)
  (new_stamina_inc_max_delta_factor -10000)
  (player_decay_delta_max 0.2)
  (player_decay_delta_min 0)
  (player_size_delta_factor -100)
  (player_speed_max_delta_max 0.2)
  (player_speed_max_delta_min 0)
  (stamina_inc_max_delta_factor 0))
(player_type (id 0)(player_speed_max 1)(stamina_inc_max 45)
  (player_decay 0.4)(inertia_moment 5)(dash_power_rate 0.006)
  (player_size 0.3)(kickable_margin 0.7)(kick_rand 0)
  (extra_stamina 0)(effort_max 1)(effort_min 0.6))
(player_type (id 1)(player_speed_max 1.1956)(stamina_inc_max 30.06)
  (player_decay 0.4554)(inertia_moment 6.385)(dash_power_rate 0.007494)
  (player_size 0.3)(kickable_margin 0.829)(kick_rand 0.0645)
  (extra_stamina 9.4)(effort_max 0.9812)(effort_min 0.5812))
(player_type (id 2)(player_speed_max 1.135)(stamina_inc_max 33.4)
```

(continues on next page)

(continued from previous page)

```
  (player_decay 0.4292)(inertia_moment 5.73)(dash_power_rate 0.00716)
  (player_size 0.3)(kickable_margin 0.8198)(kick_rand 0.0599)
  (extra_stamina 31.3)(effort_max 0.9374)(effort_min 0.5374))
(player_type (id 3)(player_speed_max 1.1964)(stamina_inc_max 31.24)
  (player_decay 0.4664)(inertia_moment 6.66)(dash_power_rate 0.007376)
  (player_size 0.3)(kickable_margin 0.88)(kick_rand 0.09)
  (extra_stamina 47.1)(effort_max 0.9058)(effort_min 0.5058))
(player_type (id 4)(player_speed_max 1.151)(stamina_inc_max 37.8)
  (player_decay 0.45)(inertia_moment 6.25)(dash_power_rate 0.00672)
  (player_size 0.3)(kickable_margin 0.8838)(kick_rand 0.0919)
  (extra_stamina 44.1)(effort_max 0.9118)(effort_min 0.5118))
(player_type (id 5)(player_speed_max 1.1544)(stamina_inc_max 34.68)
  (player_decay 0.4352)(inertia_moment 5.88)(dash_power_rate 0.007032)
  (player_size 0.3)(kickable_margin 0.8052)(kick_rand 0.0526)
  (extra_stamina 47.1)(effort_max 0.9058)(effort_min 0.5058))
(player_type (id 6)(player_speed_max 1.193)(stamina_inc_max 36.7)
  (player_decay 0.4738)(inertia_moment 6.845)(dash_power_rate 0.00683)
  (player_size 0.3)(kickable_margin 0.885)(kick_rand 0.0925)
  (extra_stamina 92)(effort_max 0.816)(effort_min 0.416))
(sense_body 0 (view_mode high normal) (stamina 4000 1) (speed 0 0)
  (head_angle 0) (kick 0) (dash 0) (turn 0) (say 0) (turn_neck 0)
  (catch 0) (move 0) (change_view 0) (arm (movable 0) (expires 0)
  (target 0 0) (count 0)) (focus (target none) (count 0)) (tackle
  (expires 0) (count 0)))
(see 0 ((f c t) 6.7 27 0 0) ((f r t) 58.6 3) ((f g r b) 73 37)
  ((g r) 69.4 32) ((f g r t) 66 27) ((f p r c) 55.7 41)
  ((f p r t) 45.2 22) ((f t 0) 6.3 -18 0 0)
  ((f t r 10) 16.1 -7 0 0) ((f t r 20) 26 -4 0 0)
  ((f t r 30) 36.2 -3) ((f t r 40) 46.1 -2)
  ((f t r 50) 56.3 -2) ((f r 0) 73.7 30) ((f r t 10) 68.7 23)
  ((f r t 20) 66 15) ((f r t 30) 64.1 6) ((f r b 10) 79 37)
  ((f r b 20) 85.6 42))
(sense_body 0 (view_mode high normal) (stamina 4000 1) (speed 0 0)
  (head_angle 0) (kick 0) (dash 0) (turn 0) (say 0) (turn_neck 0)
  (catch 0) (move 0) (change_view 0) (arm (movable 0) (expires 0)
  (target 0 0) (count 0)) (focus (target none) (count 0)) (tackle
  (expires 0) (count 0)))
(see 0 ((f c t) 6.7 27 0 0) ((f r t) 58.6 3) ((f g r b) 73 37)
  ((g r) 69.4 32) ((f g r t) 66 27) ((f p r c) 55.7 41)
  ((f p r t) 45.2 22) ((f t 0) 6.3 -18 0 0)
  ((f t r 10) 16.1 -7 0 0) ((f t r 20) 26 -4 0 0)
  ((f t r 30) 36.2 -3) ((f t r 40) 46.1 -2)
  ((f t r 50) 56.3 -2) ((f r 0) 73.7 30) ((f r t 10) 68.7 23)
  ((f r t 20) 66 15) ((f r t 30) 64.1 6) ((f r b 10) 79 37)
  ((f r b 20) 85.6 42))
...
```

You can still type commands (such as `(move 0 0)` or `(turn 45)`) that the player will then send to the server. You should be able to see the result of these commands on the monitor window.

3.6 How to stop the server

The correct procedure for stopping the server is:

1. Stop all clients (players)
2. Stop all monitors by clicking on the quit button
3. `ctrl-c` at the terminal window where you started the server in order to terminate it

If you follow this procedure, you will not only stop all visible running processes but also make sure that all those processes that may be running in the background (such as the server) are also stopped. The problem that arises when you don't properly shut down the server is that you may not be able to start another process unless you start it with different parameters.

Also, if you don't stop the simulator with a ctrl-c, then the logfiles will no be closed properly (only important if you are using compressed logging) and they will not be renamed correctly.

NOTE: It is sometimes useful and convenient to terminate processes using their name. Using the **kill** operating system command involves finding the process number of the process you want to stop using the **ps** command. A simpler way to eradicate all processes that have a specific name is by means of the **killall** command, for example: `killall rcssserver` is sufficient to kill all processes with the name **rcssserver**.

3.7 Troubleshooting

In this section we list known problems and try to give some solutions or at least point you in the right direction.

If you run into any errors in configuring, building or running the simulator, which are not reported here please submit a bug report via the RoboCup Soccer Simulator website, https://rcsoccersim.github.io/, especially if you can provide a patch or hint to the solution of the problem.

Last update: Mar 25, 2024

CHAPTER

FOUR

SOCCER SERVER

4.1 Objects

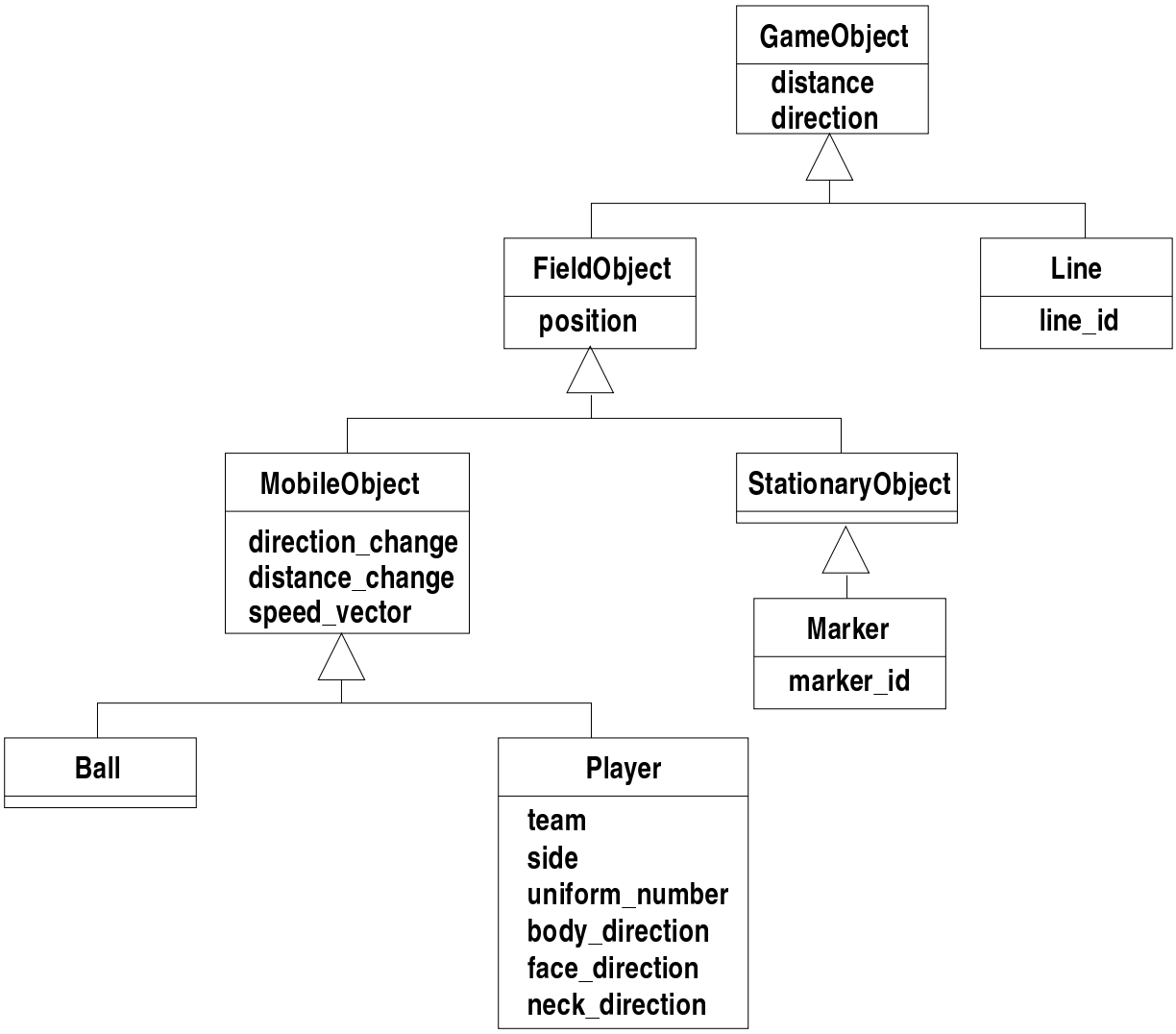


Fig. 4.1: UML diagram of the objects in the simulation

4.2 Protocols

4.2.1 Player Command Protocol

Connecting, reconnecting, and disconnecting

From player to server	From server to client
(init *TeamName* [(version *VerNum*)] [(goalie)]) *TeamName* ::= [+-_a-zA-Z0-9]+ *VerNum* ::= the protocol version (e.g. 15)	(init *Side Unum PlayMode*) *Side* ::= `l` | `r` *Unum* ::= 1~11 *PlayMode* ::= one of play modes (error no_more_team_or_player_or_goalie)
(reconnect *TeamName Unum*) TeamName := [+-_a-zA-Z0-9]+	(error *Side PlayMode*) *Side* ::= `l` | `r` *Unum* ::= 1~11 *PlayMode* ::= one of play modes (error reconnect)
(bye)	

If your client connects or reconnects successfully with a protocol version >= 7.0, the server will additionally send following messages: `server_param` (a message containing the server parameters), `player_param` (a message containing the player parameters) and `player_type` (a message containing the player types). Finally, the player will receive a message on changed players (see Sec. *Heterogeneous Players*).

Initial Settings

From player to server	From server to player
(compression *Level*) *Level* ::= zlib compression level [0,9] or negative number	(ok compression *Level*) (warning compression_unsupported)
(clang (ver *MinVer MaxVer*)) *MinVer* ::= integer *MaxVer* ::= integer	(ok clang (ver *MinVer MaxVer*))
(ear (*OnOff* [*Team*] [*Type*])) *OnOff* ::= `on` | `off` *Team* ::= `our` | `opp` | `left` | `right` | `l` | `r` | *TeamName*	no response if succeeded (error no team with name *TeamName*)
(synch_see)	(ok synch_see)
(gaussian_see)	(ok gaussian_see)

Player Control

From player to server	Only once per cycle
(turn *Moment*) *Moment* ::= **minmoment** ~ **maxmoment** degrees	Yes
(dash *Power* [*Direction*]) (dash (l *Power* [*Direction*]) (r *Power* [*Direction*])) (dash (r *Power* [*Direction*]) (l *Power* [*Direction*])) (dash (l *Power* [*Direction*])) (dash (r *Power* [*Direction*])) *Power* ::= **min_dash_power** ~ **max_dash_power** *Direction* ::= **min_dash_angle** ~ **max_dash_angle** degrees Note: backward dash (negative power) consumes double stamina.	Yes
(kick *Power Direction*) *Power* ::= **minpower** ~ **maxpower** *Direction* ::= **minmoment** ~ **maxmoment** degrees	Yes
(tackle *PowerOrAngle* [*Foul*]) *PowerOrAngle* ::= **minmoment** ~ **maxmoment** degrees : if client version >= 12 *PowerOrAngle* ::= **-max_back_tackle_power** ~ **max_tackle_power** : if client version < 12 *Foul* ::= `true` | `false` | `on` | `off`	Yes
(catch *Direction*) *Direction* ::= **minmoment** ~ **maxmoment** degrees	Yes
(move *X Y*) *X* ::= real number *Y* ::= real number	Yes
(change_view [*Width*] *Quality*) *Width* ::= `narrow` | `normal` | `wide` *Quality* ::= `high` | `low`	No
	No

Others

From player to server	From server to player
(sense_body)	sense_body message
(score)	(score *Time Our Opp*) *Time* ::= simulation cycle of rcssserver *Our* ::= sender's team score *Opp* ::= opponent team score

The server may respond to the above commands with the errors: (error unknown command) or (error illegal command form)

4.2.2 Player Sensor Protocol

The following table shows the protocol for client version 14 or later.

From server to player

(hear *Time Sender* "*Message*")
(hear *Time OnlineCoach CoachLanguageMessage*)
Time ::= simulation cycle of rcssserver
Sender ::= `online_coach_left` | `online_coach_right` | `coach` | `referee` | `self` | *Direction*
Direction ::= -180 ~ 180 degrees
Message ::= string
OnlineCoach ::= `online_coach_left` | `online_coach_right`
CoachLanguageMessage ::= see the standard coach language section

(see *Time ObjInfo+*)
Time ::= simulation cycle of rcssserver
ObjInfo ::=
(*ObjName Distance Direction DistChange DirChange BodyFacingDir HeadFacingDir* [*PointDir*] [t] [k]])
| (*ObjName Distance Direction DistChange DirChange* [*PointDir*] [{t|k}])
| (*ObjName Distance Direction* [t] [k])
| (*ObjName Diretion*)
ObjName ::= (p ["*TeamName*" [*UniformNumber* [goalie]]])
| (b)
| (g {l|r})
| (f c)
| (f {l|c|r} {t|b})
| (f p {l|r} {t|c|b})
| (f g {l|r} {t|b})
| (f {l|r|t|b} 0)
| (f {t|b} {l|r} {10|20|30|40|50})
| (f {l|r} {t|b} {10|20|30})
| (l {l|r|t|b} 0)
| (P)
| (B)
| (G)
| (F)
Distance ::= positive real number
Direction ::= -180 ~ 180 degrees
DistChange ::= real number
DirChange ::= real number
BodyFacingDir ::= -180 ~ 180 degrees
HeadFacingDir ::= -180 ~ 180 degrees
PointDir ::= -180 ~ 180 degrees
TeamName ::= string
UniformNumber ::= 1 ~ 11

(sense_body *Time*
(view_mode {high|low} {narrow|normal|wide})
(stamina *Stamina Effort Capacity*)
(speed *AmountOfSpeed DirectionOfSpeed*)
(head_angle *HeadAngle*)

4.3 Sensor Models

A RoboCup agent has three different sensors (and one special sensor). The aural sensor detects messages sent by the referee, the coaches and the other players. The visual sensor detects visual information about the field, like the distance and direction to objects in the player's current field of view. The visual sensor also works as a proximity sensor by "seeing" objects that are close, but behind the player. The body sensor detects the current "physical" status of the player, like its stamina, speed and neck angle. Together the sensors give the agent quite a good picture of the environment.

4.3.1 Aural Sensor Model

Aural sensor messages are sent when a client or a coach sends a say command. The calls from the referee is also received as aural messages. All messages are received immediately.

The format of the aural sensor message from the soccer server is:

(hear *Time Sender* ''*Message*")

- *Time* indicates the current time.
- *Sender* is the relative direction to the sender if it is another player, otherwise it is one of the following:
  - `self`: when the sender is yourself.
  - `referee`: when the sender is the referee.
  - `online_coach_left` or `online_coach_right`: when the sender is one of the online coaches.
- *Message* is the message. The maximum length is **server::say_msg_size** bytes. The possible messages from the referee are described in Section *Play Modes and referee messages*.

The server parameters that affects the aural sensor are described in Table 4.1.

Table 4.1: Parameters for the aural sensor.

Parameter in server.conf	Value
audio_cut_dist	50.0
hear_max	1
hear_inc	1
hear_decay	1

Capacity of the Aural Sensor

A player can only hear a message if the player's hear capacity is at least **server::hear_decay**, since the hear capacity of the player is decreased by that number when a message is heard. Every cycle the hear capacity is increased with **server::hear_inc**. The maximum hear capacity is **server::hear_max**. To avoid a team from making the other team's communication useless by overloading the channel the players have separate hear capacities for each team. With the current server.conf file this means that a player can hear at most one message from each team every second simulation cycle.

If more messages arrive at the same time than the player can hear, the messages actually heard are chosen randomly. This rule does not include messages from the referee, or messages from oneself. From rcssserver 8.04, players can send `attentionto` commands to focus their attention on a particular player.

Focus

If the player focuses on player A from team T (AKA pTA), the player will hear one message selected randomly from the say messages issued by pTA in the previous cycle. If pTA did not issue any say commands, the player will hear one message selected randomly from all the say messages issued by players in team T. At the same time, the player will hear one message selected randomly from the other team. If attentionto is off (default) the player will hear one message from each team selected randomly from the messages available.

The way to focus is using `attentionto` commands. See *Attentionto Model* in detail.

Range of Communication

A message said by a player is transmitted only to players within **server::audio_cut_dist** meters from that player. For example, a defender, who may be near his own goal, can hear a message from his goal-keeper but a striker who is near the opponent goal can not hear the message. Messages from the referee can be heard by all players.

Aural Sensor Example

This example should show which messages get through and how to calculated the hear capacity.

Example: Each coach sends a message every cycle. The referee send a message every cycle. The four players in the example all send a message every cycle. Show which messages gets through during 10 cycles (6 might be enough).

4.3.2 Vision Sensor Model

Basics

The visual sensor reports the objects currently seen by the player. The information is automatically sent to the player with the frequency determined by the player's view width, view quality, and the synchronous/asynchronous mode. Furthermore, the server mixes noise into the information sent to the player. There are two types of noise models: the quantiazaiton model and the Gaussian model, and the default is the quantization model. Gaussian model was introduced from version 19

Visual information arrives from the server in the following basic format:

(see *ObjName Distance Direction DistChng DirChng BodyDir HeadDir* [t|k])

Distance, *Direction*, *DistChng* and *DirChng* are calculated in the following way:

$$
\begin{aligned}
p_{rx} &= p_{xt} - p_{xo} \\
p_{ry} &= p_{yt} - p_{yo} \\
v_{rx} &= v_{xt} - v_{xo} \\
v_{ry} &= v_{yt} - v_{yo} \\
a_o &= AgentBodyDir + AgentHeadDir \\
Distance &= \sqrt{p_{rx}^2 + p_{ry}^2} \\
Direction &= \arctan\left(p_{ry}/p_{rx}\right) - a_o \\
e_{rx} &= p_{rx}/Distance \\
e_{ry} &= p_{ry}/Distance \\
DistChng &= (v_{rx} * e_{rx}) + (v_{ry} * e_{ry}) \\
DirChng &= [(-(v_{rx} * e_{ry}) + (v_{ry} * e_{rx}))/Distance] * (180/\pi) \\
BodyDir &= PlayerBodyDir - a_o \\
HeadDir &= PlayerHeadDir - a_o
\end{aligned}
$$

where (p_{xt}, p_{yt}) is the absolute position of the target object, (p_{xo}, p_{yo}) is the absolute position of the sensing player, (v_{xt}, v_{yt}) is the absolute velocity of the target object, (v_{xo}, v_{yo}) is the absolute velocity of the sensing player, and a_o is the absolute direction the sensing player is facing. The absolute facing direction of a player is the sum of the *BodyDir* and the *HeadDir* of that player. In addition to it, (p_{rx}, p_{ry}) and (v_{rx}, v_{ry}) are respectively the relative position and the relative velocity of the target, and (e_{rx}, e_{ry}) is the unit vector that is parallel to the vector of the relative position. *BodyDir* and *HeadDir* are only included if the observed object is a player, and is the body and head directions of the observed player relative to the body and head directions of the observing player. Thus, if both players have their bodies turned in the same direction, then *BodyDir* would be 0. The same goes for *HeadDir*.

The **(goal r)** object is interpreted as the center of the right hand side goalline. **(f c)** is a virtual flag at the center of the field. **(f l b)** is the flag at the lower left of the field. **(f p l b)** is a virtual flag at the lower right corner of the penalty box on the left side of the field. **(f g l b)** is a virtual flag marking the right goalpost on the left goal. The remaining types of flags are all located 5 meters outside the playing field. For example, **(f t l 20)** is 5 meters from the top sideline and 20 meters left from the center line. In the same way, **(f r b 10)** is 5 meters right of the right sideline and 10 meters below the center of the right goal. Also, **(f b 0)** is 5 meters below the midpoint of the bottom sideline.

In the case of **(l ...)**, *Distance* is the distance to the point where the center line of the player's view crosses the line, and *Direction* is the direction of the line.

Currently there are 55 flags (the goals counts as flags) and 4 lines to be seen. All of the flags and lines are shown in Fig. 4.2.

In protocol versions 13+, when a player's team is visible, their tackling and kicking state is also visible via *t* and *k*. If the player is tackling, *t* is present. If they are kicking, *k* is present instead. If an observed player is tackling, the kicking flag is always overwritten by the tackle flag. The kicking state is visible the cycle directly after kicking.

Asynchronous mode and Synchronous mode

There are two modes available for all players in in the vision sensor: asynchronous mode and synchronous mode. The asynchronous mode functions exactly like the default time step in version 11 or older. Until server version 17, asynchronous mode is the default mode for all players. Since version 18, synchrnonous mode is the default mode. If v17 or older players wish to switch to synchronous mode, they have to send the `(synch_see)` command. Once they have switched to synchronous mode, they cannot return to asynchronous mode.

In asynchronous mode, the information is automatically sent to the player every **server::sense_step**, currently 150, milliseconds, in the default setting. The frequency is changed according to the player's view width and view quality. Please note that the message arrival timig is not synchronized with the simulation step interval.

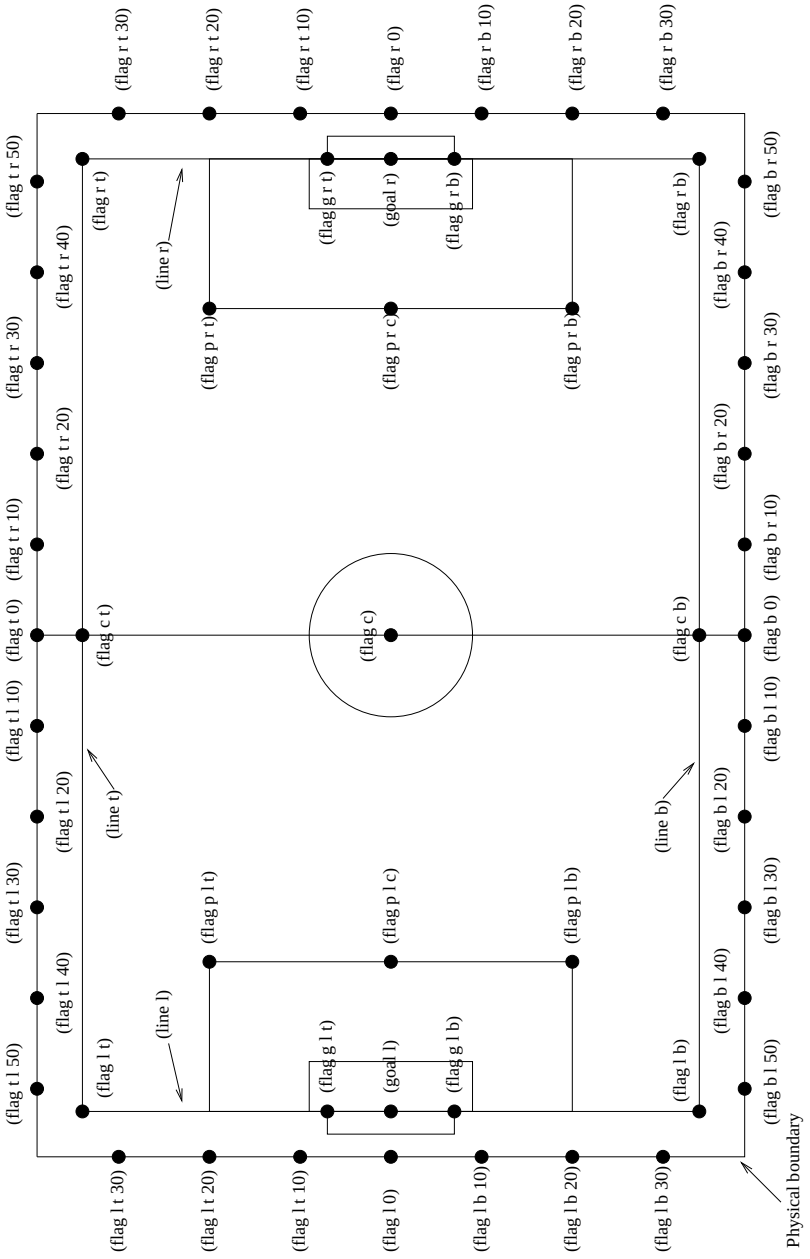


Fig. 4.2: The flags and lines in the simulation.

In synchronous mode, players' view quality is always set to 'high' and they cannot change their view quality. Instead of that, the message arrival timing of the visual information is automatically synchronized with the simulation step interval. The server executes the visual information transmission process **server::synch_see_offset** milliseconds after the simulation step update. The frequence is changed according to the player's view width.

Range of View and View Frequency in Asynchronous mode

The visible sector of a player is dependant on several factors. In asynchronous mode, we have the server parameters **server::sense_step** and **server::visible_angle** which determines the basic time step between visual information and how many degrees the player's normal view cone is. The default values in the asynchronous mode are 150 milliseconds and 90 degrees.

The player can also influence the frequency and quality of the information by changing *ViewWidth* and *ViewQuality*.

To calculate the current view frequency and view angle of the agent use equations (4.1) and (4.2).

$$view\_frequency = sense\_step * view\_quality\_factor * view\_width\_factor \tag{4.1}$$

where view_quality_factor is 1 if *ViewQuality* is `high` and 0.5 if *ViewQuality* is `low`; view_width_factor is 2 if *ViewWidth* is `narrow`, 1 if *ViewWidth* is `normal`, and 0.5 if *ViewWidth* is `wide`.

$$view\_angle = visible\_angle * view\_width\_factor \tag{4.2}$$

where view_width_factor is 0.5 if *ViewWidth* is `narrow`, 1 if *ViewWidth* is `normal`, and 2 if *ViewWidth* is `wide`.

The player can also "see" an object if it's within **server::visible_distance** meters of the player. If the objects is within this distance but not in the view cone then the player can know only the type of the object (ball, player, goal or flag), but not the exact name of the object. Moreover, in this case, the capitalized name, that is "B", "P", "G" and "F", is used as the name of the object rather than "b", "p", "g" and "f".

The following example and Fig. 4.3 are taken from [Stone98].

The meaning of the view_angle parameter is illustrated in Fig. 4.3. In this figure, the viewing agent is the one shown as two semi-circles. The light semi-circle is its front. The black circles represent objects in the world. Only objects within $view\_angle^\circ/2$, and those within visible_distance of the viewing agent can be seen. Thus, objects *b* and *g* are not visible; all of the rest are.

As object *f* is directly in front of the viewing agent, its angle would be reported as 0 degrees. Object *e* would be reported as being roughly -40°, while object *d* is at roughly 20°.

Also illustrated in Fig. 4.3, the amount of information describing a player varies with how far away the player is. For nearby players, both the team and the uniform number of the player are reported. However, as distance increases, first the likelihood that the uniform number is visible decreases, and then even the team name may not be visible. It is assumed in the server that **unum_far_length** $\leq$ **unum_too_far_length** $\leq$ **team_far_length** $\leq$ **team_too_far_length**. Let the player's distance be *dist*. Then

- If *dist* $\leq$ **unum_far_length**, then both uniform number and team name are visible.
- If **unum_far_length** $<$ *dist* $<$ **unum_too_far_length**, then the team name is always visible, but the probability that the uniform number is visible decreases linearly from 1 to 0 as *dist* increases.
- If *dist* $\geq$ **unum_too_far_length**, then the uniform number is not visible.
- If *dist* :math`leq` **team_far_length**, then the team name is visible.
- If **team_far_length** $<$ *dist* $<$ **team_too_far_length**, then the probability that the team name is visible decreases linearly from 1 to 0 as *dist* increases.
- If *dist* $\geq$ **team_too_far_length**, then the team name is not visible.

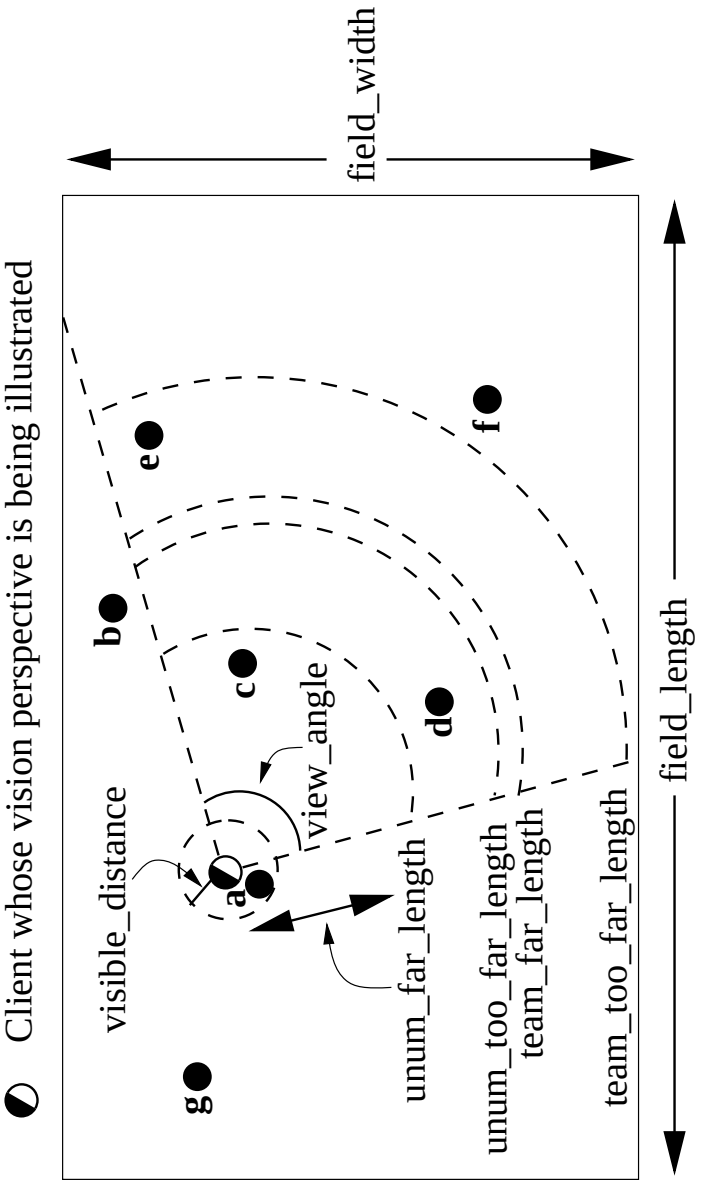


Fig. 4.3: The visible range of an individual agent in the soccer server. The viewing agent is the one shown as two semi-circles. The light semi-circle is its front. The black circles represent objects in the world. Only objects within **server::view_angle**/2, and those within **server::visible_distance** of the viewing agent can be seen. **unum_far_length**, **unum_too_far_length**, **team_far_length**, and **team_too_far_length** affect the amount of precision with which a player's identity is given. Taken from [Stone98].

For example, in Fig. 4.3, assume that all of the labeled circles are players. Then player *c* would be identified by both team name and uniform number; player *d* by team name, and with about a 50% chance, uniform number; player *e* with about a 25% chance, just by team name, otherwise with neither; and player *f* would be identified simply as an anonymous player.

Table 4.2: Parameters for the visual sensors in server.conf.

Parameter in `server.conf`	Value
server::sense_step	150
server::visible_angle	90.0
server::visible_distance	3.0
server::quantize_step	0.1
server::quantize_step_l	0.01

Table 4.3: Heterogenious parameters for the visual sensors.

Parameters in player_type	Value
unum_far_length	20.0
unum_too_far_length	40.0
team_far_length	maximum_dist_in_pitch
team_too_far_length	maximum_dist_in_pitch
player_max_observation_length	maximum_dist_in_pitch
ball_vel_far_length	20
ball_vel_too_far_length	40
ball_max_observation_length	maximum_dist_in_pitch
flag_chg_far_length	20
flag_chg_too_far_length	40
flag_max_observation_length	maximum_dist_in_pitch

Range of View and View Frequency in Synchronous mode

In synchronous mode, the "low" view quality is not available, and the view widths in Table 4.4 are available. In all view widths, rcssserver send see messages at **server::synch_see_offset** milliseconds from the beginning of the cycle.

The amount of information the player can receive changes depending on the distance to the target object, the same as in asynchronous mode.

Table 4.4: Settings of the synchronous mode

mode	view width(degree)	see frequency
narrow	60	every cycle
normal	120	every 2 cycles
wide	180	every 3 cycles

Focus Point

The focus point concept was developed in server version 18 to make observations in the game more closely resemble those made by human observers and camera lenses. The position of the focus point affects the observation noise model. In brief, the server introduces more noise to the distance of an observed object if the object is farther from the observer's focus point.

The default position of the focus point is the player's position. However, the player can change the focus point by sending the `(change_focus DIST_MOMENT DIR_MOMENT)` command. It's worth noting that the focus point cannot be outside the player's view angle, and its maximum distance from the player is 40.

This feature is available to players using version 18 or above on server versions 18 or above.

Visual Sensor Noise Model: Quantization

The quantizaiton noise model is the default mode for all players. In this mode, in order to introduce noise in the visual sensor data the values sent from the server is quantized. For example, the distance value of the object, in the case where the object in sight is a ball or a player, is quantized in the following manner:

$$
\begin{aligned}
p_{rfx} &= p_{xf} - p_{xo} \\
p_{rfy} &= p_{yf} - p_{yo} \\
f &= \sqrt{p_{rfx}^2 + p_{rfy}^2} \\
f' &= \exp(\mathrm{Q}uantize(\log(f), quantize\_step)) \\
d'' &= \mathrm{Q}uantize(max(0.0, d - (f - f')), 0.1)
\end{aligned}
$$

where (p_{xf}, p_{yf}) is the absolute position of the focus point of the observer, (p_{xo}, p_{yo}) is the absolute position of the observer, d is the exact distance of the observer to the object, f and f' are the exact distance and quantized distance of the focus point to the object respectively, and d'' is the result distance value sent to the observer. $Quantize(V, Q)$ is as follow:

$$
\mathrm{Q}uantize(V, Q) = ceiling(V/Q) \cdot Q
$$

This means that players can not know the exact positions of very far objects. For example, when distance from the focus point is about 100.0 the maximum noise is about 10.0, while when distance is less than 10.0 the noise is less than 1.0.

In the case of lines, the distance value is quantized in the following manner.

$$
d' = \mathrm{Q}uantize(\exp(\mathrm{Q}uantize(\log(d), quantize\_step\_l)), 0.1)
$$

Visual Sensor Noise Model: Gaussian

The Gaussian noise model has been introduced in server version 19. Players can change their noise model by sending `(gaussian_see)` command. All version players can use this command. If the command is accepted, rcssserver sent a reply message, `(ok gaussian_see)`.

In this model, the noised distance in the player's observation is determined by a Gaussian distribution:

$$
\begin{aligned}
s &= d \times r_d + f \times r_f \\
d' &= d \times max(0.0, NormalDistribution(d, s))
\end{aligned}
$$

where d is the exact distance from the observer to the object, f is the exact distance from the focus point to the object, and d' is the result distance value sent to the observer. $NormalDistribution(mean, stddev)$ is a random number

generator based on a Gaussian distribution with given mean and standard deviation. Therefore, the resulting value d' is generated from a Gaussian distribution with mean d' and standard deviation s.

r_d and r_f are the noise rate parameter defined as heterogenious parameters. There are four parameters, **dist_noise_rate**, **focus_dist_noise_rate**, **land_dist_noise_rate**, and **land_focus_dist_noise_rate**. The former two paramters are used for movable object (ball and players), and the latter two paramters are used for landmark objects (flags and goals). In server version 19, all heterogeneous players use same values defined in server.conf (Table 4.5).

Table 4.5: Server parameters for Gaussian model.

Parameters in player_type	Value
dist_noise_rate	0.0125
focus_dist_noise_rate	0.0125
land_dist_noise_rate	0.00125
land_focus_dist_noise_rate	0.00125

4.3.3 Body Sensor Model

The body sensor reports the current "physical" status of the player. he information is automatically sent to the player every **server::sense_body_step**, currently 100, milli-seconds.

The format of the body sensor message is:

(sense_body *Time*
(view_mode *ViewQuality ViewWidth*)
(stamina *Stamina Effort Capacity*)
(speed *AmountOfSpeed DirectionOfSpeed*)
(head_angle *HeadAngle*)
(kick *KickCount*)
(dash *DashCount*)
(turn *TurnCount*)
(say *SayCount*)
(turn_neck *TurnNeckCount*)
(catch *CatchCount*)
(move *MoveCount*)
(change_view *ChangeViewCount*)
(arm (movable *MovableCycles*) (expires *ExpireCycles*) (count *PointtoCount*))
(focus (target {none|{l|r} *Unum*}) (count *AttentiontoCount*))
(tackle (expires *ExpireCycles*) (count *TackleCount*))
(collision {none|[(ball)] [(player)] [(post)]})
(foul (charged *FoulCycles*) (card {red|yellow|none})))
(focus_point *FocusDist FocusDir*))

- *ViewQuality* is one of `high` and `low`.
- *ViewWidth* is one of `narrow`, `normal`, and `wide`.
- *AmountOfSpeed* is an approximation of the amount of the player's speed.

- *DirectionOfSpeed* is an approximation of the direction of the player's speed.
- *HeadDirection* is the relative direction of the player's head.
- **Count* variables are the total number of commands of that type executed by the server. For example *DashCount* = 134 means that the player has executed 134 **dash** commands so far.
- *MovableCycles*
- *ExpireCycles*
- *FoulCycles*

TODO: add descriptions about values. arm [8.03], focus [8.04], tackle [8.04], collision [12.0.0_pre-20071217], foul [14.0.0], focus_point [18.0.0] in NEWS

The semantics of the parameters are described where they are actually used. The *ViewQuality* and *ViewWidth* parameters are for example described in the Section *Vision Sensor Model*.

The server parameters that affects the body sensor are described in the following table:

Table 4.6: Parameters for the body sensor.

Parameter in server.conf	Value
server::sense_body_step	100

4.3.4 Fullstate Sensor Model

TODO

4.4 Movement Models

4.4.1 Basics

In each simulation step, movement of each object is calculated as following manner:

$$\begin{aligned}(u_x^{t+1}, u_y^{t+1}) &= (v_x^t, v_y^t) + (a_x^t, a_y^t) : accelerate \\ (p_x^{t+1}, p_y^{t+1}) &= (p_x^t, p_y^t) + (u_x^{t+1}, u_y^{t+1}) : move \\ (v_x^{t+1}, v_y^{t+1}) &= decay \times (u_x^{t+1}, u_y^{t+1}) : decay\ speed \\ (a_x^{t+1}, a_y^{t+1}) &= (0, 0) : reset\ acceleration\end{aligned} \tag{4.3}$$

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

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

where θ^t is the direction of the object in timestep t and power_rate is `dash_power_rate` or is calculated from `kick_power_rate` as described in Sec. *Kick Model*. In the case of a player, this is just the direction the player is facing. In the case of a ball, its direction is given as the following manner:

$$\theta_{ball}^t = \theta_{kicker}^t + Direction$$

where θ_{ball}^t and θ_{kicker}^t are directions of ball and kicking player respectively, and *Direction* is the second parameter of a **kick** command.

4.4.2 Movement Noise Model

In order to reflect unexpected movements of objects in real world, rcssserver adds noise to the movement of objects and parameters of commands.

Concerned with movements, noise is added into Eqn.:ref:*eq:u-t* as follows: **TODO: new noise model. See [12.0.0 pre-20071217] in NEWS**

$$(u_x^{t+1}, u_y^{t+1}) = (v_x^t, v_y^t) + (a_x^t, a_y^t) + (\tilde{r}_{\mathrm{r}max}, \tilde{r}_{\mathrm{r}max})$$

where $\tilde{r}_{\mathrm{r}max}$ is a random number whose distribution is uniform over the range $[-\mathrm{r}max, \mathrm{r}max]$. $\mathrm{r}max$ is a parameter that depends on amount of velocity of the object as follows:

$$\mathrm{r}max = \mathrm{r}and \cdot |(v_x^t, v_y^t)|$$

where $\mathrm{r}and$ is a parameter specified by **server::player_rand** or **server::ball_rand**.

Noise is added also into the *Power* and *Moment* arguments of a command as follows:

$$argument = (1 + \tilde{r}_{\mathrm{r}and}) \cdot argument$$

4.4.3 Collision Model

Collision with other movable objects

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

Collision with goal posts

Goal posts are circular with a radius of 6cm and they are located at:

$$x = \pm(FIELD\_LENGTH \cdot 0.5 - 6cm)$$
$$y = \pm(GOAL\_WIDTH \cdot 0.5 + 6cm)$$

The goal posts have different collision dynamics than other objects. An object collides with a post if it's path gets within object.size + 6cm of the center of the post. An object that collides with the post bounces off elastically.

4.5 Action Models

4.5.1 Catch Model

The goalie is the only player with the ability to catch a ball. The goalie can catch the ball in play mode `play_on` in any direction, if the ball is within the catchable area and the goalie is inside the penalty area. If the goalie catches into direction φ, the catchable area is a rectangular area of length **server::catchable_area_l** and width **server::catchable_area_w** in direction φ (see Fig. 4.4). The ball will be caught with probability **server::catch_probability**, if it is inside this area (and it will not be caught if it is outside this area). For the current values of catch command parameters see Table 4.7:

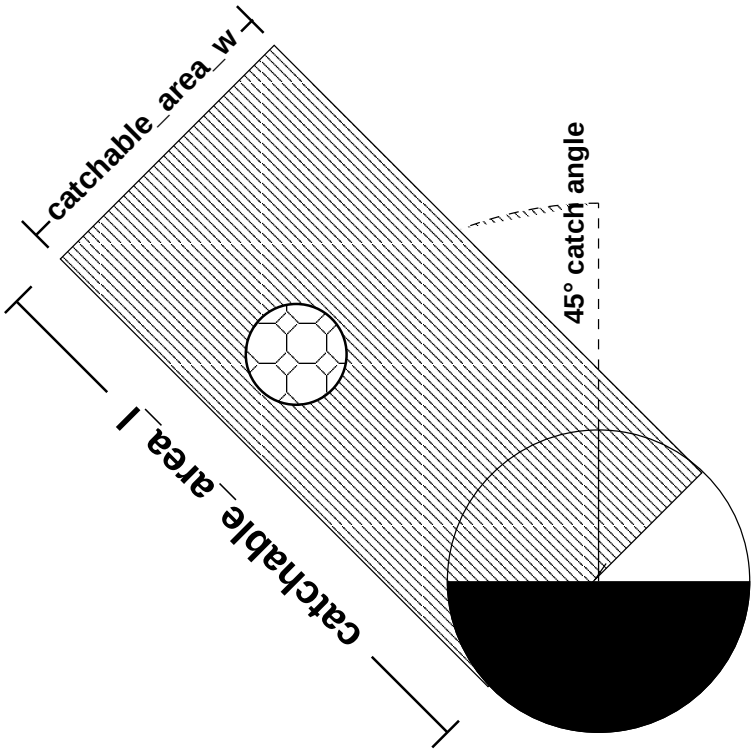


Fig. 4.4: Catchable area of the goalie when doing a (catch 45)

Table 4.7: Parameters for the goalie catch command

Parameter in `server.conf` and `player.conf`	Value
server::catchable_area_l	2.0
server::catchable_area_w	1.0
server::catch_probability	1.0
server::catch_ban_cycle	5
server::goalie_max_moves	2
player::catchable_area_l_stretch_max	1.3
player::catchable_area_l_stretch_min	1

First time when goalie has been introduced in Soccer Simulation 2D was with server version 4.0.0: When a client connects the server with '(init TEAMNAME (goalie)', the client becomes a goalies. The goalie can use '(catch DIR)' command that enable to capture the ball.

With server version 4.0.2 another parameter named **server::catch_probability** has been introduced. This parameter represents the probability that a goalie succeed to catch the ball by a catch command. (default value: 1.0)

If the goalie successfully catches a ball it is moved adjacent to and facing the ball and both the goalie and ball have their velocities set to zero. When the goalie moves, dashes or turns while the ball is caught, the ball remains adjacent to and directly in front of the goalie.

The goalie can issue catch commands at any location. If the catch is successful, and the ball is outside of the penalty area or if the goalie moves the ball outside of the penalty area and it's still in the field, an indirect free kick is awarded to the opposing team at the ball's current location. If a caught ball is moved over the goal line but not inside the goal, a corner kick is awarded. If a caught ball is moved into the goal, a goal is awarded.

Later, in server version 14.0.0 a heterogeneous goalie has been introduced. Beginning with this version online

coaches can change the player type of goalie. The 'catchable_area_l_stretch' variable was added to each heterogeneous player type through two new parameters: player::catchable_area_l_stretch_min (default value: 1.0) and player::catchable_area_l_stretch_max (default value: 1.3)

The following pseudo code shows a trade-off rule of the catch model:

```
// catchable_area_l_stretch is the heterogeneous parameter, currenlty within [1.0,1.3]

double this_catchable_are_delta = server::catchable_area_l * (catchable_area_l_stretch -␣
↪1.0)
double this_catchable_area_l_max = server::catchable_area_l + this_catchable_are_delta
double this_catchable_area_l_min = server::catchable_area_l - this_catchable_are_delta

if (ball_pos is inside the MINIMAL catch area)
{
    // the MINIMAL catch area has a length of this_catchable_area_l_min and width␣
↪server::catchable_area_w goalie
    // catches the ball with probability server::catch_probability (which is 1.0 by␣
↪default)
}
else if (ball_pos is beyond the MAXIMAL (stretched) area)
{
    // the MAXIMAL catch area has a length of this_catchable_area_l_max and width␣
↪server::catchable_area_w goalie
    // definitely misses the ball
}
else
{
    double ball_relative_x = (ball_pos - goalie_pos).rotate(-(goalie_body + catch_dir)).x
    double catch_prob = server::catch_probability
                        - server::catch_probability
                          * (ball_relative_x - this_catchable_area_l_min)
                          / (this_catchable_area_l_max - this_catchable_area_l_min)
    // goalie catches the ball with probability catch_prob it holds: catch_prob is in [0.
↪0,1.0]
}
```

If a catch command was unsuccessful, it takes **server::catch_ban_cycle** cycles until another catch command can be used (catch commands during this time have simply no effect). If the goalie succeeded in catching the ball, the play mode will change to `goalie_catch_ball_[l|r]` first and `free_kick_[l|r]`, after that during the same cycle. Once the goalie caught the ball, it can use the **move** command to move with the ball inside the penalty area. The goalie can use the **move** command **server::goalie_max_moves** times before it kicks the ball. Additional **move** commands do not have any effect and the server will respond with `(error too_many_moves)`. Please note that catching the ball, moving around, kicking the ball a short distance and immediately catching it again to move more than **server::goalie_max_moves** times is considered as ungentlemanly play.

Starting with server version 15.0.0 an improvement of the catch model has been introduced:

- If goalie fails to catch the ball beyond the fuzzy catchable area, the ball has no effect. (same as the previous model)
- If goalie fails to catch the ball within a fuzzy catchable area, the ball is accelerated to the catch command direction. (it is similar to the ball bouncing from the wall that the normal vector's direction is same as the catch command direction)

4.5.2 Dash Model

The **dash** command is used to accelerate the player in direction of its body. **dash** takes the acceleration *power* as a parameter. The valid range for the acceleration *power* can be configured in `server.conf`, the respective parameters are **server::min_dash_power** and **server::max_dash_power**. For the current values of parameters for the dash model, see the following table:

Table 4.8: Dash and Stamina Model Parameters

Default Parameters `server.conf`	Default Value (Range)	Heterogeneous Player Parameters `player.conf`	Value
server::min_dash_power	-100.0		
server::max_dash_power	100.0		
server::player_decay	0.4 ([0.3, 0.5])	player::player_decay_delta_	-0.1
server::inertia_moment	5.0 ([2.5, 7.5])	player::player_decay_delta_	0.1
		player::inertia_moment_del	25.0
server::player_accel_max	1.0		
server::player_rand	0.1		
server::player_speed_max	1.05		
server::player_speed_max_r	0.75		
server::stamina_max	8000.0		
server::stamina_capacity	130600.0		
server::stamina_inc_max	45.0 ([40.2, 52.2])	player::new_dash_power_ra	-0.0012
server::dash_power_rate	0.006 ([0.0048, 0.0068])	player::new_dash_power_ra	0.0008
		player::new_stamina_inc_m	-6000
server::extra_stamina	50.0 ([50.0, 100.0])	player::extra_stamina_delta	0.0
server::effort_init	1.0 ([0.8, 1.0])	player::extra_stamina_delta	50.0
server::effort_min	0.6 ([0.4, 0.6])	player::effort_max_delta_fa	-0.004
		player::effort_min_delta_fa	-0.004
server::effort_dec	0.3		
server::effort_dec_thr	0.005		
server::effort_inc	0.01		
server::effort_inc_thr	0.6		
server::recover_dec_thr	0.3		
server::recover_dec	0.002		
server::recover_init	1.0		
server::recover_min	0.5		
server::wind_ang	0.0		
server::wind_dir	0.0		
server::wind_force	0.0		
server::wind_rand	0.0		

Each player has a certain amount of stamina that will be consumed by **dash** commands. At the beginning of each half, the stamina of a player is set to **server::stamina_max**. If a player accelerates forward ($power > 0$), stamina is reduced by *power*. Accelerating backwards ($power < 0$) is more expensive for the player: stamina is reduced by $-2 \times power$. If the player's stamina is lower than the power needed for the **dash**, *power* is reduced so that the **dash** command does not need more stamina than available. Some extra stamina can be used every time the available power is lower than the needed stamina. The amount of extra stamina depends on the player type and the parameters **player::extra_stamina_delta_min** and **player::extra_stamina_delta_max**.

After reducing the stamina, the server calculates the *effective dash power* for the **dash** command. The effective dash power *edp* depends on the **dash_power_rate** and the current effort of the player. The effort of a player is a value between **effort_min** and **effort_max**; it is dependent on the stamina management of the player (see below).

$$\mathrm{e}dp = \mathrm{e}ffort \cdot \mathrm{d}ash\_power\_rate \cdot \mathrm{p}ower \tag{4.4}$$

edp and the players current body direction are tranformed into vector and added to the players current acceleration vector $\vec{a}_n$ (usually, that should be 0 before, since a player cannot dash more than once a cycle and a player does not get accelerated by other means than dashing).

At the transition from simulation step n to simulation step $n+1$, acceleration $\vec{a}_n$ is applied: **TODO: dash speed restriction. See [12.0.0_pre-20071217]**

1. $\vec{a}_n$ is normalized to a maximum length of **server::player_accel_max**.
2. $\vec{a}_n$ is added to current players speed $\vec{v}_n$. $\vec{v}_n$ will be normalized to a maximum length of **player_speed_max**. players, the maximum speed is a value between **server::player_speed_max + player::player_speed_max_delta_min** and **server::player_speed_max + player::player_speed_max_delta_max** in `player.conf`.
3. Noise $\vec{n}$ and wind $\vec{w}$ will be added to $\vec{v}_n$. Both noise and wind are configurable in *server.conf*. Parameters responsible for the wind are **server::wind_force**, **server::wind_dir** and **server::wind_rand**. With the current settings, there is no wind on the simulated soccer field. The responsible parameter for the noise is **server::player_rand**. Both direction and length of the noise vector are within the interval $[-|\vec{v}_n| \cdot \mathrm{p}layer\_rand \ldots |\vec{v}_n| \cdot \mathrm{p}layer\_rand]$.
4. The new position of the player $\vec{p}_{n+1}$ is the old position $\vec{p}_n$ plus the velocity vector $\vec{v}_n$ (i.e.the maximum distance difference for the player between two simulation steps is **player_speed_max**).
5. **player_decay** is applied for the velocity of the player: $\vec{v}_{n+1} = \vec{v}_n \cdot \mathrm{p}layer\_decay$. Acceleration $\vec{a}_{n+1}$ is set to zero.

Sideward and Omni-Directional Dashes

Besides the forward and backward dashes that were already described in the previous section, since version 13 the Soccer Server also supports the possibility to perform sideward and even omni-directional dashes. In addition to the already known parameter of the **dash(x)** command where $x \in [-100, 100]$ determines the relativ strength of the dash (with negative sign indicating a backward dash), the omni-directional dash model uses two parameters to the **dash** command:

$$dash(power, dir) \tag{4.5}$$

where $power$ determines the relative strength of the dash and dir represents the direction of the dash accelaration relative to the player's body angle. The format in which the command needs to be sent to the Soccer Server is `(dash <power> <dir>)`. If a negative value is used for $power$, then the reverse side angle of dir will be used. Practically, the direction of the dash is restricted to by the corresponding Soccer Server parameters to

$$dir \in [server :: min\_dash\_angle, server :: max\_dash\_angle]$$

The effective power of the dash command is determined by the absolute value of the dash direction. Players will always dash with full effective power (100%) alongside their current body orientation, i.e. when using a zero direction

angle as described in the preceding section. Two further Soccer Server parameters, `server::side_dash_rate` and `server::back_dash_rate`, determine the effective power that is applied when a non-straight dash is performed.

Thus, for example, strafing movements (90 degrees left/right to the player) will be performed with 40% of effective power, whereas backward dashes will performed with 60% (according to current Soccer Server parameter default values). For values between these four main directions a linear interpolation of the effective power will be applied. The following formula explains the maths behind the sideward dash model.

$$dir\_rate = \begin{cases} back\_dash\_rate - (back\_dash\_rate - side\_dash\_rate) * (1.0 - (fabs(dir) - 90.0)/90.0) & \text{if } fabs(dir) > 90.0 \\ side\_dash\_rate + (1.0 - side\_dash\_rate) * (1.0 - fabs(dir)/90.0)) & \text{else} \end{cases} \tag{4.6}$$

As discussed in the description of the forward/backward dash model in the preceding section, there exists the server parameter `server::min_dash_power` which determines the highest minimal value that can be used for the first parameter *power* of the dash command. It is expected that this parameter will be set to zero in future versions of the Soccer Server, while, for reasons of compatibility with older team binaries, its default value of -100 is encouraged currently.

Finally, the parameter `server::dash_angle_step` allows for a finer discreteness of players' dash directions. If this value is set to 90.0 degrees, players are allowed to dash into the four main directions, for a setting of 45.0 we arrive at eight different directions. Setting this parameter to 1.0, the Soccer Server is capable of emulating an omnidirectional movement model as it is commen, for example, in the MidSize League.

The following table summarizes all Soccer Server parameters that are of relevance for omni-directional dashing.

Table 4.9: Ominidirectional Dash Parameters

Default Parameters `server.conf`	Default Value (Range)	Heterogeneous Player Parameters `player.conf`	Value
server::server::max_dash_a	180.0		
server::server::min_dash_ar	-180.0		
server::side_dash_rate	0.4		
server::back_dash_rate	0.6		
server::dash_angle_step	1		

Bipedal Dash Model

Since rcssserver version 19, a bipedal dash model has been introduced. In the bipedal model, players can independently issue dash commands to the left and right legs. This means that players can now apply different accelerations to each leg. With the bipedal dash model, players can perform acceleration and direction changes simultaneously. The bipedal dash model is based on the differential drive dynamics model used in two-wheeled mobile robots and is available regardless of the client version.

In the bipedal dash model, parameters can be independently assigned to each leg using the dash command. The parameters, namely power and direction, are the same as the dash command in versions 18 and earlier. The derivation formula for acceleration obtained for each leg also follows the conventional model. The differences from the conventional model lie in the stamina consumption, the derivation formula for the player's body acceleration obtained as a result, and the player's rotation based on the speed difference between both legs.

Stamina consumed by each leg is half of the conventional model. The overall stamina consumption is the sum of stamina consumption for both legs. In other words, when the same power is applied to both legs (`dash (l power`

`dir) (r power dir))`, the stamina consumption is the same as applying that power to both legs in the conventional model `(dash power dir)`.

The dash command can be issued not only simultaneously to both legs but also separately. Even if the dash command is issued separately, as long as a dash command has been issued to both legs by the time of the cycle update, the same effect as issuing simultaneously can be achieved. If the dash command is issued to only one leg, rotation of the player does not occur, and acceleration is obtained solely from the individual leg.

Based on the given command parameters, velocity is first derived for each leg. This derivation formula is the same as the conventional model. Provisional accelerations $\hat{a}_L$ and $\hat{a}_R$ are independently calculated for each leg. Next, the current player velocity v_t and the provisional velocities $\hat{v}_L$ and $\hat{v}_R$ for each leg are obtained from the provisional accelerations. The provisional velocity $\hat{v}^{t+1}$ for the player's body is then determined by the average of $\hat{v}_L$ and $\hat{v}_R$. The player's body acceleration a^t is reverse-calculated from the difference between $\hat{v}^{t+1}$ and v^t. Noise is added according to the update formula in section *Movement Models*, and the velocity for the next step, $vt + 1$, is updated.

$$
\begin{aligned}
edp_L &= effort \times dash\_power\_rate \times dash\_rate \times dash\_power_L \\
edp_R &= effort \times dash\_power\_rate \times dash\_rate \times dash\_power_R \\
accel\_dir_L &= body\_angle^t + dash\_dir_L \\
accel\_dir_R &= body\_angle^t + dash\_dir_R \\
\hat{\vec{a}}_L &= edp_L \times (\cos(accel\_dir_L), \sin(accel\_dir_L)) \\
\hat{\vec{a}}_R &= edp_R \times (\cos(accel\_dir_R), \sin(accel\_dir_R)) \\
\hat{\vec{v}}_L &= (\vec{v^t} + \hat{\vec{a}}_L) \\
\hat{\vec{v}}_R &= (\vec{v^t} + \hat{\vec{a}}_R) \\
\hat{\vec{v}}^{t+1} &= \frac{\hat{\vec{v}}_L + \hat{\vec{v}}_R}{2} \\
\vec{a^t} &= \hat{\vec{v}}^{t+1} - \vec{v^t}
\end{aligned}
$$

When dash parameters are assigned to both legs and there is a difference in the velocity component of each leg in the body direction, the player rotates based on that speed difference. The rotation equation is identical to the differential drive kinematics.

$$
\begin{aligned}
\vec{e} &= (cos(body\_angle^t), sin(body\_angle^t)) \\
\hat{v}_{L,body} &= (\vec{v_t} + \vec{a_L}) \cdot \vec{e} \\
\hat{v}_{R,body} &= (\vec{v_t} + \vec{a_R}) \cdot \vec{e} \\
\\
\omega &= \frac{(\hat{v}_{L,body} - \hat{v}_{R,body})}{b} \\
body\_angle^{t+1} &= body\_angle^t + (1.0 + random(-player\_rand, player\_rand)) \times \omega
\end{aligned}
$$

where ω is the angular velocity, and b is the width between wheels (=player_size x 2).

Stamina Model

For the stamina of a player, there are three important variables: the *stamina* value, *recovery* and *effort*. *stamina* is decreased when dashing and gets replenished slightly each cycle. *recovery* is responsible for how much the *stamina* recovers each cycle, and the *effort* says how effective dashing is (see section above). Important parameters for the stamina model are changeable in the files `server.conf` and `player.conf`. Basically, the algorithm shown in the following code block says that every simulation step the stamina is below some threshold, effort or recovery are reduced until a minimum is reached. Every step the stamina of the player is above some threshold, *effort* is increased up to a maximum. The *recovery* value is only reset to 1.0 each half, but it will not be increased during a game.

```
# if stamina is below recovery decrement threshold, recovery is reduced
if stamina <= recover_dec_thr * stamina_max
  if recovery > recover_min
     recovery = recovery - recover_dec

# if stamina is below effort decrement threshold, effort is reduced
if stamina <= effort_dec_thr * stamina_max
  if effort > effort_min
    effort = effort - effort_dec
      effort = max(effort, effort_min)

# if stamina is above effort increment threshold, effort is increased
if stamina >= effort_inc_thr * stamina\_max
  if effort < effort_max
    effort = effort + effort_inc
    effort = min(effort, effort_max)

# recover the stamina a bit
stamina_inc = recovery * stamina_inc_max
stamina = min(stamina + stamina_inc, stamina_max)
```

In rcssserver version 13 or later, the **stamina_capacity** variable has been implemented as one of the player's stamina models in addition to the above three *stamina* variables. *stamina_capacity* is defined as the maximum recovery capacity of each player's stamina. When a player's *stamina* is recovered during a game, the same amount of *stamina* is also consumed from one's *stamina_capacity*. Once the player's *stamina_capacity* becomes 0, one's stamina is never recovered and the only **extra_stamina** is consumed instead of the normal *stamina*. The updated algorithm is shown in the following code block. `stamina_inc` can be available from the previous code block.

```
# stamina_inc is restricted by the residual capacity
if stamina_capacity >= 0.0
  if stamina_inc > stamina_capacity
    stamina_inc = stamina_capacity
stamina = min(stamina + stamina_inc, stamina_max)

# stamina capacity is reduced as the same amount as stamina_inc
if stamina_capacity >= 0.0
  stamina_capacity = max(0.0, stamina_capacity - stamina_inc)
```

stamina_capacity is reset to the initial value just after the kick-off of normal halves as well as the other stamina-related variables. However, *stamina_capacity* is never recovered at the half time of extra-inning games and before the penalty shootouts. The *stamina_capacity* is defined as one of the parameters of rcssserver **server::stamina_capacity** (the default value of *stamina_capacity* is 130600 as of rcsserver version 16.0.0). If *server::stamina_capacity* is set to a negative value, each player has an infinite stamina capacity. This setting makes the stamina-model including stamina_capacity completely the same with the stamina model before rcssserver version 13. *stamina_capacity* information is received as the following *sense_body message*:

(stamina <STAMINA> <EFFORT> <CAPACITY>)

4.5.3 Kick Model

The *kick* command takes two parameters, the kick power the player client wants to use (between **server::minpower** and **server::maxpower**) and the angle the player kicks the ball to. The angle is given in degrees and has to be between **server::minmoment** and **server::maxmoment** (see Table 4.10 for current parameter values).

Once the *kick* command arrived at the server, the kick will be executed if the ball is kick-able for the player and the player is not marked offside. The ball is kick-able for the player, if the distance between the player and the ball is between 0 and **kickable_margin**. Heterogeneous players can have different kickable margins. For the calculation of the distance during this section, it is important to know that if we talk of distance between player and ball, we talk about the minimal distance between the outer shape of both player and ball. So the distance in this section is the distance between the center of both objects *minus* the radius of the ball *minus* the radius of the player.

The first thing to be calculated for the kick is the effective kick power ep:

$$ep = \text{kick\_power} \cdot \text{kick\_power\_rate} \tag{4.7}$$

If the ball is not directly in front of the player, the effective kick power will be reduced by a certain amount dependent on the position of the ball with respect to the player. Both angle and distance are important.

If the relative angle of the ball is 0° wrt. the body direction of the player client - i.e. the ball is in front of the player - the effective power will stay as it is. The larger the angle gets, the more the effective power will be reduced. The worst case is if the ball is lying behind the player (angle 180°): the effective power is reduced by 25%.

The second important variable for the effective kick power is the distance from the ball to the player: it is quite obvious that - should the kick be executed - the distance between ball and player is between 0 and player's **kickable margin**. If the distance is 0, the effective kick power will not be reduced again. The further the ball is away from the player client, the more the effective kick power will be reduced. If the ball distance is player's **kickable margin**, the effective kick power will be reduced by 25% of the original kick power.

The overall worst case for kicking the ball is if a player kicks a distant ball behind itself: 50% of kick power will be used. For the effective kick power, we get the formula (4.8). (dir diff means the absolute direction difference between ball and the player's body direction, dist diff means the absolute distance between ball and player.) $0 \leq \text{dir\_diff} \leq 180^\circ \wedge 0 \leq \text{dist\_diff} \leq \text{kickable\_margin}$

$$ep = \text{ep} \cdot (1 - 0.25 \cdot \frac{\text{dir\_diff}}{180^\circ} - 0.25 \cdot \frac{\text{dist\_ball}}{\text{kickable\_margin}}) \tag{4.8}$$

The effective kick power is used to calculate $\vec{a}_{n_i}$, an acceleration vector that will be added to the global ball acceleration $\vec{a}_n$ during cycle n (remember that we have a multi agent system and *each* player close to the ball can kick it during the same cycle).

There is a server parameter, **server::kick_rand**, that can be used to generate some noise to the ball acceleration. For the default players, **kick_rand** is 0.1. For heterogeneous players, **kick_rand** depends on **player::kick_rand_delta_factor** in `player.conf` and on the actual kickable margin. .. In RoboCup 2000, **kick_rand** was used to generate some noise during evaluation round for the normal players.

- **TODO: new kick/tackle noise model. See [12.0.0 pre-20080210] in NEWS**
- **TODO: heterogeneous kick power rate. See [14.0.0] in NEWS**

During the transition from simulation step n to simulation step $n + 1$ acceleration $\vec{a}_n$ is applied:

1. $\vec{a}_n$ is normalized to a maximum length of **server::ball_accel_max**.
2. $\vec{a}_n$ is added to the current ball speed $\vec{v}_n$. $\vec{v}_n$ will be normalized to a maximum length of **server::ball_speed_max**.
3. Noise $\vec{n}$ and wind $\vec{w}$ will be added to $\vec{v}_n$. Both noise and wind are configurable in `server.conf`. The responsible parameter for the noise is **server::ball_rand**. Both direction and length of the noise vector are within the interval $[-|\vec{v}_n| \cdot \text{ball\_rand} \ldots |\vec{v}_n| \cdot \text{ball\_rand}]$‘. Parameters responsible for the wind are **server::wind_force**, **server::wind_dir** and **server::wind_rand**.

4. The new position of the ball $\vec{p}_{n+1}$ is the old position $\vec{p}_n$ plus the velocity vector $\vec{v}_n$ (i.e. the maximum distance difference for the ball between two simulation steps is **server::ball_speed_max**).
5. **server::ball_decay** is applied for the velocity of the ball: $\vec{v}_{n+1} = \vec{v}_n \cdot \text{ball\_decay}$. Acceleration $\vec{a}_{n+1}$ is set to zero.

With the current settings the ball covers a distance up to 50, assuming an optimal kick. 55 cycles after an optimal kick, the distance from the kick off position to the ball is about 48, the remaining velocity is smaller than 0.1. 18 cycles after an optimal kick, the ball covers a distance of 34 - 34 and the remaining veloctity is slightly smaller than 1.

Implications from the kick model and the current parameter settings are that it still might be helpful to use several small kicks for a compound kick – for example stopping the ball, kick it to a more advantageous position within the kickable area and kick it to the desired direction. It would be another possibility to accelerate the ball to maximum speed without putting it to relative position (0,0{textdegree}) using a compound kick.

Table 4.10: Ball and Kick Model Parameters

Default Parameters `server.conf`	Default Value (Range)	Heterogeneous Player Parameters `player.conf`	Value
server::minpower	-100		
server::maxpower	100		
server::minmoment	-180		
server::maxmoment	180		
server::kickable_margin	0.7 ([0.6, 0.8])		-0.1 0.1
		player::kickable_margin_de	
		player::kickable_margin_de	
server::kick_power_rate	0.027		
server::kick_rand	0.1 ([0.0, 0.2])		1 -0.1 0.1
		player::kick_rand_delta_fac	
		player::kickable_margin_de	
		player::kickable_margin_de	
server::ball_size	0.085		
server::ball_decay	0.94		
server::ball_rand	0.05		
server::ball_speed_max	3.0		
server::ball_accel_max	2.7		
server::wind_force	0.0		
server::wind_dir	0.0		
server::wind_rand	0.0		

4.5.4 Move Model

The *move command* can be used to place a player directly onto a desired position on the field. move exists to set up the team and does not work during normal play. It is available at the beginning of each half (play mode `before_kick_off`') and after a goal has been scored (play modes `goal_l_?` or `goal_r_?` '). In these situations, players can be placed on any position in their own half (i.e. X < 0) and can be moved any number of times, as long as the play mode does not change. Players moved to a position on the opponent half will be set to a random position on their own side by the server. A second purpose of the *move command* is to move the goalie within the penalty area after the goalie caught the ball. If the goalie caught the ball, it can move together with the ball within the penalty area. The goalie is allowed to move *goalie_max_moves* times before it kicks the ball. Additional *move commands* do not have any effect and the server will respond with (error too_many_moves).

Table 4.11: Parameter for the move_command

Parameter in `server.conf`	Value
goalie_max_moves	2

4.5.5 Say Model

Using the *say command*, players can broadcast messages to other players. Messages can be say_msg_size characters long, where valid characters for say messages are from the set sth (without the square brackets). Messages players say can be heard within a distance of *audio_cut_dist* by members of both teams . **Say messages** sent to the server will be sent back to players within that distance immediately. The use of the *say command* is only restricted by the limited capacity of the players of hearing messages.

Table 4.12: Parameter for the say command

Parameter in `server.conf`	Value
say_msg_size	10
audio_cut_dist	50
hear_max	1
hear_inc	1
hear_decay	1

4.5.6 Tackle Model

The tackle command is to accelerate the ball towards the player's body(**TODO:new tackle model [12.0.0 pre-20080210]**). Players can kick the ball that can not be kicked with the kick command by executing the tackle command. The success of tackle depends on a random probability related to the position of the ball. It can be obtained by the following formula.

The probability of a tackle failure when the ball is in front of the player is:

$$fail\_prob = (player\_to\_ball.x \div tackle\_dist)^{tackle\_exponent} + (player\_to\_ball.y \div tackle\_width)^{tackle\_exponent}$$

The probability of a tackle failure when the ball is behind the player is:

$$fail\_prob = (player\_to\_ball.x \div tackle\_back\_dist)^{tackle\_exponent} + (player\_to\_ball.y \div tackle\_back\_width)^{tackle\_exponent}$$

The probability of processing success is:

$$tackle\_prob = 1.0 \breve{} fail\_prob$$

In this case, when the ball is in front of the player, it is used to *tackle_dist* (default is 2.0), otherwise it is used to **tackle_back_dist** (default is 0.5); **player_to_ball** is a vector from the player to the ball, relative to the body direction of the player. When the tackle command is successful, it will give the ball an acceleration in its own body direction.

The execution effect of tackle is similar to that of kick, which is obtained by multiplying the parameter **tackle_power_rate** (default is 0.027) with power. It can be expressed by the following formula:

$$effective\_power = power \times tackle\_power\_rate$$

Once the player executes the tackle command, whether successful or not, the player can no longer move within 10 cycles. The following table shows the parameters used in tackle command.

TODO

- [12.0.0 pre-20080210] new kick/tackle noise model
- [12.0.0 pre-20080210] max_back_tackle_power
- [13.0.0] forbid backward tackle
- [14.0.0] increasing tackle noise using server::tackle_rand_factor

Table 4.13: Parameters for the tackle command

Parameter in `server.conf`	Value
tackle_dist	2
tackle_back_dist	0
tackle_width	1.25
tackle_cycles	10
tackle_exponent	6
tackle_power_rate	0.027
max_tackle_power	100
max_back_tackle_power	0
tackle_rand_factor	2

4.5.7 Foul Model

TODO

- [14.0.0] foul model and intentional foul option
- [14.0.0] trade off between foul detect probability and kick power rate
- [15.0.0] improve foul model (red_card_probability)

4.5.8 Turn Model

While *dash* is used to accelerate the player in direction of its body, the *turn command* is used to change the players body direction. The argument for the turn command is the moment; valid values for the moment are between **server::minmoment** and **server::maxmoment**. If the player does not move, the moment is equal to the angle the player will turn. However, there is a concept of inertia that makes it more difficult to turn when you are moving. Specifically, the actual angle the player is turned is as follows:

$$actual\_angle = moment \div (1.0 + inertia\_moment \times player\_speed)$$

server::inertia_moment is a server.conf parameter with default value 5.0. Therefore (with default values), when the player is at speed 1.0, the *maximum effective* turn he can do is ± 30. However, notice that because you can not dash

and turn during the same cycle, the fastest that a player can be going when executing a turn is $player\_speed\_max \times player\_decay$, which means the effective turn for a default player (with default values) is ± 60.

For heterogeneous players, the inertia moment is the default inertia value plus a value between $player\_decay\_delta\_min \times inertia\_moment\_delta\_factor$ and $player\_decay\_delta\_max \times inertia\_moment\_delta\_factor$.

Table 4.14: Turn Model Parameter

Default Parameters `server.conf`	Default Value (Range)	Heterogeneous Player Parameters `player.conf`	Value
Name		Name	
server::minmoment	-180		
server::maxmoment	180		
server::inertia_moment	5.0([2.5, 7.5])		
		player::player_decay_delta_	-0.1 0.1
		player::player_decay_delta_	25
		player::inertia_moment_del	

4.5.9 TurnNeck Model

With *turn_neck*, a player can turn its neck somewhat independently of its body. The angle of the head of the player is the viewing angle of the player. The *turn command* changes the angle of the body of the player while turn_neck changes the neck angle of the player relative to its body. The **minimum** and **maximum** relative angle for the player's neck are given by **server::minneckang** and **server::maxneckang** in server.conf. Remember that the neck angle is relative to the body of the player so if the client issues a *turn command*, the viewing angle changes even if no turn_neck command was issued. Also, *turn_neck commands* can be executed during the same cycle as turn, dash, and *kick commands*. turn_neck is not affected by momentum like turn is. The argument for a *turn_neck command* must be in the range between **server::minneckmoment** and **server::maxneckmoment**.

Table 4.15: Parameter for the turn neck command

Parameter in `server.conf`	Value
minneckang	-90
maxneckang	90
minneckmoment	-180
maxneckmoment	180

4.5.10 Change Focus Model

The focus point is a feature developed in server v.18, which can be used by all versions of players. It represents a position inside a player's view angle, and can be up to 40.0 meters away from the player's position. The focus point affects the visual sensor noise model, with the noise of observed objects increasing as the distance between the focus point and the object increases.

The initial position of the focus point is the player's position. Players can change the position of the focus point by sending a **change_focus** command. This command takes two parameters, *dist_moment* and *dir_moment*, and changes the position of the focus point relative to the player's neck angle.

It is important to note that players are not allowed to move the focus point outside of their view angle. Additionally, if a player changes their view angle to a smaller one, the server will automatically move the focus point back into the player's view angle.

See *Vision Sensor Model* in detail about the vison sensor.

4.5.11 Pointto Model

Players can send commands to point to a spot on the field of the form:

(pointto <DIST> <DIR>)

or

(pointto off)

The first form will cause the arm to point to the spot DIST meters from the player in DIR direction, relative to the player's current face direction. The player will continue to point to the same location on the field independent of an motion or rotation of the player for at least **server::point_to_ban** cycles, and until another `pointto` command is issued or **server::point_to_duration** cycles pass. The second form disables a previous call of pointto.

Table 4.16: Parameter for the pointto command

Parameter in `server.conf`	Value
point_to_ban	5
point_to_duration	20

Version 8+ clients can see where a player is pointing, if the player is pointing, the player is in view and they are close enough to determine their team name. In these cases the player part of the `see` message has the form (without the newline):

(p "<TEAMNAME>" <UNUM>) <DIST> <DIR> <DISTCHG> <DIRCHG>
<BDIR> <HDIR> <POINTDIR>)

or

(p "<TEAMNAME>") <DIST> <DIR> <POINTDIR>)

Where `POINTDIR` is the direction the players are is pointing with random Gaussian (normal)noise added to the actual direction, with a mean of zero and a standard deviation calculated as follows:

sigma = pow(dist / team_too_far_length, 4) * 178.25 + 1.75

This means that sigma is a minimum of 1.75 deg and reaches 180 deg when the player is observing a pointing arm from a distance of team_too_far_length. Since 95% of values in a normal distributionare within two standard deviations, then 95% of the time the noise will be in the range +- 2.5 deg when the player is very close and in the range +- 360.0 deg when the player is team_too_far_length away.

`sense_body` messages for version 8+ clients contain information about the arm actuator. The following has been inserted into the sense_body message, just before the last ')', without the new line:

(arm (movable <MOVABLE>) (expires <EXPIRES>)
(target <DIST> <DIR>) (count <COUNT>))

Where:

- <MOVABLE> is the number of cycles till the arm is movable. 0 indicates the arm is movable now
- <EXPIRES> is the number of cycles till the arm stops pointing. 0 indicates that the arm is no longer pointing,
- <DIST> and <DIR> are the distance and direction of the point the player is pointing to, relative to the players location, orientation and neck angle, accurate to 10cm or 0.1 deg.
- <COUNT> is the number of times the `pointto` command has been successfully executed by the player.

Fullstate messages have both <POINTDIST> and <POINTDIR> included between neck angle and stamina. The players own arm state has the same format as in sense body (see below) and can be found between the count and score part.

Version 8+ coaches (on and offline) can see where a player is pointing to if the player is pointing. The direction the player is pointing comes just after the players neck angle.

4.5.12 Attentionto Model

Version 8 and above players can send `attentionto` commands to focus their attention on a particular player. The command has the form:

(attentionto <TEAM> <UNUM>) | (attentionto off)

Where <`TEAM`> is

`opp` | `our` | `l` | `r` | `left` | `right` | <TEAM_NAME>

and <`UNUM`> is integer identifying a member of the team specified. Players can only focus on one player at a time (each attentionto command overrides the previous) and cannot focus on themselves.

See *Sensor Models* in detail about the aural sensor.

4.6 Heterogeneous Players

With the rcssserver version 7, heterogeneous players were introduced. For heterogeneous players, the server generates **player::player_types** random player types at startup. The player types have different capabilities based on the trade-offs defined in the player.conf file. Both teams of a match use the same player types. Type 0 is the default type and is always the same. If **player::random_seed** is not 0, the fixed set of heterogenous player paramters can be generated based on the given seed value. Table 4.17 shows the differences of heterogeneous players:

When players and coaches connect to the server, they can receive information about the available player types. The online coaches can change player types unlimited times before the first kick off and change player types **player::subs_max** times during other non-*play_on* play modes using the *change_player_type* command (see *Commands*).

The online coach can substitute a same player type within **player::pt_max** times. This restriction also applied to the default player type. This means that all field players have to be changed to the non-default type. In version 16, the goalie is still allowed to be assigned the default type. However, if server::allow_mult_default_type is false and teams use the default player type more than player::pt_max, rcssserver automatically assign the heterogeneous player type to field players just before the playmode is changed to kick-off.

The online coach can substitute a same player type within **player::pt_max** times. This restriction is not applied to the default player type. If player::pt_max is 1, each player type except the default type can be used only once.

Each time a player is substituted by some other player type, its stamina, recovery and effort is reset to the initial (maximum) value of the respective player type.

Table 4.17: The parameter differences of heterogeneous players

Parameter	Description
PlayerSpeedMax	maximum speed
StaminaIncMax	Amount of stamina recovered in one step
PlayerDecay	Player speed decay rate
InertiaMoment	Player inertia force when moving
DashPowerRate	Dash acceleration rate
PlayerSize	Player size
KickableMargin	Kickable area radius
KickRand	The amount of noise added to the kick
ExtraStamina	Extra stamina available when stamina is exhausted
EffortMax	Maximum value of the player's effort amount
EffortMin	The minimum amount of effort for the player
CatchAreaLengthStretch	Streach Length to Catch
KickPowerRate	Kick Power Rate
FoulDetectProbability	Probability that the referee will take the foul
UnumFarLength	If dist less than unum_far_length, then both uniform number and team name are visible
UnumTooFarLength	If dist more than unum_too_far_length, then the uniform number is not visible
TeamFarLength	If dist less than team_far_length, then the team name is visible
TeamTooFarLength	If dist more than team_too_far_length, then the team name is not visible.
PlayerMaxObservation-Length	If dist more than player_max_observation_length, then the player is not visible.
BallVelFarLength	If dist less than ball_vel_far_length, then ball vel is visible
BallVelTooFarLength	If dist more than ball_vel_too_far_length, then ball vel is not visible
BallMaxObservationLength	If dist more than ball_max_observation_length, then the ball is not visible.
FlagChgFarLength	If dist less than flag_chg_far_length, then the flag dist change is sent.
FlagChgTooFarLength	If dist less than flag_chg_too_far_length, then the flag dist change is not sent.
FlagMaxObservationLength	If dist more than flag_max_observation_length, then the flag is not visible.

Heterogeneous player parameters given for each match are different. Therefore, each agent does not necessarily have the parameters needed to implement the tactics. Whatever the situation, you need a way to choose the best combination of heterogeneous players.

Table 4.18: Parameter for substitutions and heterogeneous player types

Parameter in player.conf	Value
player_types	18
subs_max	3
pt_max	1

4.7 Referee Model

The Automated Referee sends messages to the players, so that players know the actual play mode of the game. The rules and the behavior for the automated referee are described in Sec. *Kick-Off*. Players receive the referee messages as hear messages. A player can hear referee messages in every situation independent of the number of messages the player heard from other players.

4.7.1 Play Modes and referee messages

The change of the play mode is announced by the referee. Additionally, there are some referee messages announcing events like a goal or a foul. If you have a look into the server source code, you will notice some additional play modes that are currently not used. Both play modes and referee messages are announced using (referee String), where String is the respective play mode or message string. The play modes are listed in Table 4.19, for the messages see Table 4.20.

Table 4.19: Play Modes

Play Mode	tc	subsequent play mode	comment
before_k	0	kick_	at the beginning of a half
play_			during normal play
time_			End of the game
kick_			announce start of play (after pressing the Kick Off button)
kick_			
free_k			
corner_k			when the ball goes out of play over the goal line, without a goal being scored and having last been touched by a member of the defending team.
goal_		play_	play mode changes once the ball leaves the penalty area
goal_			currently unused
drop_	0	play_	
offside_	30	free_k	An offside player who is closer to the opponent's goal when his teammate hits the ball, both in front of the ball and in front of the last player of the opposing team. The offside rule prevents players from concentrating in front of the opponent's goal, as no player can stand near the opponent's goal and have a chance to score by waiting for the ball, and the possibility of sending long passes close to the opponent's goal is limited. In this way, defenders can distance themselves from their own goal and participate more during the game.
penalt			When the game ends in a draw of 6,000 cycles and overtime, the winner will be determined by penalty kicks.
foul_			Pushing the opposing player
back_			A goalkeeper is not allowed to catch the ball inside his own penalty area if a teammate sends the ball to him. The opposing team will receive an indirect free-kick at the point of touch if the goalkeeper makes the mistake.
free_k			Players are not allowd to kick the ball to themselves after a free kick. If a player does kick the ball to themselves after a free kick, a free kick is awarded to the opposing team at the point that the second kick occurred.
indirect_f			In a direct free kick, the player can shoot the ball directly towards the goal, but an indirect free kick cannot and must pass the ball to a teammate.
illegal_d			

where Side is either the character *l* or *r*, OSide means opponent's side. tc is the time (in number of cycles) until the subsequent play mode will be announced

Table 4.20: Referee Messages

Message	tc	subsequent play mode	comment
goal_*Side*_*n*	50	kick_off_*OSide*	announce the *n* th goal for a team
foul_*Side*	0	free_kick_*OSide*	announce a foul
yellow_card_*Side*_*U	0		announce an yellow card information
red_card_*Side*_*Un	0		announce a red card information
goalie_catch_ball_*Si	0	free_kick_*OSide*	
time_up_without_a_te	0	time_over	sent if there was no opponent until the end of the second half
time_up	0	time_over	sent once the game is over (if the time is second half and the scores for each team are different)
half_time	0	before_kick_off	
time_extended	0	before_kick_off	

where *Side* is either the character *l* or *r*, *OSide* means opponent's side. tc is the time (in number of cycles) until the subsequent play mode will be announced.

4.7.2 Time Referee

TODO

- Judges the game time
- server::half_time
- [12.1.3] server::extra_half_time
- [13.0.0] change a length of overtime

4.7.3 Offside Referee

The offside referee is a module that observes the field, particularly passes, to check whether the offside foul happens. This module determines offside lines every cycle, then specifies several candidates from players which would result in an offside if they receive a pass.

The referee is configurable by some parameters in server.conf file. some useful parameters are explained below.

```
server::use_offside = true  // true: enable, false: disable
```

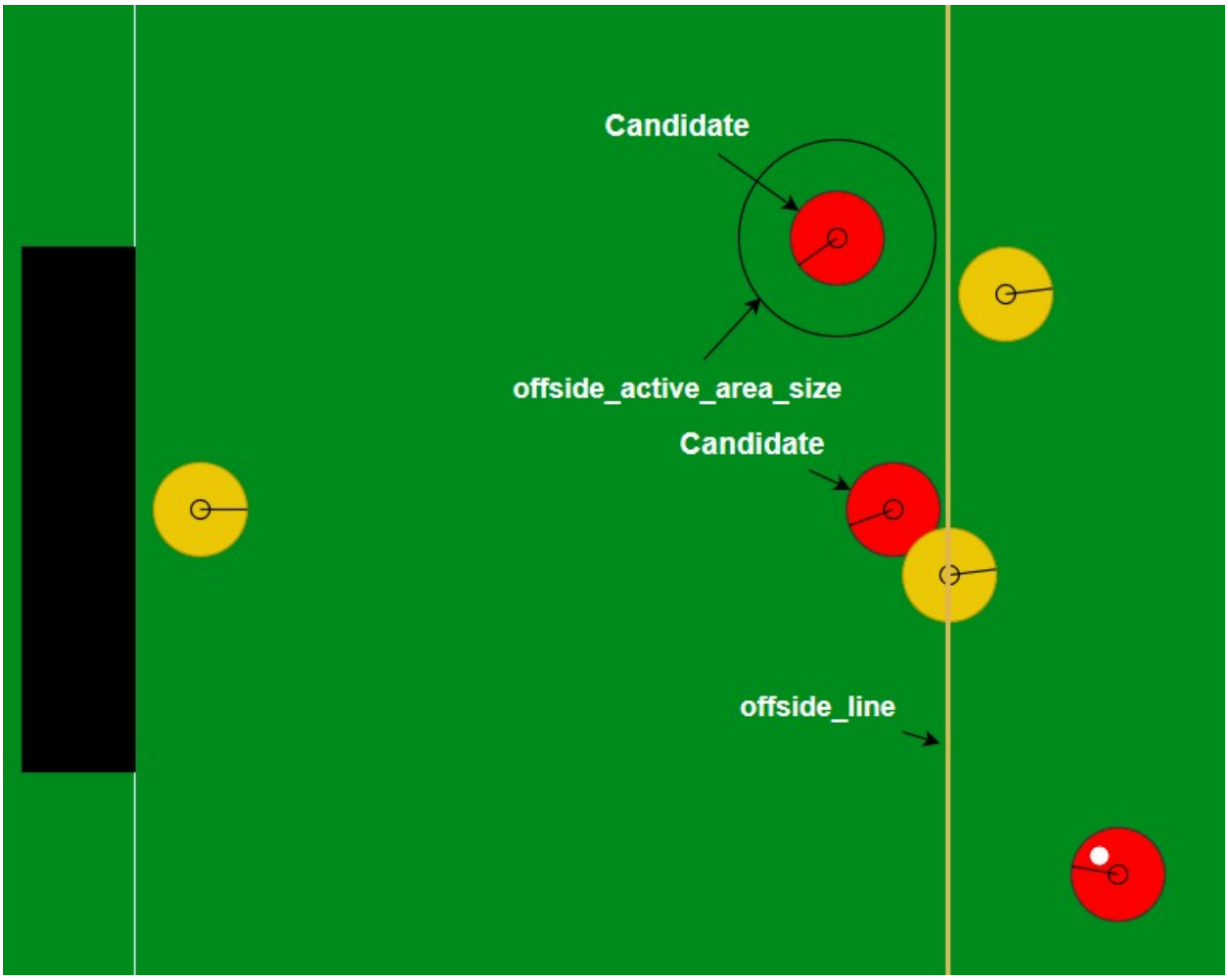
This parameter determines whether the offside referee is enabled or disabled.

```
server::offside_active_area_size = 2.5
```

This parameter determines the radius of an area around a candidate pass receiver. If the ball enters the area and the candidate performs a kick or tackle command, the offside foul is called. Offside is also called if the candidate collides with the ball.

```
offside_kick_margin = 9.15
```

This parameter determines the radius of area that every player in the team which has done offside foul must stay out when the other team wants to free-kick. If there is a player in that area, server moves them out of that.


Candidate
offside_active_area_size
Candidate
offside_line

4.7.4 FreeKick Referee

Free kicks are detected automatically by the soccer server in many relevant cases. The Free kick referee is a module that observes the play mode, to check whether the free kick foul happens and what should teams do. Some methods are explained below.

```
void FreeKickRef::kickTaken
```

This method is executed when foul has occurred by the player. This method checks whether the kick is correctly done or not.

```
void FreeKickRef::tackleTaken
```

This method is executed when a tackle foul has occurred by the player

```
void FreeKickRef::ballTouched
```

This method checks whether the ball has been touched by an unauthorized player.

```
void FreeKickRef::analyse
```

This method checks the game play mode and removes unauthorized players from the foul area due to the situation.

```
void FreeKickRef::playModeChange
```

This method provides the free kick conditions according to the game mode and occurs when the mode has changed.

```
void FreeKickRef::callFreeKickFault
```

This method is for calling the free kick and receives the side and the foul location as inputs.

```
bool FreeKickRef::goalKick
```

If the right or left goal kick has occurred, the output value of this method is true.

```
bool FreeKickRef::freeKick
```

If foul occurs, the output value of this method is true.

```
bool FreeKickRef::ballStopped
```

If the ball stops moving, the output value of this method is true.

```
bool FreeKickRef::tooManyGoalKicks
```

If the value of goal kick count is greater than maxGoalKicks the output value of this method is true.

```
void FreeKickRef::placePlayersForGoalkick
```

This method sends the opponent players out of the penalty area if a goal kick occurs.

4.7.5 Touch Referee

TODO

- Judge the goal
- [14.0.0] golden goal option, server::golden_goal

Checking for goals, out of bounds and within penalty area no complies with FIFA regulations. For a goal to be scored the ball must be totally within the goal - i.e.

$$|ball.x| > FIELD\_LENGTH \cdot 0.5 + ball\_radius$$

Similarly the ball must be completely out of the pitch before it is considered out - i.e

$$|ball.x| > FIELD\_LENGTH \cdot 0.5 + ball\_radius \; ||$$
$$|ball.y| > FIELD\_WIDTH \cdot 0.5 + ball\_radius$$

Lastly the ball is within the penalty area (and thus catchable) if the ball is at least partially within the penalty area - i.e.

$$|ball.y| <= PENALTY\_WIDTH \cdot 0.5 + ball\_radius \; \&\&$$
$$|ball.x| <= FIELD\_LENGTH \cdot 0.5 + ball\_radius \; \&\&$$
$$|ball.x| >= FIELD\_LENGTH \cdot 0.5 - (PENALTY\_LENGTH \cdot 0.5 + ball\_radius)$$

4.7.6 Catch Referee

TODO

- Judges the goalie's catch behavior
- [12.0.0 pre-20071217] change the rules of back pass and catch fault
- [12.0.0 pre-20071217] change the rule of goalies' catch vioration
- [12.1.1] fix the back pass rule

4.7.7 Foul Referee

TODO

- Judges the foul
- [14.0.0] foul model and intentional foul option
- [14.0.0] foul information in sense_body/fullstate
- [14.0.0] red/yellow card message

If an intentional and dangerous foul is detected, the referee penalizes the player and sends the yellow/red card message to clients. The message format is similar to playmode messages. Side and uniform number information of penalized player are appended to the card message:

(referee TIME yellow_card_[lr]_[1-11]) or (referee TIME red_card_[lr]_[1-11])

4.7.8 Ball Stuck Referee

TODO: server::ball_stuck_area. [11.0.0] in NEWS

4.7.9 Illegal Defense Referee

From the server version 16, a new referee module has been added to control the number of defensive players. We have four new variables in **server_param** to change the parameters of this referee.

```
server::illegal_defense_duration = 20
```

This parameter determines the number of cycles that illegal defense situation would have to remain before calling a free kick.

```
server::illegal_defense_number = 0
```

This parameter determines how many players would need to be in the specified zone before the illegal defense situation countdown starts. If the value is set to 0, the referee never detects illegal defense situations.

```
server::illegal_defense_dist_x = 16.5
```

This parameter determines the distance from the field's goal lines for detecting defensive players.

```
server::illegal_defense_width = 40.32
```

This parameter determines the horizontal distance from the horizontal symmetry line for detecting defensive players.

4.7.10 Keepaway Referee

TODO

- [9.1.0] keepaway mode

4.7.11 Penalty Shootouts Referee

TODO

- [9.3.0] penalty shootouts
- [9.4.0] pen_coach_moves_players

Rules

If defensive players exists within the rectangle defined by **illegal_defense_dist_x** and **illegal_defense_width**, they are marked as an illegal state. if the number of markerd players becomes greater than or equal to **illegal_defense_number** and this continues for **illegal_defense_duration** cycles, then play mode will change to **free_kick_[lr]** for the offensive team.

A team is considered as the offensive team when their player is the latest player to kick the ball. If both teams perform a kick on the same cycle, neither team is considered as offensive, and the countdown resets. The above rule is applied to the tackle action too. The change of play mode does not affect cycles of illegal defense situations.

4.8 The Soccer Simulation

In *Movement Models*, we gave a description on how the objects are moved with respect to their accelerations and velocities. In this section, we describe at what point in time acceleration and velocities are applied to the objects during the simulation.

4.8.1 Description of the simulation algorithm

In Soccer Server, time is updated in discrete steps. A simulation step is 100ms. During each simulation step, objects (i.e. players and the ball) stay on their positions. If players decide to act within a step, actions are applied to the players and the ball at the transition from one simulation cycle to the next. Depending on the play mode, not all actions are allowed for the players (for instance in 'before kick off' mode, players can **turn** and **move**, but they cannot **dash**), so only allowed actions will be applied and take effect. If during a step, several players kick the ball, all the kicks are applied to the ball and a resulting acceleration is calculated. If the resulting acceleration is larger than the maximum acceleration for the ball, acceleration is normalized to its maximum value. After moving the objects, the server checks for collisions and updates velocities if a collision occurred (see also Sec. 4.4.2). When applying accelerations and velocities to the objects, the order of application is randomized. After changing objects positions, and updating velocities and accelerations, the automated referee checks the situation and changes the play mode or the object positions, if necessary. Changes to the play mode are announced immediately. Finally, stamina for each player is updated.

In Soccer Server, time is updated in discrete steps. A simulation step is 100ms. During each simulation step, objects (i.e. players and the ball) stay on their positions. If players decide to act within a step, actions are applied to the players and the ball at the transition from one simulation cycle to the next. Depending on the play mode, not all actions are allowed for the players (for instance in `before_kick_off` mode, players can **turn** and **move**, but they cannot **dash**), so only allowed actions will be applied and take effect.

If during a step, several players kick the ball, all the kicks are applied to the ball and a resulting acceleration is calculated. If the resulting acceleration is larger than the maximum acceleration for the ball, acceleration is normalized to its maximum value. After moving the objects, the server checks for collisions and updates velocities if a collision occurred (see also *Collision Model*).

When applying accelerations and velocities to the objects, the order of application is randomized. After changing objects positions, and updating velocities and accelerations, the automated referee checks the situation and changes the play mode or the object positions, if necessary. Changes to the play mode are announced immediately. Finally, stamina for each player is updated.

4.8.2 Keepaway Mode

TODO: [9.1.0] in NEWS

4.9 Using Soccerserver

To start the server either type:

```
./rcssserver
```

from the directory containing the executable or:

```
rcssserver
```

if you installed the executables in your PATH.

4.9.1 Configuration Files

rcssserver will look in your home directory for the configuration files:

- .rcssserver/server.conf
- .rcssserver/player.conf
- .rcssserver/CSVSaver.conf
- .rcssserver-landmark.xml

If .conf files do not exist, they will be created and populated with default values.

You can include additional configuration files by using the `include=FILE` option to `rcssserver`.

TOOD

- [8.01] landmark reader
- [13.0.0] RCSS_CONF_DIR

4.9.2 Recording Command Log

TODO: description about .rcl file

4.9.3 Automatic Mode

TODO: [9.0.2]

4.9.4 Anonyous Mode

Anonymous Mode,which was introduced in server version 16.0.0 allows the server to hide team names from opponents. There are two parameters inside server.conf, which allow each side's name to be set to a fixed string. If the parameter is empty, the real name of the team will be reported to the opponent.

Table 4.21: Server parameters for Anonymous mode

Parameter	Description
server::fixed_teamname_l	Fixed name of the left team, which is sent to the right team. Leave empty for real name.
server::fixed_teamname_r	Fixed name of the left team, which is sent to the right team. Leave empty for real name.

4.9.5 Synchronous Mode

TODO: [7.11] in ChangeLog

4.9.6 Result Saver

TODO

- [9.4.0] StdOutSaver, MySQLSaver
- [9.4.3] CSVSaver

4.9.7 The Soccerserver Parameters

Table 4.22: Parameters adjustable in `server.conf`

Name	Current Value in `server.conf`	Description
version	VERSION	soccer server version
catch_ban_cycle	5	goalies cannot execute the next catch until this cycle has passed after the successful catch.
clang_win_size	300	time window which controls how many messages can be sent (coach language)
clang_advice_win	1	number of advice messages per window
clang_define_win	1	number of define messages per window
clang_del_win	1	number of del messages per window
clang_info_win	1	number of info messages per window
clang_mess_delay	50	delay between receipt of message and send to players
clang_mess_per_cycle	1	maximum number of coach messages sent per cycle
clang_meta_win	1	number of meta messages per window
clang_rule_win	1	number of rule messages per window
clang_win_size	1	The length of clang message window
coach_port	6001	(offline) coach port
connect_wait	300	maximum cycle to wait for client connections in automatic mode
drop_ball_time	100	number of cycles to wait until dropping the ball automatically
extra_half_time	100	length of a half time of extra halves in seconds
foul_cycles	5	idle cycles of foul charged players
freeform_send_period	20	online coaches can send a freeform message during this period after the waiting period
freeform_wait_period	600	online coaches can send a freeform message after waiting this period
game_log_compression	0	compression level of game log file
game_log_version	5	version of game log format
game_over_wait	100	maximum cycle to wait for server termination in automatic mode
goalie_max_moves	2	goalie max. moves after a catch
half_time	300	length of a half time in seconds
hear_decay	1	value that reduces the auditory capacity when receiving an auditory message
hear_inc	1	value that increases the auditory capacity when the game cycle is updated

continues on next page

Table 4.22 – continued from previous page

Name	Current Value in `server.conf`	Description
hear_max	1	maximum value of audiotory capacity
illegal_defense_duration	20	threshold count to detect illegal defense behavior
illegal_defense_number	0	number of players judged to be illegal illegal defense behavior
keepaway_start	-1	**automatic referee changes playmode to play_on after this** seconds elapsed
kick_off_wait	100	maximum cycle to wait kick-off in automatic mode
max_goal_kicks	3	(actually no effect)
max_monitors	-1	max number of monitor connections
nr_extra_halfs	2	number of extra halves in a game
nr_normal_halfs	2	number of normal halves in a game
olcoach_port	6002	online coach port
pen_before_setup_wait	10	max waiting cycles in penalty_miss_[lr] or penalty_score_[lr]
pen_max_extra_kicks	5	max extra kick trials in penalty shootouts
pen_nr_kicks	5	number of normal kick trials in penalty shootouts
pen_ready_wait	10	max waiting cycles in penalty_ready_[lr]
pen_setup_wait	70	max waiting cycles in penalty_setup_[lr]
pen_taken_wait	150	max cycles in penalty_taken_[lr]
point_to_ban	5	players cannot execute the next pointto until this cycle has passed
point_to_duration	20	point to continues automatically for up to this cycle
port	6000	player port number
recv_step	10	time step of acception of commands [unit: msec]
say_coach_cnt_max	128	upper limit of the number of online coach's message
say_coach_msg_size	128	upper limit of length of online coach's message
say_msg_size	10	string size of say message [unit:byte]
send_step	150	time step of visual information [unit:msec]
send_vi_step	100	interval of online coach's look
sense_body_step	100	time step of player's body information [unit:msec]
simulator_step	100	time step of simulation [unit:msec]
slow_down_factor	1	coefficient that slows down simulation time
start_goal_l	0	initial score of the left team
start_goal_r	0	initial score of the right team
synch_micro_sleep	1	sleep time to wait clients in synch mode [unit:msec]
synch_offset	60	offset time from the beginning of the cycle to send *think* message [unit:msec]
synch_see_offset	0	offset time from the beginning of the cycle to send *see* message if players uses *synch_see* mode [unit:msec]

continues on next page

Table 4.22 – continued from previous page

Name	Current Value in `server.conf`	Description
tackle_cycles	10	idle cycles of the players that executed *tackle*
text_log_compression	0	compression level of text log file
auto_mode	false	enable auto start of the match
back_passes	true	enable back pass rule
coach	false	
coach_w_referee	false	allows trainer with automatic referee
forbid_kick_off_offside	true	forbid kick off offside
free_kick_faults	true	enable free kick fault rule
fullstate_l	false	enable full state information for left team
fullstate_r	false	enable full state information for right team
game_log_dated	true	flag to write date in game log name
game_log_fixed	false	enable fixed name in game log
game_logging	true	flag for game logging
golden_goal	false	flag for the golden goal rule
keepaway	false	flag for keepaway mode
keepaway_log_dated	true	flag to write date in keep away log name
keepaway_log_fixed	false	enable fixed name in keep away log
keepaway_logging	true	enable logging in keep away mode
log_times	false	
old_coach_hear	false	
pen_allow_mult_kicks	true	Turn on to allow dribbling in penalty shootouts
pen_coach_moves_players	true	Turn on to have the server automatically position players for peanlty shootouts
pen_random_winner	false	enable random winner in penalties
penalty_shootouts	true	Set to true to enable penalty shootouts after normal time and extra time if the game is drawn.
profile	false	
proper_goal_kicks	false	
record_messages	false	enables recording message to game log file
send_comms	false	enables sending message to monitors
synch_mode	false	enables synchronous mode
team_actuator_noise	false	flag whether to use team specic actuator noise
text_log_dated	true	flag to write date in text log name
text_log_fixed	false	enable fixed name in text log
text_logging	true	flag for recording client command log
use_offside	true	flag for using off side rule
verbose	false	flag for verbose mode
wind_none	false	wind factor is none
wind_random	false	wind factor is random
audio_cut_dist	50.0	audio cut off distance
back_dash_rate	0.6	dash power date for the backward dash
ball_accel_max	2.7	max. ball acceleration
ball_decay	0.94	ball decay
ball_rand	0.05	noise parameter for the ball movement
ball_size	0.085	ball size
ball_speed_max	3.0	max. ball velocity
ball_stuck_area	3.0	threshold of distance to detect a stucked situation

continues on next page

Table 4.22 – continued from previous page

Name	Current Value in `server.conf`	Description
ball_weight	0.2	(not used) weight of the ball
catch_probability	1.0	default goalie catch probability
catchable_area_l	1.2	goalie's defalut catchable area length
catchable_area_w	1.0	goalie's catchable area width
ckick_margin	1.0	corner kick margin
control_radius	2.0	(not used)
dash_angle_step	1.0	minimum angle step for dash command
dash_power_rate	0.006	default dash power rate
effort_dec	0.005	dash effort decrement
effort_dec_thr	0.3	player dash effort decrement threshold
effort_inc	0.01	dash effort increment
effort_inc_thr	0.6	dash effort increment treshold
effort_init	1.0	default effort value
effort_min	0.6	min. player dash effort
extra_stamina	50.0	default extra stamina
foul_detect_probability	0.5	default foul detect probability
foul_exponent	10.0	
goal_width	14.02	goal width
illegal_defense_dist_x	16.5	
illegal_defense_width	40.32	
inertia_moment	5.0	default intertia moment for turn
keepaway_length	20	length of rectangle in keep away mode
keepaway_width	20	width of rectangle in keep away mode
kick_power_rate	0.027	kick power rate
kick_rand	0.1	base parameter for noise added directly to kicks
kick_rand_factor_l	1.0	factor to multiply kick rand for left team
kick_rand_factor_r	1.0	factor to multiply kick rand for right team
kickable_margin	0.7	default kickable margin
max_back_tackle_power	0.0	maximum back tackle power
max_dash_angle	180.0	maximum dash angle relative to player's body angle
max_dash_power	100.0	maximum dash acceleration power
max_tackle_power	100.0	maximum tackle power
maxmoment	180.0	max. moment
maxneckang	90.0	max. neck angle
maxneckmoment	180.0	max. neck moment
maxpower	100.0	max kick power
min_dash_angle	-180.0	minimum dash angle relative to player's body angle
min_dash_power	-100.0	minimum dash acceleration power
minmoment	-180.0	max. moment
minneckang	-90.0	max. neck angle
minneckmoment	-180.0	max. neck moment
minpower	-100	min kick power

continues on next page

Table 4.22 – continued from previous page

Name	Current Value in `server.conf`	Description
offside_active_area_size	2.5	**if offside marked players try to kick/tackle command and their distance from the ball is less than this value, referee detects** offside
offside_kick_margin	9.15	
offside_kick_margin	9.15	
pen_dist_x	42.5	
pen_max_goalie_dist_x	14	
player_accel_max	1.0	max. player acceleration
player_decay	0.4	default player decay
player_rand	0.1	players' movement noise parameter
player_size	0.3	player radius
player_speed_max	1.05	maxium speed of players
player_speed_max_min	0.75	The minumum value of the maximum speed of players
player_weight	60.0	(not used) player weight
prand_factor_l	1	factor to multiply prand for left team
prand_factor_r	1	factor to multiply prand for right team
quantize_step	0.1	quantize step of distance for movable objects
quantize_step_l	0.01	quantize step of distance for landmarks
recover_dec	0.002	player recovery decrement
recover_dec_thr	0.3	player recovery decrement threshold
recover_init	1.0	player's initial recovery value
red_card_probability	0.0	probability of red card in a foul
side_dash_rate	0.4	factor to multiply effective power when side dash is performed
slowness_on_top_for_left_team	1	
slowness_on_top_for_right_team	1	
stamina_capacity	130600	max. recovery capacity of each player's stamina
stamina_inc_max	45.0	default max. player stamina increment
stamina_max	8000.0	max. player stamina
stopped_ball_vel	0.01	threshold value to detect ball is moving or not
tackle_back_dist	0.0	max. x distance between player and ball that player may perform a tackle when ball is behind the player
tackle_dist	2.0	max. x distance between player and ball that player may perform a tackle when ball is in front of the player
tackle_exponent	6.0	exponent used in tackle failure probability equation
tackle_power_rate	0.027	tackle power rate
tackle_rand_factor	2.0	
tackle_width	1.25	max. y distance between player and ball that player may perform a tackle when ball is in front of the player
visible_angle	90.0	visible angle

continues on next page

Table 4.22 – continued from previous page

Name	Current Value in server.conf	Description
visible_distance	3.0	
wind_ang	0.0	
wind_dir	0.0	wind direction
wind_force	0.0	
wind_rand	0.0	
coach_msg_file	''	
fixed_teamname_l	''	fixed name of left team's opponent
fixed_teamname_r	''	fixed name of right team's opponent
game_log_dir	'./'	path to game log directory
game_log_fixed_name	'rcssserver'	fixed name of game log
keepaway_log_dir	'./'	path to keep away log directory
keepaway_log_fixed_name	'rcssserver'	fixed name of keep away log
landmark_file	'~/.rcssserver-landmark.xml	
log_date_format	'%Y%m%d%'	date format in game log
team_l_start	''	path to start script of left team
team_r_start	''	path to start script of right team
text_log_dir	'./'	path to text log directory
text_log_fixed_name	''	fixed name of text log

Table 4.23: Parameters adjustable in player.conf

Name	Current Value in player.conf	Description
version		soccer server version
player_types	18	number of random player types generated at match startup
pt_max	1	number of times that online coach can substitute a player to another player of the same type
random_seed	-1	seed to generate heterogeneous players parameters of a match if it is non zero
subs_max	3	maximum number of substitutions in a match
allow_mult_default_type	false	
catchable_area_l_stretch_max	1.3	defines the upper bound of player's catchable_area_l_stretch
catchable_area_l_stretch_min	1	defines the lower bound of player's catchable_area_l_stretch
dash_power_rate_delta_max	0	defines the upper bound of player's dash power rate when added to default dash power rate
dash_power_rate_delta_min	0	defines the lower bound of player's dash power rate when added to default dash power rate
effort_max_delta_factor	-0.004	controls the upper bound of player's effort amount

continues on next page

Table 4.23 – continued from previous page

Name	Current Value in player.conf	Description
effort_min_delta_factor	-0.004	controls the lower bound of player's effort amount
extra_stamina_delta_max	50	defines the upper bound of player's extra stamina when added to default extra stamina
extra_stamina_delta_min	0	defines the lower bound of player's extra stamina when added to default extra stamina
foul_detect_probability_delta_factor	0	defines the range of heterogeneous player's foul detect probability
inertia_moment_delta_factor	25	factor to control the length of inertia moment delta interval
kick_power_rate_delta_max	0	defines the upper bound of player's kick power rate when added to default kick power rate
kick_power_rate_delta_min	0	defines the lower bound of player's kick power rate when added to default kick power rate
kick_rand_delta_factor	1	
kickable_margin_delta_max	0.1	defines the upper bound of player's kickable margin when added to default kickable margin
kickable_margin_delta_min	-0.1	defines the lower bound of player's kickable margin when added to default kickable margin
new_dash_power_rate_delta_max	0.0008	
new_dash_power_rate_delta_min	-0.0012	
new_stamina_inc_max_delta_factor	-6000	
player_decay_delta_max	0.1	defines the upper bound of inertia moment delta when multiplied by inertia moment delta factor
player_decay_delta_min	-0.1	defines the lower bound of inertia moment delta when multiplied by inertia moment delta factor
player_size_delta_factor	-100	controls the range of heterogeneous player's size
player_speed_max_delta_max	0	defines the upper bound of player's maximum speed when added to server::player_speed_max
player_speed_max_delta_min	0	defines the lower bound of player's maximum speed when added to server::player_speed_max
stamina_inc_max_delta_factor	0	

Table 4.24: Parameters adjustable in CSVSaver.conf

Name	Current Value in CSVSaver.conf	Description
version		soccer server version
save	false	flag to save matches result in a file
filename	'rcssserver.csv'	file to save the results to. If this file does not exist it will be created. Otherwise, the results will be appended to the end.

CHAPTER

FIVE

SOCCER MONITOR

5.1 Introduction

Soccer monitor provides a visual interface. Using the monitor we can watch a game vividly and control the proceeding of the game. .. By cooperating with logplayer, soccermonitor can replay games, so that it .. becomes very convenient to analyze and debug clients.

5.2 Getting started

To connect the soccer monitor with the soccer server, you can use the command following:

```
$ rcssmonitor
```

By specifying the options, you can modify the parameters of soccer monitor instead of modifying monitor configuration file. You can find available options by:

```
$ rcssmonitor --help
```

If you use script **rcsoccersim** to start the server, a monitor will be automatically started and connected with the server:

```
$ rcsoccersim
```

5.2.1 Total number of monitor clients

By default, there is no restriction on the number of monitor clients. You can restrict the number of monitor connections by changing the value of `server::max_monitor` parameter. This feature is useful when you want to reduce the load by limiting arbitrary connections from others.

If the value of `server::max_monitor` is negative integer (default:-1), no restriction. If the value is positive integer, the total number of monitor clients that can connect to the rcssserver is restricted within that number.

Suppose a new monitor client tries to connect to the server after the number of connected monitors has reached the server limit (max_monitor). In that case, the server will refuse the connection and send **(error no_more_monitor)** back to the monitor's client.

5.3 Communication from Server to Monitor

Soccer monitor and rcssserver are connected via UDP/IP on port 6000 (default). When the server is connected with the monitor, it will send information to the monitor every cycle. rcssserver-15 provides four different formats (version 1 ~ 4). The server will decide which format is used according to the initial command sent by the monitor (see *Communication from Monitor to Server*). The detailed data structure information can be found in appendix sec-appendixmonitorstructs.

5.3.1 Version 1

rcssserver sends dispinfo_t structs to the soccer monitor. dispinfo_t contains a union with three different types of information:

- showinfo_t: information needed to draw the scene
- msginfo_t : contains the messages from the players and the referee shown in the bottom windows
- drawinfo_t: information for monitor to draw circles, lines or points (not used by the server)

The size of dispinfo_t is determined by its largest subpart (msg) and is 2052 bytes (the union causes some extra network load and may be changed in future versions). In order to keep compatibility between different platforms, values in dispinfo_t are represented by network byte order. Which information is included is determined by the mode information. NO_INFO indicates no valid info contained (never sent by the server), BLANK_MODE tells the monitor to show a blank screen (used by logplayer) (see rcssserver-*/src/types.h):

```
NO_INFO    0
SHOW_MODE  1
MSG_MODE   2
DRAW_MODE  3
BLANK_MODE 4
```

Following is a description of these structs and the ones contained:

Showinfo

A showinfo_t struct is passed every cycle (100 ms) to the monitor and contains the state and positions of players and the ball:

```
typedef struct {
   char   pmode ;
   team_t team[2] ;
   pos_t  pos[MAX_PLAYER * 2 + 1] ;
   short  time ;
} showinfo_t ;
```

- pmode: currently active playmode of the game (see rcssserver-*/src/types.h):

```
PM_Null,
PM_BeforeKickOff,
PM_TimeOver,
PM_PlayOn,
PM_KickOff_Left,
PM_KickOff_Right,
PM_KickIn_Left,
PM_KickIn_Right,
```

(continues on next page)

(continued from previous page)

```
PM_FreeKick_Left,
PM_FreeKick_Right,
PM_CornerKick_Left,
PM_CornerKick_Right,
PM_GoalKick_Left,
PM_GoalKick_Right,
PM_AfterGoal_Left,
PM_AfterGoal_Right,
PM_Drop_Ball,
PM_OffSide_Left,
PM_OffSide_Right,
PM_PK_Left,
PM_PK_Right,
PM_FirstHalfOver,
PM_Pause,
PM_Human,
PM_Foul_Charge_Left,
PM_Foul_Charge_Right,
PM_Foul_Push_Left,
PM_Foul_Push_Right,
PM_Foul_MultipleAttacker_Left,
PM_Foul_MultipleAttacker_Right,
PM_Foul_BallOut_Left,
PM_Foul_BallOut_Right,
PM_Back_Pass_Left,
PM_Back_Pass_Right,
PM_Free_Kick_Fault_Left,
PM_Free_Kick_Fault_Right,
PM_CatchFault_Left,
PM_CatchFault_Right,
PM_IndFreeKick_Left,
PM_IndFreeKick_Right,
PM_PenaltySetup_Left,
PM_PenaltySetup_Right,
PM_PenaltyReady_Left,
PM_PenaltyReady_Right,
PM_PenaltyTaken_Left,
PM_PenaltyTaken_Right,
PM_PenaltyMiss_Left,
PM_PenaltyMiss_Right,
PM_PenaltyScore_Left,
PM_PenaltyScore_Right,
    PM_Illegal_Defense_Left,
PM_Illegal_Defense_Right,
PM_MAX
```

- team: information about the teams. Index 0 is for team playing from left to right:

```
typedef struct {
  char  name[16];  /* name of the team */
  short score;     /* current score of the team */
} team_t;
```

- pos: position information of ball and players. Index 0 represents the ball, indices 1 to 11 is for team[0] (left to right) and 12 to 22 for team[1]:

```
typedef struct {
  short enable;
  short side;
  short unum;
  short angle;
  short x;
  short y;
} pos_t;
```

- time: current game time.

Values of the elements can be

- enable: state of the object. Players not on the field (and the ball) have state DISABLE. The other bits of enable allow monitors to draw the state and action of a player more detailed (see rcssserver-*/src/types.h):

```
DISABLE         0x00000000
STAND           0x00000001
KICK            0x00000002
KICK_FAULT      0x00000004
GOALIE          0x00000008
CATCH           0x00000010
CATCH_FAULT     0x00000020
BALL_TO_PLAYER  0x00000040
PLAYER_TO_BALL  0x00000080
DISCARD         0x00000100
LOST            0x00000200
BALL_COLLIDE    0x00000400
PLAYER_COLLIDE  0x00000800
TACKLE          0x00001000
TACKLE_FAULT    0x00002000
BACK_PASS       0x00004000
FREE_KICK_FAULT 0x00008000
POST_COLLIDE    0x00010000
FOUL_CHARGED    0x00020000
YELLOW_CARD     0x00040000
RED_CARD        0x00080000
    ILLEGAL_DEFENSE 0x00100000
```

- side: side the player is playing on. LEFT means from left to right, NEUTRAL is the ball (rcssserver-*/src/types.h):

```
LEFT     1
NEUTRAL  0
RIGHT   -1
```

- unum: uniform number of a player ranging from 1 to 11
- angle: angle the agent is facing ranging from -180 to 180 degrees, where -180 is view to the left side of the screen, -90 to the top, 0 to the right and 90 to the bottom.
- x, y: position of the ball or player on the screen. (0, 0) is the midpoint of the field, x increases to the right, y to the bottom of the screen. Values are multiplied by SHOWINFO_SCALE (16) to reduce aliasing, so field size

is PITCH_LENGTH * SHOWINFO_SCALE in x direction and PITCH_WIDTH * SHOWINFO_SCALE in y direction.

Messageinfo

Information containing the messages of players and the referee:

```
typedef struct {
  short board ;
  char  message[2048] ;
} msginfo_t;
```

- board: indicates the type of message. A message with type MSG_BOARD is a message of the referee, LOG_BOARD are messages from and to the players. (rcssserver-*/param.h):

  ```
  MSG_BOARD 1
  LOG_BOARD 2
  ```

- message: zero terminated string containing the message.

Drawinfo

Allows to specify information for the monitor to draw circles, lines or points.

5.3.2 Version 2

rcssserver sends dispinfo_t2 structs to the soccer monitor instead of dispinfo_t structs which is used in version 1. dispinfo_t2 contains a union with five different types of information (the data structures are printed in appendix :ref"*sec-appendixmonitorstructs*:

- showinfo_t2: information needed to draw the scene. It includes all information on coordinates and speed of players and the ball, teamnames, scores, etc.
- msginfo_t : contains the messages from the players and the referee. It also contains information on team's images and information on player exchanges.
- team graphic: The team graphic format requires a 256x64 image to be broken up into 8x8 tiles and has the form:

  ```
  (team_graphic_{l|r} (<X> <Y> "<XPM line>" ... "<XPM line>"))
  ```

  Where X and Y are the co-ordinates of the 8x8 tile in the complete 256x64 image, starting at 0 and ranging upto 31 and 7 respectively. Each XPM line is a line from the 8x8 xpm tile.

- substitutions: substitutions are now explicitly recorded in the message board in the form:

  ```
  (change_player_type {l|r} <unum> <player_type>)
  ```

- player_type_t: information describing different player's abilities and tradeoffs
- server_params_t: parameters and configurations of soccerserver
- player_params_t: parameters of players

Which information is contained in the union is determined by the mode field. NO_INFO indicates no valid info contained (never sent by the server). BLANK_MODE tells the monitor to show a blank screen:

```
NO_INFO      0
SHOW_MODE    1
MSG_MODE     2
BLANK_MODE   4
PT_MODE      7
PARAM_MODE   8
PPARAM_MODE  9
```

5.3.3 Version 3

From the monitor protocol version 3, transferred data are represented by human readable text messages. Each data is represented by S-expression and sent to monitors as one UDP packet. This protocol is also used for recording the game log format version 4. Please note that *PlayMode* and *Score* in the *show* type message are separately recorded in the game log.

Below is a list of data types sent by the version 3 protocol:

- server_param
- player_param
- player_type
- show
- msg

The format of *server_param*, *player_param*, and *player_type* messages are the same as the v8+ format for players and coaches. The *msg* type message may contain *team_graphic* data, as in the version 2 format.

The following table shows the format of other types of messages.

From server to monitor

(show *Time PlayMode Score Ball* *Player*+)
Time ::= simulation cycle of rcssserver
PlayMode ::= (pm *PlayModeID*)
Score ::= (tm *LeftName RightName LeftScore RightScore* [*PenaltyScore*])
PenaltyScore ::= *LeftPenaltyScore LeftPenaltyMiss RightPenaltyScore RightPenaltyMiss*
Ball ::= ((b) *X Y VelX VelY*)
Player ::=
((*Side Unum*) *Type State X Y VelX VelY Body Neck* [*PointX PointY*]
(v *ViewQuality ViewWidth*) (s **Stamina Effort Recovery* [*Capacity*]))
[(f *FocusSide FocusUnum*)]
(c *KickCount DashCount TurnCount CatchCount MoveCount TurnNeckCount ChangeViewCount*)
SayCount TackleCount PointtoCount AttentiontoCount))

(msg *Time Board* "*Message*"+)
Time ::= simulation cycle of rcssserver
Board ::= message board type id
Message ::= message string

5.3.4 Version 4

The version 4 protocol is almost same as the version 3. The information of players' stamina capacity is contained in each player data of the show type message.

5.4 Communication from Monitor to Server

The monitor can send to the server the following commands (in all commands, <*variable*> has to be replaced with proper values):

```
(dispinit) | (dispinit version <version>)
```

sent to the server as first message to register as monitor (opposed to a player, that connects on port 6000 as well) . "(dispinit)" is for information version 1, while "(dispinit version 2)" is for version 2. You can change the version by setting the according monitor parameter. (See *Settings and Parameters*)

```
(dispstart)
```

sent to start (kick off) a game, start the second half or extended time. Ignored, when the game is already running.

```
(dispfoul <x> <y> <side>)
```

sent to indicate a foul situation. x and y are the coordinates of the foul, side is LEFT (1) for a free kick for the left team, NEUTRAL (0) for a drop ball and RIGHT (-1) for a free kick for the right team.

```
(dispdiscard <side> <unum>)
```

sent to show a player the red card (kick him out). side can be LEFT or RIGHT, unum is the number of the player (1 - 11).

```
(dispplayer <side> <unum> <posx> <posy> <ang>)
```

sent to place player at certain position with certain body angle, side can be LEFT (1) or RIGHT (-1), unum is the number of the player(1 - 11). Posx and posy indicate the new position of the player, which will be divided by SHOW-INFO_SCALE. And ang indicate the new angle of a player in degrees. This command is added in the server 7.02.

```
(compression <level>)
```

The server supports compression of communication with its clients and monitors (since version 8.03). A monitor can send the above compression request to the server to start compressed communication. If the server is compiled without ZLib, the server will respond with `(warning compression_unsupported)` else <*level*> is not a number between 0 and 9 inclusive, the server will respond with `(error illegal_command_form)` else the server will respond with `(ok compression <level>)` and all subsequent messages to that client will be compressed at that level, until a new compression command is received. If a compression level above zero is selected, then the monitor is expected to compress its commands to the server. Specifying a level of zero turns off compression completely (default).

TODO: [12.0.0 pre-20071217] accept some coach commands from monitor

5.5 How to record and playback a game

To record games, you can call server with the argument:

```
server::game_logging = true
```

This parameter can be set in `server.conf` file. The logfile is recorded under **server::game_log_dir** directory. The default logfile name contains the datetime and the result of the game. You can use the fixed file name by using **server::game_log_fixed** and **server::game_log_fixed_name**.

```
server::game_log_fixed : true
server::game_log_fixed_name : 'rcssserver'
```

To specify the logfile version, you can call server with the argument:

```
server::game_log_version [1/2/3/4/5]
```

or set the parameter in server.conf file:

```
server::game_log_version : 5
```

You can replay recorded games using logplayer applications. The latest rcssmonitor (version 16 or later) can work as a logplayer. To replay logfiles just call rcssmonitor with the logfile name as argument, and then use the buttons on the window to start, stop, play backward, play stepwise.

5.5.1 Version 1 Protocol

Logfiles of version 1 (server versions up to 4.16) are a stream of consecutive dispinfo_t chunks. Due to the structure of dispinfo_t as a union, a lot of bytes have been wasted leading to impractical logfile sizes. This lead to the introduction of a new logfile format 2.

5.5.2 Version 2 Protocol

Version 2 logfile protocol tries to avoid redundant or unused data for the price of not having uniform data structs. The format is as follows:

- head of the file: the head of the file is used to autodetect the version of the logfile. If there is no head, Unix-version 1 is assumed. 3 chars 'ULG' : indicating that this is a Unix logfile (to distinguish from Windows format)
- char version : version of the logfile format
- body: the rest of the file contains the data in chunks of the following format:
  - short mode: this is the mode part of the dispinfo_t struct (see *Version 1 Protocol* Version 1) SHOW_MODE for showinfo_t information MSG_MODE for msginfo_t information
  - If mode is SHOW_MODE, a showinfo_t struct is following.
  - **If mode is MSG_MODE, next bytes are**
    - short board: containing the board info
    - short length: containing the length of the message (including zero terminator)
    - string msg: length chars containing the message

Other info such as DRAW_MODE and BLANK_MODE are not saved to log files. There is still room for optimization of space. The team names could be part of the head of the file and only stored once. The unum part of a player could be implicitly taken from array indices.

Be aware of, that information chunks in version 2 do not have the same size, so you can not just seek SIZE bytes back in the stream when playing log files backward. You have to read in the whole file at once or (as is done) have at least to save stream positions of the showinfo_t chunks to be able to play log files backward.

In order to keep compatibility between different platforms, values are represented by network byte order.

5.5.3 Version 3 Protocol

The version 3 logfile protocol contains player parameter information for heterogenous players and optimizes space. The format is as follows:

- head of the file: Just like version 2, the file starts with the magic characters 'ULG'.
- char version : version of the logfile format, i.e. 3
- **body: The rest of the file contains shorts that specify which data structures will follow.**
  - **If the short is PM_MODE,**
    - a char specifying the play mode follows. This is only written when the playmode changes.
  - **If the short is TEAM_MODE,**
    - a team_t struct for the left side and
    - a team_t struct for the right side follow. Team data is only written if a new team connects or the score changes.

  - **If the short is SHOW_MODE,**
    * a short_showinfo_t2 struct specifying ball and player positions and states follows.
  - **If the short is MSG_MODE,**
    * a short specifying the message board,
    * a short specifying the length of the message,
    * a string containing the message will follow.
  - **If the short is PARAM_MODE,**
    * a server_params_t struct specifying the current server parameters follows. This is only written once at the beginning of the logfile.
  - **If the short is PPARAM_MODE,**
    * a player_params_t struct specifying the current hetro player parameters. This is only written once at the beginning of the logfile.
  - **If the short is PT_MODE,**
    * a player_type_t struct specifying the parameters of a specific player type follows. This is only written once for each player type at the beginning of the logfile.

Data Conversion:

- Values such as x, y positions are meters multiplied by SHOWINFO_SCALE2.
- Values such as deltax, deltay are meters/cycle multiplied by SHOWINFO_SCALE2.
- Values such as body_angle, head_angle and view_width are in radians multiplied by SHOWINFO_SCALE2.
- Other values such as stamina, effort and recovery have also been multiplied by SHOWINFO_SCALE2.

5.5.4 Version 4 Protocol

The version 4 logfile protocol is a text-based format, that may be readable for humans, adopted in rcssserver version 12 or later. Each line contains one data in S-expression like sensory messages. Its grammar is almost the same as monitor protocol version 3.

- head of the file: Just like older versions, the file starts with the magic characters 'ULG'.
- char version : version of the logfile format, i.e. 4
- new line
- **body: In the rest of the file, one of the following data is recorded on each line:**
  - server_param
  - player_param
  - player_type
  - msg
  - playmode
  - team
  - show

`msg` may contain various string data, such as `team_graphic`, the result of the game, and so on. Starting with the server version 12.1.0, the game result is recorded using *msg* data at the end of the game log. See *Version 3* in detail.

5.5.5 Version 5 Protocol

The version 5 logfile protocol is adopted in rcssserver version 13 or later. Its grammar is almost the same as the version 4 protocol, except adding stamina_capacity information to each player data.

5.5.6 Settings and Parameters

rcssmonitor has various modifiable parameters. You can check available options by calling rcssmonitor with `--help` argument:

```
rcssmonitor --help
```

Several parameters can be modified from `View` menu after invoking rcssmonitor.

Some parameters are recorded in `~/.rcssmonitor.conf`, and rcssmonitor will reuse them in the next execution. Of course, you can directly edit this configuration file.

5.6 Team Graphic

TODO

5.7 What's New

16.0:

- Support illegal defense information.
- Integrate a log player feature.
- Implement a time-shift reply feature.
- Remove a buffering mode.
- Change the default tool kit to Qt5.
- Support CMake.

15.0:

- Support v15 server parameters.

14.1:

- Support an auto reconnection feature.

14.0:

- Reimplement using Qt4.
- Support players' card status.
- Implement a buffering mode.

13.1:

- Support a team_graphic message.

13.0:

- Support the monitor protocl version 4.
- Support a stamina capacity information.

12.1:

- Support pointto information.
- Implement an auto reconnection feature.

12.0:

- Support the monitor protocl version 3.

11.0.2:

- Support the penalty kick scores.

11.0:

- Support 64bits OS.

10.0:

- Ported to OS X.

9.1:

- Support a keepaway field.

8.03:

- The server supports compressed communication to monitors as described in section 5.4
- Player substitution information is added to the message log
- Team graphics information is added to the message log

7.07:

- The logplayer did not send server param, player param, and player type messages. This has been fixed.
- The monitor would crash on some logfiles because stamina max seemed to be set to 0. The monitor will no longer crash this way.

Parameter Name	Used Value	Default	Explanation
host	localhost	Localhost	hostname of soccerserver
port	6000	6000	port number of soccerserver
version	2	1	monitor protocol version
length magnify	6.0	6.0	magnification of size of field
goal width	14.02	7.32	goal width
print log	off	On	flag for display log of communication [on/off]
Log line	6	6	size of log window
Print mark	on	On	flag for display mark on field [on/off]
mark file name	mark.RoboCup.grey.xbm	Mark.xbm	mark on field use file name
ball_file_name	ball-s.xbm	Ball.xbm	ball use file name
player_widget_size	9.0	1.0	size of player widget

continues on next page

Table 5.1 – continued from previous page

Parameter Name	Used Value	Default	Explanation
player_widget_font	5x8	Fixed	font(uniform number) of player widget
Uniform_num_pos_x	2	2	position (X) of player uniform number
Uniform_num_pos_y	8	8	position (Y) of player uniform number
team_l_color	Gold	Gold	Team_L color
team_r_color	Red	Red	Team_R color
goalie_l_color	Green	Green	Team_L Goalie color
goalie_r_color	Purple	Purple	Team_R Goalie color
neck_l_color	Black	Black	Team_L Neck color
neck_r_color	Black	Black	Team_R Neck color
Goalie_neck_l_color	Black	Black	Team_L Goalie Neck color
Goalie_neck_r_color	Black	Black	Team_R Goalie Neck color
status_font	7x14bold	Fixed	status line font [team name and score, time, play_mode]
popup_msg	off	Off	flag for pop up and down "GOAL!!" and "Offside!" [on/off]
Goal_label_width	120	120	pop up and down "GOAL!!" label width
Goal_label_font	-adobe-times bold-r---34---- ----*	Fixed	pop up and down "GOAL!!" label font
Goal_score_width	40	40	pop up and down "GOAL!!" score width
Goal_score_font	-adobe-times bold-r---25---- ----*	Fixed	pop up and down "GOAL!!" score font
Offside_label_width	120	120	pop up and down"Offside!" label width
Offside_label_font	-adobe-times bold-r---34---- ----*	Fixed	pop up and down "Offside!" label font
eval	off	Off	flag for evaluation mode
redraw_player	on	Off	always redraw player (needed for RH 5.2)

7.05:

- For quite some time, the logplayer has occasionally "skipped" so that certain cycles were never displayed by the logplayer. This seems to be caused by the logplayer sending too many UDP packets for the monitor to receive. Therefore, a new parameter has been added to the logplayer 'message delay interval'. After sending that many messages, the logplayer sleeps for 1 microsecond, giving the monitor a chance to catch up. This is not a guaranteed to work, but it seems to help significantly. If you still have a problem with the logplayer/monitor "skipping", try reducing message delay interval from it's default value of 10. Setting message delay interval to a negative number causes there to be no delay.
- The server used to truncate messages received from the players and coach to 128 characters before recording

them in the logfile. This has been fixed.

7.04:

- If a client connects with version > 7.0, all angles sent out by the server are rounded instead of truncated (as they were previously) This makes the error from quantization of angles (i.e. conversion of floats to ints) both uniform throughout the domain and two sided. This change was also made to all values put into the dispinfo t structure for the monitors and logfiles.

7.02:

- A new command has been added to the monitor protocol:

```
(dispplayer side unum posx posy ang)
```

(contributed by Artur Merke) See *Communication from Monitor to Server*.

7.00:

- Included the head angle into the display of the soccermonitor. (source contributed by Ken Nguyen)
- Included visualization effect when the player collided with the ball or the player collided with another player. The monitor displays both cases with a black circle around the player.
- Introduced new monitor protocol version 2. (See 5.5.2 Version 2 and 5.4 Commands From Monitor to Server)
- Introduced new logging protocol version 3. (See 5.5.3 Version 3 Protocol)
- Fixed logging so that the last cycle of a game is logged.

CHAPTER

SIX

SOCCER CLIENT

6.1 Protocols

This section provides a brief overview of the protocol between the Soccer Client and the Soccer Server. More details on these protocols can be found in the Soccer Server section. Note that the init and reconnect commands should be send to the player's UDP port (default: 6000) of the Soccer Server machine, and after the response they should be sent to the port assigned to your player by the server, in a valid format. The server sends the init response from this port (refer to section 1.2.1) . All the commands sent to or received from the server are strings of common character and are in a pair of parenthesis.

6.1.1 Initialization and Reconnection

Every player wanting to connect to the server should introduce himself. This is like a handshake and is done only at the beginning and optionally in the half time when you want to reconnect.

Initialization

Your client should send an init command to the server in the following format

(init *TeamName* [(version *VerNum*)] [(goalie)])

The goalie should include the "(goalie)" in the init command to be allowed by the server to catch the ball or do another special goalie action. Note there can only be one or no goalie in each team. (You are not obliged to use a goalie) The Server welcomes you with a response to your init message in the following format

(init *Side UniformNumber PlayMode*)

Or by an error message (if there is an error, i.e. you have initiated more than two team, more than 11 players in a team or more than one goalie in a team)

(error no_more_team_or_player_or_goalie)

Side is your team's side of play, a character, l(left) or r(right). UniformNumber is the player's uniform number (the players of each team are known by their uniform number). PlayMode is a string representing one of the valid play modes.

If you connect to server with versions 7.00 or higher you will receive additional server parameters, player parameters and player types information (the last two are related to the hetero players feature). For the exact format refer to the appendix.

(server_param *Parameters* ...)

(player_param *Parameters* ...)

(player_type *id Parameters* ...)

Here the hand shaking is finished and your client is known as a valid player.

Reconnection

Reconnection is useful for changing the client program of a player without restarting the game. It can only be done in a non-PlayOn playing mode (e.g. in the half time). For reconnection you should send a reconnect command in the following format

(reconnect *TeamName UniformNumber*)

And you will receive a response in the following format

(reconnect *Side PlayMode*)

Or one of the following errors

(can't reconnect)

if the game is in the PlayOn mode.

(error reconnect)

when no client reconnected due to an error. You may also receive the following error if the team name is invalid **(error no_more_team_or_player_or_goalie)** Here again if you are connecting to the server with version 7.00 or higher you will receive additional server parameters, player parameters and player types information.

Disconnection

Before you disconnect, you can send a bye command to the server. This command will remove the player from the field.

(bye)

There will be no answers from the server.

Version Control

Due to the progressive development of the Soccer Server, new features have been added every year and this resulted in changes and improvements in the protocols to support these features. In order to keep compatibility with the older clients and making it easier to work with (specially for researchers), a system has been implemented for the Protocols Version Control. Every client should tell the server the version of its communication protocol in the **init** command so that the server would be able to send the messages in the proper format. But note that although the communication protocol remains unchanged, the judgment and the simulation rules may change and this will affect the whole game.

6.1.2 Control Commands

During the game each player can send action commands. The server executes the commands at the end of the cycle and simulates the next cycle regarding the received commands and the previous cycles data.

Body Commands

All the playing and movement behaviors of the player are consisted from a few commands known as body commands that are presented briefly below. The results of these commands are a little complicated and depend on many simulation factors. For the details of the execution of each command refer to the Soccer Server Section.

(turn *Moment*)

The Moment is in degrees from 180 to 180. This command will turn the player's body direction Moment degrees relative to the current direction.

(dash *Power*)

This command accelerates the player in the direction of its body (not direction of the current speed). The Power is between **minpower** (used value: 100) and **maxpower** (used value: 100).

(kick *Power Direction*)

Accelerates the ball with the given Power in the given Direction. The direction is relative to the the Direction of the body of the player and the power is again between **minpower** and **maxparam**.

(catch *Direction*)

Goalie special command: Tries to catch the ball in the given Direction relative to its body direction. If the catch is successful the ball will be in the goalie's hand until kicked away.

(move *X Y*)

This command can be executed only before kick off and after a goal. It moves the player to the exact position of X (between 54 and 54) and Y (between 32 and 32) in one simulation cycle. This is useful for before kick off arrangements.

Note that in each simulation cycle, only one of the above five commands can be executed (i.e. if the client sends more than one command in a single cycle, one of them will be executed randomly, usually the one received first)

(turn_neck *Angle*)

This command can be sent (and will be executed) each cycle independently, along with other action commands. The neck will rotate with the given Angle relative to previous Angle. Note that the resulting neck angle will be between **minneckang** (default: 90) and **maxneckang** (default: 90) relative to the player's body direction.

Communication Commands

The only way of communication between two players is broadcasting of messages through the **say** command and hearing through the **hear** sensor.

(say *Message*)

This command broadcasts the Message through the field, and any player near enough (specified with **audio_cut_dist**, default: 50.0 meters), with enough hearing capacity will hear the Message. The message is a string of valid characters.

(ok say)

Command succeeded. In case of error there will be the following response from the Server

(**error illegal_command_form**)

Misc. Commands

Other commands are usually of two forms:

- Data Request Commands

  (sense_body)

  Requests the server to send sense body information. Note the server sends sense body information every cycle if you connect with version 6.00 or higher.

  (score)

  Request the server to send score information. The server's reply will be in this format

  (score *Time OurScore OpponentScore*)

- Mode Change Commands

  (change_view *Width Quality*)

  Changes the view parameters of the player. Width is one of narrow, normal or wide and Quality is one of high or low. The amount and detail of the information returned by the visual sensor depends on the width of the view and the quality. But note that the frequency of sending information also depends on these parameters (e.g. if you change the quality from high to low, the frequency doubles, and the time between two see sensors will be cut to half).

6.1.3 Sensor Information

Sensor information are the messages that are sent to all players regularly (e.g. each cycle or each one and half a cycle). There is no need to send any message to the server to get these information. All the returned information of the sensors have a time label, indication the cycle number of the game when the data have been sent (indicated by Time). This time is very useful.

Visual Sensor

Visual Sensor is the most important sensor, and is a little bit complicated. This sensor returns information about the objects that can be seen from the player's view (i.e. objects that are in the view angle and not very far).

The main format of the information is

(see *Time ObjInfo ObjInfo* ...)

The ObjInfos are of the format below

(*ObjName Distance Direction* [*DistChange DirChange* [*BodyFacingDir HeadFacingDir*]])

or

(*ObjName Direction*)

Note that the amount of information returned for each object depends on its distance. The more distant the object is the less information you get. For more detailed information regarding ObjInfo refer to Appendix.

ObjName is in one of the following formats:

(p [*TeamName* [*Unum*]])

(b)

(f *FlagInfo*)

(g *Side*)

p stands for player, **b** stands for ball, **f** stands for flag and **g** stands for goal. Side is one of **l** for left or **r** for right. For more information on FlagInfo refer to Appendix.

Audio Sensor

Audio sensor returns the messages that can be heard through the field. They may come from the online coach, referee, or other players.

The format is as follows:

(hear *Time Sender Message*)

Sender is one of the followings:

- **self**: when the sender is yourself.
- **referee**: when the sender is the referee of the game.
- **online_coach_l** or **online_coach_r**
- *Direction*: when the sender is a player other than yourself the relative direction of the sender is returned instead.

Body Sensor

Body sensor returns all the states of the player such as remaining stamina, view mode and the speed of the player at the beginning of each cycle:

(sense_body *Time* (view_mode { high | low } { narrow | normal | wide }) (stamina *Stamina Effort*) (speed *Speed Angle*) (head_angle *Angle*) (kick *Count*) (dash *Count*) (turn *Count*) (say *Count*) (turn_neck *Count*) (catch *Count*) (move *Count*) (change_view *Count*))

The last eight parameters are counters of the received commands. Use the counters to keep track of lost or delayed messages.

6.2 How to Create Clients

This section provides a brief description to write a first-step program of soccer client.

6.2.1 Sample Client

The Soccer Server distribution includes a very simple program for soccer clients, called rcssclient. It is under the "src" directory of the distribution, and is automatically compiled when you make the Soccer Server. The rcssclient is not a stand-alone client: It is a simple 'pipe' that redirects commands from its standard input to the server, and information from the server to its standard output. Therefore, nothing happens when users invoke the sampleclient. The users must type-in commands from keyboards, and read the sensor information displayed on the terminal. (Actually it is impossible to read sensor information, because the server sends about 17 sensor informations (see information and sense_body information) per second.) The rcssclient is useful to understand what clients should do, and what the clients will receive from the server.

How to Use rcssclient Here is a typical usage of the sampleclient.

1. Invoke client under sampleclient directory of the Soccer Server.

```
% ./rcssclient -server SERVERHOST
```

Here, SERVERHOST is a hostname on which Soccer Server is running. Then the program awaits user input. If the Soccer Server uses an unusual port, for example 6005, instead of the standard port (6000), the users should use the following form.

```
% ./rcssclient -server SERVERHOST -port 6005
```

2. Type in init command from the keyboard.

   (init MYTEAMNAME (version 7))

   Here MYTEAMNAME is a team name the users want to use. Then a player appears on the field. In the same time, the program starts to output the sensor information sent from the server to the terminal. Here is a typical output

```
(init foo (version 7))
(init r 1 before_kick_off)
(server_param 14.02 5 0.3 0.4 0.1 60 1 1 4000 45 0 0.3 0.5 ...
(player_param 7 3 3 0 0.2 -100 0 0.2 25 0 0.002 -100 0 0.2 ...
(player_type 0 1 45 0.4 5 0.006 0.3 0.7 0 0 1 0.6)
(player_type 1 1.16432 28.5679 0.533438 8.33595 0.00733326 ...
(player_type 2 1.19861 25.1387 0.437196 5.92991 0.00717675 ...
(player_type 3 1.04904 40.0956 0.436023 5.90057 0.00631769 ...
(player_type 4 1.1723 27.7704 0.568306 9.20764 0.00746072 ...
(player_type 5 1.12561 32.4392 0.402203 5.05509 0.00621539 ...
(player_type 6 1.02919 42.0812 0.581564 9.53909 0.00688457 ...
(sense_body 0 (view_mode high normal) (stamina 4000 1) ...
(see 0 ((g r) 61.6 37) ((f r t) 49.4 3) ((f p r t) 37 27) ...
(sense_body 0 (view_mode high normal) (stamina 4000 1) ...
```

   The first line, "(init foo (version 7))", is a report what the client sends to the server. The second line, "(init r 1

   before_kick_off) is a report of the first response from the server. Here, the server tells the client that the assigned player is the right side team (r), its uniform number is 1, and the current playmode is before_kick_off. The next 9 lines are server_param and player_param, which tells various parameters used in the simulation. Finally, the server starts to send the normal sensor informations,

   sense_body and see.
   Because the server sends these sensor information every 100ms or 150ms, the client continues to output the information endlessly.

3. **Type in move command to place the player to the initial position. The player** appears on a bench outside of the field. Users need to move it to its initial position by move command like:

   (move -10 10)

   Then the player moves to the point (-10,10). Because, as mentioned before, the client program outputs sensor information endlessly, users can not see strings they type in. So, they must type-in commands blindly.[1]

4. **Click 'Kick-Off' button on the Soccer Server. Then the game starts. The users** can see that the time data in each sensor information (the first number of see and sense_body information) are increasing.

5. **After then, users can use any normal command, turn, dash, kick and so on. For** example, users can turn the player to the right by typing:

[1] Users can redirect the output to any file or program. For example, you can redirect it to /dev/null to discard the information by invoking "% client SERVERHOST > /dev/null". Then, the users can see the string they type-in.

(turn 90)

The player can dash forward with full power by typing:

(dash 100)

When the player is near enough to the ball, it can kick the ball to the left with power 50 by:

(kick 50 -90)

Note again that because of endless sensor output, users must type-in these commands blindly.

Overall Structure of Sample Client

The structure of the rcssclient is simple. The brief process the client does is as follows:

1. Open a UDP socket and connect to the server port. (init_connection())
2. Enter the read-write loop (message_loop), in which the following two processes are executed in parallel.

- read user's input from the standard input (usually a keyboard) and send it to the server (send_message()).
- receive the sensor information from the server (receive_message()) and output it to the standard output (usually a console).

In order to realize the parallel execution, sampleclient uses the select() function. The function enables to wait for multiple input from sockets and streams in a single process. When select() is called, it waits until one of the sockets and streams gets input data, and tells which sockets or streams got the data. For more details of the usage of select(), please refer to the man page or manual documents.

An important tip in the sampleclient is that the client must change the server's port number when it receives sensor informations from the server. This is because the server assign a new port to a client when it receives an init command. This is done by the following statement in "client.c" (around line 147)

```
printf( "recv %d : ", ntohs(serv_addr.sin_port));
sock->serv_addr.sin_port = serv_addr.sin_port ;
buf[n] = '\0'
```

6.2.2 Simple Clients

In order to develop complete soccer clients, what users must do is to write code of a 'brain' part, which performs the same thing as users do with the sampleclient described in the previous section. In other words, users must write a code to generate command strings to send to the server based on received sensor information.

Of course it is not a simple task (so that many researchers tackle RoboCup as a research issue), and there are various ways to implement it. Simply saying, in order to develop player clients, users need to realize the following functions

[Sensing] To analyze sensor information: As shown in the previous section, the server sends various sensor information in S-expressions. Therefore, a client needs to parse the S-expressions. Then, the client must analyze the information to get a certain internal representation. For example, the client needs to analyze a visual information to estimate player's location and field status, because the visual information only include relative locations of landmarks and moving objects on the field.

[Action Interval] To control interval of sending commands: Because the server accepts a body command (turn, dash and kick) per 100ms, the client needs to wait appropriate interval before sending a command.

[Parallelism] To execute sensor and action processes in parallel: Because the Soccer Server processes sensor information and command asynchronously, clients need to execute a sensor process, which deals with sensor information, and an action process, which controls to send commands, in parallel.

[Planning] To make a plan of play: Using sensor information, the client needs to generate appropriate command sequences of play. Of course, this is the final goal of developing soccer clients!!

6.2.3 Tips

Here we collect tips to develop soccer client programs.

- Debugging is the main problem in developing your own team. So try to find easy debuging methods.
- A nice and simple way to see your program's variables in a condition is to use an **abort()** command or some **asserts** to force the program to core-dump; And debug the core using gbd.
- Log every message received from the server and sent to the server. It is very useful for debugging.
- Using ready to use libraries for socket and parsing problems is useful if you are a beginner.
- Remember to pass the version number to the server in the init command. Although it is optional, the default is 3.00 which usually is not desired.
- Even if the catch probability is 1.00 your catch command may be unsuccessful because of errors in returned sensors about the positions.
- The first serious problem you may encounter is the timing problem. There are many methods to synchronize your client's time with server. One simple methods is to use received sense body information.
- Beware of slow networks. If your timing is not very powerful your client's will behave abnormaly in a crowded or slow network or if they are out of process resources (e.g. you run many clients on one slow machine). In this case they may see older positions and will try to act in these positions and this will result in confusion (e.g. they will turn around themselves)
- The main usage of flags are for the player to find the position of himself in the field. Your very first clients may ignore flags and play with relative system of positions. But you may need a positioning module in the near future. There are many of the in the ready to use libraries.
- The program should check the end of buffer in analyzing sensor information. The sensor information uses S-expressions. But the expression may not be completed when the sensor data is longer than the buffer, so that some closing parentheses are lost. In this case, the program may core-dump if it parses the expression naively.

CHAPTER

SEVEN

COACH

7.1 Introduction

Coaches are privileged clients that provide assistance to the players. There are two kinds of coaches, the online coach and the trainer. The latter is often called 'off-line coach' as well, but for clarity's sake we will use the term 'trainer'.

7.2 Distinction Between Trainer and Online Coach

In general, the trainer can exercise more control over the game and may be used only in the development stage, whereas the online coach may connect to official games. The trainer is useful during development for such tasks as running automated learning or managing games. The online coach is used during games to provide additional advice and information to the players.

While developing player clients, for example when applying machine learning methods to learn skills like dribbling or kicking, it might be useful to create training sessions in an automated way. Therefore, the trainer has the following capabilities:

- It can control the play-mode
- It can broadcast audio messages. Such a message can consist of a command or some information intended for one or more of the player-clients. Its syntax and interpretation are user-defined.
- It can move the players and the ball to any location on the field and set their directions and velocities.
- It can get noise-free information about the movable objects.

For details on these capabilities see Section 7.3.

The online coach is intended to observe the game and provide advice and information to the players. Therefore, its capabilities are somewhat limited:

- It can communicate with the players.
- It can get noise-free information about the movable objects.

To prevent the coach from controlling each client in a centralized way, communication is restricted in several ways as described in Section 7.7. The online coach is a good tool for opponent modelling, game analysis, and giving strategic tips to teammates. Since the coach gets a noise-free, global view of the field and has fewer real-time demands, it is expected that it can spend more time deliberating over strategies. See Section 7.6 for more details about the online coach.

7.3 Trainer

7.3.1 Connecting with and without the Soccerserver Referee

By default, an internal referee module is active within the soccerserver that controls the match (see Section 4.7). If the trainer should have complete control over the match, the soccerserver must be instructed to deactivate the referee module. This means for example, that the play-mode will not change and players will not be moved back to their sides after a goal. The trainer has to react to these events by its own rules.

The soccerserver must be informed at startup-time that a trainer-client will be used. Add the option *-coach* to the command arguments of the soccerserver application when a coach-client is used and the internal referee module of the server must be deactivated. You can also add the line *coach* to the server.conf.

If you want to connect a trainer but let the server referee remain activated, add the option *-coach_w_referee* to the command arguments of the server or add *coach_w_referee* to the server configuration file.

If the server is invoked with one of the trainer modes, it prepares a UDP socket to which the trainer-client can connect. The default port number is 6001. If a different port number is needed the new port can be set by assigning its value to the *coach_port* parameter (see Section B.1).

7.4 Commands

The trainer and the online coach can use the following set of commands. The items are listed in three categories. The first category includes commands that can be used only by the trainer, the second includes commands that can be used also by the online coach with certain restrictions, and the third lists commands that can be used by both trainer and online coach.

7.4.1 Commands that can be used only by the trainer

- **(change_mode PLAY_MODE)**

Change the play-mode to PLAY_MODE. PLAY_MODE must match one of the modesde nedin Section 4.7.1. Note that for most play-mode requests the soccerserver will only change the play-mode. The position of the ball usually remains unchanged, but in some cases players will be moved. E.g. in free-kick and kick-in playmodes they will be moved away from the ball if they stand within a certain radius. When changing to 'before_kick_off' they will be even moved to their own side.

Possible replies by the soccerserver:

- **(ok change_mode)**
  The command succeded.
- **(error illegal_mode)**
  The specified mode was not valid.
- **(error illegal_command_form)**
  The PLAY_MODE argument was omitted
- **(move OBJECT X Y [VDIR [VELx VELy]])**

This command will move OBJECT, which may be a player or the ball (see Section Sensor models for format information),to absolute position(X,Y). If VDIR is specified, it will also change its absolute facing direction to VDIR (this only matters for players). Additionally, if VELx and VELy are specified, the object's velocity will be set accordingly.

The trainer always uses left-hand coordinates.

Possible replies by the soccerserver:

- **(ok move)**
  The command succeded.
- **(error illegal_object_form)**
  The OBJECT specification was not valid.
- **(error illegal_command_form)**
  The position, direction, and/or velocity specification was not valid.
- **(check_ball)**

Ask the soccerserver to check the position of the ball. Four positions are defined:

- **in_field**
  The ball is within the boundaries of the field.
- **(goal_l)**
  The ball is within the area assigned to the goal on the left side of the field.
- **(goal_r)**
  The ball is within the area assigned to the goal on the right side of the field.
- **(out_of_field)**
  The ball is somewhere else.

Note that the states 'goal_l' and 'goal_r' do not necessarily imply that the ball actually crossed the goal line.

Possible replies by the soccerserver:

- **(ok check_ball TIME BALLPOSITION)**
  BALLPOSITION will be one of the states specified above.
- **(start)**

This command starts the server, e.g. sets the play-mode to 'kick_off_l'. This essentially simulates pressing the kick-off button on the monitor.

If the trainer does not send an init command, then the first commands of any type received from the trainer will cause the server to start, e.g. set the play-mode to 'kick_off_l'.

Possible replies by the soccerserver:

- **(ok start)**
  The command succeeded.
- **(recover)**

This command resets players' stamina, recovery, effort and hear capacity to the values at the beginning of the game.

Possible replies by the soccerserver:

- **(ok recover)**
  The command succeeded.
- **(ear MODE)**

It turns on or off the sending of auditory information to the trainer. MODE must be one of **on** and **off**. If **(ear on)** is sent, the server sends *all* auditory information to the trainer. See Table 7.3 for the format. If **(ear off)** is sent, the server stops sending auditory information to the trainer.

Possible replies by the soccerserver:

- **(ok ear on) and (ok ear on)**
  Both replies indicate that the command succeeded.

- **(error illegal_mode)**
  MODE did not match **on** or **off**.
- **(error illegal_command_form)**
  The MODE argument was omitted.

7.4.2 Commands that can be used only by the online coach

- **(init (version VERSION))** for the trainer and
- **(init TEAMNAME (version VERSION))** for the online coach.

These commands tell the server which protocol version should be used to communicate with the trainer or coach. In the case of the online coach TEAMNAME has to be specified to indicate which team the coach belongs to. Note that the coach must connect after at least one player from its team.

The trainer is *not* required to issue an init command. However, it is recommended that the trainer does so. Otherwise, the server will communicate with an older protocol.

It should be mentioned that the default port is 6001 for the trainer and 6002 for the online coach.

Possible replies by the soccerserver:

- **(init ok)**
  The command succeeded in case of the trainer.
- **(init SIDE ok)**
  The command succeeded in case of the online coach. SIDE is either 'l' or 'r'.
- **(say MESSAGE)**

Note that the online coach can use this command with the same syntax, but there are more restrictions. See Section 7.6.2 for details.

This command broadcasts the message MESSAGE to all clients in the case of the trainer and only to teammates in the case of the online coach. For the trainer, the format of MESSAGE is the same as for a player-client. It must be a string whose length is less than *say_coach_msg_size*(see Section B.1) and it must consist of alphanumeric characters and/or the symbols().+/?<>_*

The format in which the players hear these messages can be found in Section 4.3.1.

Possible replies by the soccerserver:

- **(ok say)**
  The command succeeded.
- **(error illegal_command_form)**
  MESSAGE did not match the required format.
- **(change_player_type TEAM_NAME UNUM PLAYER_TYPE)** for the trainer and
- **(change_player_type UNUM PLAYER_TYPE)** for the online coach.

These commands can be used to change the heterogeneous player type (see Section 4.6) of the player with the number UNUM of team TEAM_NAME to the type PLAYER_TYPE. PLAYER_TYPE is a digit between 0 and 6, where 0 denotes the default player type. Note that in the case of the online coach the argument TEAM_NAME is missing, because it can only change player types in its own team.

The trainer does not have to comply with the rule that a maximum of three (specified by *subs_max*) players of each type can be on the field.

See Section 7.6.3 for details about the restrictions as to when and how the online coach may substitute players.

Possible replies by the soccerserver:

- **(warning no_team_found)**
  The team does not exist.
- **(error illegal_command_form)**
  If **change_player_type** is not followed by a string, two integers and a close bracket.
- **(warning no_such_player)**
  If there is no player with that uniform number on that team.
- **(ok change_player_type TEAM UNUM TYPE)**
  The command succeeded.

Additionally, the soccerserver can send the following replies to the online coach:

- **(warning cannot_sub_while_playon)**
  If the play-mode is **'play-on'**.
- **(warning no_subs_left)**
  If the coach has already made its three (specified by *subs_max*) subs for the game.
- **(warning max_of_that_type_on_field)**
  If the player-type is not the default and there are three (specified by *subs_max*) of that type already on the field.
- **(warning cannot_change_goalie)**
  If the coach tries to change the player type of the goalie.

The server responds to the teammates with:

- **(change_player_type UNUM TYPE)**

and opponents (including opponent coach) with:

- **(change_player_type UNUM)**

TODO: team_graphic

7.4.3 Commands that can be used by both trainer and online-coach

- **(look)**

This command provides information about the positions of the following objects on the field:

- The left and right goals.
- The ball.
- All active players.

Note that the trainer and online coach for *both* sides receive left-hand coordinates. That is, the coaches receive information in the global coordinates that the left-hand team uses. In general, the players receive no global information (the one exception being the **move** command), but it is common for teams to localize themselves so that the negative x direction is towards the goal they defend.

Possible replies by the soccerserver:

- **(ok look TIME (OBJ1 OBJDESC1) (OBJ2 OBJDESC2) . . .)**
  OBJj can be any of the objects mentioned above. See Section 4.3 for information about the way the names of those objects are composed. OBJDESCj has the following form:
  - For goals: X Y
  - For the ball: X Y DELTAx DELTAy
  - For players: X Y DELTAx DELTAy BODYANGLE NECKANGLE [POINTING_DIRECTION]

The coordinates are always in left-hand orientation, no matter whether a trainer or online coach is used.

If the trainer/coach should receive visual information periodically, use the **(eye on)** command.

- **(eye MODE)**

MODE must be one of **on** and **off** . If **(eye on)** is sent, the server starts sending **(see_global ...)** information (see Section 7.5) every 100ms (the interval is specified by the *send_vi_step* parameter automatically to the client. If **(eye off)** is sent, the server stops sending visual information automatically. In this case the trainer/coach has to ask actively with **(look)**, if it needs visual information.

Possible replies by the soccerserver:

- **(ok eye on) and (ok eye off)**
  Both replies indicate that the command succeeded.
- **(error illegal_mode)**
  MODE id does not match **on** or **off**.
- **(error illegal_command_form)**
  The MODE argument was omitted.
- **(team_names)**

This command makes the trainer/coach receive information about the names of both teams and which side they are playing on.

Possible replies by the soccerserver:

- **(ok team_names [(team l TEAMNAME1) [(team r TEAMNAME2)]])**
  Depending on whether the teams already connected no, one, or both team name(s) will be supplied. Recall that the first team that connects will be on the left side.

7.4.4 Commands that can be used only by the online-coach

- **(team_graphic (X Y "XPM line" ... "XPM line"))**

The online coach can send teams-graphics as 256x64 XPM to the server. Each **team_graphic**-command sends a 8x8 tile. X and Y are the coordinates of this tile, so they range from 0 to 31 and 0 to 7 respectively. Each XPM line is a line from the 8x8 XPM tile. Monitors that are connected to the server will receive the following message on the message-board after each of the coach's **team_graphic**-commands: **(team_graphic_l|r (X Y "XPM line" ... "XPM line"))**

Possible replies by the soccerserver:

- **(ok team_graphic X Y)**
  For each tile the server sends this string in order to signal its arrival.

7.5 Messages from the Server

Apart from the replies to the commands mentioned above the server also sends some messages to the trainer and online coach. If the clients connect to the server with a version >= 7.0 (using the **init**-command), they will receive the following parameter messages just like player clients:

- **(server_param ...)** once
- **(player_param ...)** once
- **(player_type ...)** once for each player type

See Section 4.2.2 for details on the parameter messages.

If the client chooses to receive visual information in each cycle by sending **(eye on)** it will receive messages in the following format every 100ms *(send_vi_step)*:

`class center`

(*see_global* (OBJ1 OBJDESC1)(OBJ2 OBJDESC2) . . .)

OBJj denotes the name of the object. See Table 4.3 for information about the way the names for those objects are composed. OBJDESCj has the following form:

- For goals: X Y
- For the ball: X Y DELTAx DELTAy
- For players: X Y DELTAx DELTAy BODYANGLE NECKANGLE [POINT-ING_DIRECTION]

The syntax is the same as in the reply to the **(look)** command, so coordinates are always in left-hand orientation.

If the client wants to receive auditory information and sent **(ear on)** to the server, it will receive all auditory information, from both the referees and all of the players. There are two kinds of hear messages:

- **(hear TIME referee MESSAGE)** for all referee messages, such as "play_on" and "free_kick_left". See Section 4.7 for a list of the valid messages from the referee.
- **(hear TIME (p "TEAMNAME" NUM) "MESSAGE")** for all player messages. Note the quotes around the message.

See Section 4.3.1 for more details about the players speaking and listening abilities.

7.6 Online Coach

7.6.1 Introduction

The online coach is a privileged client that can connect to the server in official games. It has the capability of receiving global and noise-free information about the objects on the field. In order to encourage research in this area there are special coach contests since 2001. This way, research groups that do not want to develop a team of player clients can participate in the RoboCup challenge by focusing on the online coach. Additionally, to make it possible to use a single coach with a variety of teams, a standard coach language (CLang) has been developed that can be used to communicate with the players.

See Sections 7.4 and 7.5 for details about the commands that can be used by the online coach and messages that will be sent by the server.

7.6.2 Communication with the players

Before version 7.00, the online coach could say short (128 characters, *say_coach_msg_size*) alphanumeric (plus the symbols().+*/?<>) messages when the play-mode is not 'play_on'. This type of message still exists as a "freeform" message, but there are now other standard message types. Since version 8.05 there are also certain intervals in which freeform messages can be sent even during 'play_on'. Every 600 cycles (specified by *freeform_wait_period*) of 'play_on' the coach can send freeform-messages for 20 cycles (specified by *freeform_send_period*). For example, if the play-mode changes to 'play_on' at cycle 420 and stays in 'play_on' till the end of this example, the coach can send freeform-messages between 1020 and 1040, 1620 and 1640, etc. The coach can send *say_coach_cnt_max* freeform messages per game. The length of these messages has to be less than *say_coach_msg_size*. If the game continues into extended time, the online coaches are given an additional *say_coach_cnt_max* messages to say every additional 6000 cycles (or whatever the normal length of a game is). Allowed messages are cumulative, so if the coach does not use all

its allowed messages, it can use them in the extended time. The server will send **(error said_too_many_messages)** if the coach tries to send messages after it reaches the maximum number.

It should be noted that freeform-messages are not allowed in coach-competition-games, and are only supported by CLang for compatibility reasons.

In the standard coach language, there are three other types of messages: rule-, define-, and delete- messages. To prevent coaches from micro-controlling every single action of the players communication is restricted in the following ways.

Every 300 cycles (specified by *clang_win_size*) the coach can send one of each. Note that the number of allowed messages can be changed by setting the *clang_define_win*, *clang_del_win*, and *clang_rule_win* parameters (see SectionB.1). The messages are heard by the players 50 (specified by *clang_mess_delay*) cycles later. If the play-mode is not 'play_on', one (specified by *clang_mess_per_cycle*) message is sent to the players in each cycle, even if the delay time has not elapsed. Messages that are sent while the play mode is not 'play_on' do not count towards the message number restrictions. For example, if the default values are used the coach can send one message per cycle during breaks that will be heard by the players without delay. The server guarantees that messages of each type will be sent to the players in the same order in which they were received from the coach.

The language grammar developed below does not place restrictions on the length of the messages that can be sent to the server. However, for very practical reasons, any message in the standard language cannot be longer than 8154 characters (this is so the maximum message that should be sent to the player is 8K).

The first version of the coach language (Clang) was developed for server version 7.x. For server version 8.x the language has been extended. Because of this, clients that want to receive messages from their coach have to explicitly advise the server, which version of CLang they support. This is done by sending

- **(clang (ver MIN MAX))**

where MIN and MAX are unsigned integers denoting the earliest and latest supported version of CLang, respectively. Clients that do not send such a message will not receive coach messages. The server can determine the version number of coach messages and will filter out any messages that are not supported by the player. If a message has been filtered out, the players will receive

- **(hear TIME online_coach_left|right (unsupported_clang))**

The coach will receive a message from each player that informs it about the supported versions:

- **(clang (ver (PLAYER_NAME) MIN MAX))**

This means that you have to add the sending of **(clang (ver 7 7))**, if you use version 7 source code of players with newer server versions.

The standard coach language will be described in detail in Section 7.7.

7.6.3 Changing Player Types

Using the **change_player_type**-command (described in Section 7.4) the online coach can change player types unlimited times in **'beforekickoff'** play-mode. Of course, these changes have to comply with the general rules about heterogeneous players (see Section 4.6). After kick-off player types can be changed three (*subs_max*) times during play-modes that are not 'play_on'.

See the description of the **change_player_type**-command in Section 7.4 for details about the possible replies from the server.

Note: A player client will be informed about substitutions that occurred before the client connected by the message **(change_player_type UNUM TYPE)** for substitutions in it own team and **(change_player_type UNUM)** for substitutions in the opponent team.

7.6.4 Team Graphic

TODO

7.7 The Standard Coach Language

7.7.1 General Properties

The standard coach language was developed to enable coaches to work together with teams from different research groups. One of the design goals was to have clear semantics that should prevent misinterpretation from both the players and the coach. The language is based on low-level concepts that can be combined to construct new high-level concepts.

Additionally, coaches can communicate a certain number of freeform messages that may be arbitrary strings to the players during non-*'play_on'*-modes. See Section 7.6.2 for details. Be aware though, that freeform messages probably will not be understood by other teams if you plan to use your coach with other teams.

The language description below is the improved and extended version of the language developed by the community, as it is supported by server version 8.x. While the first version of CLang is still supported by the server, its use is not encouraged. A complete description of this first version can be found in the manual for server version 7. It is hoped that all interested researchers will continue to develop CLang to make it a useful tool for multi-agent research.

Some concepts were derived from Unilang [14] (e.g. definitions and several actions) and SFL[12] (e. g. variables and point arithmetic).

Note that the server itself parses all the coach messages using flex and bison (the GNU replacements for lex and yacc) and constructs a simple representation based on a C++ class hierarchy. Please feel free to use and modify this code from the server to handle the parsing of the coach messages. In particular, look at the *coach_lang** files.

7.7.2 Example Language Utterance

The general idea of CLang is to describe tactics and behaviours as rules which map directives to conditions. Each rule consists of a component that denotes a situation (the *condition*) and a list of *directives* that are applicable if the situation description is true in the given world state. Rules can either be used as advice that tells the player how to act, or as information that, for example, describes how the opponent behaves in certain situations. In CLang rules also have an ID, so that the coach can refer to them later.

A simple rule which advises the player number 5 to pass to his teammate with the number 11 if it has the ball and is in the middle of the field can be defined as follows:

```
(define
    (definerule
        MyRule1
        direc (
        (and
            (bowner our 5)
            (bpos (rec (pt -10 -10) (pt 10 10))))
        (do our 5 (pass 11)))))
```

Each of the primitives will be explained in detail later. For now, it should suffice to get the idea that the rule is assigned the ID "MyRule1" and is defined as a directive (as compared to a model-rule that describes observed behavior). **bowner** determines that player 5 of the coach's team is the ball owner. **bpos** specifies the ball position using a rectangle. Finally,

the directive advises player number 5 to pass to his teammate 11. In CLang lingo **(pass 11)** is an *action* and **(do our5 (pass 11))** is a *directive*.

Rules are off by default. So the coach has to turn them off by sending a message like **(rule (on MyRule1))**

Now the language concepts will be looked at in more detail.

7.7.3 Overview of the Five Message Types

There are five types of coach messages in the standard coach language: Rule, Define, Delete, and Freeform. Their purpose and format will be described in this section, and some examples will be given.

In the following format description elements in capitals denote non-terminal symbols which are defined in section 7.7.7.

Define-message: Define messages are the most complex messages in CLang, because they define and combine the components that the coach wants to share with the players, like conditions, directives, regions, actions, and rules. By defining a component it is assigned an ID which the coach can use to refer to it at later messages.

Conditions: Format for defining a condition: **(definec CLANG_STR CONDITION)**

Example: **(definec "Defense" (bowner opp 0))** This defines the condition in which any player of the opponent team owns the ball.

Actions: Format for defining an action:**(definea CLANG_STR ACTION)**

Example: **(definea "Pass7" (pass 7))**

Directives: Format for defining a directive:**(defined CLANG_STR DIRECTIVE)**

Example: **(defined "Pass10to11" (doour 10 (pass 11)))** This directive denotes player 10 passing to player 11.

Regions: Format for defining a region:**(defined CLANG_STR REGION)**

Example: **(defined "OURHALF" (rec (pt -52.5 -34) (pt 0 34)))** A rectangle which covers the team's own half is defined.

Rules: Format for defining a rule:**(definerule CLANG_VAR model RULE)** or **(definerule CLANG_VAR direc RULE)**

Example: **(definerule Rule1 direc ((playm bko) (do our 7 (pos (pt -20 20)))))** This rule states that player 7 should position itself at the given point before kick-off. See also section 7.7.4 about defining rules.

Rule-message: Rule messages are used to turn previously defined rules on or off. After defining a rule, it is off by default.

Format: **(rule ACTIVATION_LIST)**

Example: **(rule (on rule2) (off rule1))**

Delete-message: The delete message tells a player that a rule will not be used again and can be removed from the memory. This also means that after deleting a rule, its ID should not appear in other nested rule definitions (see section 7.7.4) anymore.

Format: **(delete ID_LIST)**

Examples: **(delete Rule1) (delete (Rule1 Rule2)) (delete all)** Deletes one rule, a list of two rules, or all rules, respectively.

Freeform-message: Free-form messages are arbitrary strings and can be sent according to the aforementioned restrictions in section 7.6.2.

Format: **(freeform "STRING")** Note that STRING must be included in quotes.

7.7.4 Defining Rules

The definition of rules is an important part of CLang, so it will be looked at in more detail in this section. Remember that a rule consists of a condition and a list of directives, which again contain actions.

As stated above the format for defining a rule is **(definerule DEFINE_RULE)** using the following components:

```
<DEFINE_RULE>: <CLANG_VAR> model <RULE>
             | <CLANG_VAR> direc <RULE>
```

```
<RULE>: (<CONDITION> <DIRECTIVE_LIST>)
        | (<CONDITION> <RULE_LIST>)
        | <ID_LIST>
```

Each rule is assigned a name complying with the definition of **CLANG_VAR**. Additionally, rules are in one of two modes, either **model** which states that the rule is a description of observed behavior, or **direc** which states that the rule is a directive to behave in a certain way.

Now, the actual content of a rule can be specified in several ways:

- (CONDITION DIRECTIVE_LIST)

This is the straightforward way. The example in section 7.7.3 complies with this format. The CONDITION denotes a situation, and DIRECTIVE_LIST denotes the appropriate directives. Note that the list can contain directives for one, several, or **all** players, or even several directives for the same player. In the latter case, it is up to the player to decide which directive is to be followed.

- (CONDITION RULE_LIST)

This is a very powerful format for combining rules with larger tactics. Since each rule in RULE_LIST already contains a condition, a definition of this form results in nested rules. It can for example be used to activate several rules simultaneously. Suppose, there are already several rules specifying the home positions of the defenders: pos2a and pos2b for player 2, and pos3a and pos3b for player 3. Now, by using

(definerule defenseformation direc ((bowner our {0}) (pos2a pos3a)))

and

(definerule offenseformation direc ((bowner opp {0}) (pos2b pos3b)))

it can be specified when the rules are supposed to be active (depending on which team owns the ball). For evaluating such definitions, the outer condition is assumed to be distributed into the inner conditions, being combined with logical **and**. E.g. assume that pos2a was specified as

((time > 20) (do our {2} (pos (pt -40 10))))

then the above definition would create

((and (bowner our {0}) (time > 20)) (do our {2} (pos (pt -40 10))))

- ID_LISTS

Similar to the above format, this way several existing rules can be combined. Suppose, there have been defined two rules:

(definerule position2 direc ((true) (home (pt -40 -10))))

(definerule mark2 direc ((bowner opp {10}) (mark 10)))

These can be combined into a behavior for player 2:

(definerule player2 direc (position2 mark2))

7.7.5 Semantics and Syntax Details of the Components

In the following the syntax and semantics of the non-terminal symbols that were used in the format outlined above will be described. Rules have a condition on the left-hand side and a set of actions on the right-hand side. Thus each rule can be thought of as essentially specifying an if-then statement:

```
if CONDITION
then { DIRECTIVE_1 DIRECTIVE_2 ... }
```

In the player's programs, it is easy to represent all the advice given by the coach as a small rule-base. Following the advice would be easy by matching the current world state against the condition, and trying to act on the directives. Note: If more than one condition applies to the current situation and the corresponding directives differ, it is up to the player to choose the directive. Note that the player should also exercise some discretion in following directives. For example, if the only directive that matches is to pass to player 5, but player 5 is well-covered by opponents, the player with the ball may choose to ignore the directive for now.

- Conditions:

  A condition is made from the logical connectives over atomic state description propositions:

  - **(ture)**
    Always true.
  - **(false)**
    Always false.
  - **(ppos TEAM UNUM SET INT INT REGION)**
    The first INT is the MINIMUM and the second is the MAXIMUM At least MINUMUM but no more than MAXIMUM players in UNUM SET from team TEAM are in region REGION. Regions and unum sets are more precisely defined below. TEAM is either "our" or "opp". There is no ambiguity since the coach can only be heard by its own players.
  - **(bpos REGION)**
    The ball is in region REGION.
  - **(bowner TEAM UNUM SET)**
    The ball is controlled by some player in UNUM SET of team TEAM. The ball-owner is the last player that had ball contact (i.e. the ball was in his kickable area), even if the ball left his control after that.
  - **(playm PLAY MODE)**
    The play-mode is PLAY MODE. See Section 7.7.7 for the valid values of PLAY MODE.
  - **(COND COMP)**
    The time, goal-difference, number of own or opponent goals can be compared with constants, using the operators < > <= == != >=. Examples: (time > 20) (2 >= opp goals)
  - **unum CLANG VAR UNUM SET**
    If CLANG VAR is instantiated, it is checked whether the unum denoted by the variable CLANG VAR is in the set UNUM SET. If the variable is still unbound, it is bound to the specific set.

  The logical connectives are:

  - **(and CONDITION_1 CONDITION_2 … CONDITION_n)**
  - **(or CONDITION_1 CONDITION_2 … CONDITION_n)**
  - **(not CONDITION)**

  An example condition: "When opponent player 3 is in region X and controls the ball" would be **(and (ppos opp {3} X) (bowner opp {3}))**

- Directives:

  Directives are lists of actions for individual sets of players and come in two forms:

  - **(do TEAM UNUM SET ACTION LIST)** (affirmative mode: players should take these actions)
  - **(dont TEAM UNUM SET ACTION LIST)** (negative mode: players should avoid taking these actions)

  If the actions in the affirmative mode are mutually exclusive, it is up to the player to decide which one is to be followed. In rules that are in the model mode, directives convey knowledge about the plans/behaviors of the players or their opponents.

- Actions:
  - **(pos REGION)**
    The player should position himself in REGION.
  - **(home REGION)**
    The player's default position should be in REGION. This directive is intended largely to specify formations for the team.
  - **(mark UNUM SET)**
    The player should mark some opponent player in UNUM SET.
  - **(markl REGION)**
    The passing lane from the current ball position to REGION should be marked.
  - **(markl UNUM SET)**
    The passing lane from the current ball position to some opponent player in UNUM SET should be marked.
  - **(oline REGION)**
    The offside-trap line for the player/team should be set at REGION.
  - **(htype TYPE)**
    The player is of heterogeneous type TYPE. The TYPE number is as described in Section 4.6. A value of -1 should clear the player's idea of the heterogeneous type.
  - **(pass REGION)**
    The ball should be passed to some player in REGION.
  - **(pass UNUM SET)**
    The ball should be passed to some player in UNUM SET.
  - **(dribble REGION)**
    The ball should be dribbled to REGION.
  - **(clear REGION)**
    The ball should be cleared from REGION, which means to shoot the ball to a point outside of REGION.
  - **(shoot)**
    The ball should be shot at the goal.
  - **(hold)**
    The player should hold the ball, i. e. stand at his position and keep the ball away from opponents.
  - **(intercept)**
    The player should go to the ball and try to control it.
  - **(tackle UNUM SET)**
    The player should tackle some player in UNUM SET (or the ballowner?).

- Regions:

Any REGION token can be any of the following:

- **a POINT**
  This is defined more precisely below
- **(rec POINT 1 POINT 2)**
  Defines a rectangle with its sides parallel to the pitch lines, respectively.
- **(tri POINT 1 POINT 2 POINT 3)**
  Defines a triangle made up of the given points.
- **(arc POINT RADIUS SMALL RADIUS LARGE ANGLE BEGIN ANGLE SPAN)**
  Defines a donut-arc: the area between two circles co-centered at point POINT, having the given radii, with the arc defined starting at the beginning angle and covering the spanning angle. For example, a circle with a radius r could be defined as "(arc (pt 0 0) 0 r 0 360)", and a U-shaped region could be defined as "(arc (pt 0 0) 5 10 0 180)"
- **(null)**
  The null (empty) region.
- **(reg REG_1 REG_2 ... REG_n)**
  Defines a region made up of the union of the given regions.

A POINT is any of the following:

- **(pt X Y)**
  X and Y are real and in global coordinates. This is the absolute position (X,Y);
- **(pt ball)** The current global position of the ball.
- **(pt TEAM UNUM) The current position of player number UNUM on team**
  TEAM (either 'our' or 'opp'). Remember that UNUM can be a variable.
- **(POINT 1 OP POINT 2)**
  This arithmetically combines two points into a new point. POINT i can be made up of arithmetic operators, resulting in a recursive structure. The operators are defined in a natural way, for example: **(pt** $X_1 Y_1$ **) OP (pt** $X_2 Y_2$ **)** = **(pt** X_1 **OP** X_2 Y_1‘ $**OP**$: $math$: ‘Y_2 **)** where **OP** is one of $+ * /$

The use of these relative points makes it easy to express ideas such as "Move to the ball", "If there are 2 teammates within 10m of the ball", etc. Remember that the online coach receives visual information always in left-hand orientation, no matter which side its team plays on. Yet, when sending messages to a team that plays on the right side, the coach must use right-hand orientation in the messages. Transforming coordinates from left- to right-hand orientation is done by negating them.

- UNUM SETS:

  Unum sets are sets of player numbers. These are sets in the sense that order does not matter and may be changed by the server. If 0 is included anywhere in the set, then the set contains all players 1 - 11. The set can contain variables.

  Format: { :math: *NUM_1 NUM_2 ... NUM_n* }

- Variables:

  Technically, everywhere where UNUM occurs, a variable can be used. Yet, it is important to make sure that the variables are instantiated or ground. The scope is the innermost spanning rule, e.g. in

```
1   (definerule rule1 model
2       (bowner our {0})
3       ((true)             (do our {5} (mark 11)))
```

(continues on next page)

(continued from previous page)

```
4       ((bowner our {X}) (do our {X} (shoot)))
5   )
```

the scope of **X** is the complete line 4. This also shows how variables can be instan- tiated: Only in conditions that have UNUMs as fixed argument (i. e. UNUMs in POINTs do not count as condition UNUMS) a variable may be introduced. Its value is set by checking which unums make the condition true. In the example **X** is instantiated with the uniform number of the ball owner. In a condition like **ppos** it can be necessary to instantiate the variable as a set of unums:

(ppos our {X} 1 11 REGION) In this example **X** has to be instantiated as the set of unums which are in **REGION**. Note that an instantiation as in (ppos our {5} 1 1 (rec (pt ball) (pt our {X}))) is not supported.

7.7.6 Futher Resources

- **The CLang Corpus contains examples of actual CLang messages:**
  http://www-2.cs.cmu.edu/ pfr/soccer/clang corpus.html
- **The Multi-Agent Modeling Special Interest Group (MAMSIG) provides binaries**
  and sources of coachable teams and online coaches:

  http://www.cl-ki.uni-osnabrueck.de/ tsteffen/mamsig
- **The Coach-mailing-list discusses Clang details, competition rules, and coaching**
  methods: http://robocup.biglist.com/coach-l/

7.7.7 Syntax

The complete grammar of the standard coach language:

<MESSAGE> : <FREEFORM_MESS> | <DEFINE_MESS> | <RULE_MESS> | <DEL_MESS>

<RULE_MESS> : (rule <ACTIVATION_LIST>)

<DEL_MESS> : (delete <ID_LIST>)

<DEFINE_MESS> : (define <DEFINE_TOKEN_LIST>)

<FREEFORM_MESS> : (freeform <CLANG_STR>)

<DEFINE_TOKEN_LIST> : <DEFINE_TOKEN_LIST> <DEFINE_TOKEN>
| <DEFINE_TOKEN>

<DEFINE_TOKEN> : (definec <CLANG_STR> <CONDITION>)
| (defined <CLANG_STR> <DIRECTIVE>)
| (definer <CLANG_STR> <REGION>)
| (definea <CLANG_STR> <ACTION>)
| (definerule <DEFINE_RULE>)

<DEFINE_RULE> : <CLANG_VAR> model <RULE>
| <CLANG_VAR> direc <RULE>

<RULE> : (<CONDITION> <DIRECTIVE_LIST>)
| (<CONDITION> <RULE_LIST>)
| <ID_LIST>

<ACTIVATION_LIST> : <ACTIVATION_LIST> <ACTIVATION_ELEMENT>
| <ACTIVATION_ELEMENT>

<ACTIVATION_ELEMENT> : (on|off <ID_LIST>)

<ACTION> : (pos <REGION>)
| (home <REGION>)
| (mark <UNUM_SET>)
| (markl <UNUM_SET>)
| (markl <REGION>)
| (oline <REGION>)
| (htype <INTEGER>)
| <CLANG_STR>
| (pass <REGION>)
| (pass <UNUM_SET>)
| (dribble <REGION>)
| (clear <REGION>)
| (shoot)
| (hold)
| (intercept)
| (tackle <UNUM_SET>)

<CONDITION> : (true)
| (false)
| (ppos <TEAM> <UNUM_SET> <INTEGER> <INTEGER> <REGION>)
| (bpos <REGION>)
| (bowner <TEAM> <UNUM_SET>)
| (playm <PLAY_MODE>)
| (and <CONDITION_LIST>)
| (or <CONDITION_LIST>)
| (not <CONDITION>)
| <CLANG_STR>
| (<COND_COMP>)
| (unum <CLANG_VAR> <UNUM_SET>)
| (unum <CLANG_STR> <UNUM_SET>)

<COND_COMP> : <TIME_COMP>
| <OPP_GOAL_COMP>
| <OUR_GOAL_COMP>
| <GOAL_DIFF_COMP>

<TIME_COMP> : time <COMP> <INTEGER>
| <INTEGER> <COMP> time

<OPP_GOAL_COMP> : opp_goals <COMP> <INTEGER>
| <INTEGER> <COMP> opp_goals

<OUR_GOAL_COMP> : our_goals <COMP> <INTEGER>
| <INTEGER> <COMP> our_goals

<GOAL_DIFF_COMP> : goal_diff <COMP> <INTEGER>
| <INTEGER> <COMP> goal_diff

<COMP> : < | <= | == | != | >= | >

<PLAY_MODE> : bko | time_over | play_on | ko_our | ko_opp
| ki_our | ki_opp | fk_our | fk_opp
| ck_our | ck_opp | gk_opp | gk_our
| gc_our | gc_opp | ag_opp | ag_our

<DIRECTIVE> : (do|dont <TEAM> <UNUM_SET> <ACTION_LIST>)
| <CLANG_STR>

<REGION> : (null)
| (arc <POINT> <REAL> <REAL> <REAL> <REAL>)
| (reg <REGION_LIST>)
| <CLANG_STR>
| <POINT>
| (tri <POINT> <POINT> <POINT>)
| (rec <POINT> <POINT>)

<POINT> : (pt <REAL> <REAL>)
| (pt ball)
| (pt <TEAM> <INTEGER>)
| (pt <TEAM> <CLANG_VAR>)
| (pt <TEAM> <CLANG_STR>)
| (<POINT_ARITH>)

<POINT_ARITH> : <POINT_ARITH> <OP> <POINT_ARITH>
| <POINT>

<OP> : + | - | * | /

<REGION> : <REGION_LIST> <REGION>
| <REGION>

<UNUM_SET> : { <UNUM_LIST> }

<UNUM_LIST> : <UNUM>
| <UNUM_LIST> <UNUM>

<UNUM> : <INTEGER> | <CLANG_VAR> | <CLANG_STR>

<ACTION_LIST> : <ACTION_LIST> <ACTION>
| <ACTION>

<DIRECTIVE_LIST> : <DIRECTIVE_LIST> <DIRECTIVE>
| <DIRECTIVE>

<CONDITION_LIST> : <CONDITION_LIST> <CONDITION>
| <CONDITION>

<RULE_LIST> : <RULE_LIST> <RULE>
| <RULE>

<ID-LIST> : <CLANG_VAR>
| (<ID_LIST2>)
| all

<ID-LIST2> : <ID_LIST2> <CLANG_VAR>
| <CLANG_VAR>

<CLANG_STR> : "[0-9A-Za-z().+-*/?<>_]+"

<CLANG_VAR> : [abe-oqrt-zA-Z_]+[a-zA-Z0-9_]*

Parameter name	Used value	Default value	Explanation
coach_port	6001	6001	The port number the trainer connects to.
say_msg_size	512	256	Maximum length of a freeform message a player, trainer, or coach can say.
say_coach_cnt_ma	128	128	Upper limit of freeform messages an online coach can say
send_vi_step	100	100	Interval of online coach's look.
clang_win_size	100	100	Number of cycles that lie between online coach messages
clang_define_win	1	1	Number of define messages that can be sent in the aforementioned interval.
clang_rule_win	1	1	Number of rule messages that can be sent in the aforementioned interval.
clang_del_win	1	1	Number of delete messages that can be sent in the aforementioned interval.
clang_mess_delay	50	50	Number of cycles messages from the online coach will be delayed.
clang mess per cycle	1	1	Number of messages that will be sent to the players during non-play on modes.

Table 7.1: Trainer Interactions with the Server

From trainer to server	From server to trainer
(init (version VERSION)) VERSION ::= a real number	trainer: (init ok)
(change mode PLAY_MODE) PLAY MODE ::= one of the play-modes	(ok change_mode) (error illegal_mode) (error illegal_command_form)
(move OBJECT X Y [VDIR [DELTA_X DELTA_Y]]) OBJECT ::= One of object names X ::= -52–52 Y ::= -32–32 VDIR ::= -180–180 DELTA_X, DELTA_Y ::= [float]	(ok move) (error illegal_object_form) (error illegal_command_form)
(check_ball)	(ok check_ball TIME BPOS) TIME ::= sim. time of server BPOS ::= in_field | goal SIDE | out of field SIDE ::= l | r
(start) (recover)	(ok start) (ok recover)
(change_player_type TEAM_NAME UNUM PLAYER_TYPE) TEAM_NAME ::= string UNUM ::= 1–11 PLAYER_TYPE ::= 0–6	(warning no_team_found) (error illegal_command_form) (warning no_such_player) (ok change_player_type TEAM UNUM TYPE)
(ear MODE) MODE ::= on | off	(ok ear on) (ok ear off) (error illegal mode) (error illegal_command_form)

Table 7.2: Online Coach Interactions with the Server

From trainer to server	From server to online coach
(init TEAMNAME (version VERSION)) VERSION ::= a real number TEAMNAME ::= string	(init SIDE ok) SIDE ::= l | r
(change_player_type UNUM PLAYER_TYPE) UNUM ::= 1–11 PLAYER TYPE ::= 0–6	(warning no_team_found) (error illegal_command_form) (warning no_such_player) (ok change_player_type TEAM UNUM TYPE) (warning cannot_sub_while_playon) (warning no_subs_left) (warning max_of_that_type_on_field) (warning cannot_change_goalie)

Table 7.3: Server Interactions with Trainer/Coach

From client to server	From server to client
(say MESSAGE) (see Section 7.4.2)	(ok say) (error illegal command form)
(look)	(ok look TIME (OBJ_1 $OBJDESC_1$) (OBJ_2 $OBJDESC_2$)..) OBJ_j ::= object name (see Section *Sensor Models*) $OBJDESC_j$ ::= X Y | X Y $DELTA_x$ $DELTA_y$| X Y $DELTA_x$ $DELTA_y$ BODYANG NECKANG
(eye MODE) MODE ::= on | off	(ok eye on) (ok eye off) (error illegal mode) (error illegal command form)
This message is sent automatically ev- ery send_vi_step milliseconds when the coach/trainer eye is on (see the "eye" commands below).	(see_global TIME (OBJ_1 $OBJDESC_1$) (OBJ_2 $OBJDESC_2$)...)
The trainer must use the 'ear' command to get these messages. The online coach always gets these messages.	(hear TIME referee MESSAGE) (hear TIME (p "TEAMNAME" NUM) "MESSAGE") TIME ::= time message was sent TEAMNAME ::= string NUM ::= 1–11 MESSAGE ::= string
(team_names)	(ok team_names [(team l TEAMNAME1) [(team r TEAMNAME2)]])

CHAPTER

EIGHT

REFERENCES AND FURTER READING

BIBLIOGRAPHY

[AK99] Minoru Asada and Hiroaki Kitano, editors. RoboCup-98: Robot Soccer World Cup II. LNAI 1604. Springer, Berlin, Heidelberg, New York, 1999.

[Burkhard97] Hans-Dieter Burkhard, Markus Hannebauer, and Jan Wendler. AT Humboldt — Development, Practice and Theory. In Hiroaki Kitano, editor, RoboCup-97: Robot Soccer World Cup I, volume 1395 of Lecture Notes in Computer Science, pages 357–372. RoboCup Federation, Springer–Verlag, 1997.

[RoboCup99proc] Silvia Coradeschi, Tucker Balch, Gerhard Kraetzschmar, and Peter Stone, editors. Team Descriptions Simulation League RoboCup'99, Stockholm, Sweden, July 1999.

[JFK61] John F. Kennedy. Urgent National Needs. Congressional Record – House (25 may 1961), 1961.

[PreRoboCup96] Hiroaki Kitano, editor. Proceedings of the IROS-96 Workshop on RoboCup, Osaka, Japan, November 1996.

[RoboCup97] Hiroaki Kitano, editor. RoboCup-97: Robot Soccer World Cup I. Springer Verlag, Berlin, 1998.

[Kitano95IJCAI] Hiroaki Kitano, Minoru Asada, Yasou Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The Robot World Cup Initiative. In Proc. of IJCAI-95 Workshop on Entertainment and AI/Alife, pages 19–24, 1995.

[Kitano97] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The robot world cup initiative. In W. Lewis Johnson and Barbara Hayes-Roth, editors, Proceedings of the First International Conference on Autonomous Agents (Agents '97), pages 340–347, New York, 5–8 1997. ACM Press.

[Lanser97] Stefan Lanser, Christoph Zierl, Olaf Munkelt, and Bernd Radig. MORAL - A Vision-based Object Recognition System for Autonomous Mobile Systems. In 7th International Conference on Computer Analysis of Images and Patterns, Kiel, pages 33–41. Springer–Verlag, September 1997.

[Luke97] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Co-evolving Soccer Softbot Team Coordination with Genetic Programming. In Hiroaki Kitano, editor, Proceedings of the RoboCup97 Workshop at the 15 th International Joint Conference on Artificial Intelligence (IJCAI97), pages 115–118, 1997.

[Mackworth93] Alan Mackworth. On Seeing Robots, chapter 1, pages 1–13. World Scientific Press, 1993.

[Nie01] Andreas G. Nie, Angelika Honemann, Andres Pegam, Collin Rogowski, Leonhard Hennig, Marco Diedrich, Philipp Hugelmeyer, Sean Buttinger, and Timo Steffens. the osnabrueck robocup agents project. Technical report, Institute of Cognitive Science, Osnabrueck, 2001.

[Noda97RoboCup97] Itsuki Noda, Shoji Suzuki, Hitoshi Matsubara, Minoru Asada, and Hiroaki Kitano. Overview of RoboCup-97. In Hiroaki Kitano, editor, RoboCup-97: Robot Soccer World Cup I, pages 20–41. Springer–Verlag, 1997.

[Reis01] Luis Paulo Reis and Nuno Lau. Coach unilang - a standard language for coaching a (robo)soccer team. In Andreas Birk, Silvia Coradeshi, and Satoshi Tadokoro, editors, RoboCup-2001: Robot Soccer World Cup V. Springer, Berlin, 2002..

[RoboCup2000] Peter Stone, Tucker Balch, and Gerhard Kraetszchmar, editors. RoboCup-2000: Robot Soccer World Cup IV, Berlin, 2001. Springer Verlag.

[Dorer99] Klaus Dorer. Motivation, Handlungskontrolle und Zielmanagement in autonomen Agenten. PhD thesis, Albert-Ludwigs-Universität Freiburg, Freiburg, December 1999. (German only).

[Stone98] Peter Stone. Layered Learning in Multi-Agent Systems. PhD thesis, School of Computer Science, Carnegie Mellon University, December 1998.

[Kummeneje01PhL] Johan Kummeneje. RoboCup as a Means to Research, Education, and Dissemination. Ph. Lic. Thesis, March 2001. Department of Computer and Systems Sciences, Stockholm University and the Royal Institute of Technology.

[Heintz00] Fredrik Heintz. RoboSoc a System for Developing RoboCup Agents for Educational Use. Master's thesis, IDA 00/26, Linköping university, Sweden, March 2000.

[Murray99] Jan Murray. My goal is my castle – Die höheren Fähigkeiten eines RoboCup-Agenten am Beispiel des Torwarts. Studienarbeit, Universität Koblenz-Landau,Germany, March 1999. (German only).

[Murray01] Jan Murray. Soccer Agents Think in UML. Diploma thesis, Universität KoblenzLandau, 2001.

[Obst99] Oliver Obst. RoboLog: Eine deduktive Schnittstelle zum RoboCup Soccer Server. Diploma thesis, Universität Koblenz-Landau, February 1999. (German only)

[Buck00] Sebastian Buck and Martin A. Riedmiller. Learning situation dependent successrates of actions in a robocup scenario. In Pacific Rim International Conference on Artificial Intelligence, page 809, 2000.

[Stone00] Peter Stone. Layered Learning in Multiagent Systems: A Winning Approach toRobotic Soccer. MIT Press, 2000.

[Subrahmanian00] Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Ozcan, and Robert Ross. Heterogeneous Agent Systems. MIT Press, Cambridge, Massachusetts, 2000.

[FIFA01] Laws of the games. by FIFA on http://www.fifa.com, 2000. Verified on 12th February 2001.

[Stevens90] W.R. Stevens. UNIX Network Programming. Prentice Hall, 1990.

[CMUnited98] Peter Stone, Manuela Veloso, and Patrick Riley. The CMUnited-98 Champion Simulator Team. In Minoru Asada and Hiroaki Kitano, editors, RoboCup-98: Robot Soccer World Cup II. RoboCup Federation, Springer–Verlag, 1998.

[CMUnited99] Peter Stone, Manuela Veloso, and Patrick Riley. The CMUnited-99 Simulator Team. In Silvia Coradeschi, Tucker Balch, Gerhard Kraetzschmar, and Peter Stone, editors, Team Descriptions Simulation League RoboCup'99, pages 7–11. RoboCup Federation, Linköping University Electronic Press, 1999.

INDEX

C