Robocup 2D Soccer Simulation

Holger Endert, Robert Wetzker, Thomas Karbe
Axel Heßler, Philippe Büttner, Felix Brossmann

DAI-Labor, TU Berlin
Faculty of Electrical Engineering
and Computer Science

The Dainamite Agent Framework

Team-Description

November 23, 2006

Contents

"The ultimate goal of the RoboCup project is By 2050, develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer."

1 Introduction

1.1 Introduction - The Robot World Cup Initiative (RoboCup)

Robocup [10] is described best by stating the goal of the project, which is 'by the Year 2050, develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer'. Developing a team of robots for such a task includes dealing with fields of research like artificial intelligence, multi-agent collaboration, robotics, image recognition and much more. In order to make contributions to some or at least one of the related topics the robocup initiative holds different kinds of competitions, whereas each of them emphasises only parts of the complete robocup problem. That is, there are some competitions for real robots of varying size playing against each other as well as for simulated environments, where software agents are used instead of real robots, ignoring the details of hardware design.

1.1.1 The Robocup Simulated Soccer Environment

When developing a robocup team for simulated soccer, one has to deal with the Robocup Soccer Simulator [1] (short: soccer server). This is a program that can simulate the environment and the players of a soccer game, using some degree of abstraction. A detailed description of the soccer server, its behaviour and protocols can be found in the programs manual [7] or in this master thesis [8]. This subsection gives only a brief overview about the general architecture.

A simulation of a soccer game is usually done by executing several programs distributed in a client-server style. The soccer server is responsible for simulating the environment, the players, the ball, the time and the soccer rules. Since it is a server, it provides an interface for agents and monitors to connect for different purposes via the UDP protocol. Each software-agent connects to the server in order to take control of a single player, e.g. the goalie or a forward. Controlling is done by sending action commands to the server within discrete time-steps (cycles), whereas the server responds with perceptions, that are available to the agent (visual, acustic and sense). The latter are used to orientate and are the basis for further decisions of the agent. A monitor is a program, which is able to display the state of a current or past simulation, and hence can connect to the server as well. It receives the actual game

[1] The actual version is 10.0.7, which is the official tournament release in the year 2006. See at *http://sserver.sourceforge.net* for details.

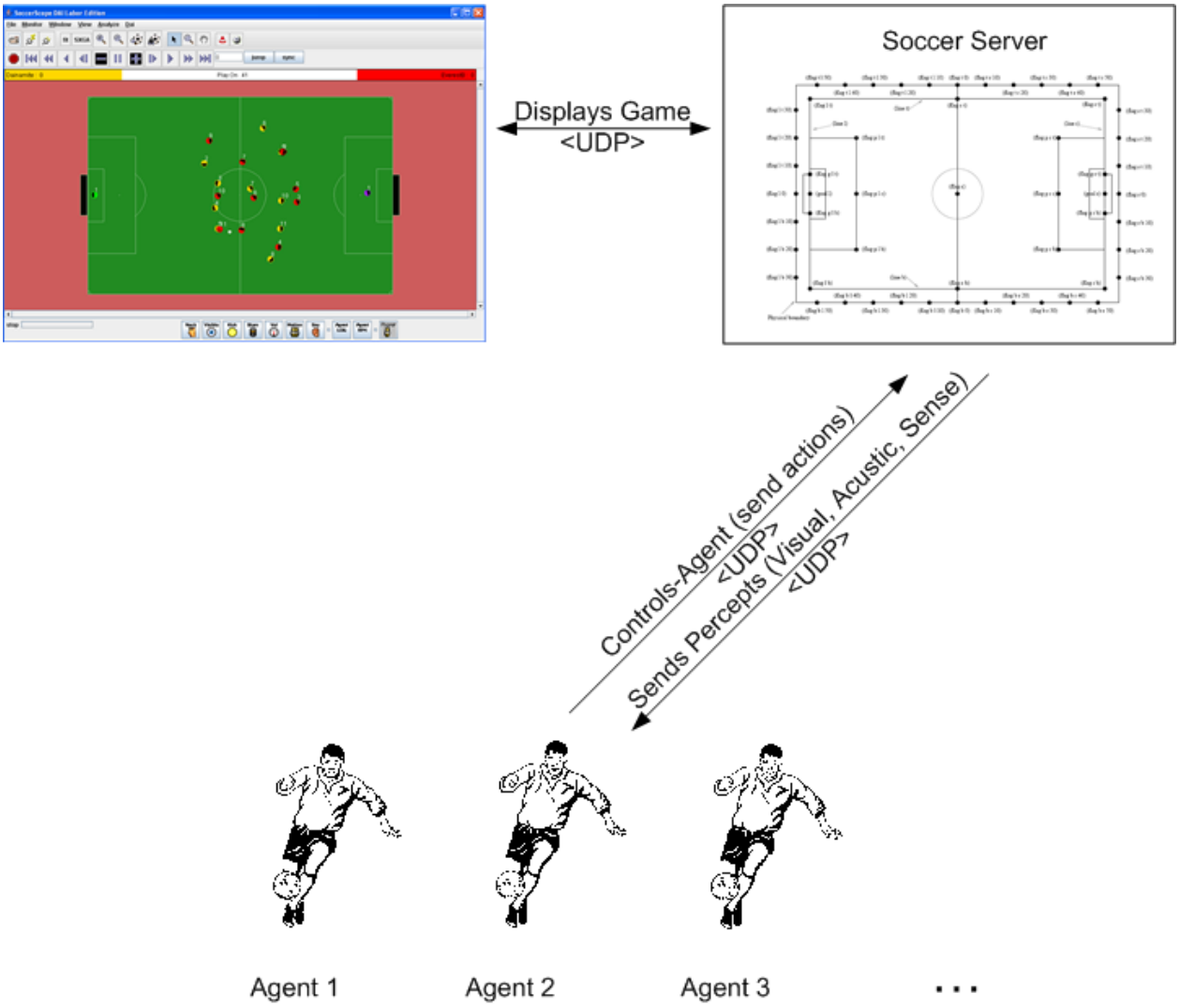


Figure 1.1: Overview about the Client-Server-Based Simulation

state every cycle such that it can update its view. An overview of the general architecture is given in figure 1.1.

1.1.2 Release Notes

The following documentation refers to the agent framework developed at the DAI-Labor [2], which participated on the robocup in the year 2006 in Bremen and placed ninth out of sixteen teams. The team is called *Dainamite*, and as far as the author knows, it was the first team, which was written in Java, that qualified for and participated in an official tournament. In its first game, this team was able to beat the champion of the previous year (*Brainstormers*), and won the group ranking of the first round. However, the second round was disillusion-

[2] www.dai-labor.de

ing, and we missed the quarter finals by a narrow margin. The given release includes the complete framework as well as the most basic skills like passing and ball-interception, so that new developers can begin immediately extending tactic and specialized higher-level tasks, e.g. machine learning. Other tactic relevant skills like attacking and the back-four were removed to enforce new developers to think about other solutions. With this release we hope to open the Robocup-World for other Java Developers and especially for educational institutions. Therefore, this documentation and all code contained in the dainamite framework is published under the terms of the GNU GPL (GNU General Public License) [3]. Feel free to use, extend and redistribute all parts as granted in [2]. We do not provide any warranty for the code or its functionality. We also have to note, that the contained libraries are property of the corresponding owning institution and or developers, and we therefore are not responsible for their correct functionality. A listing of these libraries is given in 2.2.

1.1.3 Structure of this Document

This document is organized as follows: The next chapter gives an overview about the agent classes like the RobocupAgent itself, or the Coach. After that, the building blocks of the agents are presented, each in a separate chapter. These are the parser, the synchronization, the world-model and the particle filter, the prophet, the action plus neckrotator and message-factory and the tactic. Then, there are some issues concerning the coach, the goalie and the Monitor, whereas the latter is an extension of SoccerScope2003 [11], developed by the YowAI Team [4]. Finally, this document closes with a general assessment of this framework and gives rise to some extensions and improvements that can be done.

[3] http://www.fsf.org/licensing/licenses/gpl.txt

[4] Special thanks to Shuhei Shiota for allowing us to release our SoccerScope variant.

2 Getting Started

2.1 Introduction

This chapter deals with starting the team and explains some of the related basic mechanisms like the configutation files. The source-code release already contains some basic skills, which provides the means for a simple kick-and-run tactic, and these can and should be tested to get a feeling for what is going on in Robocup. The development of *Dainamite* was primarily done using Eclipse [1], a very powerful Java IDE, and therefore its usage is explained as well. On the other hand, starting the team independant from eclipse saves resources such as memory and cpu-time, what may be necessary on older machines in order to obtain better simulations. Hence, starting the team from command-line is explained too.

2.1.1 Requirements

The first requirement for executing simulations is an executable version of the soccer server. This can be downloaded from http://sserver.sourceforge.net/. Each version above 9.4.5 should be compatible to our team. The version used in Bremen was 10.0.7 with 10.0.11 as base-version. A description of how to compile and run the soccer server can be found at http://www.cs.utexas.edu/ AustinVilla/sim/keepaway/tutorial.html. For training scenarios described in section 12.3, an adapted version of the server has to be used, which is also available on our website. The one and only real requirement for the team itself is Java in version 1.5 or higher. You can get it from http://java.sun.com/j2se/1.5.0/download.jsp. If you want to use Eclipse, you have to use at least the version 3.1 for Java 1.5 compatibilty, available at http://www.eclipse.org/downloads/.

2.2 Project Structure

The Dainamite Framework was created as an Eclipse Project, and therefore a project structure was used to organize the contained resources. The project is distributed as zip-file, which after unpacking has the structure as presented next. All resources are located in a project folder called *Robocup*. In this, the source code, configuration files, libraries, scripts and release notes are contained. The source code is splitted into two parts, one contains the code for the team, and the other the extended SoccerScope version. The former is located in the

robocup module, whereas the source code root is in *robocup/src/main/java*[1]. The SoccerScope code is located in a similar module (called *soccerscope*), and the source root is therefore in *soccerscope/src/main/java*. Additionally, there are configuration files for the agents, which have to be passed to them on startup. These are located in the sub-folder *etc/agents*. Also located in the *etc/* folder are configuration files for database functionality of the monitor (*database.config* and *robocup.sql*), and start- and build-scripts for the team (*etc/jar/*). Finally, the framework requires some extra Java libraries, which are located in the *lib/* sub-directory of the project. Currently, these libraries are required:

- ssj.jar - a library for stochastic calculations [5].
- jcommon-1.0.0-pre2.jar - dependency for jfreechart (GNU LGPL).
- jfreechart-0.9.4.jar - a library for creating charts with Java [9] (GNU LGPL).
- xpp3_min-1.1.3.4.O.jar - dependency for xstream (used for Training-Scenario creation) (BSD Free Software License, copyright 2003-2006 by Joe Walnes).
- xstream-1.1.2.jar - a library for XML serialization of Java Objects (used for Training-Scenario serialisation), (BSD Free Software License, copyright 2003-2006 by Joe Walnes) [6].
- mysql-connector-java-3.0.10-stable-bin.jar - a library for adding database connectivity to the monitor (published under GPL by MySQL [4])

2.3 Starting the Team From Command-Line

In order to start the team from command-line, the project contains start-scripts for Windows (bat-files) and Unix (shell-scripts). There are different ways to start the team, e.g. out of a created jar-file or directly from classes. It is also possible to start the team on more than one machine, which is sometimes required for robocup competitions. In this years competition, we had to start the complete team on one machine, but we had to use 12 JVM instances, one for each agent, in order to forbid inter-process communication between them. For testing and usage with the monitor, we recommend to start the complete team in a single JVM, hence the given start-scripts are fitted to that use. Simply adapt the files *StartTeamIn1JVM.** on the */etc/jar/* subdirectory. You probably have to change the JAVA_HOME envrionment variable, the servers hostname and the teamname. There exists also a manifest file for creating a jar archive (*manifest*). Only starting the team requires only one library (ssj.jar). For using the monitor with charts, database and training scenarios, additional jars have to be added (directly after the given lib, leaving a space between them).

[1] This corresponds to a Maven2 module.

2.4 Embedding Dainamite in Eclipse

The subsequent sections detail how to setup the working environment in order to start developing and testing on the *Dainamite Agent Team*. Generally, there are more than one way to do that - every Java based Development Tool will work, but this description refers to the recommended way using Eclipse as IDE and/or Maven as Build-System. If the developer is trained on other IDE's (e.g. Netbeans), he might want to use them instead. The most basic steps for including the project into a workspace might be similar there.

Include Project into the Workspace:

- Download the project (zip) and save it to your local file system.
- Open Eclipse, and import the project via *File* → *Import* → *Existing Projects into Workspace*.
- Click *Next*, choose *Select from Archive File* and browse the project.
- Click *Finish*, and the project will be extracted and copied into your workspace.

Set properties:

- Select the *Robocup* Project
- Go to Eclipse menue → *Project* → *Properties*

  A Window as seen in figure 2.1 should appear.

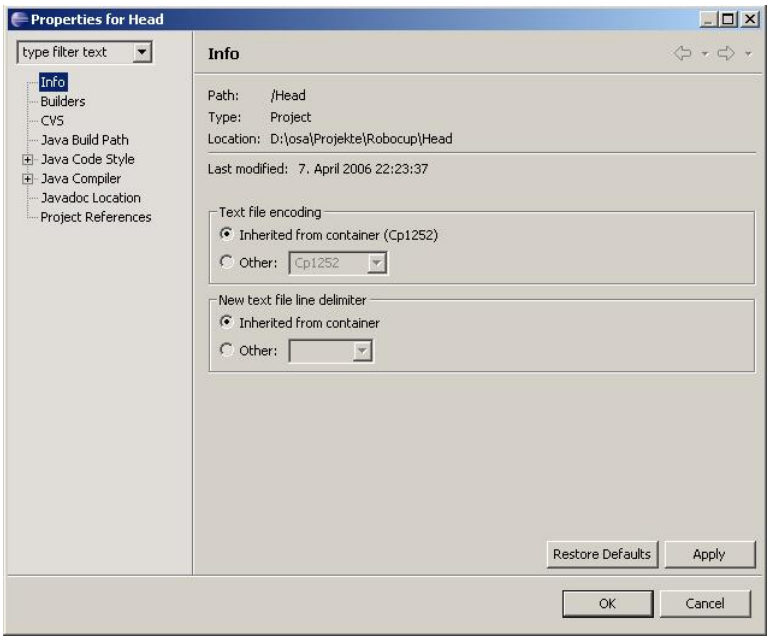


Figure 2.1: Properties

- Select on the left tree *Java Build Path*
- Click on *Source-Tab* → *Add Folder...*
- Select [Project Name]/robocup/src/main/java and [Project Name]/soccersope/src/main/java
- Click *Ok* and *Apply*
  This should now look like figure 2.2.

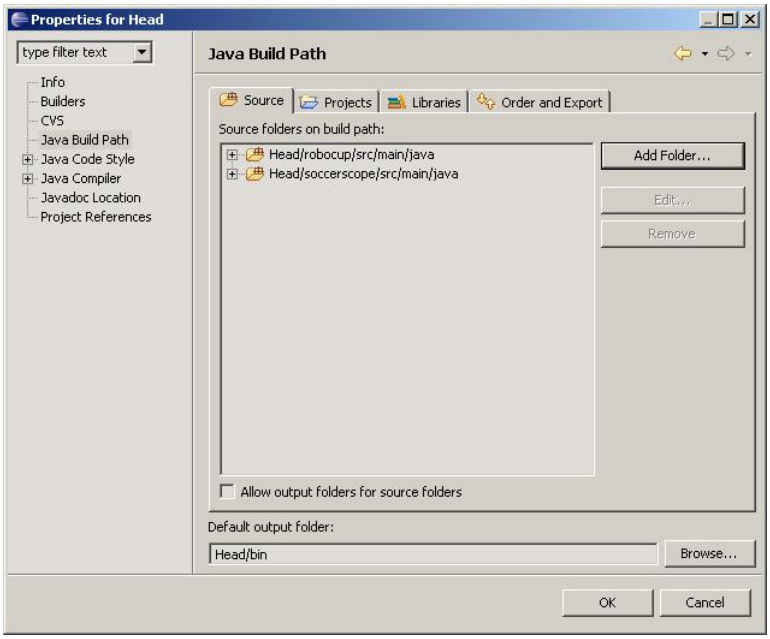


Figure 2.2: Properties

- Now switch to *Libraries-Tab* → *Add JARs...*
- Open the *Robocup/lib* directory and select all Jar-Files and apply.
  This should now look like figure 2.3

Set Run-dialog:

- Close the Properties dialog and click in eclipse menue *Run* → *Run...*
  A Window named *Run* should appear.
- Click on *Java Application*
- Now create a new Application by clicking the *New* button
- The Main-Tab should contain following values:
  Project: `Robocup or the name you called it`
  Main class: `robocup.component.RobocupAgent`

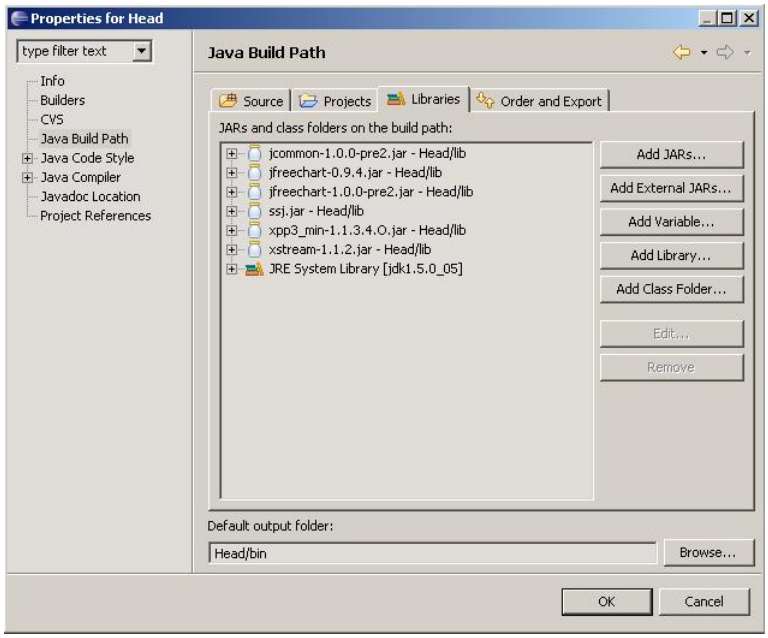


Figure 2.3: Properties

- The Arguments-Tab should contain following values:
  Program arguments: `[TEAMNAME] [HOST] goalie.conf defender.conf defender.conf defender.conf defender.conf player.conf player.conf player.conf player.conf player.conf player.conf coach.conf`

  e.g.: `DAINAMITE localhost goalie.conf defender.conf defender.conf defender.conf defender.conf player.conf player.conf player.conf player.conf player.conf player.conf coach.conf`
  VM arguments: `-Xmx512M`
  Working directory: `${workspace_loc:Robocup}/etc/agents`

  Now the Dainamite Project should run by clicking the run button.

2.5 The Agent Configuration Files

When starting the team, each agent needs a configuration file containing required information, such as the teamname or the servers location (hostname, portname). Therefore, the configuration files are simple property-value pairs, which can be read using a Java-Properties object. For each configuration file, which is given as program argument, an agent instance is created, whereas the order is relevant: First, the goalie has to be started, since he should have the tricot number 1. Thereafter the defenders, followed by all other player agents should be instantiated. Finally, the coach should be started, because his work requires that all agents are in place (i.e. for substituting the heterogeneous player types). Since the teamname and the hostname are also given as program argument, the values of the configuration files are overwritten, such that each agent uses the same values. In the following, the properties are

listed and explained briefly:

- teamname - The name of team, with which the agents connect to the server. Note, that each agent of a team must use the same teamname, else the server would't recognize the members correctly. Therefore, the first program argument is used to specify a global teamname.
- hostname - The address of the server, either in text (e.g. localhost) or as ip-address. This value must be set via the second program argument, which is also global.
- port - The port on which the server is listening for connections. The default pots are already set here.
- mode - An ID for the agent type: 0 - FieldPlayer, 1 - Goalie, 2 - Coach and 4 - Trainer (defined in AbstractAgent).
- gui-connect - Defines, if the agent connects to the gui (for debugging purposes) or not. A global value can be set optionally using the VM argument -Dgui-connect=true—false.
- statelist - The states (tactical classes), which should be instantiated by the agent, together with the corresponding playmodes, in which that state should be active. Note, that the names are the class-names of the states (subclasses of *AbstractState*) located in package *robocup.component.tactics*, and the playmodes are the constants defined in the enum *ControllerNumbers.PLAY_MODE*.

Changing the configuration files is especially neccessary, if new states should be added to the agents. Therefore use the same syntax as already given in the files. Note, that there shouldn't be an empty line between the states, and each new line should be closed by a '\', which indicates, that the next character (a newline) is ignored, and the property value continues.

3 The Dainamite Agent Architecture

3.1 Introduction

The core elements of this framework are the agent classes, which are topic of this chapter. In the following, their structure is given by presenting the corresponding class hierarchy, and the general control flow within them is shown using sequence diagrams. This should help to gain a better understanding, how the framework as a whole works, before going into details of the specific components of the agents.

3.2 Agents of the Dainamite Team

The agents that take part in a robocup simulation are the player-agents, the coach and sometimes a trainer. However, the trainer does not belong to the team (at least conceptual), hence its usage is explained in section 12.3. The following section gives an overview about the basic structure and capabilities of these agent types.

3.2.1 The AbstractAgent

The basic class for each agent is called *AbstractAgent*. It contains all type independent data and functionality, that a robocup agent must implement. Generally, this class provides the means for four tasks, which have to be solved within each agent type. These are initializing correctly, staying alive, maintaining a bi-directional connection to the server (send and receive) and understanding the servers messages. Each of them are important in different lifetimes of the agent, which are explained next.

In order to instantiate and use an agent-object, one has to follow three basic steps. Each of them is responsible for different sub-tasks. Generally, the following order must hold, else instantiation may fail because of connection problems to the server.

1. Create an instance of the corresponding agent class.
2. Call the *initialize()* method of the agent object.
3. Call the *start()* method (inherited from the Thread-class) to activate the agent directly after initialization.

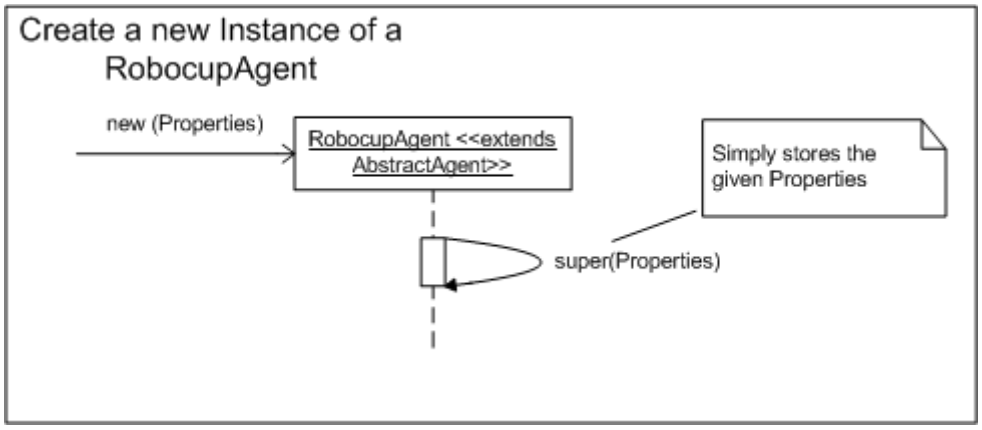


Figure 3.1: Creation of an Instance of the RobocupAgent class

On creation, the agent-object needs all necessary data, which is used to instantiate its components correctly. For instance, in order to connect to the soccer server, the agent must become aware of the hostname and the port, on which the server is listening for requests. All this information is given as a propery-object [1] to the constructor. In the current implementation, these objects are loaded from configuration files (one file for each agent), whose paths are given as command line arguments to the JVM. Additionally, there are some global command line arguments, which may overwrite properties of the specific property files (e.g. the teamname or the server hostname). The process of creating an instance of an agent with the *new*-operator (here for the class RobocupAgent) is given in 3.1.

Initialization of the agent contains reading all necessary information from the properties, which were given on instantiation, creating all components, e.g. the world-model, the parser, and so on, and establishing a connection to the server. The involved steps can be seen in 3.2, where at first in *initBaseConfig()* and *initConfig()* the basic and specific information from the properties are read. In *initComponents()*, all components are instantiated (the dots indicate their creation), and in *initConnection()*, the agent-type specific *initProtocol()* is called, which creates the RCSSConnection, sends an init-command to the server, and uses the responses to finish initialization of the agent and its components (e.g. the unum, the server-settings and the heterogeneous player-types are provided there, so that they can be set only after receiving them). Some of these methods are equal for different agents, others not, hence the AbstractAgent can only provide abstract methods for some tasks, which were overwritten in the specific sub-classes.

The run-method of the agent is responsible for keeping him alive, i.e. maintaining an active state. It also determines the order in which specific tasks are executed, which are in general receiving data from the server, process these data into readable objects and finally decide how to act after receiving them. Each of these tasks are delegated to the main components, which are triggered there. These are for the *AbstractAgent*:

1. RCSSConnection: This class contains simple methods for sending and receiving strings

[1] java.util.Properties

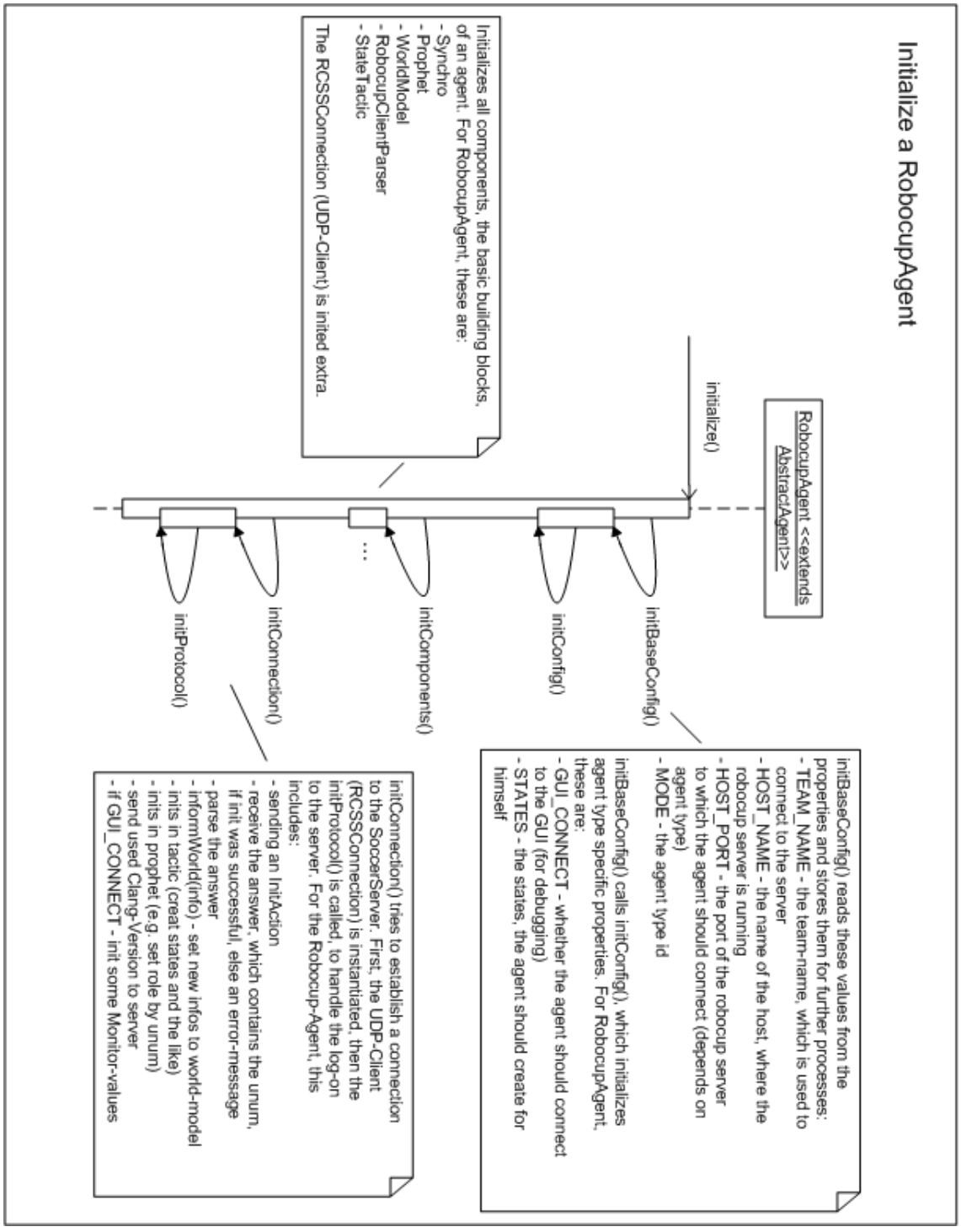


Figure 3.2: Initialize the RobocupAgent object

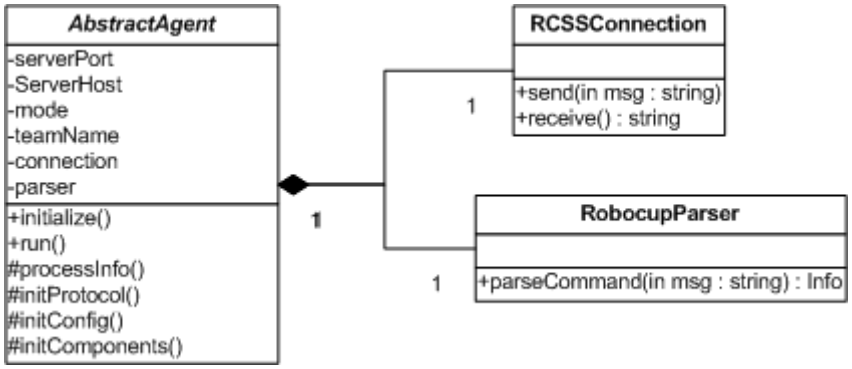


Figure 3.3: Overview about the class AbstractAgent

from and to the soccer server.

2. RobocupParser: This class can transform strings into objects, which are needed to extract and process their content later on.

Finally, the objects created by the parser are further processed by the components of each specific agent type (see for example RobocupAgent therefore), hence here the abstract method *processInfo* is called, which is overridden in the subclasses respectively. In Figure 3.3, the class *AbstractAgent* and its two components are displayed in UML notation.

3.2.2 The RobocupAgent

This class is the main class for all player controlling agents, and it also contains the main method for all agent types, which is used to start the entire team. It extends the *AbstractAgent* and implements the methods that have to be overridden. It also incorporates some more components additionally to those mentioned in its super class, which are explained in this work more detailed in the following chapters. Here, the general structure of this class is presented briefly.

As an extension of the *AbstractAgent*, this class provides some additional attributes. These can be seen in Figure 3.4, as well as its components and methods. The number is the tricot number (corresponds to the unum received from the server after initialization), which is unique for his team and hence can be used to identify an agent. The attributes *useGui* and *useTactic* can be used to switch certain functionalities on and off, and the list of states captures the agents capabilities, which were passed within his config-file, and constitutes his tactical decisions.

The *RobocupAgent* works by delegating specific tasks in a certain order to its components in the *processInfo()*-method. Within those, the main logic for maintaining a worldmodel and deciding how to act is implemented. The components are:

- Synchro: This component is responsible for deciding, on which percepts to react in order to stay synchronized to the server. Its main purpose is to avoid holes (cycles, in which no actions are send) and clashes (cycles, in which two competing actions are send).

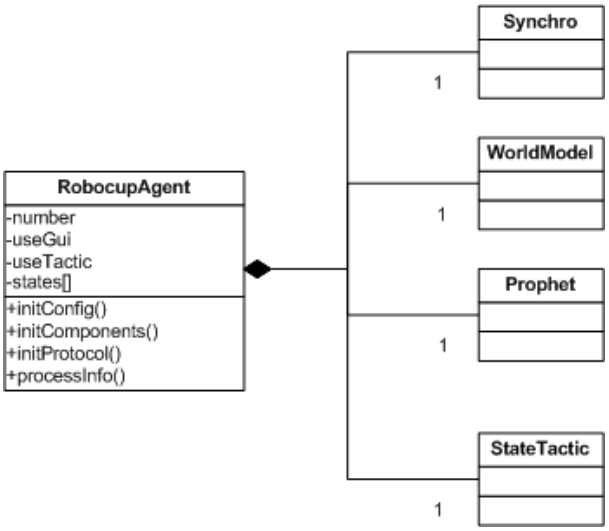


Figure 3.4: Overview about the class RobocupAgent

- WorldModel: This class contains the knowledge of the agent about the state of the world, and is updated using the percepts received from the server.
- Prophet: The Prophet is used to predict possible future states of the world, using the current knowledge and possible actions available to the agent. This knowledge is also used to decide, which actions are more preferable than others by examining their expected outcomes.
- StateTactic: This is where the agent decides, how to behave in a current situation. Therefore, it uses the WorldModel and the Prophet and produces a set of actions, which were given to the synchro (which forwards them to the server in an appropriate moment).

In 3.5, the control-flow of the RobocupAgent during simulation is given as sequence diagram. It shows, how and when the different components are used to handle certain tasks. First of all, the *RCSSConnection.receive()*-method waits for data from the server, and after that, the reveiced string is parsed into machine-usable objects within the *RobocupClientParser*. This is implemented inside the *run()*-method of *AbstractAgent*. The *Info*-Objects is forwarded to the *processInfo()*-method, which is defined abstract in *AbstractAgent*, but overridden in the *RobocupAgent*. There, these objects are used to update the *WorldModel* (*informWorld()*) and to calculate the *Tactic* (*informTactic()*). The *Synchro* handles the sending times, and hence is informed each time, a message was received (*informSynchro()*). If for certain time (configured in RCSSConnection) no message was received, the agent exits the main-loop and terminates, assuming that the server is not running anymore. This is evaluated in *alt_1*. In *alt_2*, it is evaluated, if the data should be used to update the *WorldModel*, which in some cases is not meaningful, e.g. if the second visual information arrived in the same cycle. *alt_3* checks, if an action should be computed by evaluating the *Tactic*, which should be done almost always after receiving actual visual information. However, sometimes it is needed to compute an action after a body-sense information as well, because it might happen, that the visual information is received very late, so that time is too short here.

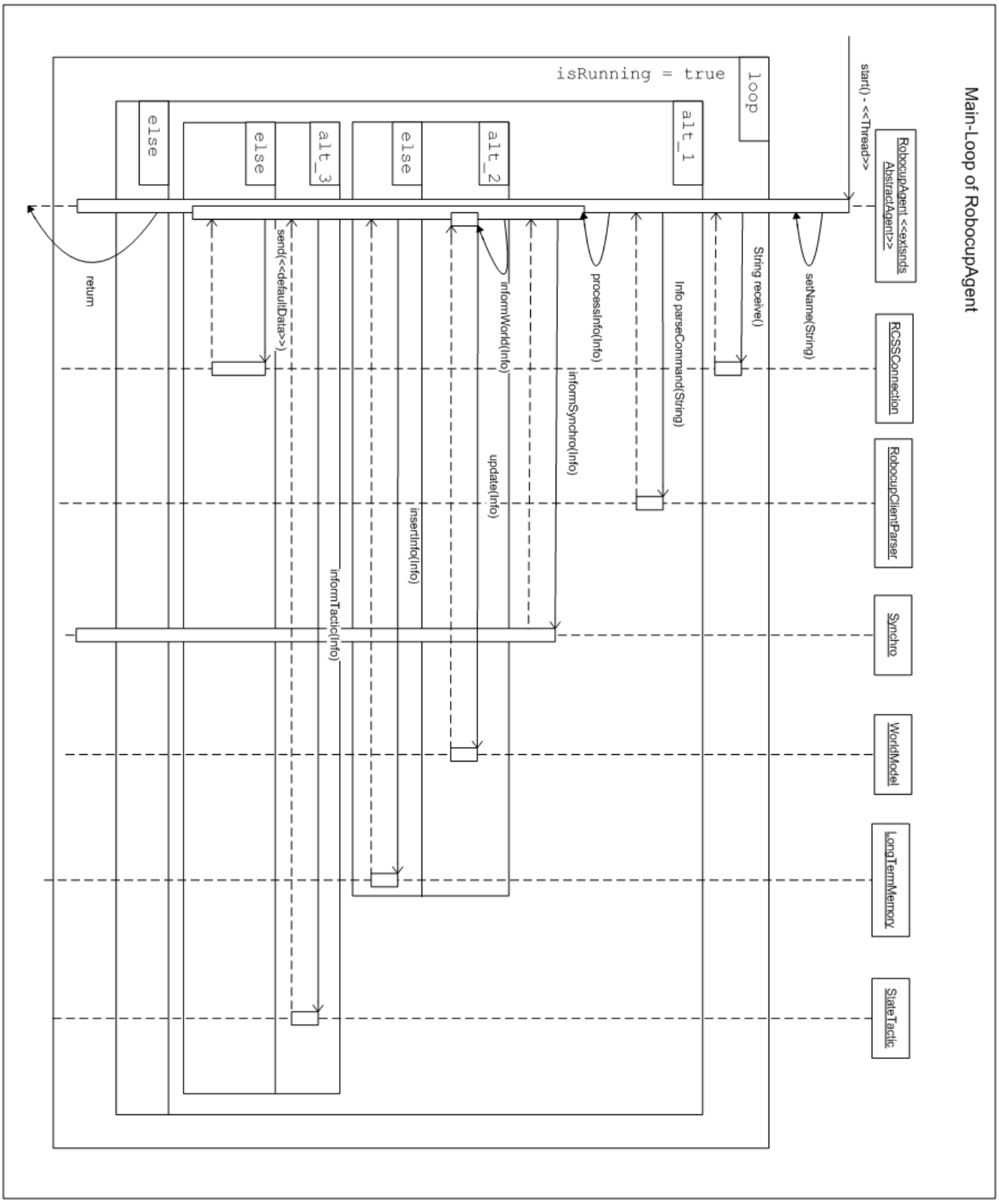


Figure 3.5: The Control-Flow of the RobocupAgents main thread, after starting.

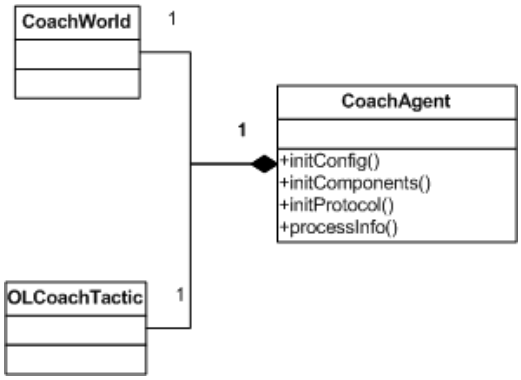


Figure 3.6: Overview about the class CoachAgent

3.2.3 The CoachAgent

The *CoachAgent* is the agent class for the coach, and therefore extends the abstract agent as well. Because it has unrestricted and noise-free perception, which already is synchronized with the server, its structure is little less complicated than that of the *RobocupAgent*. There are no extra attributes (like tricot-number), and only two additional components with regard to the *AbstractAgent*. These are the following:

- CoachWorld: The knowledge of the coach about the state of the world. It contains more precise data than those of the player agents, and its maintainance is easy due to better information from the server.
- OLCoachTactic: This class calculates the actions of a coach, which usually are speech-acts in freeform or clang format.

The *CoachAgent* also implements the abstract methods inherited from *AbstractAgent*, which are the *initConfig*, *initComponents*, *initProtocol* and the *processInfo* methods. They serve for the same purposes as within the *RobocupAgent*, but are less complicated, because no explicit synchronization is needed. The tactics for the coach are given in chapter 11. In the following Figure 3.6, its structure in UML notation is given.

3.2.4 The TrainerAgent

The *TrainerAgent* has very few functionality within the Dainamite-Framework, but is extended in the SoccerScope (Monitor) section 12.3. Here, it provides a shell for a trainer, which can connect to the server and interact through UDP communication in order to execute training scenarios. Hence it has to implement the abstract methods (see *AbstractAgent* therefore) in order to connect and stay active.

The following Figure 3.7 shows a complete view about the agent classes and their components.

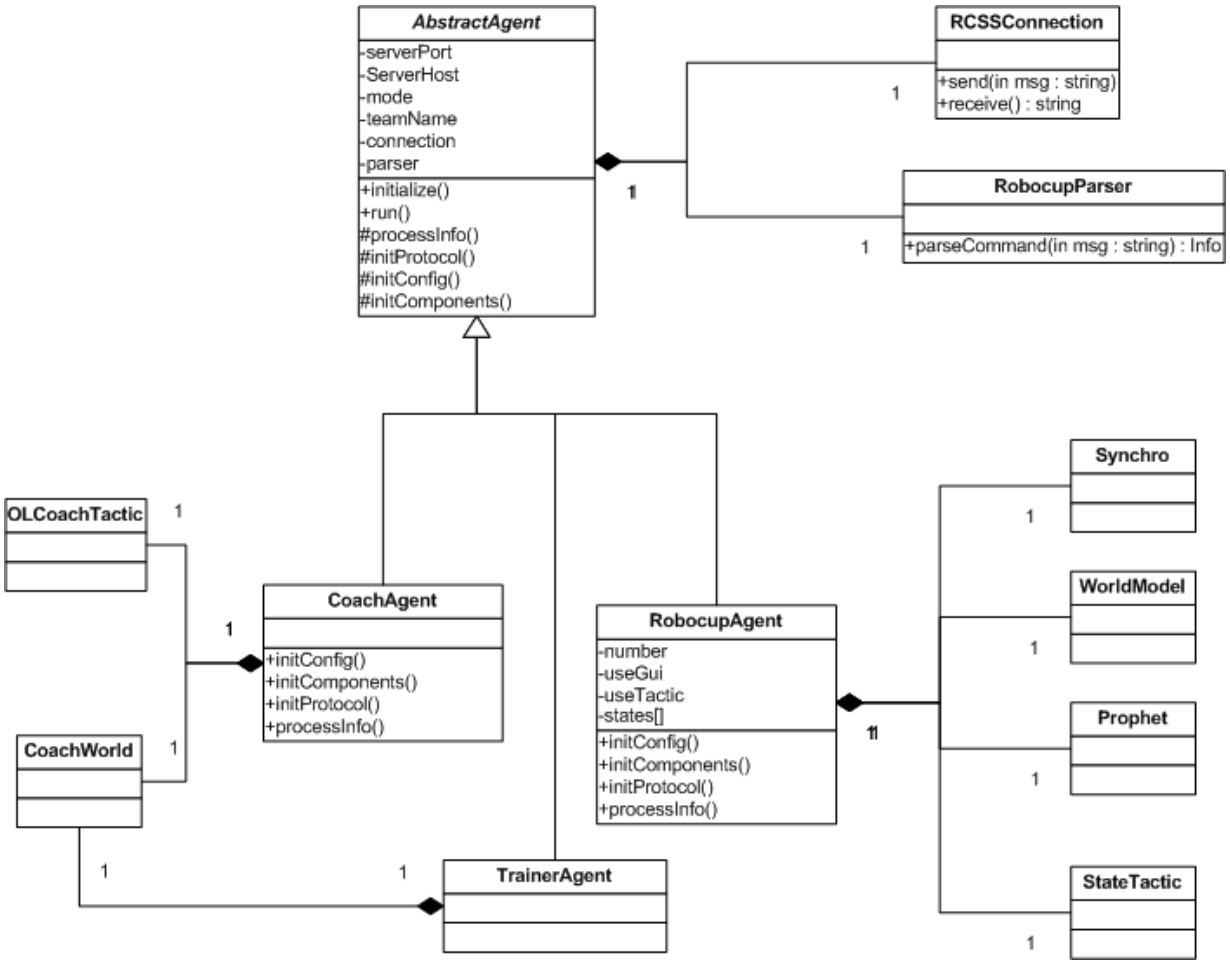


Figure 3.7: Overview about all Agent Classes

The next chapters cover the components of the agents in more detail. The order corresponds to the order of workflow within the agents. First, received messages have to be translated into usable objects by the parser. Then synchronisation assures the correct activity thereafter. The worldmodel is updated, and after that, tactical decisions are computed using different mechanisms.

4 Parser

4.1 Introduction to the Parser Component

The Parser is responsible for converting the text based messages received from the SoccerServer into information that can be handled by the various components of the framework. Therefore its general purpose is to translate strings into objects (of type *Info*), using a method with the following signature:

```
public Info parseCommand(String message);
```

The message (parameter) for this method is provided by the RCSSConnection (see AbstractAgent in section 3.2), and the resulting *Info*-object is processed by other components (WorldModel) of the corresponding agent. The parser is written using JavaCC [3], a parser generator for java, which provides the possibility to create a set of parser-classes out of a grammar specification[1]. Important for the parser, however, is the initialization of the agent, because the server version, or more precisely, the protocol that corresponds to a certain server version, that should be used, can be defined there. This implementation connects with version 9.4.5[2], and hence the parser is written to understand all message-types that belongs to that protocol version. As far as we know, the protocols didn't change in later releases of the server[3], so that this parser should be compatible to later versions as well.

4.2 The Different Parts of the Parser

There are multiple different parts of the parser, some of which correspond to the type of information processed (see figure 4.1 below). The complete parser is actually a set of different sub-parsers, each of them for different messages, so that it is possible to combine these parts for the different agent-types. For instance, some messages received by the coach and the player have the same form, like the player-type messages, such that here the same sub-parser could be used. Other messages are quite different, like the visual information, because these are global and precise for the coach, but noisy with a relative perspective for the players. Here,

[1] Note that the sources for the parser-classes are all files ending with .jj, because the Java-Files are generated out of these.

[2] See therefore the *toString()*-method of the class *robocup.component.actions.InitAction*

[3] The version 10.0.7 was used in Bremen 2006

different parsers have to be implemented. Some messages are unique for certain agent-types, e.g. the sense-body information, which is only sent to the player-agents. In the following, the parser-aggregation out of sub-parsers is explained for the specific agent-types.

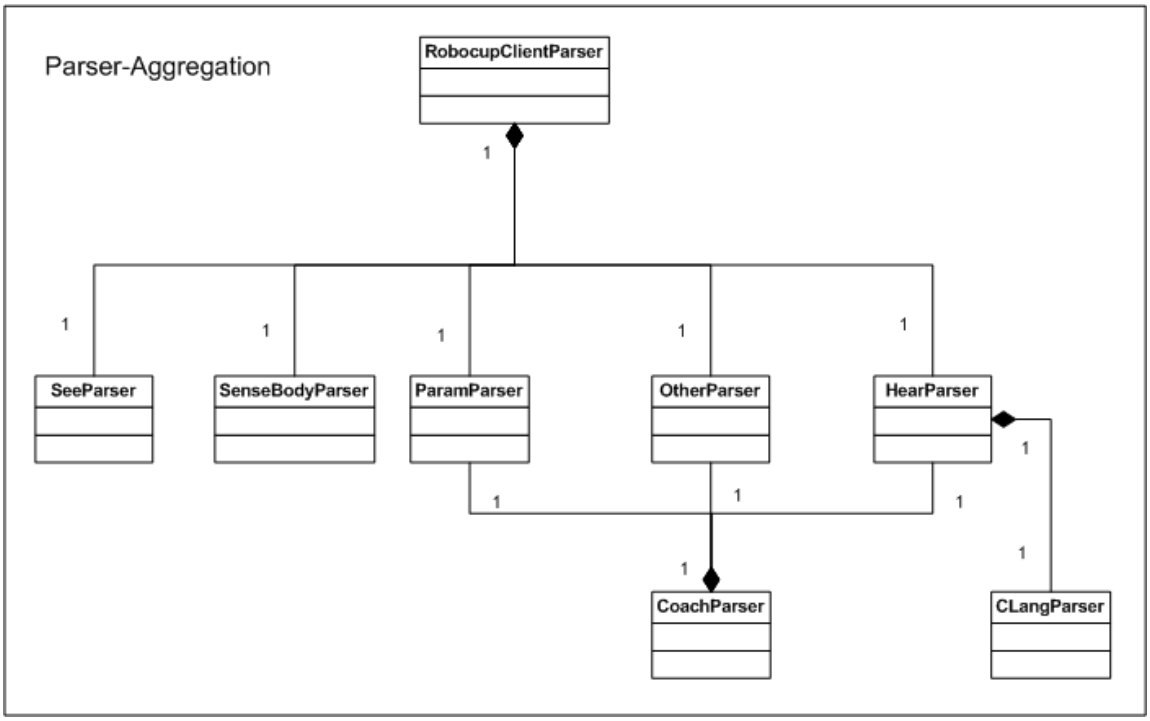


Figure 4.1: The structure of the Parser Parts

4.2.1 The Player Parser

RobocupClientParser :

The main parser for all player-agents, which decides what type of information is to be processed and which sub-parser will do this.

- Type of information processed
  - All
- Other parsers used
  - HearParser
  - OtherParser
  - ParamParser
  - SeeParser
  - SenseBodyParser

4.2.2 The Coach/Trainer Parser

CoachParser :

The main parser for the Coach (and Trainer) messages which uses some of the specific player parsers.

- Type of information processed
  - All except SenseBody
- Other parsers used
  - HearParser
  - OtherParser
  - ParamParser

4.2.3 The Hear Parser

HearParser :

Parses received messages heard from other teammates, the referee or the coach (and maybe sometimes in the future messages of the other team too). Messages from the coach are sent via CLang (coach-language), and hence parsed inside the *CLangParser*, which is part of the *HearParser*. This parser generates all types of *AuralInfo*'s, .i.e. *CoachSayInfo, CoachPlayerSayInfo, PlayerSayInfo, PlayModeInfo and RefereeInfo.*

- Type of information processed
  - Messages sent by other players
- Example
  - (hear 2 135 our 2 "t0218_-30")

4.2.4 The See Parser

SeeParser :

Parses visual information like Flags, Lines, Players and the Ball. The resulting info-object is a collection containing all these objects, i.e. a *VisualInfo* consists of the seen *LineInfo*'s, *FlagInfo'*s, a *BallInfo* and *PlayerInfo*'s. Note, that these classes have an inheritance structure similar to that of the corresponding classes of the *WorldModel.*

- Type of information processed
  - Visual information
- Example
  - (see 61 ((f t l 10) 37.3 9) ((f t l 20) 37.7 -6 0 0) ((f t l 30) 40.4 -20) ((p "a4ty" 10) 16.4 -1 0 0 153 154) ((l t) 32.1 -86))

4.2.5 The SenseBody Parser

SenseBodyParser :

Parses information provided by sensebody messages. These messages are only received by the player-agents, which need to know for example about their stamina, recovery, etc. The resulting info-object is called *SenseBodyInfo*, which provides the nessessary attributes for all values.

- Type of information processed
  - SenseBody messages
- Example
  - (sense_body 61 (view_mode high narrow) (stamina 3548.34 0.864) (speed 0 111) (head_angle 0) (kick 0) (dash 17) (turn 126) (say 88) (turn_neck 77) (catch 0) ...

4.2.6 The Parameter Parser

ParamParser :

Parses the information regarding server parameters and player types. These messages are sent to the agents initially, and are all the same for each agent type. The resulting info-objects are either *PlayerTypesInfo*, which contains the heterogeneous player-variants, *PlayerParamInfo*, containing the general players physics, and the *ServerParamInfo*, which contains all other server-related constants. Note that the *PlayerParamInfo* and the *ServerParamInfo* are configurable in the files *player.conf* and *server.conf* respectively, which are located in the *.rcssserver*-folder of the users home-directory. These files are generated by the soccer-server on first startup, and should be edited for example when a training-scenario should be used.

- Type of information processed
  - Server parameter
    - Example:
      (server_param (audio_cut_dist 50) (auto_mode 0) (back_passes 1) (ball_accel_ max 2.7) (ball_decay 0.94) (ball_rand 0.05) (ball_size 0.085) (ball_speed_ max 2.7)
  - Player parameter
    - Example:
      (player_param (dash_power_rate_delta_max 0) (dash_power_rate_delta_min 0) (effort_max_delta_factor -0.002) (effort_min_delta_factor -0.002)
  - Player types
    - Example:
      (player_type (id 0)(player_speed_max 1.2)(stamina_inc_max 45)(player_ decay 0.4) (inertia_moment 5)(dash_power_rate 0.006)(player_size 0.3)

4.2.7 The Other Information Parser

OtherParser :

Parses everything not belonging into one of the above categories. This includes error-messages received from the server and confirmation messages. Errors are not translated into info-objects, hence the resulting infos are either *ChangePlayerTypeInfo* or *ChangeOpponentTypeInfo*, which are received from the server after player-exchange.

- Type of information processed
  - Errors
    - Example:
      (error only_init_allowed_on_init_port)
  - Playertype changes
    - Example:
      (change_player_type 3 4)

4.3 The Different Messagetypes Explained

Each message has a certain content and format, and is mapped by the parser to specific *Info*-objects, containing the corresponding data. In the following section, the message-types are explained, closing with the resulting objects in a class-diagramm. A more sophisticated, but not really actual description of the message formats can be found in [7].

4.3.1 General Message Layout

Most of the messages the player receives have a common structure:

- they start and end with brackets (....)
- the first entry after the bracket is the type of the message (see/hear/ ...)
- the second entry is the cycle the message was sent (see 0 ...)
- for more complicated message they'll be also bracketed inside (see 0 (...) (...) ...)

4.3.2 The Hear Message

The structure of the hear message
(hear [cycle] [source] [message])
There are four sources for something the player hears:

- The Referee
  The source would be ***referee***
- The Coach
  The source would be ***ol_coach_left*** or ***ol_coach_left***.
- Own Player
  The source would be ***[direction] our [no of player]***
- Other Player
  The source would be ***[direction] opp***

If the source is the online coach, *message* is one of the different clang message types. For a further description of them, have a look at chapter 11. Examples of received hear messages:

- (hear 0 referee drop_ball)
- (hear 2 135 our 2 "t0218_-30")
- (hear 2 59 opp "b52342727")

4.3.3 The See Message

The structure of the see message
(see [cycle] [object]+)
There are five objects a player might see

- flag
  object would look like ***((f [flag identifier])[position&movement information])***
- line
  object would look like ***((l [line identifier])[position&movement information])***
- goal
  object would look like ***((g [side of goal])[position&movement information])***
- ball
  object would look like ***((b)[position&movement information])***
- player
  object would look like ***((p "[teamname]" [player no] goalie)[position&movement information] t)***
  Where ***goalie*** would only appear if the player seen is a goalie and ***t*** if the player tackles. The teamname and number are also not visible every time because the player may be too far away to see this information.

Example parts of received see messages:

- ((f c) 13.7 18 0 -0)
- ((l l) 72.2 -66))
- ((g l) 67.4 -18)
- ((b) 10 7 0.2 -0.1)
- ((p "a4ty" 6) 9 -2 0.18 1.8 41 5)

Position and movement information may be any of the combinations below (what we actually see depends on the distance of the object, if it's father away we get less information)

- [direction]
- [distance] [direction]
- [distance] [direction] [point direction] (player only)
- [distance] [direction] [distance change] [direction change]
- [distance] [direction] [distance change] [direction change] [point direction] (player only)
- [distance] [direction] [distance change] [direction change] [body facing direction] [head facing direction] (player only)
- [distance] [direction] [distance change] [direction change] [body facing direction] [head facing direction] [point direction] (player only)

4.3.4 The Sense_Body Message

- The structure of the sense_body message:
  - (sense_body [cycle] [info]+)
- Example:
  - (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)))

4.3.5 The Server_Param Message

- The structure of the server_param message:
  - (server_param [parameter]+)
- Example:
  - (server_param (audio_cut_dist 50) (auto_mode 0) (back_passes 1) (ball_accel_max 2.7) (ball_decay 0.94) (ball_rand 0.05) (ball_size 0.085) (ball_speed_max 2.7) (ball_weight 0.2) (catch_ban_cycle 5) (catch_probability 1) (catchable_area_l 2) (catchable_area_w 1) (ckick_margin 1) (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 0) (coach_port 6001) (coach_w_ referee 0) (connect_wait 300) (control_radius 2) (dash_power_rate 0.006) (drop_ball_time 200) (effort_dec 0.005) (effort_dec_thr 0.3) (effort_inc 0.01) (effort_inc_thr 0.6) (effort_init 1) (effort_min 0.6) (forbid_kick_off_offside 1) (free_kick_faults 1) (freeform_send_period 20) (freeform_wait_period 600) (fullstate_l 0) (fullstate_r 0) (game_log_compression 0) (game_log_dated 1) (game_log_dir "./") (game_log_fixed 0) (game_log_fixed_name "rcssserver") (game_log_version 3) (game_logging 1) (game_over_wait 100) (goal_width 14.02) (goalie_max_moves 2) (half_time 100)

4.3.6 The Player_Param Message

- The structure of the player_param message:
  - (player_param [parameter]+)
- Example:
  - (player_param (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) (player_speed_max_delta_min 0) (player_types 7) (pt_max 3) (random_seed -1) (stamina_inc_max_delta_factor 0) (subs_max 3))

4.3.7 The Player_Type Message

- The structure of the player_type message:
  - (player_tpye [id] [parameter]+)

- Examples:
  - (player_type (id 0) (player_speed_max 1.2) (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 1.2) (stamina_inc_max 34.18) (player_decay 0.595) (inertia_moment 9.875) (dash_power_rate 0.007082) (player_size 0.3) (kickable_margin 0.822) (kick_ rand 0.061) (extra_stamina 10) (effort_max 0.98) (effort_min 0.58))

4.3.8 Overview About Generated Info-Types

In this subsection, an overview about the *Info*-objects generated by the parser is presented. Therefore, figure 4.2 provides a class diagram, which contains those important for the player agent (some more actually exists, especially for the coach language). Within this, frequently received messages are those inside the blue box, whereas initially received are inside of the orange one. Frequently received messages represent the perceptions of the agents by either hearing, seeing or sensing. Initially received messages contain information which will not change during simulation (e.g. the field-size within the ServerParamInfo or the tricot number of the agent). The attributes of these types were omitted here in order to obtain a better overview. For getting an insight about their content, either refer to the messages aforementioned or to the code of the objects within the project itself.

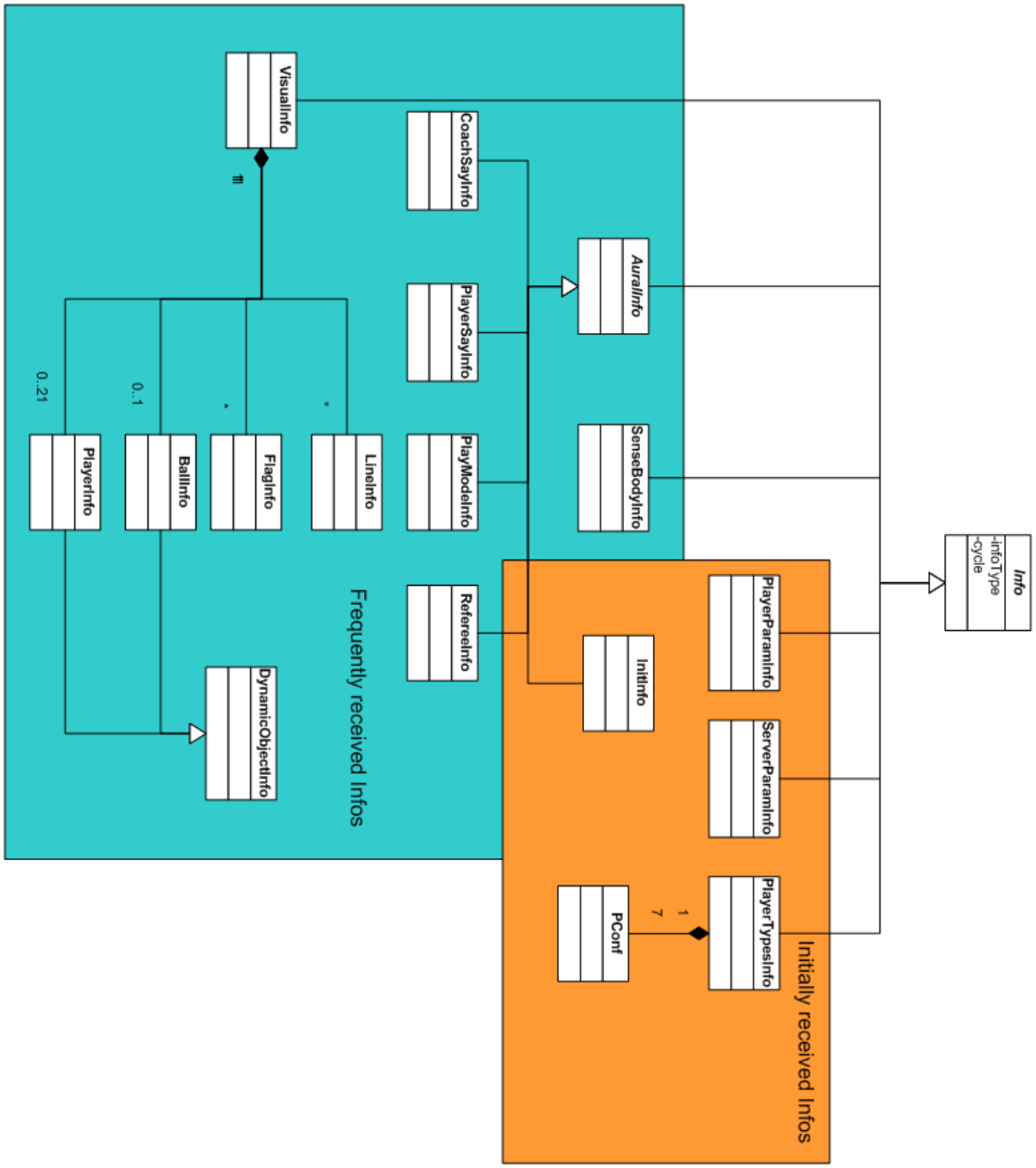


Figure 4.2: UML Class Diagramm for Info-Types, which are generated by the Parser (focusing on those for the Player-Agents)

5 Synchronisation

5.1 Overview

Like in almost every other client/server-environment it is necessary to synchronize calls to the other party. In robocup the amount of time available to calculate an action is limited because actions have to arrive at the server before the end of each cycle. On the other side we would like to spend as much time as possible for the calculation of an action to get a better action quality. The synchronisation tries to resolve this conflict by sending the action as late as possible based on the expected cycle length and network latency.

A second task accomplished by our Synchronisation is the regulation of the VI arrivals. This is needed as we would like to base our decisions on visual data instead of an estimation. Using different Viewmode sequences the Synchro manages the arrival times in a way that we have one VI each cycle.

This chapter takes a deeper look into the ongoing communication between an agent and the soccer server in order to get an overview about the message-flow. Then the problems that the Synchro should solve are introduced, and after that, this document will give an idea about the logic used by our synchro implementation.[1]

5.2 Communication

A player (client) is informed about what is going on in his environment by three types of messages sent from the soccer server. He is informed about what he is feeling by a sensebody info (SBI) that arrives at the beginning of each cycle, about what he is seeing by a visual info (VI) and about what he heard from teammates, the referee or the coach by an aural info (AI). The SBI is sent by the server in steps of (by default) 100ms. However, due to network latency and/or a slow running server this value may be above 100ms.

The VI arrival frequency depends on the current view angle and view quality (the values for view angle and quality also arrive as part of the previous SBI) and can be influenced by the client sending **ChangeViewModeActions**. The interval between 2 VIs can be derived by the following formula:

$$Interval = send\_step * view\_angle\_factor * view\_quality\_factor$$

[1] This section mainly describes our solution of the synchronisation problem giving only a limited overview of the problem itself. For a more detailed problem discussion refer to [8].

The send_step is set in the server configuration (default 150ms). The view angle can be set to narrow (45°), normal (90°) and wide (180°). The factors for these angles are 0.5, 1.0 and 2.0 in the same order. The view quality is set to low (factor 0.5) or high (factor 1.0).

The client communicates to the server by sending action messages. The challenge of the Synchro is to determine the sending time of these action messages to the server, such that the conditions discussed in the next section hold.

5.3 Problem

The Synchro is responsible for two main tasks. To avoid holes and/or clashes and to optimize the VI arrival times.

5.3.1 Holes and Clashes

Cycles with an action sent to late to the server for beeing processed are called holes. Cycles with more than one action beeing sent are called clashes[2]. Both cases should be strictly avoided as they miss a chance to act or lead to non predictable results.

5.3.2 Waiting for a VI

A VI is an important source of information that we would like to base our decisions on. In the default setting, only 2 VI's arrive every 3 cycles, due to the cycle length of 100ms and the VI arrival interval of 150ms (normal view angle and high view quality), thus leading to a very high percentage (33%) of actions that were calculated based only on a BSI. Furthermore we have to make sure that all VIs arrive early enough to allow the calculation of an action without risking a hole. How this is achieved is described in the next section.

5.4 Synchronization Concept

As mentioned above, it isn't a good idea to determine the send time by a constant value. It has to be established a basis to choose the send time in a dynamic manner. For this purpose the master thesis [8] was used as a repository of ideas and tried to find a method of resolution in it. The following solution was inspired by the "Flexible External Windowing" concept:

"The arrival times of visual informations form a pattern that repeats itself every three cycles. Due to the repetitive nature of this pattern it is possible to predict in each cycle whether a VI will arrive or not"

[2]Note that these denotations are adopted from [8] - were an extensive analysis of the synchronisation-problem is given.

In the following, it is described now how this pattern was used to build up a dynamic synchronization mechanism.
For the Synchro, every cycle consists of four quarters as visualized in figure 5.1. The Synchronisation problem therefore can be solved by arranging that a VI arrives each cycle in one of the first three quarters. An arrival in the fourth quarter should be avoided as we would risk not to have enough time for further calculations.

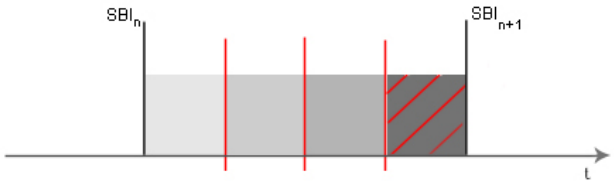


Figure 5.1: Interpretation of a cycle

Combining this with the VI interval formula, it is possible to derive an algorithm, that produces a pattern, which always lets a VI arrive in one of the three former quarters.

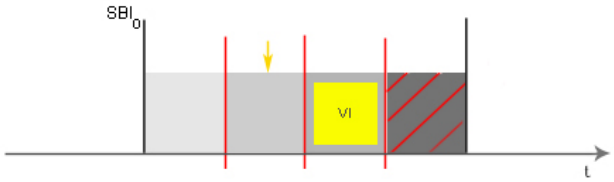


Figure 5.2: Cycle (t) with a VI in the 3rd quarter

As it can bee seen in figure 5.2 a VI arrives in the third quarter. Assume the send step has it's default value of 150ms and was set by the server. Now let the view angle factor for the next VI be 0.5, i.e. set it to the narrow mode and the view quality factor 1.0, that means the view quality is set to "high". The result for the next interval is 75ms. Since the length of the cycle is ideally 100ms, using one with 75ms leads to a new VI arriving in 3 quarters in the future. (The orange pointer in the figures shows the VI arrival quarter of the next cycle)

Figure 5.3 shows the next VI that arrived 3 quarters later in the second quarter of the following cycle. If this interval isn't changed, the next VI will arrive again in 75ms (3 quarters), and the arrival time in the next cycle is now in the first quarter.

Now we switch the view mode from "narrow" to "normal" leading to a view angle factor of 1.0. This will result in a VI interval of 150ms, thus six quarters and that is the third quarter of the next cycle. The next VI will be received on the same position as it has been three cycles earlier. The player will always see twice $\frac{1}{8}$ (45°) and once $\frac{1}{4}$ (90°) of the world. Thus during three cycles he sees the same amount of the world as when using only the standard view mode

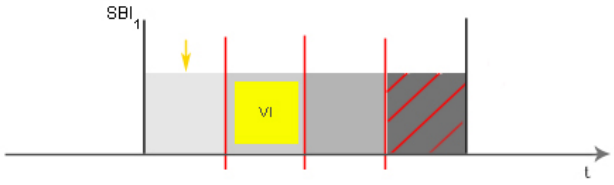


Figure 5.3: Cycle (t+1) with a VI in the 2nd quarter

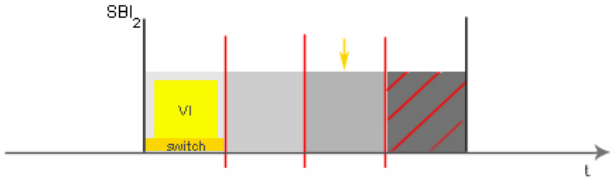


Figure 5.4: Cycle (t+2) with a VI in the first quarter

of twice $\frac{1}{4}$ (90°) and once empty. However, now he may see important objects like the ball or near opponents every cycle and he is more flexible of where to look to.

5.4.1 Emergency sending

The solution mentioned above assumes that we are always capable of calculating an action in the time left after a VI arrival. Unfortunately, as by now our action calculation blocks till an action was calculated instead of continously improving a preliminary action, this fact cannot always be assured. Especially situations where the player has ball control and has to decide between passing and dribbling etc. are expensive to calculate. Therefore for some cycles we additionaly calculate an action based only on the BSI. This action is deleted whenever a better action based on a VI could be calculated in time. Otherwise it is send to the server some ms before the end of the cycle. This currently happens only in less than 1% of all cycles.

For a more detailed description of the synchro functionality see our source code documentation.

6 WorldModel

6.1 Overview

The world model is a representation of the current knowledge about the world of a single player. He perceives the environment using visual, aural and body sense information which he receives from the soccer server. The function of the world model is to sum up the information of these three info types and provide methods to easily access the state of the world out of other components like the different situations and the tactic components. Gaining accurate information out of the info-objects is not always trivial, hence some aspects are explained in a separate chapter (see chapter 6.6 for a deeper look into estimation of position and speed of dynamic objects). Additionally, deriving knowledge out of given data is also excluded from the world model. This is the topic of the prophet, where relevant information (e.g. the interception point of the ball) is calculated. See in chapter 8 for further details on that.

This chapter is organized as follows: First of all, the differences between the world model of the soccer server and that of the agents are compared against each other. These concern mainly the coordinate space and the corresponding geometry. After that, the building blocks of the world model are presented, e.g. the basic classes such as the ball or the player together with some of their important attributes and methods. After that, the modularized concept of the world model itself is presented, and this chapter closes with a discussion on improvements and future work.

6.2 Global vs. Agent Perspective on the Environment

In order to understand, how an agents world model works, the distinction between the global and the agent perspective on the environment (i.e. the soccer-field) must be clarified. When looking at the soccer field from above, e.g. as shown in figure 6.1, lines or flags can said to be on the left, right, top or bottom side. This perspective is referred to as *global*, because it doesn't care about information, which is relevant to the player agents. The corresponding coordinate system is given in figure 6.2, where the X- and Y-axis and the angles are annotated. This system is a common left-hand coordinate system.

For the player agents on the other hand, it is not so helpful to know whether a goal is located on the left or on the right side, but to know if it is the own goal or belongs to the other team. Hence the world model of the agents is translated into the agents perspective. All objects (flags, lines) in the agents world model are named with suffixes *own*, *other*, *left* or

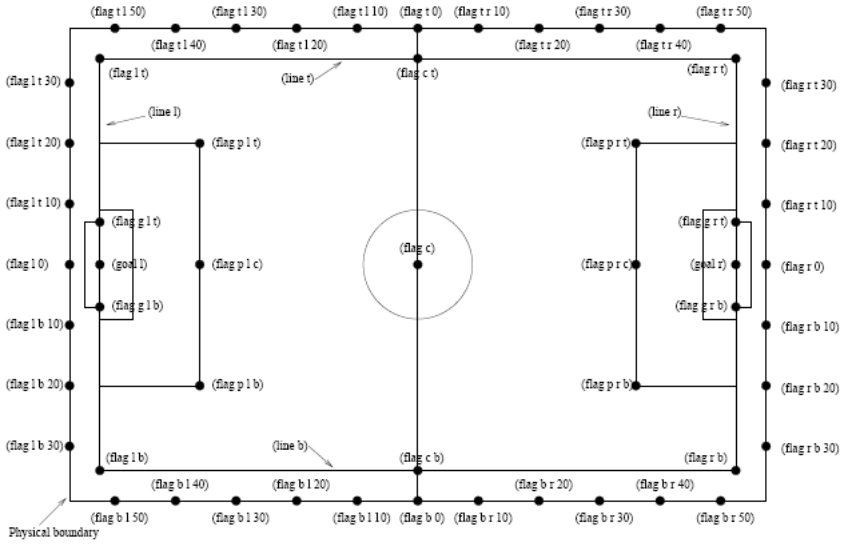


Figure 6.1: Layout of the static objects in the robocup environment (token from [7]).

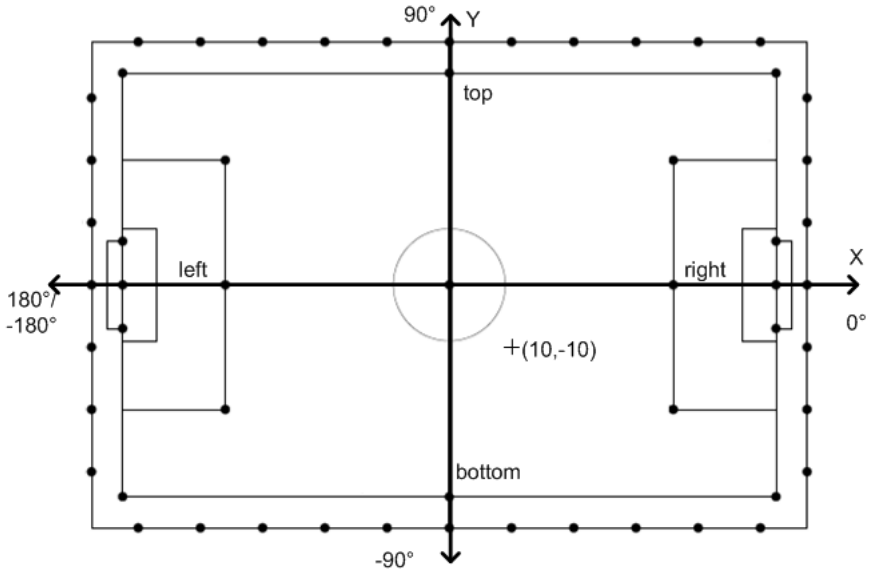


Figure 6.2: The coordinate space of the servers internal world model.

right, indicating to which side of the field relative to the playing direction of the agent they belong. With *own* the side the agent is playing from is referred to, whereas with *other* the opponents side is meant. *Left* or *right* are relative from the agents playing direction, i.e. when looking to the opponents goal. In figure 6.3, the corresponding coordinate system is presented. As it can be seen, not only the coordinate system layout is different, also the angle-metric has changed.

The angles within the agents perspective might appear confusing, because they increase or decrease into the opposite direction as usual, and the null-angle lies on the Y-axis. The idea

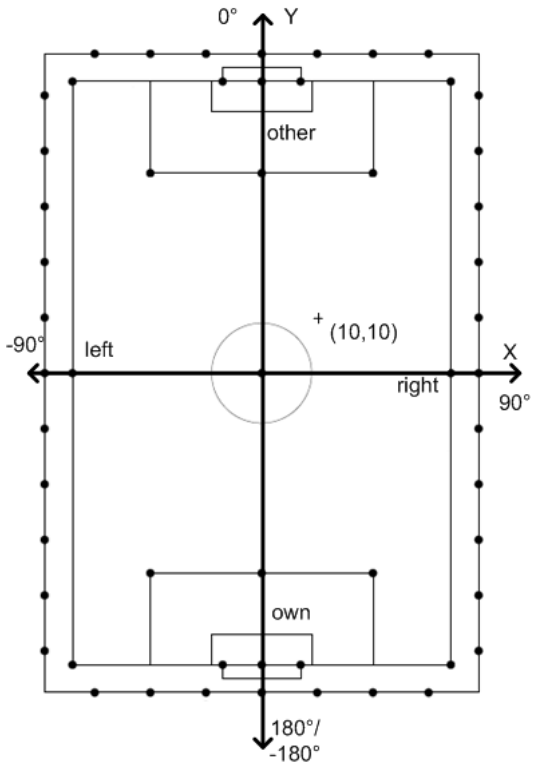


Figure 6.3: The coordinate system of a robocup agent.

behind this is, that the developers found it intuitive to let the null-angle point towards the opponents goal, and to have increasing angles to the right side. However, since developers have to care only for the agents perspective, there shouldn't be too much confusion at all, at least if the developer uses the classes and methods for geometric calculations that are part of this framework.

Another problem for the agents perspective is, that it depends on whether a team plays from left to right or the other way round, because flags and lines are fixed within the servers internal world model. For instance (flag l t 20) refers to the a flag on the own side, 20 meters left from the goal, when playing from left to right, but to a flag on the on the other side, 20 meters right to the opponents goal otherwise [1]. Therefore a mapping from the global to the agents perspective is necessary, which in this framework is located already in the parser. Each information, which is forwarded to the world model is already represented in the agent perspective, and the developer of all other components must only use the agents perspective. The same counts for actions, which have to be sent to the server. For instance, the *Move*-Action has absolute coordinates as parameters. Here, coordinates from the agents perspective have to be mapped to global coordinates. This happens in the BasicActionFactory, which can be called using the agents perspective, but creates actions, which are translated to the servers global perspective. The following subsection now presents the classes for geometric

[1] All flags are instantiated within the class *SConf*, where all server-specific parameters and values are stored.

calculations of the dainamite framework, which ease the use of the agents perspective.

6.2.1 Geometry Classes and their Usage

The Dainamite Framework provides some basic classes for geometric calculations, such as a vector, line or circle. These can be used by the developer to calculate positions and areas for certain purposes. They are located in the package robocup.component.geometry, and all these classes are wideliy used within the whole framework. Thus changes have to be made carefully, otherwise strange behaviour of the agents might result.

Vektor

The most basic class is *Vektor*[2], which is used to represent positions and speeds of mobile objects as well. It incoorporates a large set of methods making calculations very easy. Since this class is used very widely, it provides the possibilty to reuse previously instantiated objects comfortably [3]. Further information can be found in the JavaDoc comment of this class.

Class

Name	robocup.component.geometry.Vektor
Description	This class represents a position or a velocity in the 2D space. It can be instantiated using polar or cartesian coordinates. The default is polar (i.e. when only two values, the *length* and the *angle* were given to the constructor). Angles should always be given in *degree*.

Attributes

x	The x-coordinate of that vector.
y	The y-coordinate of that vector.
length	The length of that vector, which is its distance from the coordinate origin (0,0).
angle	The angle from the coordinate origin to the vector.

Methods

add(Vektor)	Adds the given vector and retrieves the result as a new instance.
addToThis (Vektor)	Adds the given vector to *this* instance.
cloned()	Clones this vector and retrieves it.
compareWithVariation (Vektor, double)	Checks, if the given vector is as near as the variation to *this* vector.

[2]The strange spelling was used in order to avoid conflicts with many other equally named classes, that are inherent in Java.

[3]Object instantiation is one of the performance killers in Java.

copy(Vektor)	Overwrites this vector with the values of the given one.
cross(Vektor)	Calculates the cross-product of the two vectors.
div(double)	Divides the vector by a constant value (and changes it).
getAngleBetween (Vektor)	Returns the global angle between the angle of this vector and the angle of the specified vektor.
getAngleBetween (Vektor, Vektor)	Returns the angle between two positions seen from this vektor.
getAngleTo(Vektor)	Returns the global angle to the specified vector relative from the position denoted by this vector.
getDistance(Vektor)	Returns the distance to a given vector.
getVektorBetween (Vektor)	This method returns a vector that is interjacent to the vector and the given vector.
getVektorTo(Vektor)	Returns a Vektor from this Vektor to the global vector. Note that this method is equal to v.sub(this).
mult(double)	Multiplies the vector with a constant factor.
pointAt (double, double)	Is a setter for the coordinates x and y.
pointAtPolar (double, double)	Sets the Vektor by polar data.
reset()	Sets the Vektor to (0,0).
rotate(double)	Rotates the vector by the given angle.
scalar(Vektor)	Scalar product of the two vectors. Changes the instance.
sub(Vektor)	Subtracts the given vector from *this* and retrieves the result as a new instance.
subFromThis(Vektor)	Subtracts the given vector from *this*. Changes the instance.
***static* normalize (double)**	Converts an angle such that it lies between [180,-180) degree.

StraigtLine

This class models a straight line in $\Re^2$. A straight line is represented by an equation of the form: y = mx + n.

Class

Name	robocup.component.geometry.StraightLine

Description	This class represents a line in the 2D space. It can be instantiated using a gradient plus constant (m and n), a position (Vektor) and an angle, or two positions.

Attributes

m	The gradient of the straight line.
n	The intersction height with the y-axis.

Methods

getDistanceToPoint (Vektor)	This method calculates this line's distance to a given point.
getIntersectionPoint (StraightLine)	This method calculates the intersection of this line and another given line. Returns *null*, if lines are parallel.
getSlope()	returns the slope of a line in degree.
getVektorToLine (Vektor)	This method calculates the shortest Vektor from a given point to a line.
isOnLine (Vektor)	This method checks if the given Vektor is on line.

Circle

This class represents a circle, which is defined in terms of a center vector and its radius.

Class

Name	robocup.component.geometry.Circle
Description	This class represents a 2D circle. It can be instantiated using either a vector and the radius, or by three vectors whereas all of them have to be placed on a line.

Attributes

center	The circles center (Vektor).
radius	The circles radius.

Methods

getArea()	Retrieves the area of this circle.
getCircumference()	Retrieves the circumference by $2\Pi * radius$
getDiameter()	returns the circles diameter..
getIntersectionPoints (Circle)	Returns the intersection points of these two circles. If none exist, the returned list is empty.
getIntersectionPoints (StraightLine)	Returns the intersection points of this circle with a StraightLine. If none exist, the returned list is empty.

liesWithinCircle (Vektor)	returns true, if a position is inside the circle.

Remarks

Some improvements of the geometry package may be the implementation of a meaningful class hierarchy and of composite objects, where intersection and union can be defined. On the other hand, calculating with areas of any shape might be not very performant, hence it was not included at the moment.

6.3 World Model Content

The simplified world of a Robocup soccer game consists of a few elements, whereas some of them can be known with a different level of detail. Beside some other data, the robocup world has players, a ball and a field on which two goals are located. In the simulator, the field is sourrounded with *Flags*, which have a fixed known position. These can be used by the players to orientate. Goals are also modelled as flags. As shown in figure 6.4, a flag extends a *StaticObject*, which generally is a thing with a fixed position. The *Flags* id can be used to distinguish them. *Ball* and *Player* have the additional property of being mobile, hence they have a speed as common attribute, which they both inherit from the class *DynamicObject*. *Players* on the other hand have some more attributes, e.g. the body- or neck-direction, stamina, etc. Note, that this class diagram is very similar to that presented in [7].

6.3.1 Class Descriptions

This section summarizes the main classes together with some their important attributes and methods. Some of them are dependent on the geometry mentioned in the previous section, others implement the behaviour of *DynamicObjects* from the soccer server (see in [7] for the dynamics of these objects). These can be used to forecast certain situations, e.g. where the ball is in n cycles, assuming it has a certain speed and will not be kicked in the mean time. Before changing any of these objects during a running simulation, make sure they are copies of the corresponding objects of the world model. For instance, if the position of an object is changed by a calculation, this change will not reflect the actual perceived environment any more, and other classes using this information later on might produce strange results. As with the class *Vektor*, use the methods *cloned()* and *copy()* and work with copies instead.

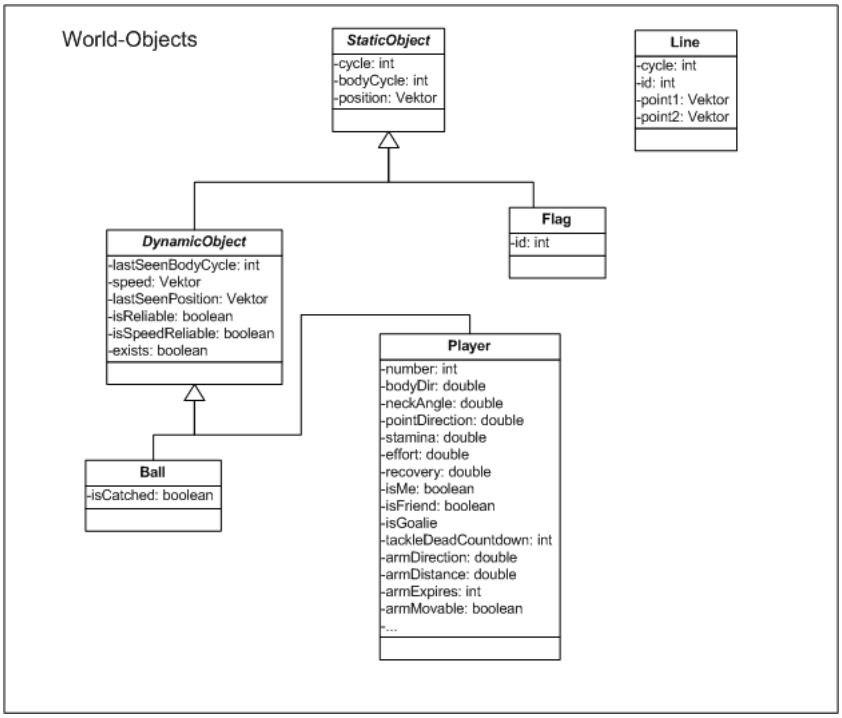


Figure 6.4: Class hierarchy of the important objects.

StaticObject

This class is an abstract representation for each things, which has a certain position.

Class

Name	robocup.component.worldobjects.StaticObject
Description	This is an abstract class, which is extended directly by *DynamicObject* and *Flag*.

Attributes

cycle	The game cycle, in which this object was last noticed.
bodyCycle	The body cycle (corresponds to the amount of sense body infos received), in which this object was last noticed.
position	The position of the object (Vektor). Note, that usually this is the thought position, derived from visual and acustic information. Only the coach agent knows exact positions.

Methods

getAngleTo (StaticObject)	Angle from the position of this StaticObject to the global position of the specified StaticObject.

getAngleTo (Vektor)	Angle from the position of this StaticObject to the global position of the specified Vektor.
getDistance (StaticObject)	Distance from the position of this StaticObject to the global position of the specified StaticObject.
getDistance (Vektor)	Distance from the position of this StaticObject to the global position of the specified Vektor.
getVektorTo (StaticObject)	Returns the Vektor which points from this StaticObject to the position of the specified StaticObject.
getVektorTo (Vektor)	Returns the Vektor which points from this StaticObject to the position of the specified Vektor.

Flag

This class represents a marker on the soccer field, which is visible by the agent. For the occurrence of the flags, see figure 6.1. The global positions of all flags are known by the agents, and the goals are implemented within the soccer server as flags as well. It is a direct extension of the class *StaticObject*.

Class

Name	robocup.component.worldobjects.Flag
Description	Instantiation of this object is done by giving a cycle, an id and a position to the constructor. Note, that usually no further instances are needed, because all known flags are implemented within the *SConf* class.

Attributes

id	The flags id, which is used to distinguish them.

Line

This class represents a field line, but also has some methods for geometric calculations. [4] Since this class is not an extension of *StaticObject* (because it has no unique position), there are some attributes with the same meaning as there.

[4] Perhaps an inheritance relation is adequate here.

Class

Name	robocup.component.worldobjects.Line
Description	This is a field line. The server knows five lines: The center line and four field border lines. A line is instantiated by giving two vectors, or a cycle, and id plus two vectors.

Attributes

cycle	The game cycle, in which this object was last noticed.
id	The line id, which is used to identify its position.
point1	The first point, which defines the line.
point2	The second point, which defines the line.

Methods

changeDirection()	Swaps the two points of the line, such that the direction is changed.
getAngle()	Returns the angle from *point1* to *point2*.
getDistanceToPoint (Vektor)	The shortest distance from the line to the vector.
getGradient()	Returns the gradient of this line.
getIntersectionPoints (Circle)	Returns a list of intersection points of this line and and the given circle. If none exist, the list is empty.
getIntersectionPoint WithLine(Line)	Returns a list of intersection points of this line and and the given one. If the lines are parallel, the list is empty.
getPerpendicularBisector ()	Returns the perpendicular bisector of the side.
isOnLine(Vektor)	Returns true, if the given point lies on the line.

DynamicObject

This class is an abstract representation for any mobile objects, i.e. for *Players* and the *Ball*. Hence it contains an attribute for their speed.

Class

Name	robocup.component.worldobjects.DynamicObject
Description	This classe extends the *StaticObject* by adding velocity attributes and others. It is also abstract, and involves all common aspects of a *Player* and the *Ball*.

Attributes	
exists	Is true, if the object exists. E.g. in training scenarios only a few players may participate. Hence some of them are marked as non-existant in the world model.
isReliable	Is true, if the agent beliefs, the corresponding knowledge about a player or ball is to some degree accurate. This will not be the case, if the object wasn't seen for longer time, or wasn't seen at an expected position.
lastSeenPosition	Stores the last seen position of this object. The actual position may be actualized due to acustic messages, which may not be accurate in some cases.
lastSeenBodyCycle	The time, when the object was seen latest.
speed	The objects speed.
speedReliable	Is true, if the speed was perceived, and could't change in the meantime.

Methods	
extrapolate (int, double)	Extrapolates Dynamic Objects based on their speed and their decay. This can be used to see, what an object may look like in n cycles, if it isn't affected by any actions (i.e. if the ball isn't kicked, or a player doesn't dash). This and the corresponding methods of *Player* and *Ball* are used extensively in the most Situations of the Prophet.
getFinalPosition (double)	Returns the final position (Vektor) of an object, assuming no acceleration accurs. The parameter (decay) stands for the speed decreasing factor in each cycle, which depends on the player type (default 0.4) or the ball (default 0.94).
getFuturePos (int, double)	Returns the position of the object in n cycles, if no acceleration occurs.
getFutureVelocity (int, double)	Returns the speed of the object in n cycles, if no acceleration occurs.

Ball

This class is a representation of a ball. Note that there is always one ball in a simulation, no more, no less.

Class

Name	robocup.component.worldobjects.Ball
Description	This class extends the *DynamicObject*, and has a few new attributes and methods for the balls state. Since its constructor requires some parameters (cycle, bodyCycle, position, speed), there exists a static method to create a default instance, called *getDummyInstance()*.

Attributes

isCatched	Is true, if the ball was catched by the goalie. This might not not always be known by the agent.

Methods

cloned()	Returns an exact copy of the given ball.
copy(Ball)	Writes all values of the given ball (e.g. speed, position, etc.) into this ball-object.
predictBallAfterAction (Player, Action)	This method predicts the next ball state, assuming the given player executes the given action. It returns a new ball object, i.e. this ball is not changed.
predictThisAfterAction (Player, Action)	This method predicts the next ball state, assuming the given player executes the given action. It directly changes this ball-object, and is hence more performant.

Player

This class is the representation of a player. Here, the detail of knowledge is quite different. An agent knows quite much about himself, i.e. his stamina or neckangle, which he could't really know about any other player. Additionally, the available information about his teammates might be somewhat higher than that about opponents, because of communication and the robocup rules (e.g., if the coach changes the player type of an agent, every teammate is informed about that, which doesn't count for opponents). Information on position and speed vary by distance and visual perception quality as explained in [7].

Class

Name	robocup.component.worldobjects.Player
Description	This class extends the *DynamicObject*. Since its constructor requires many parameters , there exists a static method to create a default instance, called *getDummyInstance()*.

Attributes

armDirection	The direction the arm was set with the point-to action.
armDistance	The distance the agents arm is pointing to. This is only known by the agent himself, because visual information only contain the pointing direction.
armExpires	The amount of cycles the arm stays pointing to a certain direction.
armMovable	Is true, if the arm is movable. After setting a point-to direction, the arm has to point at least for 5 cycles (default) there.
attention	The player, an agent sets his attention to using the attentionto action.
bodyDir	The absolute body direction.
effort	The effort factor refers to how effective dash actions may be executed. This can only be known for oneself.
isFriend	Is true, if an agent belongs to the same team.
isGoalie	Is true, if the agent is a goalie. This is not always player with number 1, but can be perceived via visual information.
isMe	Is true, if the player is the one controlled by the agent himself.
number	The tricot number of the agent. This is given by the server after initializing a connection, Numbers are always from 1 to 11.
pConf	The heterogeneous configuration for each player is stored in this class (PConf). For instance, the dash-power or the extra stamina is stored here. The default type (type 0) is set initially, and after the coach changed them, they were updated by the players.
recovery	This is a factor, which determines, how effective stamina regeneration is. This is only an estimated value for oneself, since it is not communicated by the server.
stamina	Stamina is the energy source of an agent, which is consumed by dash-actions and can be regenerated each cycle to the maximum value (default 4000). If stamina is low, the agent can't run with full speed.
tackleDeadCountdown	The amount of cycles, an agent is unable to move after a tackle-action was executed. The default value after tackling is 10 cycles. Note, that tackling doesn't always succeed, which increases these costs additionally.

Methods

canCatch (Ball, WorldModel)	Returns true, if the player is a goalie, and the ball is inside the catchable area (ca. 2m distance).
canKick(Ball)	Returns true, if the ball is within the kickable margin of the player (ca 1m).
canKick (Ball, double)	Returns true, if the ball is within the kickable margin of the player minus a security.
cloned()	Creates and returns an exact copy of that player object.
copy(Player)	Sets the values of the given player to this one, such that it is an exact copy.
correctDashPower (Action, STAMINA_LEVEL)	If the action is a dash-action, the dash-power will be reduced such that the given stamina level is not underrun. This depends on the heterogeneity of the player.
equals(Object)	Players are equal here, if they have the same number and belong to the same team.
getActualKickPowerRate (Ball)	This method calculates the actual power rate for a kick command considering the ball's location relative to the player. This is lower, if the ball is behind him or when its distance is higher.
getAngleAbs (double, RELATIVE)	Returns the absolute angle from the specified relative angle, which is either relative to the head or relative to the body, defined within the second parameter.
getAngleAbs (StaticObject, RELATIVE)	Returns the absolute angle to the relative position of the specified StaticObject. If the position of the StaticObjct is relative to the head the enum (second param) should be RELATIVE.TO_HEAD, else is should be RELATIVE.TO_BODY.
getAngleAbs (Vektor, RELATIVE)	Returns the absolute angle to the relative Vektor. If the Vektor is relative to the head the second parameter should be RELATIVE.TO_HEAD, else it should be RELATIVE.TO_BODY.
getAngleForTurn (double)	This method calculates the angle that has to be given to a turn command in order to turn the body by a given angle. This depends on the agents speed and his heterogeneity.
getAngleRel (double, RELATIVE)	Returns the relative angle from the specified absolute angle. If the returned angle should be relative to the head the second parameter should be RELATIVE.TO_HEAD, else it should be RELATIVE.TO_BODY.

getAngleRel (StaticObject, RELATIVE)	Returns the relative angle from the absolute position of the StaticObject. If the returned angle should be relative to the head second parameter should be RELATIVE.TO_HEAD, else it should be RELATIVE.TO_BODY.
getAngleRel (Vektor, RELATIVE)	Returns the relative angle to the absolute Vektor. If the returned Vektor should be relative to the head the second parameter should be RELATIVE.TO_HEAD, else is should be RELATIVE.TO_BODY.
getKickDistance()	Returns the maximum distance a ball may habe to the player, if this should be able to kick it. This is kd = (player-size + kickalble-margin + ball-size).
getMaxBalAccelVektor (Ball, Vektor)	Returns a vector containing the maximum acceleration, which can be set on the given ball, when trying to kick to the given position.
getMaxBallAccelOptimized (Ball, Vektor)	Same as above, but this method is more efficient.
getMaxTurningAngle1()	The maximum angle, a player can turn using a singe turn action in a certain situation (depends on his speed and his heterogeneity).
getMaxTurningAngle2()	The maximum angle, a player can turn using two turn actions in a certain situation (depends on his speed and his heterogeneity). It is assumed, that a player can turn into any direction with three turn-actions.
getPosAbs (StaticObject, RELATIVE)	Returns an absolute position denoted by the relative position of the specified StaticObject. The second parameter specifies if the position of the StaticObject is relative to the head or relative to the body.
getPosAbs (Vektor, RELATIVE)	Returns an absolute position denoted by the specified relative Vektor. The second parameter specifies if the Vektor is relative to the head or relative to the body.
getPosRel (StaticObject, RELATIVE)	Returns a relative position from the absolute position of the StaticObject either relative to the head or relative to the body. This is specified by the second parameter.
getPosRel (Vektor, RELATIVE)	Returns a relative position denoted by the specified absolute Vektor. The second parameterspecifies if the returned Vektor should be relative to the head or relative to the body.
getPowerForDash (Vektor)	This method calculates the power that should be given to a dash command in order to come as close to the position as possible.

getPowerForDash (Vektor, int)	This method calculates the power that should be given to a dash command in order to come as close to the position as possible. it differs from the simple version of this method in that the drift is considered. (i.e.: if you wan't to be at a position in 2 cycles it doesn't make sense to dash full first and stop afterwards (looses 1 action) instead the speed resulting from dashing + the drift in the next cycle should bring you to the position).
predictPlayerAfterAction (Action)	Extrapolates the player assuming he will execute the given action. The result is a new instance, which will be returned by this method.
predictThisAfterAction (Action)	Extrapolates the player assuming he will execute the given action. Here, the actual object is modified.
predictThisAfterDash-Action (DashAction)	Extrapolates the player assuming he will execute the given dash-action. Here, the actual object is modified.
predictThisAfterPointTo-Action (PointToAction)	Extrapolates the player assuming he will execute the given pointto-action. Here, the actual object is modified.
predictThisAfterState (AbstractState)	Extrapolates the player assuming he will execute the given state, which might result in different actions. Here, the actual object is modified.
predictThisAfterTurn-Action (TurnAction)	Extrapolates the player assuming he will execute the given turn-action. Here, the actual object is modified.
tackleFailureProbability (Vektor)	Returns the probability of failure when trying to tackle the ball, whereas the given vektor is the ball-position.
tackleSuccessProbability (Vektor)	Returns the success-probability when trying to tackle the ball, whereas the given vektor is the ball-position. This simply is sp = 1 - *tackleFailureProbability*.

Remarks

In the tables of classes above, some things were omitted, because only the most important attributes and methods should be presented here. For instance, most attributes have setter- and getter-methods, which weren't listed here. Others have a very special meaning and their effects are observable not very often, such that these were omitted as well.

6.4 Structure of the WorldModel

The WorldModel contains data of different types and precision. In order to achieve some kind of structure, certain categories of knowledge are grouped and added to subparts of the WorldModel, sharing a common interface. This interface is called IModel, and provides abstract methods for updating the corresponding model due to the receivement of info-objects. In detail, the following models exist:

- MeModel - All data concerning the self-agent.
- PlayersModel - All data concerning all players, including the self-agent.
- BallModel - All data concerning the ball.
- PlayModeModel - All knowledge about the current playmode.
- CLangModel - Storage for all received CLang statements.
- CollisionModel - Model to detect collisions and to update the WorldModel correspondingly.
- NeckRotator - Stores seen parts of the field in order to determine next looking directions (see therefore 7.3).

An overview about the models and their content is given in Figure 6.5. Generally they can be seen as the link between the objects and the WorldModel. The most important ones are the PlayersModel, the MeModel and the BallModel, which are briefly described next:

6.4.1 PlayersModel

PlayersModel can be found in *robocup.component.worldmodel.PlayersModel*.
The PlayersModel represents the part of the world that affects other players. All the players are stored in one of two Vectors "'teammates"' and "'opponents"'. An array containing all players is named "'allPlayers"'. It is possible to calculate relations between players like "'who is the closest player to player X"' etc. As a result you will extract a single player for further evaluations or iterate on an array of players which fulfill the requested criteria (like "'get my teammates / opponents"').

6.4.2 MeModel

MeModel can be found in *robocup.component.worldmodel.MeModel*.
The MeModel references the class Player. In addition to the basic methods gained from Player there are methods which refer to the actual player thread (according to every player having

its own thread). There are several methods which refer to the players position, body and neck direction, stamina, player type, number, team membership etc. Obviously the PlayersModel behaves comparably to the MeModel. But there is one major difference between them. Every player only receives a part of the whole information of the current game e.g. he can only see or hear other players who are in range and so on. Unfortunately the information is distorted by noise which leads into increased expenditure for calculations. This behaviour may lead into an unclear state of information which means that there may be no reliable data in the PlayersModel. However the MeModel will supply the player (itself) with the required data at any cycle.

6.4.3 BallModel

BallModel can be found in *robocup.component.worldmodel.BallModel*.
The BallModel references Ball. Besides some basic methods gained from Ball it implements methods for updating the ball after some kind of incoming information. Most methods of the WorldModel that affect the ball are forwarded to this class.

Finally there are some classes in figure 6.5 which were not explained yet. These are ShortTermMemory, LongTermMemory, PConf and SConf.

6.4.4 ShortTermMemory

ShortTermMemory can be found in *robocup.component.worldmodel.ShortTermMemory*. This class represents the short term memory of a player and is used for optimization purposes. All the information stored here only lasts for one cycle. The purpose of the ShortTermMemory is to save computational power by storing a value, which is called very frequently from various methods and providing access to this value for further calls. E.g. the next position of the player as it is predicted is first computed and then stored in the ShortTermMemory only for the first call in every cycle. For further calls in the same cycle the position just has to be read out of the ShortTermMemory.

6.4.5 LongTermMemory

LongTermMemory can be found in *robocup.component.worldmodel.LongTermMemory*. The LongTermMemory is currently nearly deactivated because the situations used by the class Prophet do its job now. It only stores the arrival times of the three different message types.

6.4.6 PConf and SConf

The classes PConf and SConf contain player and server related constants. The former holds all data referring to the heterogeneous player types, containing the values of e.g. KICK-

ABLE_MARGIN, MAXPOWER or STAMINA_MAX. These values are set dynamically after connecting to the soccer server, and have a big impact upon the abilities of the agents. See chapter 11 for details about the assignment of heterogeneous player types.
The class *robocup.component.SConf* contains simulation related constants. These are the same for all players.

6.5 Updates of the WorldModel

The update of the WorldModel is done for each info object, which is produced by the parser after a message was received from the server. These objects contain different information, as shown in Figure 4.2 of the parser chapter. In the following, the update methods are explained.

6.5.1 The update methods for information types

The WorldModel component represents the current world of an agent as it is perceived through different infotypes. There are tree different infotypes like mentioned before:

- aural info (robocup.component.infotypes.AuralInfo),
- body sense info (robocup.component.infotypes.SenseBodyInfo) and
- visual info (robocup.component.infotypes.VisualInfo).

Every time a new info is perceived by the RobocupAgent, it is passed on to the WorldModel, which updates itself according to the new infomation. The WorldModel then forwards the information to MeModel, BallModel and PlayersModel. At first the MeModel computes the position, speed, body- and neckangle, stamina and other playerspecific values of the player according to the incoming info and the last performed action. The precision of the player's position and its speed is very important, because the calculation of position and speed of other objects are based on them. After this is done, the BallModel computes the position and speed of the ball. Finally the position and speed of other players are updated.

Figure 6.6 shows the information flow if a new information arrives at the WorldModel.

The following subsections give an overview of the methods called by WorldModel.update(Info). For detailed information on how to update the world model see [8].

6.5.2 The body sense update methods

MeModel.updateSense(SenseBodyInfo, int)

This method is called by the WorldModel.update(Info). It updates the player according to the player's position and speed of the last cycle and the last executed actions. The last executed

actions can be accessed by SenseBodyInfo.getLastSentActions(). An estimation of the player's current speed relativ to the player's head angle (= body angle + neck angle) is included in the parameter of type SenseBodyInfo.

BallModel.updateSense(SenseBodyInfo, int)

This method updates the ball according to the position and speed of the ball of the last cycle. If the player sent a kick action in the last cycle, the resulting acceleration vektor is added to the speed vektor of the ball in the last cycle previously. Kick actions of other players are not included in the calculation, because the player doesn't know anything about actions sent by other players. He only "sees" (by receiving a visual info) the position of the ball in every cycle.

PlayersModel.updateSense(SenseBodyInfo, int)

This method updates the positions and speeds of all other players, according to their positions and speeds in the last cycle. Therefore actions other players executed are disregarded.

6.5.3 The visual update methods

The visual update methods form probably the most important part of the WorldModel. All activities of the player depend mostly on the visual input he receives from the environment.

MeModel.updateVisual(VisualInfo, int)

This method's objective is to calculate the position of the player. The VisualInfo object holds arrays of information objects about the field flags and lines the player sees. Based on this information the position calculation algorithm is invoked.

BallModel.updateVisual(VisualInfo, int)

The goal of this method is to update all information about the ball every time a VisualInfo arrives. The position of the ball is calculated and all attributes concerning the reliability are reset. Furthermore, the method is supposed to calculate the speed of the ball, if needed. The server does not send the velocity of the ball, if the player is too far away from the ball. In such case the speed vector needs to be predicted. This has not yet been implemented. If the ball has not been seen, i.e. there is no BallInfo object in the received VisualInfo, this method handles all reliability issues. For example, if the ball has not been seen more than WM_V_AGE_CONFIDENCE number of cycles, the isReliable attribute is set to false.

PlayersModel.updateVisual(VisualInfo, int)

Here the information about other players on the field is being updated. The PlayersModel possesses three Vectors with players:

- teammates - holds players of the player's team
- opponents - holds the opponents of the player
- unknown - holds players whose team can not be determined

These Vectors are initially filled due to the amount of received information within the VisualInfo. After this, the identity of unknown players have to be guessed, which is a complicated task at all.

6.5.4 The aural update methods

Some aural information arrives as PlayerSayInfo which will be handled in *robocup.component.speechacts.MessageFactory*. The MessageFactory evaluates the content of the information for the different models.

MeModel.updateAural(AuralInfo)

Currently this method only computes an InitInfo, which is sent once during the initializing process.

BallModel.updateAural(AuralInfo)

The intention of this method is to use information regarding the play modes and referee messages for predicting the position and speed of the ball.

PlayersModel.updateAural(AuralInfo)

The PlayersModel uses the MessageFactory like mentioned above. Received messages contain for example the position of one of the other players, which are updated here.

6.5.5 NeckRotator (robocup.component.NeckRotator)

This class computes the best visual angle for a player in a certain state of game. There is a seperate chapter discussing the Neckrotator.

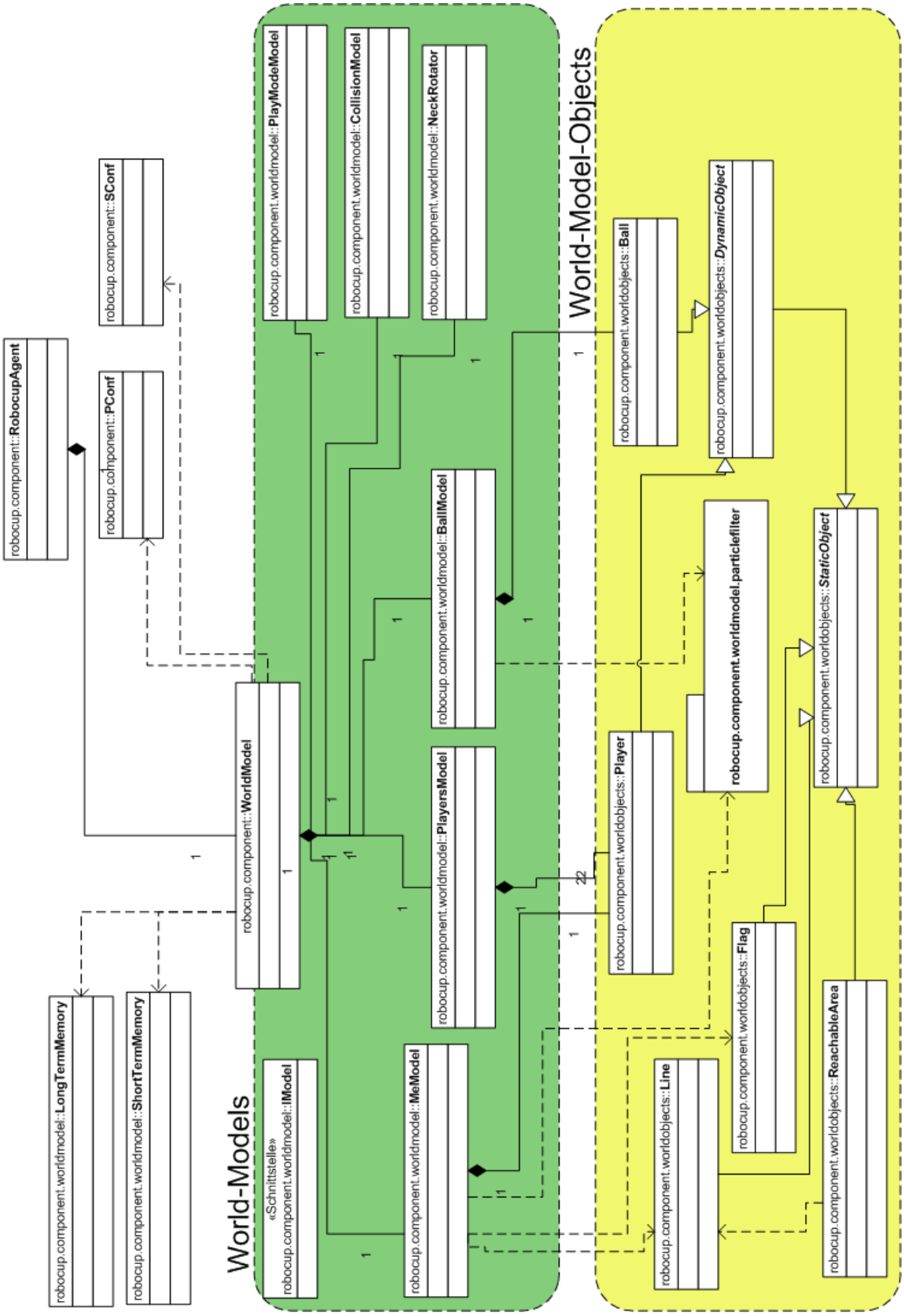


Figure 6.5: Part of the class model including most important references of the WorldModel.

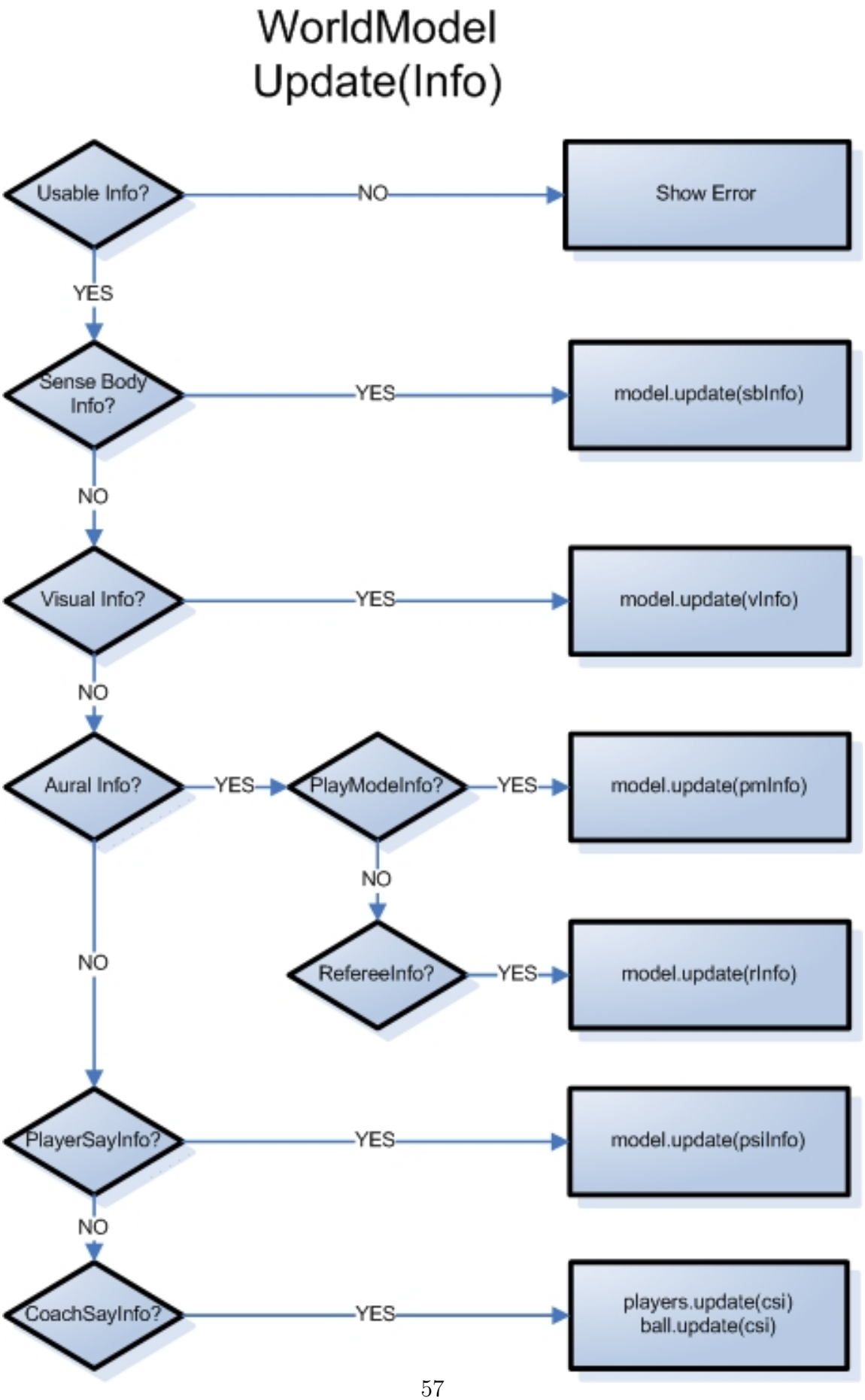


Figure 6.6: Information flow of WorldModel Update()

6.6 Particle Filter

6.6.1 Overview

The particle filter is another step towards an optimised solution to the position estimation problems that occur in the robocup project such as estimating the players own position or the balls position. This chapter first sketches the idea behind the particle filter and gives an overview over the current implementation. After the theoretical part there will be an overview over the involved classes an their public methods. Finally I will provide some outlook on how the usage of the particle filter may be improved in the future.

6.6.2 The particle filter in theory

In the previous chapters we learned what kind of Information we get from the server. The particle filter is an approach to use as much of the information we get as possible to narrow down the player's position. The information as we learned is quantised so every bit of information gives us a range of possible positions. As we only have to deal with quantisation noise we can assume that our actual position must be inside all ranges we can determine with the information we got from the server. As we combine all the information the remaining ranges get quite complicated and that is where the particle filter comes into play. By simulating the two dimensional shape that is given by all the determined ranges, using discrete particles, the particle filter gives a good estimate for the actual shape. Now we can take the barycentre of our shape as position estimate.
During the following paragraphs I will explain how different types of information are processed in the current particle filter implementation. However if you want to know more about the theory behind the particle filter see Chapter 6 of [8] and follow the corresponding bibliography.

Particle filter for player's position

Visual Info

A visual-info gives us a lot of information about the players own position. Most important to narrow down his position are the seen flags. Every flag gives us a region of possible positions in the shape of a ring-sector. The region can be described as follows. The flags position which is known to the player is the centre point of the ring (x_{flag}, y_{flag}). The ring has got an inner and an outer radius d_{min} and d_{max} that can be derived from the quantised seen distance d_{quant} included in the visual-info. Further it has got a starting and ending angle ϕ_{min} and ϕ_{max} that can be derived from the seen direction included in the visual-info plus the payers head-angle. To obtain this head-angle I use a method that already was implemented, so I will only mention that it is computed with an error of ± 0.5 degrees.
As we now intersect all of the obtained regions the remaining region gets smaller and smaller so that our position estimate gets better and better. Figure 6.7 shows two of the described

region intersecting one another and thus narrowing down the possible position of our player. I will now list all relevant equations that are necessary to compute all the values that describe

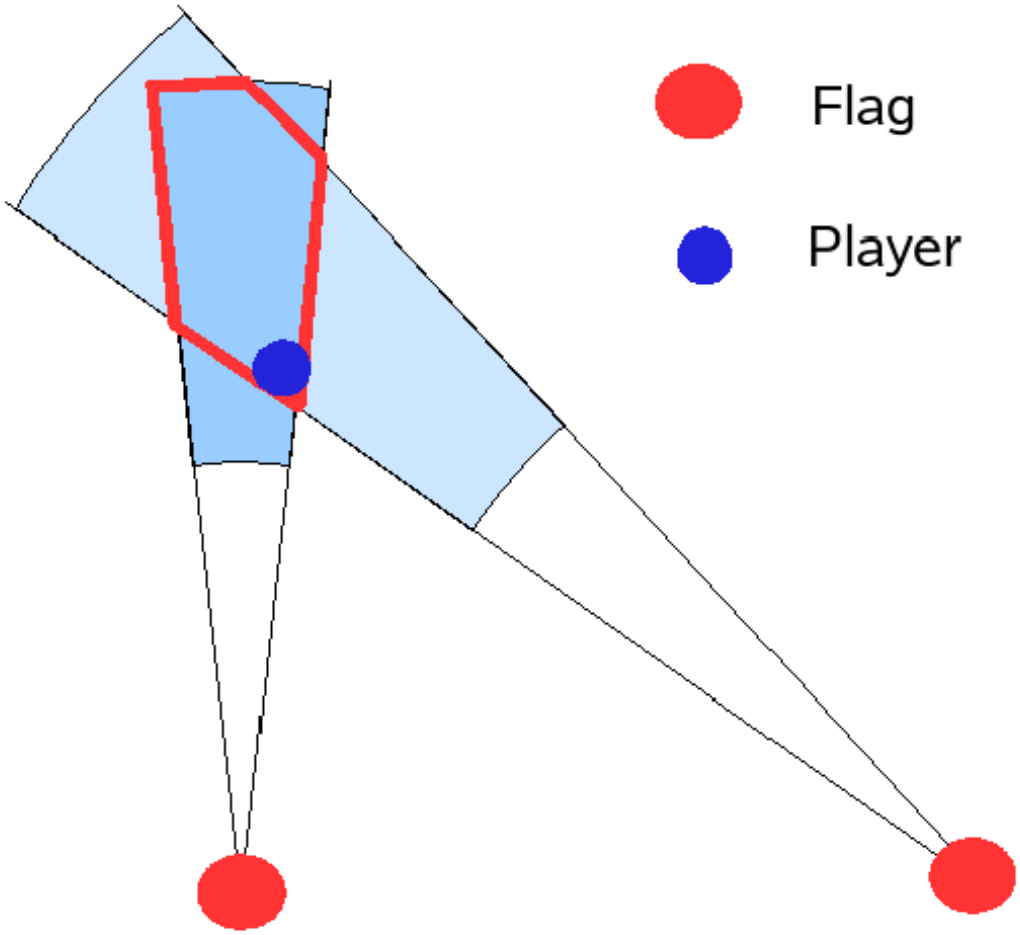


Figure 6.7: Overlapping ring sections derived from two seen flags.

our region. Most of them are already mentioned in the UVA master-thesis [8], but some I added to make this document a complete reference. The seen distance of the flag is quantised depending on the actual distance like this:

$$d_{quant} = Quantize(e^{Quantize(ln(d),StepValue)}, 0.1) \tag{6.1}$$

with

$$Quantize(x, step) = rint\left(\frac{x}{step}\right) \cdot step \tag{6.2}$$

and $StepValue = 0.01$ in the case of seen flags. Transforming equation 6.1 with

$$rint^{-1} = rint(x) \pm 0.5 \tag{6.3}$$

we get our inverse quantisation formula

$$d_{max/min} = e^{\left[rint\left(\frac{ln\left(\left(rint\left(\frac{d_{quant}}{0.1}\right)\pm 0.5\right)\cdot 0.1\right)}{0.01}\right)\pm 0.5\right]\cdot 0.01} \tag{6.4}$$

which is much more readable like this

$$dummy = ln\left(\left(rint\left(\frac{d_{quant}}{0.1}\right)\pm 0.5\right)\cdot 0.1\right)$$
$$d_{max/min} = exp\left(\left(rint\left(\frac{dummy}{0.01}\right)\pm 0.5\right)\cdot 0.01\right).$$

Now we take care of the the angles. As I mentioned before the quantised angle ϕ_{quant} is a combination of the seen direction ϕ_{seen} and the player's head-angle ϕ_{head}. ϕ_{seen} is quantised using 6.2 with a step-value of $step = 0.1$. As ϕ_{head} has an error of ± 0.5 the overall error of ϕ_{quant} is ± 1.0 degrees. Now we have to take into account that the flags position is the centre of our polar system and not the player's position, so we add 180° to our seen direction. All this leads to

$$\phi_{max/min} = 180^\circ + \phi_{seen} + \phi_{head} \pm 1.0. \tag{6.5}$$

Sensbody Info

The relevant information of the sensbody-info is the felt speed of the player. From this speed we can tell where all of our particles have to move in the next cycle. In fact the particles are translated in the beginning of a cycle using the speed that was determined in the last cycle. As the speed information again is quantised we get a region where the player could have moved during the last cycle. This region again is a ring-sector with the players old position as the centre, the minimum and maximum possible amount of speed ($d_{min/max}$) as inner and outer radius and the minimum and maximum possible direction of the speed ($\phi_{min/max}$) as angle boundaries. To translate the particle set now every valid particle is taken as the centre of such a region and a new random particle is chosen from the resulting region. Luckily direction and amount of the felt speed are quantised in a less complicated manner then the seen flag's visual-info. Namely:

$$QuantizedAmountOfSpeed = Quantize(AmountOfSpeed, 0.01) \tag{6.6}$$

and

$$QuantizedDirection = Quantize(Direction, 1.0^\circ) \tag{6.7}$$

which leads to

$$d_{max/min} = QuantizedAmountOfSpeed \pm 0.005 \tag{6.8}$$

and

$$\phi_{max/min} = QuantizedDirection \pm 0.5^\circ. \tag{6.9}$$

Particle filter for ball position

The particle filter for the ball position works in a different way. We get much less information compared to the particle filter for the player's position. That is why we will have to take more past information into account. As soon as the the ball is seen by the player we initialise a set of particle with random values of position and speed from ranges that can be derived from the quantised visual-info. Yes the particles are four-dimensional now, two for the position coordinates and two for the speed. Every cycle the particles are translated according to the known server dynamics. If a new visual-info is received all particles are checked whether they still fit into the derived region and are dropped if the don't. If the particle filter runs empty a new set of particles is initialised. This should only occur if the ball was kicked and thus does not move according to the server dynamics. Unfortunately this is not the case.

Visual Info

The region for the ball's position is very similar to the region derived from a seen flag in the particle filter for player's position because the seen distance is quantised using equation 6.1 and the seen direction using 6.2 with a step-value $step = 0.1$. The differences are that $StepValue$ in 6.1 is equal to 0.1 this time and the players position is the centre of our ring-sector so that we can also omit the 180° in 6.5.

$$d_{max/min} = e^{\left[rint\left(\frac{ln\left(\left(rint\left(\frac{d_{quant}}{0.1}\right)\pm 0.5\right)\cdot 0.1\right)}{0.1} \right)\pm 0.5 \right]\cdot 0.1} \tag{6.10}$$

$$\phi_{max/min} = \phi_{seen} + \phi_{head} \pm 1.0 \tag{6.11}$$

The region that describes possible speed values is quite different. Although the values we get are called "distance changed" and "direction changed" they are not to be understood as polar but as Cartesian values. As both again are quantised the resulting region is a rectangle which unfortunately is rotated. I will now list all the equations that are needed to do the reverse quantisation and the transformations necessary to obtain a speed vector that conforms with coordinate system used in the dainamite robocup project. (If you compare them to the ones in the UVS Master Thesis [8] you might notice that some signs are different, this is due to the fact that someone liked the idea of confusing us a little by switching the names of the X- and Y-axis. But never mind the following equations use the dainamist system.)
First of all the reverse quantisation formulas:

$$\Delta d_{max/min} = d_{max/min} \cdot 0.02 \cdot \left(rint\left(\frac{\Delta d_{quant}}{0.02 \cdot d_{quant}} \right) \pm 0.5 \right) \tag{6.12}$$

$$\Delta \phi_{max/min} = rint\left(\Delta \phi_{quant}\right) \pm 0.5° \tag{6.13}$$

with

$\Delta d_{max/min}$	:	maximum/minimum possible distance change
$\Delta \phi_{max/min}$	:	maximum/minimum possible direction change
d_{quant}	:	Quantised seen distance as received from the Server
Δd_{quant}	:	Quantised distance change as received from the Server
$\Delta \phi_{quant}$	:	Quantised direction change as received from the Server

Now we have to transform distance change and direction-change values into into Cartesian coordinates conforming the dainamite coordinate system:

$$v_y = \Delta d \cdot e_{dy} \quad - \quad \Delta \phi \cdot \frac{\pi}{180} \cdot d \cdot e_{rx} \tag{6.14}$$

$$v_x = \Delta d \cdot e_{dx} \quad + \quad \Delta \phi \cdot \frac{\pi}{180} \cdot d \cdot e_{ry} \tag{6.15}$$

with

$$e_{dx} \quad = \quad sin\,(\phi_{quant}) \tag{6.16}$$

$$e_{dy} \quad = \quad cos\,(\phi_{quant}) \tag{6.17}$$

Now we have a speed-vector relative to the player's headangle and speed. Rotate the speed-vector by the current headangle and add the player's to obtain a global speed-vector.

Sensebody Info

From the sensbody-info there is no actual information drawn. But as in the particlefilter for player position all particles are translated when a sensbody-info is received. This is done by applying the known server's movement model to every valid particle in the particle set. Please consult [8] page 27 and following for detailed information on the movement model.

6.6.3 Important Classes

ParticleFilter

`Package: robocup.component.worldmodel.particlefilter.ParticleFilter`

The ParticleFilter-class is the main class for estimating the players own position. It provides public methods to add information such as visual or sense infos and methods to retrieve the result of the current position estimation. Further there is a method to reset the position if there is a better source from which the position may be determined.

Methods

addFlag With this method the visual-info of a seen flag is added to the particle filter. A region is initialised which helps to check whether particles are valid or out of range. Before the remaining particles are checked they will be re sampled. In the case that no particle remains valid after checking a new set of particles will be initialised.

translatePosition This method is called every time a new cycle starts on a received bodysens-info. It is handed the "felt" speed so that every particle can be translated to a new position.

getPosition With this method one can retrieve the current position estimate. It computes the mean of all currently valid particles every time it is called. So beware.

resetPosition May be used to set the particle filter to a specified position. Exactly one particle is created and set to the specified position. The number of valid particles is set to one.

Region

`Package: robocup.component.worldmodel.particlefilter.Region`

The Region-class is used to check particles validity. It gets initialised by the particle filter when ever quantized information is put in. The Region-class computes a region from this quantised data which can then be used to verify whether the position of a particle is inside the region or not. It can also be used to generate particles that are inside the represented region.

Methods

reinitRegionFlag This method is used to initialise the Region-object whenever a flag is seen. The origin is set to the flags position and a minimum and maximum value for the distance and angle in which the flag was seen is derived from the visual-info.

reinitRegionSpeed This method is used to initialise the Region-object on particle translation. A minimum and maximum value for the distance and angle the player could have moved is derived from the given "felt" speed. As origin one can set the position of the particle that is to be translated.

makeRandomInsideRegion This method can be used to obtain a position that is inside the represented region. e.g. when a set of particles is to be initialised.

setOrigin Sets the origin of the represented region.

insideRegion This method is used to check whether a particle is inside the represented region. It returns 1.0 if that is the case and -1.0 if otherwise. The fact that a double is returned may be considered legacy and may be changed into a boolean value in the future.

BallParticleFilter

`Package: robocup.component.worldmodel.particlefilter.BallParticleFiler`

The BallParticleFilter-class is the main class for the estimation of the balls position and speed. The public interface is roughly the same as the one of the ParticleFilter. Visual info can be supplied and estimations retrieved. Particles may be translated at the beginning of every cycle. The particle filter can be reset to a certain position and speed.

Methods

resetPosition This method resets the particle filter to a certain speed and position.

addVI This method is called when a visual info of the ball was received. Like in addFlag a region is initialised and remaining particles are validated or the set is reinitialised. In the case that no speed has been seen yet but a position was seen in the previous cycle the speed is initialised using the difference between the last and the current position.

getPosition With this method one can retrieve the current position estimate. It computes the mean of all currently valid particles every time it is called. So beware.

getSpeed With this method one can retrieve the current speed estimate. It computes the mean of all currently valid particles every time it is called. So beware.

isBallKicked This method returns true if the particle filter ran out of particles during the last addVI-call and thus ad to be reinitialised. This occurs if the ball "appears", in a VI, where it is not expected due to the translation model. This happens when the ball was kicked. Unfortunately it happens even more often so this is not a reliable source for information.

translatePosition This method is called every time a new cycle starts on a received bodysens-info. Every particle is translated to a new speed and position according to the known server dynamics.

BallRegion

`Package: robocup.component.worldmodel.particlefilter.BallRegion`

The BallRegion-class similar to the Region-class represents a region of possible positions. In fact this class represents the position region only speed-regions are represented by the BallSpeedRegion-class.

Methods

reinitRegionPos This Method is used to initialise a BallRegion-object whenever a visual-info of the ball is received. The origin of the region is set to the players position and a minimum and maximum value for the distance and angle in which the ball was seen is derived from the visual-info.

makeRandomInsideRegion Is used to obtain a random particle inside the represented region. This is done by choosing a random distance and a random angle from the possible ranges.

setOrigin Sets the origin of the region to the given position.

insideRegion This method is used to check whether a particle is inside the represented region. It returns 1.0 if that is the case and -1.0 if otherwise. The fact that a double is returned may be considered legacy and may be changed into a boolean value in the future.

BallSpeedRegion

`Package: robocup.component.worldmodel.particlefilter.BallSpeedRegion`

The BallSpeedRegion-class represents a region of possible speeds. Opposite to all other regions so far, speed regions are represented as two ranges of Cartesian coordinates relative to the players own speed and headangle which must be stored in the Region as well.

Methods

reinitRegion Is called by the BallParticleFilter when a visual-info is added and the speed of the ball was seen.

makeRandomInsideRegion Is used to obtain a random particle that is inside of the designated region.

insideRegion This method is used to check whether a particle is inside the represented region. It returns 1.0 if that is the case and -1.0 if otherwise. The fact that a double is returned may be considered legacy and may be changed into a boolean value in the future.

6.6.4 Outlook

The particle filter for player position estimation has worked out quite well after some problems in the beginning. We were able to reach the marks set in the UVA master thesis [8]. The particle filter for ball position estimation however is not as mature as the first one. This might be due to the fact that it is hard to verify the results for it is not very well in intergrated into the existing BallModel. There are some sideffects e.g. aural comunication that manipulate the player's ball-model the particle filter is not aware of yet. There is still some potential with the ball-particle-filter but as one can see from [8] it is not as prommissing as the particle-filter for player position estimation. Of cause every reader is welcome to contribute even better ideas for solving the position estimation problem as so to push the dainamite team further beyond the milestones set by the well known [8].

6.7 ReachableArea

6.7.1 What is the Reachable Area?

The ReachableArea-class represents the area that can be reached by a player in a given number of cycles. So, when the area is initialized with an player and a number of cycles, it provides some useful informations. The class can answer simple questions like: "Is a given point reachable by this player?" but also much more complicated questions like "What sequence of actions makes my player to reach point x under the condition that my player dashes at most 5 times backwards or under the condition that my player can watch the given point all the time?"

6.7.2 Usage of ReachableArea

To use the ReachableArea-class to obtain informations about the player is very easy. First step is to create an object of the ReachableArea. Next step is, to initialize the created object with values (sometimes done directly by a constructor). And last step is, to obtain the needed values.

Construction

For construction there are given some different types of constructors for the use in different situations.

- `public ReachableArea()`
  This constructor builds a ReachableArea without any information. Since the informations can be given to the area later, this constructor is used very often. The normal steps to work are than the same for each calculation of another player or situaion:

  1. Set new informations to area if needed
  2. obtain the needed informations for this setup

  This kind of usage is recommanded, if the informations for this Reachable Area change very often, because of less object-creation.

- `public ReachableArea(int cycles, int reactCycles, Player p, double reachDist)`
  By using this constructor, the ReachableArea is initialized directly. Here the arguments have the following meaning:

  - `cycles`
    Gives the time for the player to move. Of corse in a bigger time, the player can reach a bigger area.

- `reactCycles`
  Means how long the player needs to realize, where he wants to go. So this value can be used to increase the security of the calculation, but a high value means also that the area gets smaller.
- `p`
  This is the used player. By giving this player, our class gets informations about the `decay`, `turn_fact`, `max_speed` and `speed` of the player.
- `reachDist`
  Gives a buffer distance, since in many situations it is needed only to come *near* a given point and not directly onto the point. Here the `kick_dist` of a player is a useful candidate.

In general this constructor shouldn't be used very often, since it isn't performant to create a new ReachableArea-object for each calculation.

- `public ReachableArea(int cycles, Player p, double reachDist)`
  By using this constructor, the ReachableArea is initialized directly.The arguments are the same like in the constructor before, but the argument `reachDist` misses. It is set to zero by default here.

  In general this constructor shouldn't be used very often, since it isn't performant to create a new ReachableArea-object for each calculation.

Setting the area

As already said before, the general way of using the ReachableArea is, first to give informations to the object, and then to calculate new informations, based on the given ones. After this calculation is done, new informations (for other situations) can be calculated by setting other base informations. That's why in general there is no need of having more than one ReachableArea-object. It can be reused as as often as needed.

Corresponding to the 2 constructors with arguments there are 2 `setArea`-functions to set the same arguments.

Calculating informations

After setting some informations to the ReachableArea, one can obtain many useful informations about it.

Overview

Here is a complete list of what information can be calculated:

- Is a given point inside the area?

- What sequence of actions does the player need to do, to reach a given point under several conditions?
- What actions does the player need to do, to reach a given point under several conditions *without* calculating the following actions?
- Where are the intersection-points of a given line and the border of the area?

Is a point reachable?

By using `public boolean inArea(Vektor p, int backcycles)` can be calculated, if or if not the point `p` can be reached by the player in the given time. The argument `backcycles` is optional and limits the number of allowed cycles to dash backwards. If there is given only the point `p`, backcycles is zero as a default. If backward-dashing should be allowed in general, `backcycles` should be set to -1.

What actions are optimal to reach a given point as fast as possible?

There is an optimal way to reach a point as fast as possible. This way is first to turn the body in direction of the point, second, to dash to this point. This sequence of actions is calculateable by `public LinkedList<Action> getMovesToPos(Vektor pos, double reachedDist, double turnBuffer, boolean backwards, int backCycles, Vektor watchPos, Vektor turnToPos` The function is usable in many other variants with less arguments. Here we just discuss this variant, since it is the most general.

- `pos`
  This is the position to reach.
- `reachedDist`
  How close does the player need to come to the point?
- `turnBuffer`
  If the needed turn-angle is smaller than `turnBuffer` the turn will be ignored. It is useful to have the turnbuffer, since a turn with a very small angle has almost no effect and the dash would be the better alternative here.
- `backwards`
  If this is `true` the player is allowed to dash backwards, otherwise he cannot do any backward dashes and `backCycles` and `watchPos` will be ignored.
- `backCycles`
  It has an effect only, if `backwards` is `true`. This value limits the number of allowed back-dashes. -1 means that there is no limitation.

- `watchPos`
  This is a position that should stay watchable in the full movement. If such an position is given and `backCycles` allows to dash backwards, then this position determines if to dash forward or backward.

  Attention: In a sequence of moves it can happen, that the player passes the `watchPos`. In this case the player will move in that way, that allows him to watch the position at the beginning of the movement.

  This argument is important for the goalie. Using this ball-position here makes him, to not turn his back to the ball. The argument can be `null` if it is not needed.

- `turnToPos`
  This position is used, when the player has reached his position successfully. It makes the player to turn to this position afterwards.

What is the next best action to reach a given point?

This calculation is the same like the calculation of a sequence of actions, but after calculating the first action, this function finishes and saves performance.

What are the intersection-points of a line and the ReachableArea of a player?

These insersection-points become interesting, if one needs to find out, what positions on a line are reachable in general. Herefore use the function `public LinkedList<Vektor> getIntersectionPointsWithLine(Line l)`.

Attention: The resulting list of points is *not* sorted.

Example of usage

To show the usage of ReachableArea, here is shown a little piece of code to calculate if a given player p can kick the (also given) ball b in at most x cycles, and if he can reach it, give the optimal sequence of actions to reach it.

First we need to create a global object of the ReachableArea that can be used afterwards:

```
ReachableArea ra= new ReachableArea()
```

Now let us assume that b is a copy of the actual ball and p is the actual player.

```
boolean found= false;
LinkedList<Action> actionList= null;
```

```
for (int cycle= 0; (cycle <= x) && !found; cycle++) {
    ra.setArea(cycle, 1, p, kickDist);
    found= ra.inArea(b.getPosition(), true);
    if (!found) b.extrapolate(1, sConf.BALL_DECAY);
}
if (found) {
    actionList= ra.getMovesToPos(b.getPosition(), kickDist, 10, true,
        -1, b.getPosition(), sConf.GOAL_POS_OTHER=;
}
```

Here the program stays in the loop as long as the number of used cycles is smaller than x and there is no way found to reach the ball. The `reactCycles`-argument in `setArea` is set to 1 here to gain some security. Next there is checked if the ball is reachable when backward-dashing is allowed. When the loop is calculated and there is a way to reach the ball, then the action-list will be calculated. These actions in the list fulfill the following conditions:

- the player reaches the ball (or a position with a distance of maximal `kickdist` to the ball)
- there is no turn under 10 degree
- the player is allowed to dash backwards and has no limitation on how often he dashes backwards
- the player doesn't turn his back to the interception-point
- after reaching the desired position the player turns towards the opponent goal

6.7.3 Internal work of ReachableArea

Setting Informations

When informations are set to the reachable area, it divides these informations into some subinformations:

- The central point of the area is calculated by the actual position of the given player and his actual speed.
- Depending on the actual speed there is a change in the maximum turn that the player could do. So the fields `turn1`, `turn2` and `turn3` give how far a player could turn in one, two or three cycles. A value for `turn4` doesn't exist since in 4 cycles every player can turn the full 360 degree, even when he was on full speed before.

- Each turn that is done by the player means that he cannot dash. So depending on the number of turns, there are different distances to dash. So the fields `dash0` to `dash4` give the distance that the player can dash after zero to four dashes. These dash-distances become reduced by `reactDist`

  Attention: `a= ReachableArea(5,1,p,0)` and `b= ReachableArea(4,0,p,0)` are *not* the same, since they have the same dash-distances, but the central point of `a` was extrapolated one cycle more than the central point of `b`.

Is a point inside the area?

To calculate if an point is inside the area, first is calculated if the player is allowed to dash backwards, and what distance he could run backwards in maximum case. If this distance is smaller than the distance from the central point to the desired point, then it makes no sense to check backwards-dashing.

Now for each possible number of turns x is checked if the desired point has a distance smaller than `dash`x and is inside `turn`x and `-turn`x. If backwards-dashing should be checked too, the same calculation is repeated, but the desired angle is turned around 180 degree.

What actions make the player reach the desired point?

First step is to decide, if to dash forward or backwards to the point. This decision is made like it s described before, but if a special point should stay watchable all the time, the decision must not be the fastest way to reach the point.

Next step is to calculate the needed turnAngle and from this angle to create the first turn-actions. Afterwards dashes are created as long as the player is near the point or passed the point and at last there are some turns, if wished.

Where are the intersection-points between the area and a line?

To calculate all intersection-points the border of the ReachableArea is divided into its subparts (lines and parts of circles) and for each subparts is done a separate calculation of intersection.

6.7.4 What parts need to be improved?

- Since the functionality of this class is very similar to the Movements-class, they could be merged together.
- The argument `reachDist` in `setArea` and in the constructor is very similar to `reachedDist` in `inArea`. One of them could be replaced with the other one.

7 Action

The Robocup agents can perform actions, by sending special messages to the SoccerServer. These actions are divided in main-actions like turn, kick and dash and secondary actions like turning the neck. In Each cycle it is only possible to send one of the main actions, but as many of the secondary actions as needed. Most of these actions need some extra parameters to be given, like for example angle and power of a kick. Since these actions are at a very basic level, we combined them to more complex actions, which are easier to handle. To be able to save actions and to handle them easier, we created classes for each type of action. In the first part of this chapter we'll introduce these action classes. The second part of this chapter explains the combined actions, which are collected in so called action factories.

7.1 Action classes

The Action classes were made to handle actions. By creating objects for each action, we are able to collect actions and to decide for the best of them. We can change parameters of an action and at least form the string that must be send to the server to perform the action.

In the following, we'll explain each of the action classes for the player.

7.1.1 Action

class: `robocup.component.actions.Action`

This is the superclass of all actions. It contains an enum `TYPE` to mark the type of the action. On can avoid the slow `instanceof`-command by checking the `TYPE`. Next one can check if this action is a main-action by `TYPE.isMain`. Another feature is the `isWorthSending`-method which gives information about useless parameters like a dash with speed zero or a turn with turn-angle zero.

7.1.2 DashAction

class: `robocup.component.actions.DashAction`
constructor: `DashAction(int power)`

The `DashAction` is a main action that makes the player to dash into the direction of his body. The `power` argument can be a value between -100 and 100 where negative values mean backwards dashing. If `power` is 0, the action isn't sent.

7.1.3 TurnAction

class: `robocup.component.actions.TurnAction`
constructor: `TurnAction(double angle)`

The `TurnAction` is a main action that makes the player to turn his body aound `angle` degrees. The `angle` argument can be a value between -180 and 180. If `angle` is 0, the action isn't sent. The players real turn angle depends on his current speed. If he has no speed, the real turn angle is near `angle` (changes occur only by a small random effect). If the player has a high speed, the turn angle is reduced, what makes it harder to turn while the player is running fast.

7.1.4 KickAction

class: `robocup.component.actions.KickAction`
constructor: `KickAction(double power, double direction)`

The `KickAction` is a main action that makes the player to kick the ball, if it is in a small distance to the player. The ball is kicked into the given `direction` with the given `power`. The power is reduced by a factor depending on the position of the ball. Note that the resulting ball speed is combined from the kick of the player AND from the speed that the ball had before. The `power` argument can be a value between 0 and 100 and the direction argument can be an value between -180 and 180. The kick action isn't sent if the `power` argument is 0.

7.1.5 CatchAction

class: `robocup.component.actions.CatchAction`
constructor: `CatchAction(double direction)`

The `CatchAction` is a main action that makes the goalie to catch the ball. Therefore the ball must be close to the goalie and the `directon` argument must be the ball direction. Otherwise

the catch command fails and the goalie cannot perform any actions for a number of cycles, what often leads to a goal. Other player than the goalie cannot perform the catch command. If the goalie performs a catch outside the penalty area the the opponent team gets an indirect free kick. The `direction` argument can be a value between -180 and 180.

7.1.6 MoveAction

class: `robocup.component.actions.MoveAction`
constructor: `MoveAction(int x, int y)`

The `MoveAction` is a main action that sets the player to the given position. Normal players can move themselves only when the game isn't running. The goalie can move himself after he catched the ball. Such a move must be performed inside his penalty area. There are two moves allowed after a goalie catch. A third move results in an indirect free kick for the opponent.

7.1.7 TackleAction

class: `robocup.component.actions.TackleAction`
constructor: `TackleAction(int power)`

The `TackleAction` is a main action that lets the player perform a kick with a bigger distance than with the `KickAction`, but it has also some big disadvantages, so that it should be used only in special situations. The first disadvantage is that the player needs to be turned in ball direction to perform a successful tackling. Even then the tackling could fail. The probability to fail increases the further away the ball is. The next disadvantage is that the player who performed the tackling cannot do any actions in the next 10 cycles, so use this command with care. The kick direction of the ball cannot be influenced very much. In general it is the direction of the body of the player. The `power` argument can be a value between -100 and 100 where a positive argument means a kick in body direction and a negative kick means a kick in the opposite direction. By calculating the resukting ball speed from the old ball speed and the argument, is is possible to kick to different angles.

7.1.8 AttentionToAction

class: `robocup.component.actions.AttentionToAction`
constructor: `AttentionToAction(Player player, String teamname)`

The `AttentionToAction` is a secondary action that lets the performing player hear only messages from the given `player`.

7.1.9 PointToAction

class: `robocup.component.actions.PointToAction`
constructor: `PointToAction(double dist, double dir)`

The `PointToAction` is a secondary action that lets the player point his arms to a position given by `dist` and `dir`, where `dir` is a value between -180 and 180. Other players can see the direction of the arm and use this as information.

7.1.10 TurnNeckAction

class: `robocup.component.actions.TurnNeckAction`
constructor: `TurnNeckAction(double angle)`

The `TurnNeckAction` is a secondary action that makes the player turn his head around `angle` degrees. The resulting head angle must be between -90 and 90 and so the maximum values for `angle` are between -180 and 180.

7.1.11 ChangeViewModeAction

class: `robocup.component.actions.ChangeViewModeAction`
constructor: `ChangeViewModeAction(VIEW_QUALITY quality, VIEW_ANGLE angle)`

The `ChangeViewModeAction` is a secondary action that changes the information quality and quantity that a player receives by watching around. There are two view qualities possible where the better one gives more detailled information, while the other one is twice as fast as the first one. Next there are three different view angles available. The player has the choice between `NARROW, NORMAL` and `WIDE`, where bigger angles lead to slower sending steps for information.

7.1.12 SayAction

class: `robocup.component.actions.SayAction`
constructor: `SayAction(String message)`

The `SayAction` makes the player send a message to all player (also opponents) in a certain distance around him. Note that this message can have a maximum length of 10 characters. Each player can hear only one message per cycle. If more messages arrive him, a random message is chosen. To choose, what messages the player wants to hear, he should use the `AttentionToAction`.

7.2 Action factories

The action factories are collections of static methods, that create action-objects for certain purposes like turning the body to a special point or kicking the ball to a certain point. These methods are very helpful, since now we only have to say, what the player should do. We don't need to think about how the player should do this, because the details are calculated from the factory-method. We divided our methods to six factories for different purposes. These 6 factories I'll explain now.

7.2.1 AttentionToActionFactory

class: `robocup.component.actionfactories.AttentionToActionFactory`
methods: `AttentionToAction getAction(WorldModel model)`

This action factory provides methods, to find the best player to hear now. Presently it containes only one simple method, so it should be expanded soon.

getAction

This method delivers an AttentionToAction. The delivered action makes our player to hear another player each cycle, running through the numbers 1 to 11 and starting again with 1 afterwards.

7.2.2 BasicActionFactory

class: `robocup.component.actionfactories.BasicActionFactory`
methods:

```
KickAction accelerateBallToVelocity(Player p, Ball b,
  Vektor velocityToReach)
Action alternate(Player player, Vektor pos1, Vektor pos2,
  double maxDist, STAMINA_LEVEL staminalevel)
DashAction dashToPoint(Player p, Vektor point)
DashAction dashToPoint(Player p, Vektor point, int time)
DashAction dashToPoint(WorldModel world, Vektor point)
Action freezeBall(WorldModel world, Player p, Ball b,
  double angle, double distance)
KickAction kickFastest(Player p, Ball b, Vektor posToKickTo)
KickAction kickInTime(Player p, Ball b, Vektor lastPos,
  int cycles, double error)
KickAction kickToPlayer(Player p, Ball b, Player mate,)
  int maxCycles
MoveAction move(Vektor pos)
```

```
Action moveToPos(Player player, Vektor point,
  double maxDist, STAMINA_LEVEL staminalevel)
Action moveToPos(WorldModel world, Vektor pos,
  STAMINA_LEVEL staminaLevel)
Action moveToPos(WorldModel world, Vektor q, double angle,
  boolean b)
Action moveToPosWatching(Player player, Vektor point,
  Vektor pointToWatch, double maxDist, STAMINA_LEVEL staminalevel)
Action moveToPosXTolerance(Player player, Vektor point,
  double maxXDist, STAMINA_LEVEL staminalevel)
TurnAction turnBackToPoint(Player player, Vektor point)
Action turnBackToPoint(WorldModel world, Vektor point)
TurnAction turnBodyToPoint(Player player, Vektor point)
TurnAction turnBodyToPoint(Player player, Vektor point, int delay)
TurnAction turnBodyToPoint(WorldModel world, Vektor point)
TurnNeckAction turnNeckToObject(WorldModel world,
  Action firstAction, double viewWidth, StaticObject object)
```

This action factory provides the basic methods, to run, kick and turn. So it is the most important factory that we have and as you can see it is also the biggest one.

accelerateBallToVelocity

Given as parameters are the ball, the player and a wished velocity. From the current ball speed and the possible kick power of the player is calculated, if the wished resulting ball speed is possible to reach or not. If it is possible to reach, the method gives the needed `KickAction`, otherwise it delivers `null` to show, that such a kick is impossible to produce.

alternate

This method makes the player move between two positions on the field. Normally it is used when a player reached a good position. The aim is, to make it hard to defend alternating player since he always is in movement. The function never returns `null` and when using it, one should check, if the positions maybe are offside positions or outside the field. Besides, it is important to use a `STAMINA_LEVEL` since a player in optimal position should not waste too much stamina. Otherwise, his good position only seems good for other players.

dashToPoint

This method makes the player to dash to a given point as close as possible. Since the given point is not nessicarily on the dash line of the player, it is calculed how to come to a point on the dash line that lies vertically to the desired point. There are two variants of this function.

One function consideres, in what time the player wants to be at the position (using the drift too) and the other function ignores the drift.

freezeBall

This method delivers the `KickAction` that is needed, to stop the ball at a desired position. The desired position can be given as a parameter and is seen relative to the position of the kicking player in the following cycle. This is very useful, since you can use this command to say: *In the next cycle I want to have the ball 30cm away from me on my left side.* All the nessecary calculations are done by the method then. Note, that often such a kick is impossible. Then a good alternative kick will be translated, or in the worst case `null` is returned.

kickFastest

This method returns the `KickAction` that kicks the ball as fast as possible to a desired position. Here three cases are possible:

1. the desired position is reachable in the next cycle:
   The ball is kicked exactly to the desired position.
2. it is possible to kick the ball into the desired direction:
   The ball is kicked into the desired direction with maximum power.
3. it isn't possible to kick the ball into the desired direction:
   `null` is returned.

kickInTime

This method is used if one wants to have the ball at a desired position in a given number of cycles. If it is possible to kick the ball in the desired way, the corresponding `KickAction` is delivered. Otherwise `null` is returned.

kickToPlayer

This method gives a `KickAction` for a direct pass to a teammate. That means, that the ball is kicked to the actual position of a player. In general you shouldn't use such an easy way of a pass, since the teammate could reach a much better position while the ball is on the way to him. So a good pass maybe should calculate this better position instead of forcing the teammate to wait for the ball. If the direct pas sis impossible, `null` is returned.

move

This method calculates a `MoveAction` to a given position. Use this method instead of generating the `MoveAction` yourself since this method uses our coordinate system in the right way.

moveToPos

This method makes the player to move to a desired position. So it returns a `DashAction` or a `TurnAction`. It always returns an action, never `null`. There are different variants of this method, forcing the player to move backwards or to specify, how much error distance is tolerated.

moveToPosWatching

This is almost the same like `moveToPos`, but here the player decides whether to run forward or backwards by the direction of a position that he wants to see while running.

turnBackToPoint

The player turns his back to the desired point.

turnBodyToPoint

The player turns his body to the desired point.

turnNeckToObject

The player tries to turn his neck in the way that he can see a given object in the next cycle. Therefore he assumes the next position of the object and he assumes his own next position by the main action that he wants to do.

7.2.3 PointToActionFactory

class: `robocup.component.actionfactories.PointToActionFactory`
methods: `PointToAction pointToPositionAction(WorldModel model, Vektor pointTo, Actio`

This action factory provides a method, that helps to point the arm of a player to desired positions. Presently here is only one method, but this factory could be expanded, when you have a good idea, what to show with the arm.

pointToPositionAction

This method makes the player to point to a given position on the field. Here you need a mainAction as an argument, since the pointing works only in the next cycle after the player did the main action.

7.2.4 SayActionFactory

class: `robocup.component.actionfactories.SayActionFactory`
methods: `SayAction getAction(WorldModel model)`

This action factory provides methods, that help to communicate with the other teammates on the field. Presently here is only one method, but there are many other possibilities to communicate important data,

getAction

This method makes the player to communicate his own position and his stamina to all players who hear to him. Since the opponent team should not understand this message, it is encoded.

7.2.5 TurnNeckActionFactory

class: `robocup.component.actionfactories.TurnNeckActionFactory`
methods:
```
TurnNeckAction getAction(WorldModel world, STATES state,
   Action mainAction, ChangeViewModeAction viewModeAction)
TurnNeckAction getAction(WorldModel world, STATES state,
   Action mainAction, ChangeViewModeAction viewModeAction,)
   double[] preferred)
TurnNeckAction getAction(WorldModel world, STATES state,
   Action mainAction, ChangeViewModeAction viewModeAction,
   DynamicObject preferredObject)
TurnNeckAction watchPoint(Player p, Vektor point)
```

This action factory provides methods, to turn the neck effectivly into an informative direction. Most of the methods use the `Neckrotator` class that makes the player watching always different but interesting regions to gain as much actual information as possible.

getAction

This methods use the `Neckrotator` to always find a new interesting place to watch, depending on my position, on what I am doing now and on my watch history of the last cycles. It is

possible to give a an extra parameter if it is very important to see or not to see one or more special objects.

watchPoint

Makes the player simply to watch this point if possible. This method shouldn't be used very often, since the Neckrotator mechanism is deactivated then.

7.2.6 ViewModeActionFactory

class: `robocup.component.actionfactories.ViewModeActionFactory`
methods: `ViewModeAction getAction(Synchro synchro)`

This action factory provides methods, that set the view mode in each cycle in that way, that we get visible informations in each cycle. It isn't used to make the view width bigger or smaller depending on game situations, but depending on a special rhythm that has the desired effect.

getAction

This method produces the rhythm of different view modes that lets us receive information in each cycle.

7.3 NeckRotator

7.3.1 Overview

The NeckRotator is one of the new features of the Dainamite Team. The aim is, that we want to see as much as possible and that all relevant informations should not be too old. The normal way to choose the neckangle of a player was, to let him turn the neck towards an special object, for example the ball or another player. But in some situations it is not very interesting to see the ball, because we have seen the ball even the last 5 cycles before. The NeckRotator works with a new idea. Here a single object is not so important, but it is checked, what areas of the field are of a certain interest. This depends on 3 questions: when did we see the area for the last time? how big is the area? is the area important in the actual game situation?

All these questions together are leading to the best neckangle in each cycle.

7.3.2 The NeckRotator in detail

What are the Slices?

Up to here it was spoken about "areas". Presently for every player the field is cut in 32 slices around the player. Each of the slices has the same size of the angle. Slice number 0 begins at -180 degrees. The cirle around the player is completed with slice number 31, that finishes at 180 degrees. The number of slices is calculated by $2^{numberSlicesExp}$ and so it can be changed, by changing `numberSlicesExp = 5` to another value. A higher number of slices would give more detailed values, but it costs also more time, so we think that 5 is the optimal value here.

How to fill the slices with a "sense"

The NeckRotator is is caluculating several values for each slice and combines them to a final value. Slices with a high final value are more important than slices with smaller values. If the player could see only one slice in each cycle it would be easy to give him just the center–angle of the best slice as the optimal angle, but the view-width of a player is variable and always bigger then one slice. So at first we have to calculate how many slices the player will see and then search for the row of slices with the biggest sum–value. The center–angle of these slices finally is returned as best neck–angle. This main–calculation is done in the `bestViewDirAbs`–method.

The final Value of a slice is calculated from 4 other values and weights for 3 of these values. The formula is:

$$final = sliceHistory^{historyWeight} * estimatedSliceArea^{areaWeight} * stateSlices^{stateWeight} * seeableSlices \quad (7.1)$$

The weights have changed from time to time and possibly there can be found even better weights. Presently the history and the area have the same weight and the weight of the state is a small step higher, because the actual situation seems to be the most important fact in this calculation.

sliceHistory

The sliceHistory is the difference between the actual cycle and the last cycle in that the slice was seen. Everytime when a bodySense arrived, the sliceHistory of all slices increases by 1. Arrives a visualInfo, then the sliceHistory decreases to almost 0 if the slice was seen completely. If only a part of the slice was seen, the sliceHistory decreases to percentage of the seen area.

With this calculation, the sliceHistory of some slices sometimes increases up to more than 1000 and full seen slices have values of about 0,000001. This would mean too big differences for our final calculation, so that these values will be mapped to more usable values by the `roundHistory`–function. The detailed mapping can be seen in the following table:

sliceHistory	mapped Value
< 1	$0,1$
$[1;2[$	$1,5$
$[2;3[$	$2,5$
$[3;4[$	$3,5$
$[4;5[$	$4,0$
$[5;6[$	$4,5$
$[6;8[$	$5,0$
$[8;10[$	$6,0$
$[10;25[$	$7,0$
$[25;50[$	$8,0$
$[50;100[$	$9,0$
> 100	$10,0$

So we see, that a fresh seen slice will not be very important in the next cycle, but at the beginning the importance grows very fast. Was a slice not seen for a longer time, this can be interpreted as the slice is not very important for this player. So the function does not need such a big rate of growth for bigger history–values.

estimatedSliceArea

The meridian of a slice crosses the border of the field in one point. Let the distance between the player and this point be the sliceLength, then the estimatedSliceArea is calculated by this

formula:

$$estimatedSliceArea = sliceLength^2 \tag{7.2}$$

Of corse this is not the real area of the slice, but the calculated value grows almost in the same way like the real area and it is calculated faster. A small error occures, if the slice ends at a corner of the field. Then the area is calculated some percent bigger than in reality, but this error has no effect on our calculation.

The estimatedSliceArea is not calculated new in each cycle. If a player does not move, or if he moves only little, then the areas of the slices don't change enough to be calculated new. So the calculation starts new if the player moved away more than 5m from the last point of calculation. This value can be changed in `minPosDiff`.

In the final calculation the estimatedSliceArea is narrowed down to values between 0 and 10. This has no effect on the calculated neck–angle, but it gives finalValues as a result, that are better to compare for humans.

The sense of using the area is that players should not look to slices where almost nothing interesting is to see for them. For the goalie it makes no sense to look behind his own goal. A dangerous ball cannot come from behind him.

stateSlices

The value of the stateSlices is calculated in very different ways, depending on the State in that the player is at the moment. Some states, like DRIBBLING, SEARCHBALL and 2 special goalie-states have their own methods to calculate the stateSlices but in general the results are between 0 and 10.

For the most states is used the function `setDefaultValues`. This function has many parameters, so that it can be used for almost all situations. Here comes the detailed table of parameters:

parameter	explanation
pos	position of the player
ballWeight	how important is the ball
weight3m	how important are other players who are within 3m around the player this should be a small value because these players are already sensed with body
weight10m	how important are other players who are within 3 - 10m around the player
weight25mm	how important are other players who are within 10 - 25m around the player
weight40m	how important are other players who are within 25 - 40m around the player
farAwayWeight	how important are other players who are more than 40m away from the player
kickableTeam	ball is more important, if it can be kicked by a teammate? often set to false because ball positions are given over aural info when we kick it
kickableOpp	ball is more important, if it can be kicked by an opponent? often set to true because we cannot know where the ball will be in the next cycle
teamWeight	how important are teammates? important for example, when we want to pass
oppWeight	how important are opponents? important for example, when we want to mark an opponent
oppGoalieWeight	to check if I can shoot a goal

If one of the kickable–parameters is switched on, the the player will look with a very high percentage to the ball, if it can be kicked by a teammate / opponent. Many of the parameters for the several states are only first estimations, but they seem so work quite well, so that changing these parameters would not make the team play much better.

Often the ball is at the border of the chosen view–cone and so sometimes the player doesn't see the ball because he missed it for about 1 or 2 degree. To avoid such missings, the neighbour-cycles of important cycles get a certain weight too. So the important objects or not directly at the border of the seen area and the risk to miss it is smaller.

seeableSlices

This value can only be 1 or 0, so if a slice can be seen by the player it will be 1 and otherwise it will be 0. In our final formula this has the effect that unseeable slices will have definitely the finalValue 0. All other values are not affected.

7.3.3 The special state "SEARCHBALL"

For the SEARCHBALL–state the NeckRotator works in a different way, because here the best turn–angle for the body of the player is calculated. That means that the finalValue function has changed in the way, that it is not interesting if a slice is unseeable. With turning the body the player could see it anyway. So the formula is now:

$$\begin{aligned} final &= sliceHistory^{historyWeight} * estimatedSliceArea^{areaWeight} * \\ & \quad stateSlices^{stateWeight} \end{aligned} \tag{7.3}$$

For SEARCHBALL is not used the `setDefaultValues`–method but a special `setSearchBallValues`–method. It gives the highest value to the slice with the estimated next ball. All other slices get their value depending on their distance to the favourite slice. After setting these values this method will not be called again for the next 5 cycles. The stateValues will be changed now after each visual info. The values of all seen slices will be reduced to 10 percent, so that in the next cycles other slices are more interesting. If the ball was not found after 5 cycles, then the procedure starts again.

7.3.4 Many advantages - Any disadvantages?

The testing and using of the NeckRotator showed us, that it brings us many advantages. We have a fine overview over the complete situation on the field, informations are very fresh, we don't see many uninteresting parts of the field and we can look depending on situations very variable. But does it mean that everything is perfect?.

In my eyes the Neckrotator brings a small disadvantage too. Everything that we do with the weigth gives only a kind of "indirect influence" on the player. We cannot say in a special situation that a player has to see a special point. All we can do is to say him that this special point is very interesting... If there are other interesting points for him too, the player could choose another angle. Anyway this disadvantage is only a small one. With setting extreme weight we can reach almost "direct control" again, so the advantages are much much bigger.

7.3.5 Outlook

After some problems, bugfixes etc. now the NeckRotator works relatively fine. The biggest work is done, but small things can always be made better. Of corse it can bring an advantage to find better values for the over 100 weights that are used until now, but it will have more effect to add some functionalities to the NeckRotator. Here I have two basic ideas.

The one idea is, that neighbour–slices of important slices should not get values in a static way, but depending of the distance to the player. For example, if the ball is very far away from the player, then it is not nessecary to use many neighbour–slices. The main slice will be

wide enough then. Otherwise, when the ball is very close, it is easy to miss the ball with a single slice, but even 3 or 5 slices couldn't be enough.

The second idea is, that in some situations a rotation is not useful. So in a very "hot" situation it could be possible that only the ball is interesting for the player and in this moment he should forget that didn't see some slices for a long time now. This can be done with non–static weights for history and states. Is the player far away from ball, his state is not very important, but he has a chance to look places that he didn't see for a long time. So historyWeight should increase with the distance to the ball and the stateWeight should decrease in the same way. A player close to the ball will not have such a big rotation than.

8 Prophet

8.1 Overview

The prophet is used to predict future states based on WorldModel data by means of planning. This is done by making certain assumptions for different analysis and by applying possible actions to agents. However, there is no specific planning algorithm used - it is rather the case that it is coded into the corresponding classes [1] by providing a high amount of domain knowledge in order to reduce the search space and to keep them applicable in real time. Though planning is much more performance consuming than applying rules, it provides the only possibility to make use of opportunities that will arise in the near future or to express conditions which are difficult to include into rules. Whenever the developer is confronted with implementing a new situation, he/she should beware on its performance - the best plan is useless if not generated in the moment it should be executed (usually within (much) less than 100ms).

The following sections first provides an overview about the Prophet in general and then briefly describes the situations refered by the Prophet.

8.2 The structure of the Prophet and Situation class

The Prophet itself is responsible for delegating planning reqests to a responsible Situation, hence it consists of a collection of them. Each Situation can plan towards a certain goal to achieve, e.g. if an agent wants to intercept the ball, a specific Situation called *InterceptBall-Situation* is used to calculate if this is possible and how to achieve this goal. Other (more or less sophisticated) Situations are responsible for dribbling, passing, intercepting an opponent, etc. The Situations are triggered by States (tactical components), which possess a reference on the Prophet. There, generated plans are integrated into the action selection mechanism of the agent by providing their benefit (utility value) and applicability (preCondition - does a plan exist?). The structure of the Prophet can be seen in figure 8.1, containing an overview of the most common Situations. It can be seen that each of them extends the abtract class *Situation*, which contains methods for calculating the benefit of a goal and how (if possible) to achieve it by providing an action (or plan of actions).

Another side-effect of some situations is providing game-information to the agents. For example, if the *InterceptBallSituation* results in a plan, where the agent can get the ball prior

[1] called Situations, because they try to predict world states up to specific situations

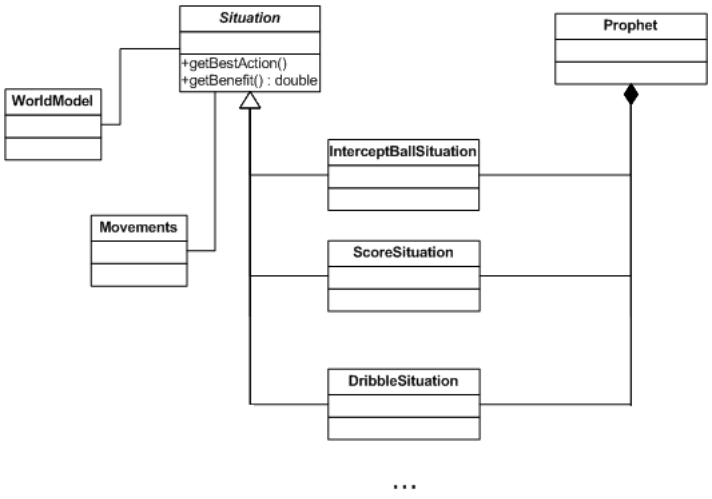


Figure 8.1: Overview about the Situations of an Agent

to all others, he (and his team) is known to be in *BallPossession*. If no plan is createable, where he (or his teammates) can get the ball first, the opponent team is. This information also influences other decisison, because an agent behaves different in both cases respectively.

8.3 Situations Used by the Prophet

The following section provides some information to the most important situations implemented so far. As some of these siutations have become very complex the description given here includes only there main features. For a more detailed description of each situation refer to the source code documentation.

8.3.1 The abstract Situation

The abstract *Situation* class defines methods that are needed by all situations such as:

- **getBenefit()** The benefit of the situation resulting from the best plan in the given context (i.e. the benefit of the best pass when passing). The benefit is a value between `0` and `1`. `0` meaning that there was no plan found. `1` meaning that the plan will 100% result in a goal (which will never happen due to noise). Determining the benefit for a situation or a plan inside of a situation is the most complicated part of the framework espescially as the benefit functions of all situations have to be consistent.
- **getBestAction()** The next (main) action needed to fulfill the plan. `NULL` if no plan was found.

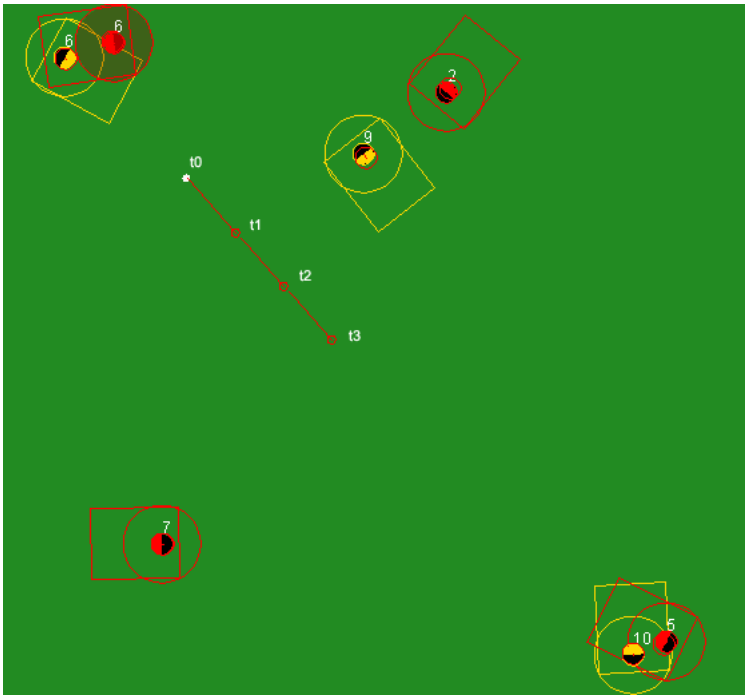


Figure 8.2: Ball Interception using its Speed

- **wasUpdated()** Based on a given world a situation will always deliver the same plan. Therefore we only calculate it once for a given world (i.e. after the arrival of a VI) saving ressources. `wasUpdated()` returns `true` if the situation was already calculated for this world and `false` otherwise.
- **setSubStateString()** The substate string will be visualized in the SoccerMonitor and shows the chosen plan or in other words the leaf of the situations decision tree that produced the best plan.

8.3.2 InterceptBallSituation

The *InterceptBallSituation* calculates, which player can intercept the ball in a given situation first, assuming all agents will try to intercept it. It makes use of the movement knowledge for the ball and the players (both precalculated and stored within the *Movements* class). Since the ball can't change its movement on its own (only some noise is added to its direction and velocity by the server)[2], the course it will take and the time spent therefore can be extrapolated very easily, as long as no one can kick it. In this case, the ball is already in possession of someone and the situation stops planning. For instance in figure 8.2, the future positions of a ball in cycles t_0 (actual cycle) and following is given by a red line and a resulting point at the end.

Knowing for each point how long the ball needs to reach it, it is analyzed, if any other player can get there in the same amount of time, assuming they will try to get there as fast as they

[2]There is also noise contained in the visual ball information a play gets.

can. This is checked for every cycle starting from the current until a cycle where an agent can kick and hence control the ball (indicated by the circles around the players) was found. This method retrieves information on:

- Where will the ball be intercepted (InterceptionPoint).
- When will the ball be intercepted (InterceptionTime).
- By whom will the ball be intercepted (sometimes, this is more than one agent, if each of them needs the same time to reach the interception point).
- What action will (probably) be done by the intercepting player(s) when intercepting the ball as fast as possible. This information is used in order to predict the future movements for some players.[3]

These information are important for checking for instance, which team is in ball-possession. If one agent sees, that he is fastest to the ball - he usually tries to intercept it. In the current implementation, an agent can also compare different interception points, if he is the very most fastest player, in order to select the best.

8.3.3 PassSituation

Until now, the *PassSituation* is the most complex one. Within it, if an agent is in ball-possession, he can analyze, which kick-action will result in situations, where a successful pass was played. This is done in the following way:

1. The Situation checks for an amount of kick-directions and strengths, how the resulting ball will look like (its speed and direction).
2. For each combination, the algorithm checks, who will intercept the ball first and where does this take place, similar to the *InterceptBallSituation*.
3. Finally, it compares the outcomes of all possibilities and selects the best as suggested pass. The pass is only executed, if it is assessed better than all other actions, e.g. as dribbling. As passes that are intercepted by opponents are filtered a pass in a perfect world without noise should always go to a teammate.

It is easy to see, why this algorithm is very complex. The amount of analyzed pass directions can be reduced in order to fasten the algorithm, even dynamically during simulation (see the Synchro-Chapter to get an idea, when the time to compute an action is less available).

The following methods are used on the Prophet to access the *PassSituation*:

[3]It has shown to be pretty simple to predict the movement of players intercepting the ball. The real challenge is to predict other players especially the opponents.

- **public boolean canPass()** `True` if the best pass benefit is greater `0`.
- **public double getPassProb()** The probability the a pass reaches a teammate. (NOT used yet.)
- **public Vektor getPassPoint()** The next interception point resulting from the best pass.
- **public Player getPassTo()** The teammate that will receive the pass.
- **public Action getKickAction()** The next action for the chosen pass plan.
- **public Ball getResultingPassBall()** The ball that will result from the pass. (Used mainly for communication.)
- **public double getPassBenefit()** The benefit of the chosen pass plan.

For details about the calculation of the pass benefit see the `SituationEvaluator` class.

8.3.4 ScoreSituation

This situation simulates goal kicks. It tries to pass between each pair of neighboured opponents inside the goalcone and calculates the score point if a pass reaches the goal before an opponent is able to intercept it. For each resulting score point it determines the probability that the ball goes into the goal when kicking to this point.[4]

The `ScoreSituation` also decides wether the goalie is important enough to be watched in the next cycle. The goalie is important if he is the only opponent preventing us from scoring and every action of him might result in a chance to score. The goalie is less important if we won't have a chance to score against him (.e. as we are to far away from the goal). In this case it's more important to watch out for teammates to pass to.

Corresponding Prophet Methods:

- **public double getScoreBenefit()** The benefit of the score plan. (This will be named getScoreProb() in future releases as the benefit of a goal is always 1!)
- **public Action getScoreAction()** The next action for the chosen plan.
- **public Ball getResultingScoreBall()** The ball resulting from the next action.
- **public boolean shouldWatchGoalie()** True, if it is important to watch the opponent's goalie waiting for a mistake.

[4]The kicking itself and the ball movement are noisy. For (much) more details see [8].

8.3.5 GoalkeeperSituation

The GoalkeeperSituation is described in detail in section **??**, where it is explained in the context of the complete agent (states and situations). Therefore this subsection provides only a summary of methods provided by the Prophet.

Corresponding Prophet Methods:

- **public Action getBestGoaliePositioningAction()**
- **public Action getBestGoalieDefensivePositioningAction()**
- **public Action getGoalieWaitAction()**
- **public int getGoalieInterceptCatchBallCycles()**
- **public boolean goalieCanCatchFastest()**
- **public boolean teamToGoaliePass()**
- **public boolean goalieCanKickFastest()**
- **public boolean ballInDangerousGoalDistance()**

8.3.6 DribbleSituation

This situation calculations a dribbling plan. First it checks wether dribbling is save. Then it determines where to dribble to and what action needs to be done for that. We try to dribble with the ball on our side thus avoiding collisions and keeping control over the ball the whole time.

The dribble situation also handles the waiting with the ball. This might prove useful when the player has no more stamina and no passing is possible or there are to many opponents around so the player tries to keep the ball as long as possible until he finds someone to pass to.

Corresponding Prophet Methods:

- **public double getDribbleBenefit()** The benefit when dribbling.
- **public Action getBestDribbleAction()** The next action for fulfilling the dribble plan.

9 Message-Factory

9.1 Introduction

Communication in Robocup serves mainly for two purposes. First of all, it can be used to share knowledge such as positions of players or the ball. This is a very effective way of improving the agents world-model accuracy. The second purpose, which certainly is on a higher level of abstraction, is that of giving or requesting tactical information. However, since message receivement is not assured, implementing some kind of dialogue is very risky and has a high probabilty of failure.

The *Message-Factory* provides the means for encoding and decoding messages, and therefore serves directly for the first purpose. All other tasks, especially those which need more logic using *AttentionTo* and communication protocols, are not yet part of this framework. The CLang, the messages that are sent by the coach, are not included here as well (see chapter 11 instead), but freeform-messages, i.e. messages which contain simple strings, may make use of the given functions. In order to explain the *Message-Factory*, this chapter is organized as follows: The next section gives an overview about the message format. Then the following explains, how different values are encoded and decoded into and from strings. For instance, floating point values can be encoded with a different kind of precision.Finally, a conclusion is given and some extensions are suggested.

9.2 Message Format

Usually, messages which are exchanged between player agents, have a fixed maximum length and can be composed by a subset of ASCII[1]. However, since the maximum length is configurable (server.conf), and the allowed charset may change in newer versions, the *Message-Factory* is implemented in a flexible way, such that changes to these values may not affect its operational reliability, or it may at least be adopted with a few lines of code, such that it will work again.

In general, all messages implemented here contain an *id*, which tells the receiver, what kind of messages was received, and the *message-body*. Up to now, an *id* is the first character of a string, which allows to differentiate 73 kinds of message types. If further messages are needed, simply use one of the **ids** as indicator for a second thereafter, which would increase the number

[1] In the server-version 10.0.7, a message is limited to 10 characters length, using only a set of about 73 different sympols

of definable messages again. Each *id* is associated with a message length, which defines the number of characters used for the *message-body*. This allows to combine different messages into one string, as shown in the following two lines.

$$string : \underbrace{c_0 \mid c_1 \mid c_2 \mid c_3 \mid c_4}_{message_1} \mid \underbrace{c_5 \mid c_6 \mid c_7 \mid c_8 \mid c_9}_{message_2}$$

$$message_1 : \underbrace{c_0}_{id} \mid \underbrace{c_1 \mid c_2 \mid c_3 \mid c_4}_{message-body}$$

Here, a string out of ten characters is shown (c_0 to c_9). The string contains two messages ($message_1$ and $message_2$) each of five chars length. Each message is again made up of an *id* (c_0 in $message_1$ and c_5 in $message_2$), which tells the receiver, what is the content and the length of the corresponding message. This is important, because otherwise the *MessageFactory* wouldn't know how to decode the message-body. For instance, in this example each message could be the position of a different player.

9.2.1 Definition of Messages

In order to define a new message, the following steps are neccesary:

1. A message type constant has to be defined. This is done by adding/changing a constant to the enum *MESSAGES*. This automatically reserves an id for that message, because the index (*MY_MESSAGE.ordinal()*) of that constant points at the corresponding index of the array of available characters defined in *NumberConverter.digits*.

2. The message length without its id has to be specified. This should be done within the constructor of the MessageFactory. There, the length has to set to the array *lengthOfMessage*, using the value of the id as index, which enables a fast access:

   lengthOfMessage[NumberConverter.digits[MESSAGES.MY_MESSAGE.ordinal()]] = 4;

9.2.2 Receiving a Message

If a string message is received, e.g. from a say-action executed by another agent, this has first to be passed to the method *MessageFactory.generateMessages(String, int)*, whereas the first parameter is the corresponding message itself, and the next is the bodyCycle indicating when the message was received. The MessageFactory splits the messages by analyzing the ids and the corresponding lengths of the messages, and stores them in the lists *messageTypes* and *messageContent*. In the former, all received types are remembered, in the latter the corresponding part of the message, i.e. the substring that makes the content of the message. Note that the indices of both lists refer to the messages and message-types, which belong

together. If the given bodyCycle is higher, the lists were cleared, before new data is added, such that only those messages are buffered, which are received in the actual cycle. Still missing is the encoding and decoding of the messages, which is explained in the next section.

9.3 Encoding and Decoding Numerical Values

Sharing information with messages of about ten characters imply, that the content has to be encoded efficiently. The approach taken here is straightforward, and allows to encode values with different precision in order to give exact information if needed, or to save the message capacity for other content, if precision is not so important. However, there are also some restrictions, but these are not very serious.

The method for the encoding of floating point and integer values by the message factory is nearly the same. Generally, a numerical value is mapped to a fixed set of characters, which represent approximately the same value, but are stored in an *n-ary* number system. Concretely, the usual decimal numbers (i.e. 10-ary numbers) are translated into numbers of the base 73 (i.e. 73-ary numbers, which is the amount of allowed characters for communication). In order to define an encoding, three decisions have to be made. First of all, the number of characters into which a number should be converted have to be defined. The more characters are used, the higher is the precision of the encoded value. After that, the range of the value must be specified by giving the minimum and the maximum value. The larger the range, the lower is the precision. The decoding of floating point values imply always a range of [0,1], which allows the use of a fixed point. An appropriate scaling has to be made beforehand.

9.3.1 Converting Small Positive Integer Values

Small integer values, especially those smaller than 73, can be mapped directly to a character using the array *NumberConverter.digits*. Therefore the number itself can be used as index for retrieving the corresponding character, i.e.:

```
char c = NumberConverter.digits[value];
```

Be sure that the values are always smaller than the length of the array, else an Exception will occur, which influences the process of the agent significantly. For instance encoding a tricot number this way would be feasible. The revers operation is nearly as trivial, because of the existance of an array for the decoding process, called *valuesOfChars[]*, which maps each character to the value it stands for. Simply give the character as index:

```
int i = this.valuesOfChars[c];
```

Note, that because the character-values (their Unicode-values) are not zero-based, the array is larger than *NumberConverter.digits*, and has some unused fields. This was accepted, because we gave priority to fast access instead of memory usage (which isn't that critical here).

If two smaller values should be encoded, they can share a character. Each character is able to hold little more than 6 bits of information, such that for example two bits can be used for a simple status, and the other for a tricot number. Encoding must be done by hand, e.g. if the first value has sixteen different states (four bits), and the latter four (two bits), use the following code to create the corresponding character:

```
char c = NumberConverter.digits[(value1 * 4) + value2];
```

The used values should start at zero, else this calculation will fail. For the decoding there are also precalculated arrays, which retrieve the corresponding values. These are *sharedFrontValue* and *sharedBackValue* respectively. The former is used to address the first (higher) bits, the latter for the lower bits. The corresponding decoding of the example above is given as:

```
number1 = this.sharedFrontValue[char1][3];
number2 = this.sharedBackValue[char1][3];
```

The arrays are two-dimensional, and store the reverse operation of the encoding shown above. For example, the *sharedFrontValue* is calculated as:

$$v_f = v_{char}/(j+1)$$

Here v_{char} is the integer value of the character (given by the array valuesOfChars[c]). The result is the number of times $j+1$ (the number of states of the first value) fits into v_{char}. The *sharedBackValue* is correspondingly calculated as:

$$v_b = v_{char} mod (j+1)$$

The array stores precalculations allowing four bits for the first value. This is no restriction, because if more bits are required, the second value has to be smaller, and the order of them can be swapped correspondingly.

9.3.2 Converting Other Integer Values

Converting integer values in general is done by specifying a minumum and a maximum value, and hence implicitly its range. This value is scaled to the available range of the selected amount of characters. Decoding is simply done by scaling back. There exists a method for converting a number (stamina) to a single character, called *convertIntToCharacter(int, int, int)*, whereas the first is the value to convert, the second the minimum, and the third the maximum value. This method may be extended to fit for a general mapping quite easy.

9.3.3 Converting Floating Point Values

As mentioned above, converting floating point values is allowed only for values of the interval [0,1], hence these have to be scaled first using again a minimum and maximum value. The en- and decoding is more flexible than that for integer values. We can define a set of doubles to be converted to a set of chars and vice versa. For instance, it is possible to map four double-values to three characters, whereas each of the doubles is mapped to approx. a $frac34$ char.

The methods used therefore are also contained in the class NumberConverter, having the following signatures:

- *char[] convertToChars(double[] doubles, int numberChars)*
- *double[] convertToDoubles(char[] chars, int start, int length, int numberDoubles)*

The first is responsible for encoding values to chars. The first parameter is an array of the values to encode, the second defines the numbers of characters which should be created. The decoding method needs a character array as parameter, together with a start and an end-index (usually 0 and length-1, if the complete array is used for decoding), and the number of doubles which are contained in the encoded characters.

Example

A very often used encoding is that of a Vektor, i.e. a position on the field. A Vektor is given by an x- and y-coordinate. In order to convert them, values have to be given in the interval of [0,1]. Therefore we decided to encode the Vektor using polar coordinates, which is an angle and a distance. The angle is scaled from [0,360] to [0,1] and the distance is assumed to be maximal 70m from the coordinate center. Thus, a double array containing the corresponding two values is encoded using the method *convertToChars(double[], int)*. Using three characters for encoding, the maximal error we get for the position is about 15cm. If we use four characters, the error is about 1cm. Note that usually the error of visual data is approx. equal to the encoding error using three characters. This way, a message can contain the position and the speed of two players quite accuaret, or those of a single player very precisely.

9.4 En- and Decoding Processes

The MessageFactory provides an encoding- and decoding process, which was optimized for performance issues. Thus, if extending, it is recommended to follow this approach. The given section outlines the processes of encoding and decoding subsequently.

9.4.1 Encoding Messages

For the encoding of a message, a simple method called *encodeMyMessage(..)* should be implemented, which directly returns the string. This method has to be public, because it is accessed directly by other components, e.g. the SayActionFactory. The only thing to consider here is, that the message format is kept, i.e. the id is the first character of the message, and the length of the returned string is that specified for the message type (plus 1, because the id counts extra).

9.4.2 Decoding Messages

As with encoding, this process requires a method called *decodeXY(String)*, which handles the decoding. The first thing to note here is, that this method doesn't return any values. It rather sets the results to attributes of the MessageFactory, which can be retrieved by getter-methods. The second thing is, that this method isn't called directly, but must be integrated into the decoding process as follows: As mentioned in section 9.2.2, after receiving a message (or a string containing more messages), this should be passed to the method *generateMessages(String)*. After that, all contained message types and message contents are stored separately. Usually, not every message is interesting for all components (i.e. the position of the ball is irrelevant to the MeModel). Hence the component can get all contained message-types using *getMessageTypes()*, and when iterating them, it can decide, which one contains relevant data. Then it invokes the corresponding decoding using the method *decodeMessageNumber(int)*, whereas the parameter refers to the index of the message-type from iteration. This one will call, depending on the message type, the implemented decoding message. Therefore each new decoding method has to be added to the decodeMessageNumber-implementation in order to be invoked subsequently.

9.5 Conclusion

The MessageFactory currently implements many message-types for positions and/or speeds of a player or a ball, together with some other data. The approach taken allows to flexibly combine messages, to vary in precision of the content and to extend it easily. However, it only allows to define simple messages. Communication protocols or dialogue systems are beyond its scope. Implementing these would require to make use of *AttentionTo*, some kind of communication rules and a model of the process of a protocol, i.e. a new *IModel* added to the *WorldModel*. Extending the MessageFactory with new types is currently not restricted, because only 20 out of 73 ids are yet used. If the maximum amount of messages is reached, new ids can be obtained by using two characters as id (i.e. a distinguished first character denotes that the second character is another id).

10 Tactic

10.1 Introduction

The most crucial element of an agent is his action selection, sometimes referred to as tactic, because this component decides how he behaves. The implementation of the tactic is also the most challenging aspect, because its measurement of quality is very vague and hence an objective assessment is difficult. We believe that a simple, but flexible and intuitive mechanism for implementing tactical decisions is required in order to obtain a good understanding of what is going on inside an agent, and subsequently to achieve a well implemented system. Therefore we used a simple state-based reactive approach, which is combined with reasoning mechanisms (Situations) if needed. This chapter introduces the concepts of our tactic, and therefore is organized as follows: First we present the process of action selection together with the structure of the tactical components. After that, some of the implemented tactical classes are shown, and finally we give a conclusion and an assessment of this approach.

10.2 States and StateEvaluation

Before describing the tactical parts of our framework in detail, this section starts with a brief motivation of our approach. As already known, a robocup-agent is able to act by sending action commands like dash, kick, etc. to the soccer server, whereas a tactical component is reponsible for selecting an appropriate action in a given situation. If looking at real soccer, actions are not only distinguished by their constituting motions, but also by their intended *goal*. For instance, a kick could be executed as a pass in one situation, or as a goal-shot in another one. Correpondingly, a soccer player intentionally decides how to behave on the higher level, which is, if to dribble, to pass, to hold or reach a certain position or to intercept the ball. These *behaviours* usually involve the execution of a sequence of actions or motions, e.g. if dribbling, the player must decide, if a kick, a dash or a turn should be performed. In order to reflect this relation, we use the concept of states. This is well suited for representing in an abstract manner, what an agent is currently doing, similar to a state machine. Therefore each state implements a defined behaviour, and the set of states that an agent owns describes his possible options for acting.

Along with the states, the tactical decision of an agent is divided into two layers. The first and higher layer is the selection of one of the states in each cycle, which corresponds to the decision of the soccer player to dribble or to pass. If a state is selected, an action fitting best for that state has to be selected. Thus their implementation involves two related activities to be

done by the developer. The first is the implementation of an assessment of a state for a given situation, which enables to compare them with the other available states. The second task is the determination of the lower level actions leading to a concrete motion for a behaviour. From these requirements we obtain an abstract definition of a state that is implemented in the class *AbstractState* by the following methods:

- public boolean preCondition()
- public double successProbability()
- public double successBenefit()
- public void calculateMainAction()
- public void calculateViewModeAction()
- public void calculateAttentionToAction()
- public void calculateTurnNeckAction()
- public void calculatePointToAction()
- public void calculateSayAction()

As denoted by their names, each of them serves for a certain purpose, which is stated next. The method *preCondition()* has to make sure, that the corresponding state can be executed in a given situation. For instance passing requires, that the ball is kickable by the player. If not, this method should return *false*. *successProbability* and *successBenefit* are used to assess a state for a given situation. The former should calculate or estimate the probability of successful execution of an action. This should be returned as a value of the interval [0,1], whereas a zero means, execution is impossible. The latter should estimate the benefit of executing the action. This is also done by returning a value of the interval [0,1], whereas a 1 should relate to the direct scoring of a goal. The product of the *successProbability* and the *successBenefit* serves as an assessment for a state, which nearly corresponds to the expected reward for executing a state[1]. Thus, our agents will naturally keep a balance between risk and benefit.

These three methods are now used for determining the state that should be selected, following this basic algorithm.

Let S be the states available for agent a. Then the state to select is given by

$$S_{valid} = \{s \in S | s.preCondition()\} \tag{10.1}$$

$$s_{opt} = \arg \max_{s \in S_{valid}} (s.successBenefit() * s.successProbability()) \tag{10.2}$$

[1]Note that the penalty for failure is not included, hence this approach is not directly applicable with RL algorithms

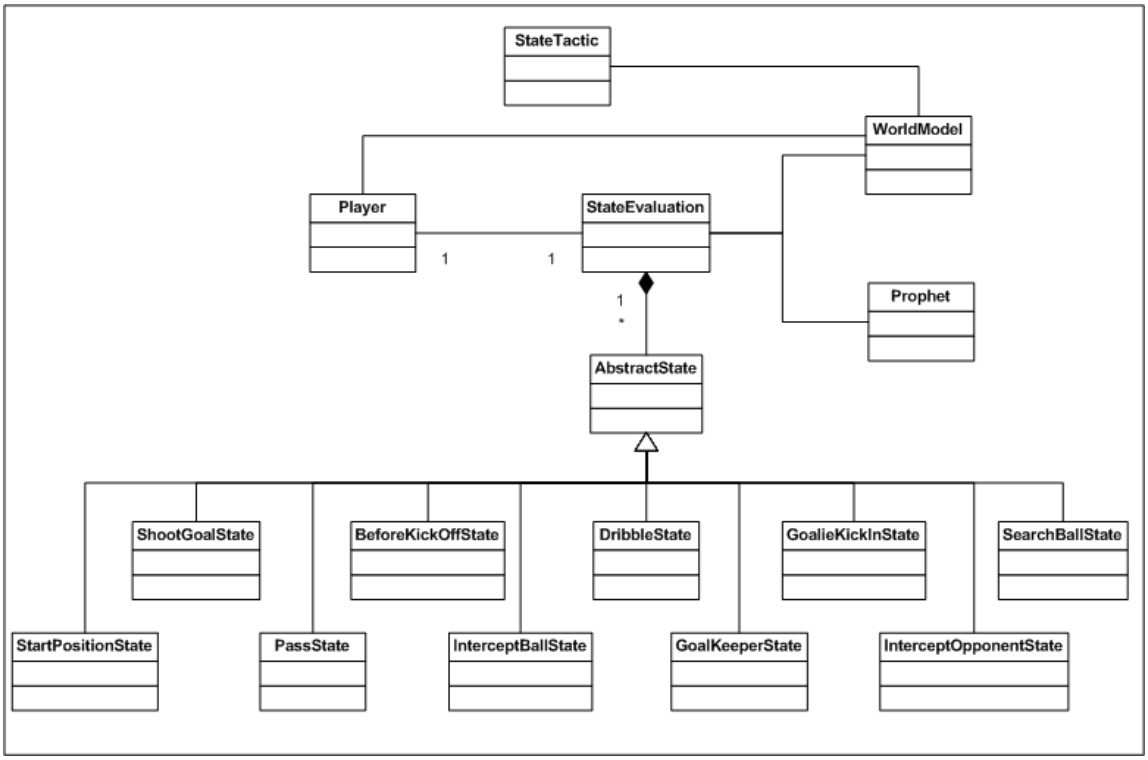


Figure 10.1: Overview about the tactical classes - the StateEvaluation and the States together with their important relations

First, the precondition of all states is evaluated, and those returning false are removed from consideration. Thereafter, each is assessed, and the best one, i.e. that with highest assessment value, is kept for execution. This now means, that the corresponding actions have to be determined. Therefore, the other methods are called in the order, which is given above. First, the main-action (dash, turn, etc.) has to be calculated, followed by the viewmode-action, and so on. This order is neccessary, because later determined actions require to know, which of the previous ones are selected (e.g. it is required to know, which view-mode is used in order to determine the turn-neck action).

The class, that is responsible for the selection of a state is called *StateEvaluation*, which is included into a *Player* (see Figure 10.1), and contains the collection of *States*, that constitutes the possible behaviours of an agent. Additionally, the *StateEvaluation* maintains a reference to the *WorldModel* and the *Prophet*. Both are very important: The former provides the information about the world, which has to be evaluated and assessed by each state. The latter holds all *Situations*. These can derive necessary information from the *WorldModel*, implement the agents reasoning capabilities and therefore are also very important for action selection (see Chapter 8) for details). Finally the StateTactic is the interface of the agent for triggering the calculation of actions in combination with the Synchro (see Chapter 5 for details).

States combined with *Situations* usually work little different than simple states. There, most of the work work is done within the *Situations*, and a state is almost only needed as mediator, such that a precondition and an assessment are also given. The reason for that is as

follows: The reasoning done in a *Situation* can be interpreted as planning, although no explicit or formal mechanism is used. If a valid plan was found by the *Situation*, the precondition may be true (there might be additional conditions, e.g. a playmode, which should be evaluated before starting to plan in order to save performance). Also the assessment of that situation-based state is directly related to the assumed resulting world-model (situation), such that this is done after planning. If for example a plan leads to goal, the state should return 1 as benefit. Subsequently, the calculation of the actions in the state is easier. Since a situation creates a plan (or something similar) the main-action are already determined there and can be retrieved by the states.

Finally it should be mentioned, that there are no explicit rules determining, which functionality has to be implemented within a state or in a situation respectively. For instance a state may be implemented to contain complex reasoning methods without referring to any situation. But there is one aspect giving hints to that question: If the reasoning process provides information that is important to more than one state, a situation should be used, because this is accessible by all states via the Prophet per default. States usually do not refer one another (and creating these references is not simple, because they are instantiated using reflection). An example for this is the *InterceptBallSituation*, which determines who is in ball-possession. Other states may depend on that information, e.g. a forward would behave different, if either a teammate owns the ball or an opponent does.

10.3 Implemented States

As shown in Figure 10.1, there are a few states already implemented, such that a simple kick'n'run tactic is given. This section gives an overview about these states by describing them briefly. For details have a look at their JavaDoc, and in order to see, which state belong to which player. and what play-modes they support, refer to the agents configuration files of the project (Section 2.5). Looking at the source-code files, you will notice that some of the state-assessment values do not to fit to the aforementioned guidelines (e.g. the ball-interception is always assessed 0.95). Keep in mind, that without being in ball-possession, the best that an agent can do is to get the ball. Important is only, that the agent selects in each situation the best option, which is, the best state that is applicable in a current situation. Thus one has to care, that states, which can be selected under equal circumstances (e.g. passing or dribbling) are adjusted against each other. In the following, the available states are listed:

AbstractState: As the name intends, this state is defined abstract, and is the superclass for all other states. Therefore it provides methods, which can be overriden by the extending classes for their own behaviour definition. Additionally it provides some useful basic method for the other states.

BeforeKickOffState: The BeforeKickOffState is triggered for all Situations with playmode BEFORE_KICK_OFF. It's the the state that lets the player move at its start position. All players will be turned in the direction of the opponent's goal. However, if its our kick

off player number 10 will be the fastest to the ball and this state will not be triggered for him, because this will execute the kick-off.

DribbleState: This state represents the players dribbling intention. It is nearly always triggered when the player has the ball under control. The decision is made in the corresponding *DribbleSituation*.

GoalieKickInState: This state defines the behaviour of the goalie after catching the ball. Then, he is able to move twice inside his penalty area in order to make a kick-in.

GoalkeeperState: This state implements most parts of the behaviour of the goalie. Especially his positioning and when to catch the ball.

InterceptBallState: The state for intercepting the ball. This state is triggered when the player can not kick the ball and is the one of the fastest player to the ball. Most of the logic used here can be found in the InterceptBallSituation.

InterceptOpponentState: The state for intercepting an opponent with ball. This state is triggered when the player can not kick the ball and is the one of the fastest player to the opponent with ball. Note that the implementation is simple and hence not very effective.

PassState: This state is responsible for passing. In combination with the PassSituation, this behaviour is the most complex and comsumes most performance.

SearchBallState: The SearchBallState is triggered when a player lost the ball-position. It triggeres the algorithm that searchs the ball. This state is also the default state triggered when no other state is possible (which should never ever happen!!).

ShootGoalState: The state that that tries to score. Triggered when with ball and near the opponents goal.

StartPositionState: State that moves a player to his start-position when in play-mode before kick-off. The start-position depends on the number of the player and is encoded into the FormationData-class.

10.4 Conclusion and Outlook

The tactical framework presented here has some interesting properties. The first is, that it provides an easy way to implement a certain behaviour. Extending the AbstractState and adding this to an agents configuration file is all that is necessary. However, since state selection depends on numerical values (their assessment, which may depend on multiple parameters), the explicit definition of state transition is not so easy. Balancing and adjusting these values is very time consuming and it is possibly helpful to enhance them using learning algorithms. Another property so far is, that states allow both, the implementation of simple reactive behaviour, or behaviour which relies on more sophisticated reasoning using situations. In

the latter case, they can be used as interface to the state-selection mechanism. It should be possible to add learned behaviours into this framework as easiliy as the others, but this was not tested so far. The assessment procedure of the states should also allow to use learning on a higher level, such that adjusting their assessment values can be done automatically. Finally, configuration of states is an easy way of defining roles of the team. Therefore, the roles goalie, defender and player already exist. Following this approach each agent can have its own defined behaviour by providing an individual configuration plus the states for them.

The given tactical framework also lacks some functionality. One is for instance explicit teamwork. Implementing a team would include to desing behaviours of two or more agents to be executed simultaneously. The current approach doesn't provide this. Up to now, we only made use of roles, and in a given situation, an agent has to recognize his tasks depending on that. Another missing feature is long-term planning. However, since robocup is highly dynamic, these might be applicable only in special situations (e.g. standards), which are more predictable.

11 Coach

11.1 Overview

The coach is a non-visible agent with different functions. The coach sets the heterogeneous players, it is also able to change players during the game, it can communicate with the players and gets noise-free information about all movable objects. Consequently the coach is a tactical instrument to analyse the game and to give strategically information to the players.

11.2 Structure

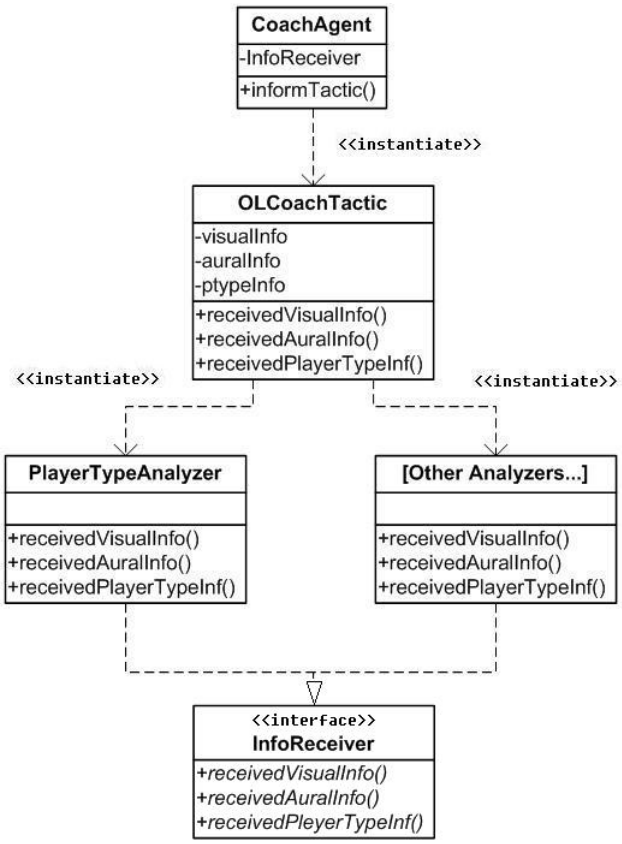


Figure 11.1: Coach Structure

The server data is forwarded by the CoachAgent to the OLCoachTactic. The server data contains visual info, aural info or the player type info. The OLCoachTactic contains three linked lists, one list for each perception. The analyzers are now being listed in the appropriate perception lists (see figure 11.1). For example the PlayerTypeAnalyzer only needs player-type information for analyzing. So this analyzer is only registered in this perception list. Finally an analyzer has the possibility to unsubscribe from a perception list with the removeFromXList method (X can be Visual, Aural, Ptype).

11.3 Coach Language

The Coach Language was concepted in order to communicate strategically information from the coach to the players. The grammar was designed in such a way, that there is no misunderstanding in their meaning, such that theretically coaches could be used by any team.

To communicate with the players, first a version has to be determined with the players via server. This happens in RobocupAgent in package robocup.component. The command that is communicated to the server is:
`(clang (ver 7 8))`
This simply means that we communicate Coach Language (short: clang) at least with version 7 and not more than version 8. This is also binding for the Freeform Messages, since they are part of the Coach Language. After the coach is sending a clang message, the sequence (see figure 11.2) is as follows:

1. Receiving and parsing the clang string by the Robocup Agent.

   The CLangParser (14.1) builds an object tree and returns an info object to the Robocup Agent. The components for the object tree can be found in package robocup.component.speechacts.coachlanguage.

2. Evaluating and interpreting the clang object tree by the CLangModel (11.3.2).
   Notice: That does not apply to Freeform Messages. They will be directly send to the WorldModel and decoded by the MessageFactory (9)

3. The results from CLangModel may flow into the action selection by any state.

11.3.1 CLang Grammar

`Package: robocup.parser9.CLangParser`

The CLang Grammar (14.1) specifies the standard coach language and complies with version 7 and 8. The implementation can be found in CLangParser.jj in package robocup.parser9.

For example a RULE consists of a CONDITION and a DIRECTIVE_LIST or of a CONDITION and a RULE_LIST or of an ID_LIST. The CONDITION can be TRUE or FALSE

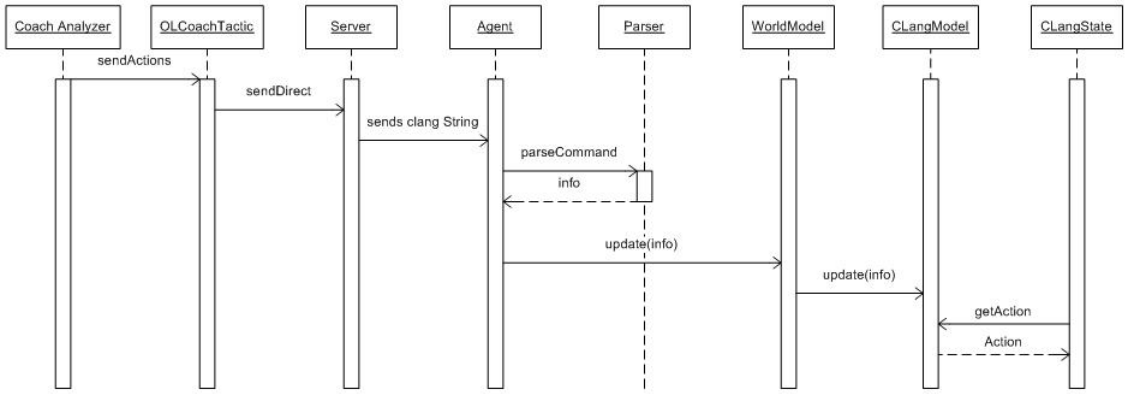


Figure 11.2: Sequence of the clang message

or it consists of other statements which can be examined for truth. The DIRECTIVE_LIST contains a list of DIRECTIVEs, which can be assigned to all players or special players. So if a CONDITION applies, the DIRECTIVE has to be followed. The following expression means that our goalie should position itself at the point xy before kick off:
`definerule((playm bko) (do our 1 (home (pt x y))))`

11.3.2 The CLangModel

`Package: robocup.component.worldmodel.CLangModel`

The CLangModel is an extension of the WorldModel. After parsing a clang string by the CLangParser the CLangModel receives a parsed object tree, which now can be interpreted and evaluated. The results are then available for the states for further processing.

11.3.3 Freeform Message

To communicate via a Freeform Message, first the message has to be encoded with the Messagefactory (9). To send this message to the players the following command is used:
(say (freeform "[MESSAGE]"))
The players are able to understand the message by decoding it with the Messagefactory.

11.3.4 Broadcasting Messages

`Package: robocup.component.coach.BroadcastCommand`

The coach only can communicate messages in specific intervals to the players. There are different intervals for freeform-messages and clang-messages. The BroadcastCommand controls the dispatch of the messages under attention of the message send periods and the maximum

counts of each message type. If multiple messages of one message type arrive in one period, all messages will be merged into one message to keep the message counts low.
Notice: You can send messages via OLCoachTactic. There are two methods to handoff the messages to the BroadcastCommand and one method to send messages directly to the server:

- `sendDirect(String)` To send the messages directly to the server. This will only be used for committing the player types (11.4.1).
- `sendFreeform(String)` To send Freeform Messages via the BroadcastCommand.
- `sendActions(CoachSayInfo)` To send Clang Messages via the BroadcastCommand.

11.4 Heterogeneous Player Types

`Package: robocup.component.coach.PlayerTypeAnalyzer`

Before the game starts seven heterogeneous player types are given by server. Each player type has different pros and cons. The goalie can only be assigned to default player type 0. The opponent receives the same player types by server. So the opponent goalie also has the same values like our goalie. After receiving the player types, they are being ranked.

Now several ranked arrays are being offered. In this ranking the best player type is the first one and the worst is the last one. Here is a list of the values of the current array ranking:

- ranked_KM indicates, which player type has the best "Kickable Margin".
- ranked_DP indicates, which player type has the best "Dash Power Rate".
- ranked_IM indicates, which player type has the best "Inertia Moment".
- ranked_St indicates, which player type has the best Value in "Stamina".
- ranked_KM_IM indicates, which player type has the best "Kickable Margin" and "Inertia Moment" with equal weighting.
- ranked_KM_St indicates, which player type has the best "Kickable Margin" and "Stamina" with equal weighting.
- ranked_DP_St indicates, which player type has the best "Dash Power Rate" and "Stamina" with equal weighting.
- ranked_2DP_1St indicates, which player type has the best "Dash Power Rate" and "Stamina" with 2 to 1 weighting.
- ranked_1DP_2St indicates, which player type has the best "Dash Power Rate" and "Stamina" with 1 to 2 weighting.

The team–formation bases on this ranking. It happens in package robocup.component.tactics in FormationData. Since each player type can only be assigned three times, the order of the player type categorization is relevant.

11.4.1 Change Player Type

The command to set a heterogeneous player is the same as the one to change a player.
(change_player_type [PLAYER] [TYPE])
PLAYER stands for the player number and TYPE for the player type. Three players can be replaced maximum.

12 SoccerScope and Tools

12.1 Introduction

Implementing and debugging a team of robocup software-agents involves a lot of testing. A general problem with this is, that the most relevant aspects of the agents are not visible during runtime, i.e. their knowledge about the state of the world and their tactical decision process. In order to make these accessible, different methods can be used - logging to a file or the shell, or even to a database is one option. A different method is to use a visualisation software, which is able to display the internals of the agents during simulation. Yet another problem the developers are faced with is, that testing by running complete games is very inefficient, if only subparts of the whole simulation should be analyzed. This counts especially for situations that do not occur very often, e.g. corner-kicks or situations near the opponents goal, if that team is very strong. Therefore the soccer server provides training capabilities by means of a trainer agent, which is able to manipulate the game and its setup in order to repeatedly run certain situations. However, the soccer server lacks in providing an implementation of the trainer, so teams have to care for that by themselves.

The given monitor is based on the *SoccerScope2003* implementation of the team *YowAI* [11], which we decided to extend for solving the aforementioned problems. Therefore it uses other (freely available) software, and thus became a powerful tool for developing a simulation team based on the dainamite framework. First of all, it is able to display the knowledge of each dainamite agent by simply selecting it on the monitor. Additionally it is possible to see the results of their tactical decisions, i.e. their states plus the corresponding assessments. All these information can be serialized in two ways, either by writing into a file or by updating a database, and can be evaluated lateron efficiently. Finally, the monitor can be used to *graphically* design and store training scenarios, which can be executed for certain purposes (e.g. machine learning) from the monitor as well.

In order to give an overview about the monitor and its extensions, this chapter is organized as follows: In the next section, the monitor itself is presented together with its visualisation capabilities. After that, the database connectivity is presented, and then the use of training scenarios is shown. Finally, this chapter closes with a short discussion of improvements and future work.

12.2 Monitor

12.2.1 Overview

The graphical user interface must comply two main features. Firstly the classic monitoring and secondly the extended analyse, statistic and manual user interaction assistance feature for the DAI-Namite framework.
To achieve these requirements the developers of the DAI-Namite team decided to develop further SoccerScope 2003. SoccerScope 2003 fulfils the first requirement completely and is available for free. It was written in JAVA so it's relative easy to develop further and add functions of the second main feature within the DAI-Namite framework.

Added visible gui functions:

1. Show the actual world model of each player within the field.
2. Show the executable states of each player for each cycle together with its assessment.
3. Display the world model data of a player using charts.
4. Save chart images to file

12.2.2 Using Soccerscope

The SoccerScope monitor starts automatically with the dainamite team, if configured this way (see 2.5 for details). This should look like Figure 12.1. On startup, it automatically connects to the soccer server as monitor instance, so that it is able to show the original game data. Additionally, agents that are connected to the gui, send their world model and tactical data to SoccerScope as well. Starting the game can be done by:

- Menu *Monitor* → *Kick-Off*
- Menu *Monitor* → *Drop-Ball*
- Menu *Monitor* → *Drop-Ball to Center*
- Toolbar-Button for *Kick-Off* (displays time 0:01)
- Toolbar-Button for *Drop-Ball* (displays ball with arrow down)

During game, (the original) SoccerScope provides different visualisation tools. These can be found in the menus *View* and *Analyze*. With the *View*-menu, details of the simulation can be shown or hidden, e.g. the kickable-area or the stamina of each player, their view-cone. It also provides a submenu for selecting specific parts of the agents world model (called

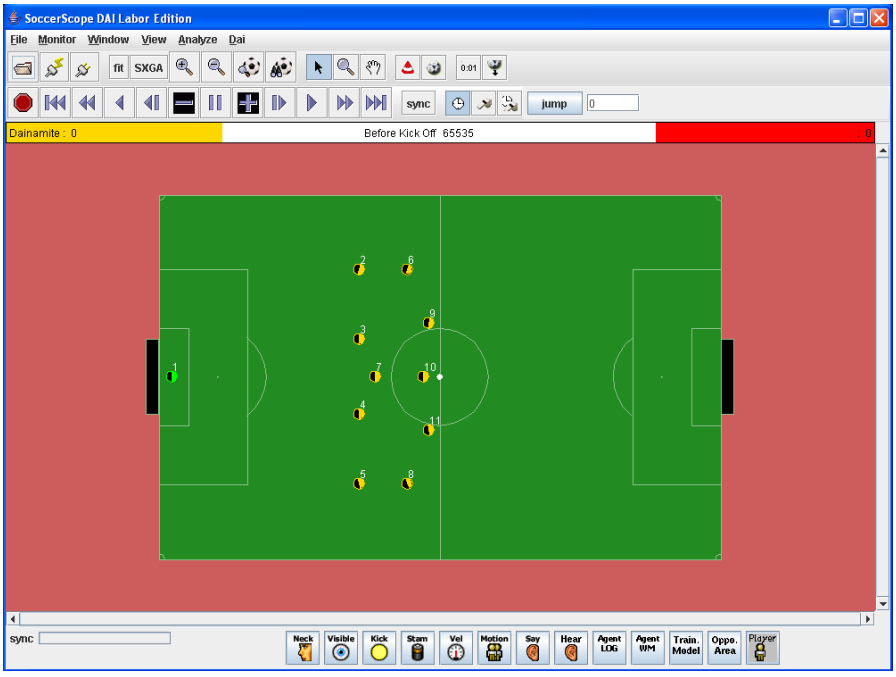


Figure 12.1: The Soccerscope main window.

Figure 12.2: WM On/Off.

AgentWorldModel). Some of the actions, which are accessible via this menu are also executable by the button-group on the bottom of the monitor. With the *Analyze*-menu, different analysis-tools can be switched on or off, e.g. the offside line, the pass courses or dominant regions. The visualizations are quite intuitive, so just try them out to see, how they work. Note, that some of them consume quite much cpu-time, such that they should be used on logfiles rather than on running games.

All added functionalities are placed in the menu called *Dai*. With the *actions* item it is possible to send actions for a player (its tactic has to be switched off therefore). *Player Data* shows a dialog with the current player data (e.g. its position). Statistics opens the the world model statistic selection window. *Tactic off* turns all dainamite players decision processes off, and to reduce the cpu usage it is possible to turn off the recording off all world model data. It is also possible to turn off only a couple of values (via Refine).

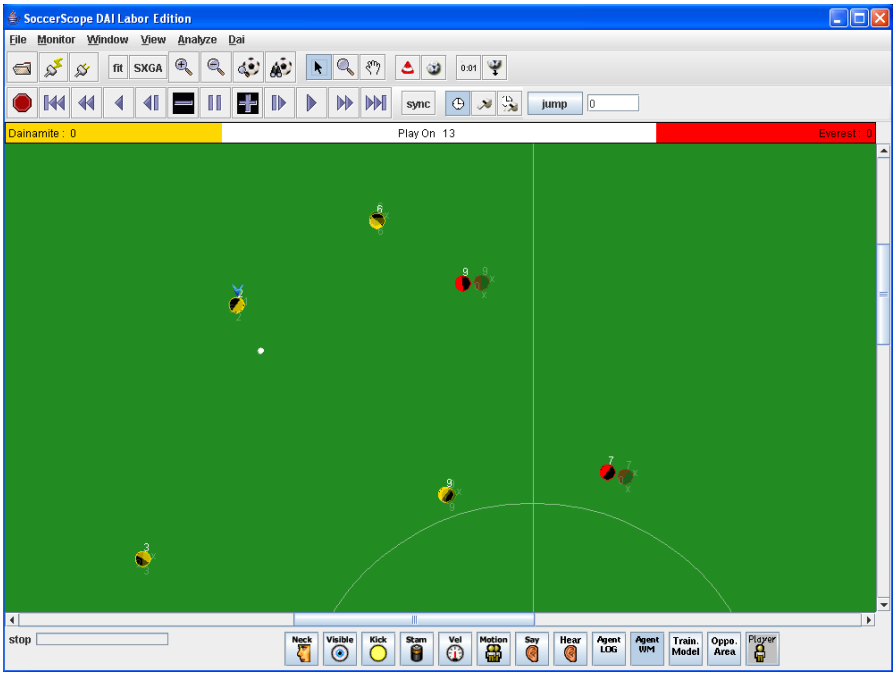


Figure 12.3: SoccerScope showing the World-Model Data of Player number 2 (Yellow).

To show the world model of a dainamite player click on *Agent WM*-Button (located on bottom right, see Figure 12.2) and then on the player of interest. Now SoccerScope additionally shows players which looks like shadows. These are the player locations of the world model from the selected player, as shown in Figure 12.3. The ball has a white *shadow* as well, and a players expected stamina may also be shown (use menu *View* → *AgentWorldModel* to refine the view). If shadows of players become darker, the corresponding information is set to *unreliable*. That happens for example in cases, when the players weren't seen for long. Additionally, each player-shadow has some numbers next to it, as shown in Figure 12.4. These are the players seen tricot number (above the shadow), its role number (below), and its expected ball-interception time (right to the shadow), which in the figure for player 11 is 2. The expected interception point can be displayed by switching the information on within the corresponding *View*-menu.

A short summary of tactical information of a certain player is accessible via the context menu, which is shown by right-clicking on a player (Figure 12.5). In the submenu *States*, all executable states of the current cycle are listed together with their assessment value. In brackets, more detailed information about the state can be shown. Most other entries of that menu allow the same actiosn as the *View*-menu. In the *Property*-menu, its knowledge about himself can be seen textually, together with information about its communication (heard and send messages).

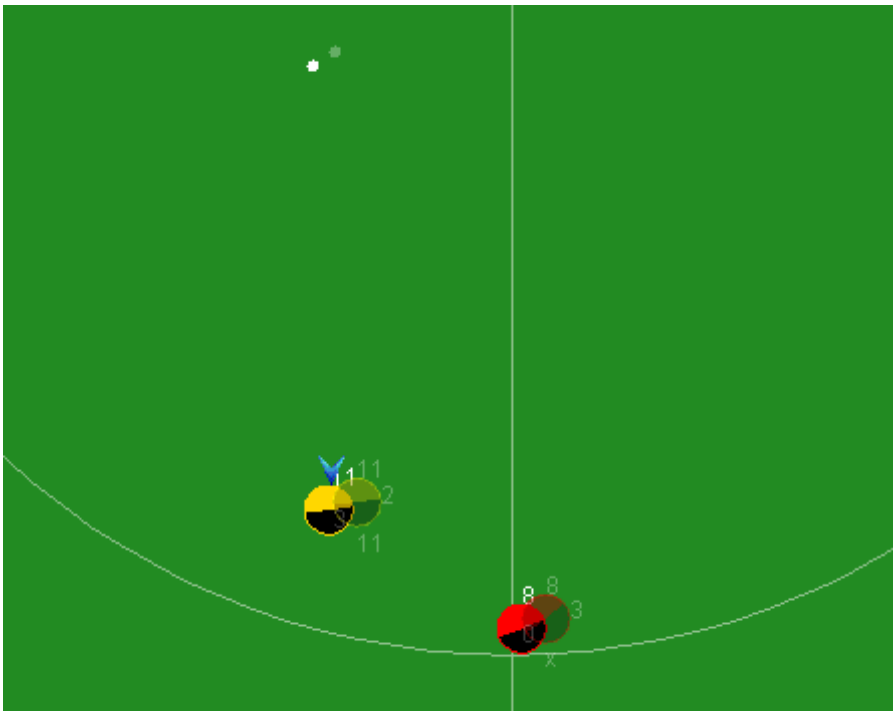


Figure 12.4: The World-Model Information of a Player

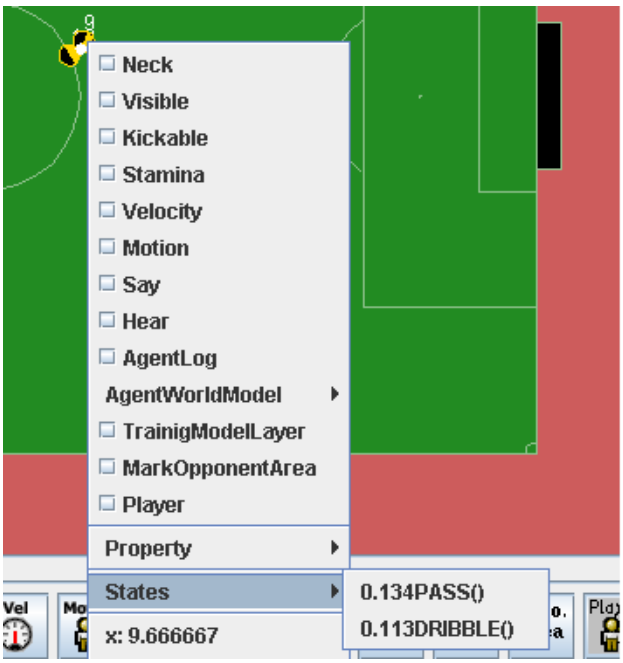


Figure 12.5: The World-Model Information of a Player

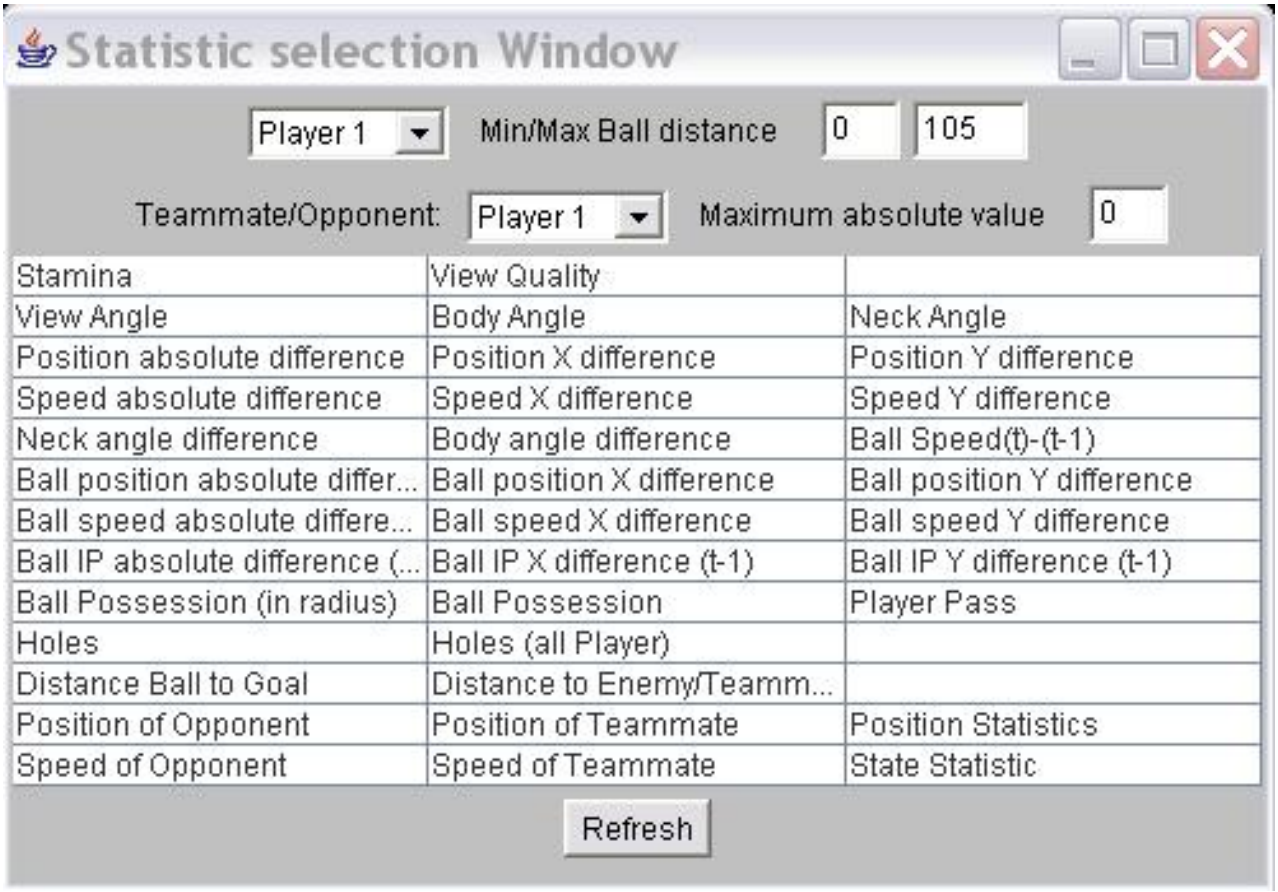


Figure 12.6: The statistic selection window.

The World Model Statistics

The statistic selection window (Figure 12.6) shows all available statistics and analyses. The first choice selector selects the dainamite player of interest. The second selector selects a teammate or an opponent player. That player number is used by analyses that depends on a second player (like distance to enemy player). Min/max ball distance ignores analyses depends on the ball distance less or greater than the given values. For example, it is not really interesting how big is the difference of the ball positions in a distance of 100 Meter. That value will result in a bad scaling. Min/max ball distance prevent that. Also for a better scaling it is possible to cut off the charts by a given value (maximal absolute value). Note that some statistics currently do not work (e.g. the state overview).

12.3 The Training scenario editor

12.3.1 Prerequisite

In order to use the training scenarios described subsequently, you have to use an adpated version of the soccer server, which is available at our website *http://www.dainamite.de*. This was extended to support a new option (called *train*), that disables the *move*-command for players in order to ease the scenario setup. Additionally a new command for the trainer is implemented, which allows to disconnect players. This is important, if testing is done with fewer than 11 agents per team.

12.3.2 Introduction

What is a training scenario? If you want to test your changes of the code, fix bugs or simply analyse the agents' behavíour for the sake of improvement, it is not merely sufficient to observate a running game of 22 twin-coloured points fighting for the ball. Indeed, you have to expose the agents to a specific test-situation where their reaction can be easily associated to but a few simple influencing factors. Such a situation is called trainings scenario. Such situations can be created, saved to a file, and run in the "train" mode of the soccer server.

Example (for the training of the defender): Set one opponent in the middle of the field, the ball at his feet. Put your own defender between him and the goal while the goal keeper has taken his place in the goal. When the scenario is kicked off, the opponent will try get the ball in the goal, having to bypass your defender. This is a wonderful way to observate your own defender's behaviour and to find out about the mistakes he makes.

Let us firstly take a closer look at soccerscope.

12.3.3 How to create a training scenario with Soccerscope

Starting Soccerscope

Soccerscope is a viewer of the playing field that can connect to the soccerserver, displaying the game that is currently played by the help of this very server. However, for the purpose of creating a training scenario we do not need to start a server, we will simply run Soccerscope on its own. The executing class is `Main` in the package `src.soccerscope`.

A screenshot, showing the settings, can be seen in 12.7

Please, do not forget to set the "Working directory" in the next tab:
${workspace_loc:Robocup}/etc/agents/

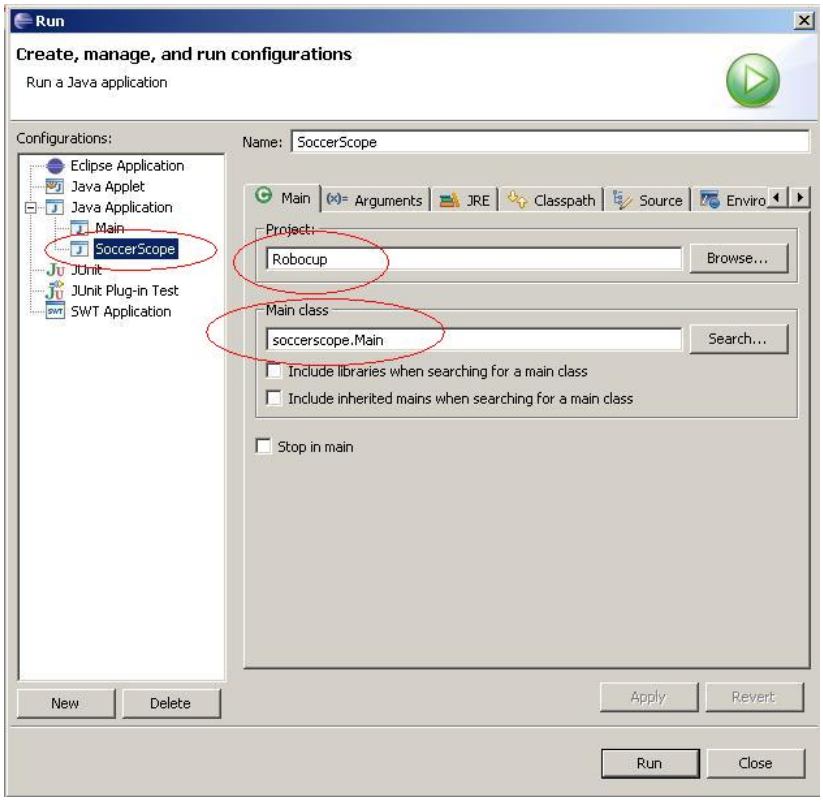


Figure 12.7: "Run Soccerscope" in Eclipse

Use Soccerscope and the Training Scenario Editor

To open the Training Scenario Editor make a middle-click on the playing field or press the button Open Training Scenario Editor (1.3). The window 12.9 will open.

The buttons "Load" and "Save" (2.1) at the bottom of the window indicate that this window not only displays the tool that will configure the settings of the training scenario, but that we also have to lateron save it to a file in order to let it run from that file. So let us start with the configuration of the training scenario.

12.3.4 How to configure a training scenario

We will now take a look at how to configure an examplary training scenario which simulates an attack by a single opponent on a single one of our defenders.

Placing players

Remember that a training scenario is certain game-situation that a game-simulation starts to run from. So how are we going to start? The first thing to do is to put the players to the

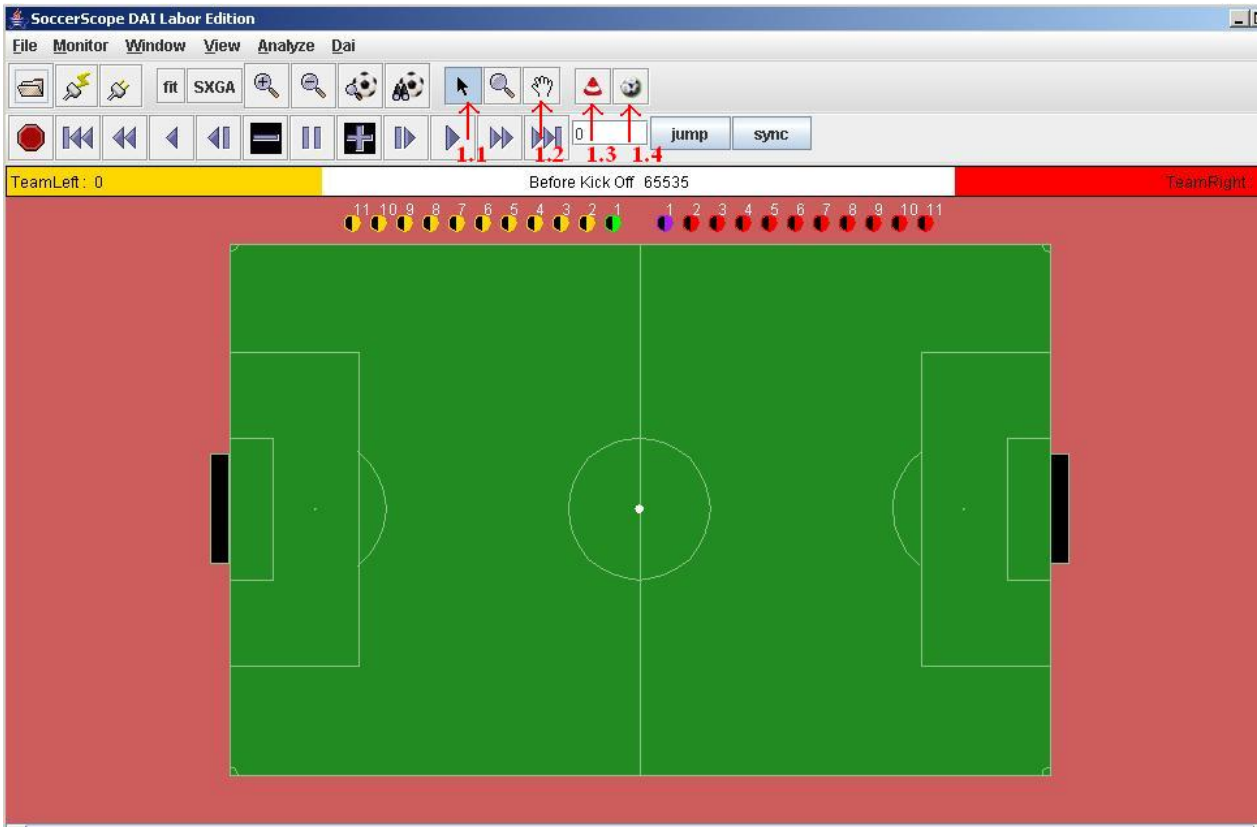


Figure 12.8: Describing Soccerscope's training functionalities

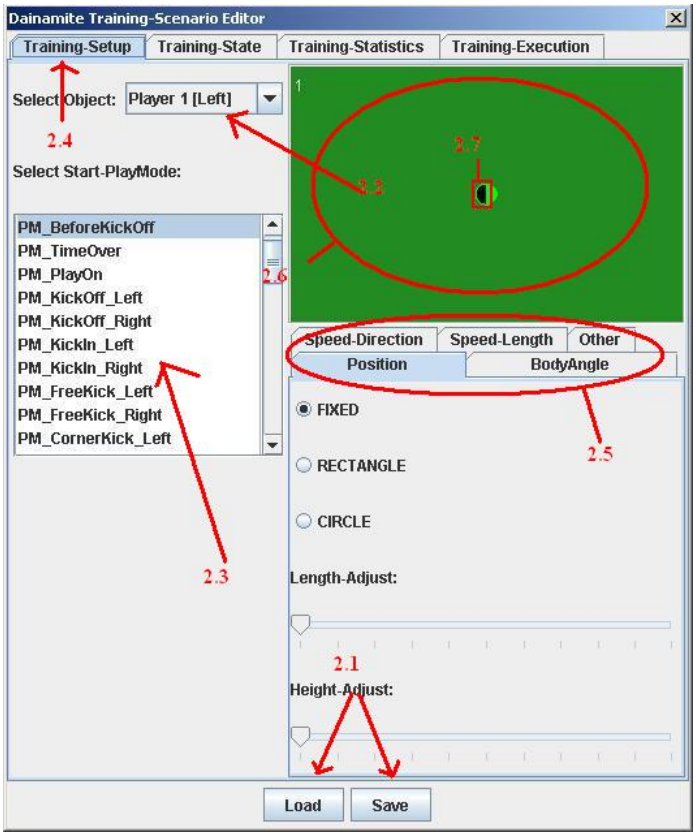


Figure 12.9: The Training Scenario Editor

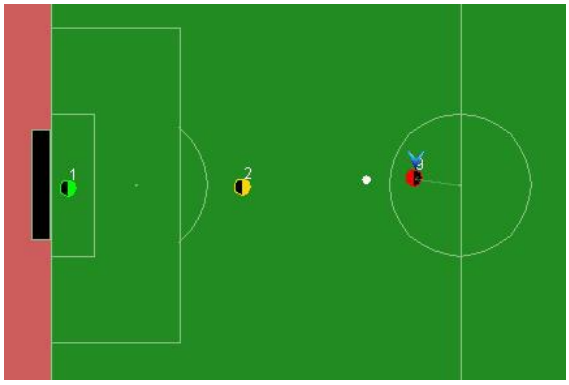


Figure 12.10: Players in their positions to form a typical defending-situation

points they are to occupy in the first cycle of the simulation. Click on the "Hand"-button (1.2) to be able to move the players (and the ball!) with the mouse and let them form the constellation you desire. (Leave those you do not need keep standing at the side, they will disappear when the game starts.) Our example would look like figure 12.10

Please note that you can not (yet) chose any player number for any purpose (defender, goalie, ...) Each number of the players in our team are attributed to a specific task on the playing field. In our example we have chosen number two because number two is a defending character. The opponent is number nine, which is of no special importance as the restrictions in the usage of our own players are not valid with other known teams. Check out the file `team.conf` in the package `etc.agents` to know the "who-is-who" of our team.

Now there are some configurations that are player-specific and others which are not. Take a look the Training Scenario Editor again. There is a drop-down-menu (2.2) that lets you select each one of the 22 players on the field. Select one of the players you just set on the field. Alternatively use the select-mode by clicking the blue button displaying a mouse-pointer-symbol (1.1). In this mode you can select players by the help of the mouse, but once you click on one of them please note that the pull-down-menu (2.2) in the Training Scenario Editor is automatically adjusted. We start with the settings that are player-specific.

Player specific settings

The first tab ("Training Setup", 2.4) should be opened already as this is the default tab the trainings scenario editor starts with. Take a look at the five tabs (2.5) in the downer right corner. The player configurations are only set in these five tabs. The following text describes the functionality of each tab and how to set it up:

- "Position": The player is usually set to exactly the same coordinates each time you run the scenario ("FIXED"). You can change this by selecting a "CIRCLE" or a "RECTANGLE". The player is then set into a random position within the selected area. Please do not forget to adjust lenght AND height, otherwise the setup position of the player

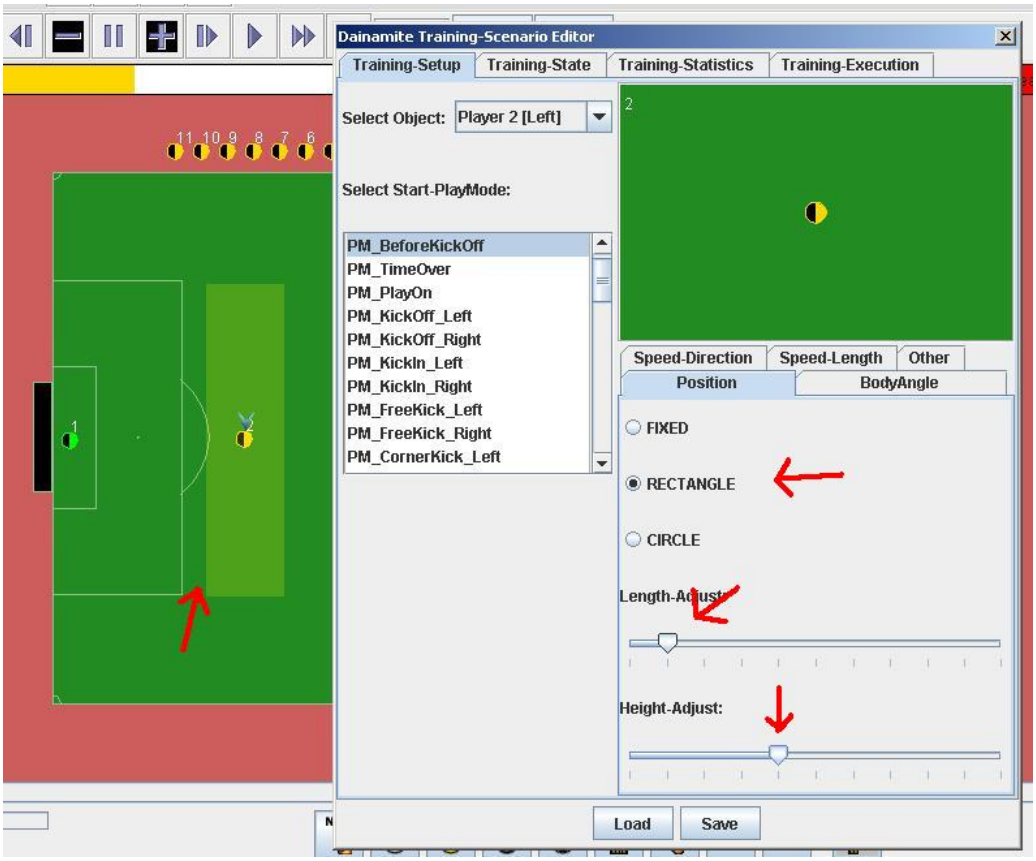


Figure 12.11: Example for a player having a rectangular starting area

will stay a line.

- "BodyAngle": Before you make a change to the configuration in this tab you should give the player a body-direction. To this end, click at any point in the green field over the tab (2.6). The player will turn his body to the point you have clicked at. This will show in the display as a move of the black half-round, which is the players back. (2.7) Now you can select a bunch of possible directions that the body of the player points to. You can do this by the help of the ruler.

  The wider an angle you select, the wider is the "bunch". At the start of the game one of those body-directions is randomly chosen from the range you have permitted.
  Our example does not need such a diffusion. However, the body direction has necessarily to be adjusted, as every player initially looks right, but has to look left for our purposes (to see the ball and try to bypass our defender with it)!

- "Speed-direction": The player has an initial speed. Select the check box if you want this speed's direction to be the same as the body-direction. (This is not necessarily so: The inertia of the player might cause his body to further move in a certain direction while his body-direction has already deliberately turned in order to move elsewhere (because you can only start a move in a certain direction if your body angle aligns its direction)).

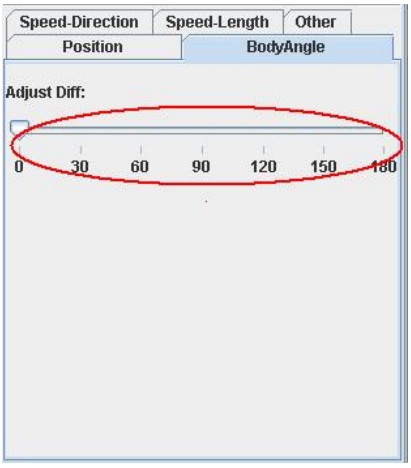


Figure 12.12: Body-Angle Ruler

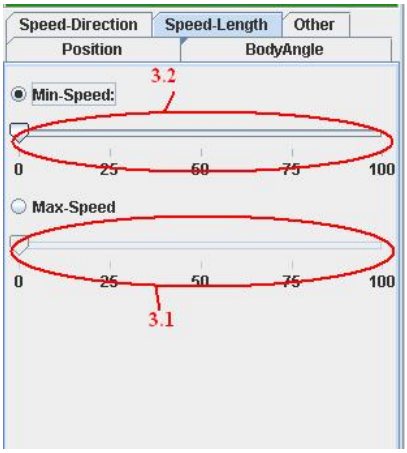


Figure 12.13: Speed-Length Ruler

You can make the speed's direction vary by plus/minus a range of angles that you can determine by the help of the ruler.
There is no need of a specific direction in our example.

- "Speed-Length": Speed-length means the length of the speed-vector, which is last of all how we generally measure the speed in this project. Select either an upper (3.1) or a lower limit (3.2) for the speed.

  At the beginning of the scenario a certain speed according to the selected criteria is chosen. (In our example you can let the minimum speed remain zero.)

- "Other" (#): To set a player "Active" is to let him participate in the game. If not "Active", the trainer will disconnect that player at the beginning of the scenario. Select "Round-View", to let the player have a good sight around. Actually, the trainer will send a message to the agents with all positions and speeds of the dynamic objects. Select both check box' for our example.

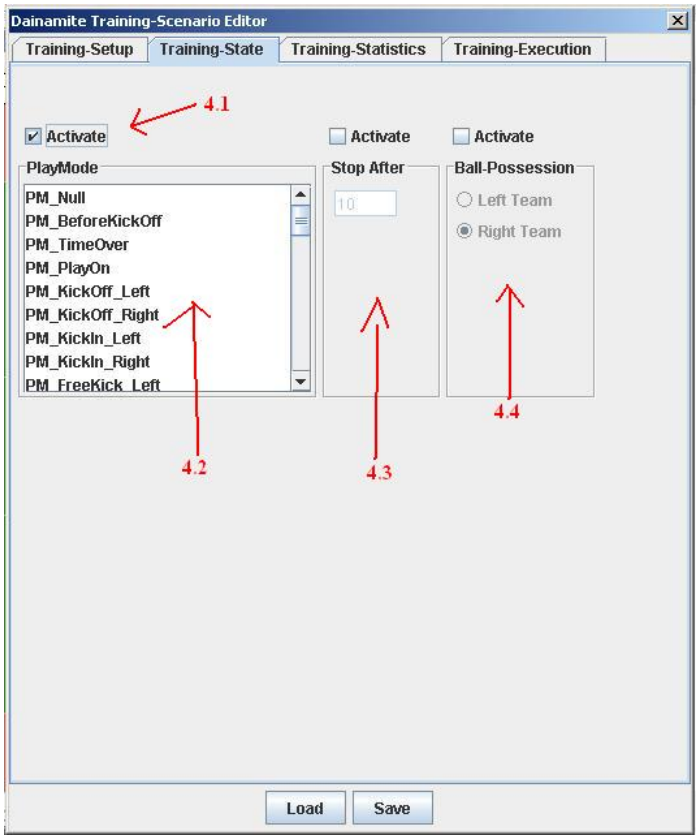


Figure 12.14: Second tab: "Training-State"

General (player-independent) scenario settings

First tab, "Training-Setup" (which should still be opened) We have already described the "Select Object"-pull-down-menu (2.2). There is just one last object we have to adjust before we can switch to the "Training-State"-tab (2.4): The menu "Select Start-Playmode" (2.3):

- The concept of playmodes is first and foremost to enable the soccer server to apply different "rules of the game" during situations such as "Corner Kick", "Free Kick", "Offside" and so forth. For a player, it is also necessary to know which playmode the game is in, as he is p.e. not allowed to kick the ball in "Corner-Kick"-mode, rather should he look for an advantageous position.
  "Play On" is the normal playmode during a football game. Now, we have to chose the playmode that our scenario is to start with. As with most scenarios, "Play On" is the right one for our defender's scenario.

Second tab, "Training-State"(#): Open the second tab now, marked "Training-State". There are three different configurations to set in this tab:

- The leftmost select box ("Playmode", 4.2) contains a lot of different playmodes to chose. Once chosen, as soon as they are activated during the run, they will immediately cancel the scenario's execution. This can happen because the playmode often changes during the execution of a game/scenario. Check the "Activate"-box (4.1) to use this function and select several break-conditions (4.2).
  For the defending scenario we might chose break conditions such as FreeKick_Left, GoalKick_Left, Foul_Charge_Left/-_Right, etc, as all these mean that the test situation is over or not longer valuable to observate.

- The number "Stop After" (4.3) indicates the number of cycles which a scenario terminates after. Check "Activate" if you want to use the function. If you enter a number, please, confirm the number with Enter! Otherwise it will not take effect. This is not needed for the defending scenario, as we want to observate the behaviour of the player till a situation appears where the defense is either succesful or not, only then can we draw conclusions. However, this setting is helpful for situations where the player is to react (quickly) in a certain range of cycles and later behaviour is not relevant.

- The scenario can also abort as soon as a certain team get hold of the ball. In order to use this function, check "Activate" in the rightmost check box (4.4) and chose a team. This function is very necessary for the defending scenario, as the ball possession of our team means a succesful defense and suffices to justify the termination of the scenario.

Third tab, "Training-Statistics": Not implemented yet.

Fourth tab, "Training-Execution": Not implemented yet.

Do not forget to save your scenario to a file by clicking "Save" (2.1) and following the usual procedure. Save the file with the extension .xml because this is what it is. You can open it with an ASCII-editor and have a look. Most entries are self-explanatory if you know the Trainings Scenario Editor's GUI. For the creation of the XML the XStream-library (xstream.codehaus.org) was used. However, it occured already that the Trainings Scenario Editor did not save the right coordinates of a player. Then you must edit his position in the XML.

12.3.5 Running the scenario

First of all, how to start a trainings scenario from the file you have just saved? These are the steps to take:

1. Start the soccer server in train mode.
   You can do so by running it like this in the command line:
   $rcssserver train server::coach=true server::coach_w_referee=true

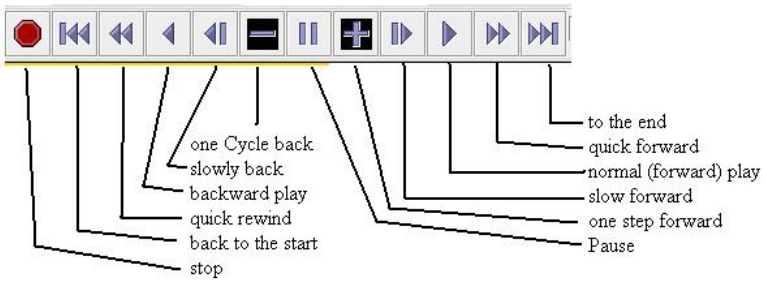


Figure 12.15: Scenario-Run Control-buttons

server::forbid_kick_off_offside=false server::half_time = -1

If you are a person of good taste and find this too long a line, simply edit the file server.conf, which you find here: [home directory]/.rcssserver/server.conf. Set the parameters mentioned in the command line in the same way. You can then do without them in the command line; it will only look like this:
$rcssserver train

2. Start your team as usual. But beware that this part is a little tricky: If you have edited the scenario with our team left and the opponent's at the right, you have to make sure that the scenario runs in the same way! Our team is not yet able to handle the situation if you don't. You should know that the team started first is the one that is set up at the left side, the second is right.

3. Once the players are all present, klick on the ball-button (1.4).

4. Select the file that you previously saved the scenario to and click "Open". The scenario should run now.

5. You can control the run of the scenario by turning the mouse wheel (advance - rewind), or by clicking on these buttons in Soccerscope:

12.3.6 Outlook

The Trainings Scenario Editor is our most important tool when designing the players intelligence, and it is already highly usable. However, there are still things to be done:

- There is a bug in saving complex scenarios to a file, which occurs from time to time. The data is then not saved the way desired. This is what I mentioned above, referring to the misplaced position of a player.

- Two out of four tabs within the Training Scenario Editor are not yet filled with the functionality attributed to it.
- Our team is not yet able to handle the situation of a scenario run where sides are exchanged compared to the creation of this scenario. This could be adjusted.
- Players of other teams can adapt to any necessary character, regardless of the number on their back. Our defender would not "dare" to kick the ball in the opponents' goal if he was the only one of his team on the field (for example in trainings scenarios).

12.4 Database

12.4.1 Introduction

Until now scenes are stored by robocup soccer server (logfile *.rcg) and all scenes of a game are stored in only one file. When we later want to open the file, we must have access to the robocup soccer server folder on the network. SoccerScope is now added with database functions. Storing scenes to the database equals to recording, a new game has to be openend, and each scene is added into this subsequently, such that only interesting parts may be stored. Besides, we can use SQL-Queries to make some certain statistics more efficiently.

12.4.2 Setup

This section details how to setup a database server, so SoccerScope's functions will work.

SoccerScope works with MySQL. A free version (Commnunity Edition) of the database server MySQL can be downloaded at `http://dev.mysql.com/downloads/`. SoccerScope's database functions were developed and tested with the old release MySQL 4.1, but current release 5.0 should work well.

After installing the database server you need to create one database and working tables. In robocup folder/etc/ you will find the script file robocup.sql for this. Import this script and all required tables will be created for you. It is assumed that the database robocup doesn't exists and will be created. To import this script type the following command on the shell

mysql -u *username* -p *password* < robocup.sql

The importing can be done on the command line as described but to manage a database easily, it is suggested to use a database management program. One very popular web based program is phpMyAdmin (current version is 2.9.0.2), which can be freely downloaded at `http://www.phpmyadmin.net/`. However one must install a web server, for example an

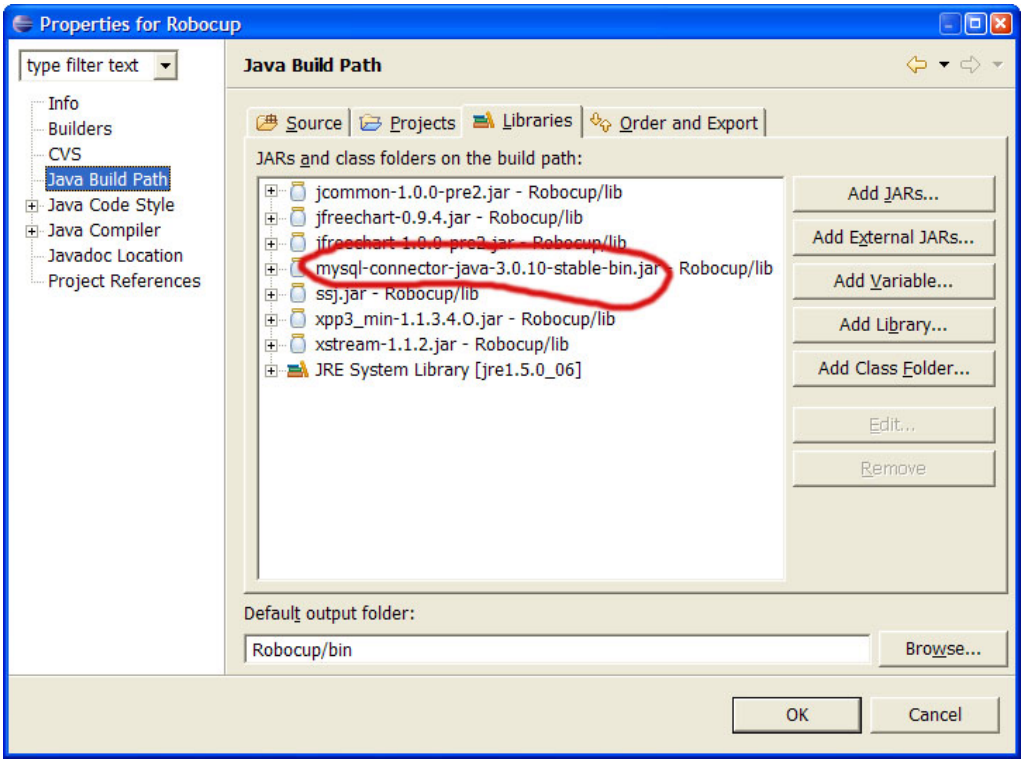


Figure 12.16: MySQL in project

Apache (`http://httpd.apache.org/`) and the PHP-Interpreter (`http://www.php.net/`).

Installing and configuring these mentioned programs are a bit complex. Alternatively one can use the all in one package XAMPP (`http://www.apachefriends.org/en/xampp.html`). XAMPP includes Apache, PHP, MySQL and phpMyAdmin. There are versions for Windows and Linux) and they can be even used without installation.

The easiest and most comfortable way to work with MySQL is working with EMS SQL Manager for MySQL. A trial version is available at `http://www.sqlmanager.net/en/products/mysql/manager`.

Java programs work with MySQL thanks to the library mysql-connector (`http://www.mysql.com/products/connector/j/`), which is available in robocup repository/lib. You muss ensure that the connector is bound with robocup project as in Figure 12.16.

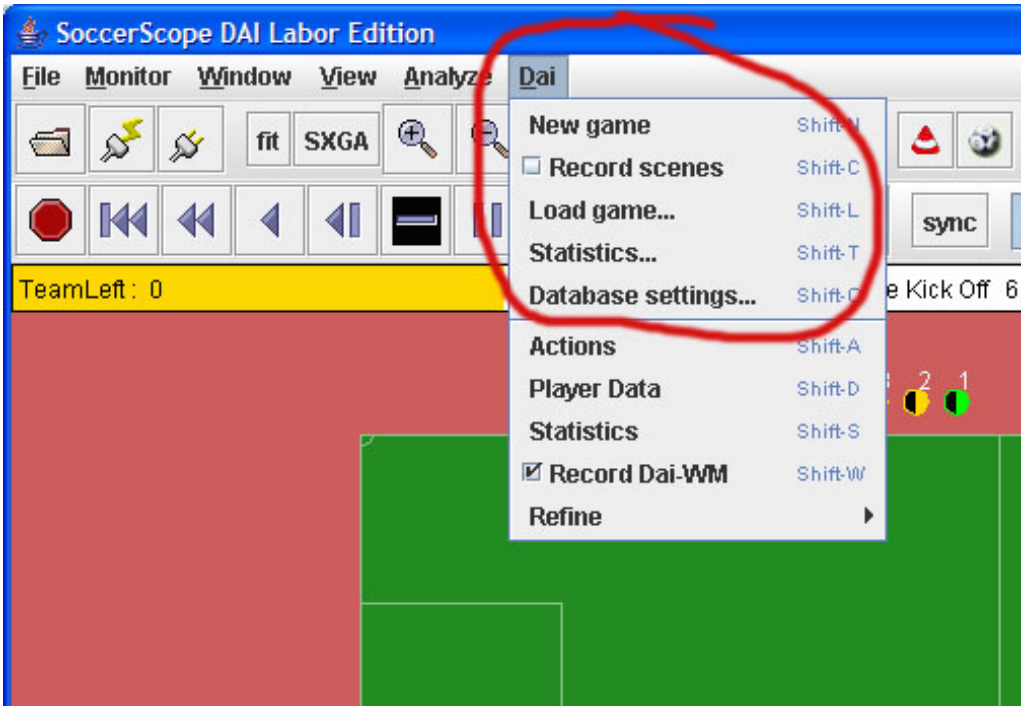


Figure 12.17: Added items to Dai menu

12.4.3 Usage

In order to connect with the database server soccerscope must know its connection settings. These settings about server name, database name, user name and password... are in the file robocup/etc/database.config and can be edited directly. It can also be done later through menu. Run soccerscope and one will see under menu Dai (Figure 12.17) five new menu items at the top.

Figure 12.18: Database settings

Database settings

With *Database setting...* information about the database can be edited (Fig. 12.18). In addition to server name, database name and password there are two options

- When the check box *Record only after both teams are loaded* is checked, recording starts only when two teams are loaded.
- When the second check box is checked no scene is recorded until the game starts.

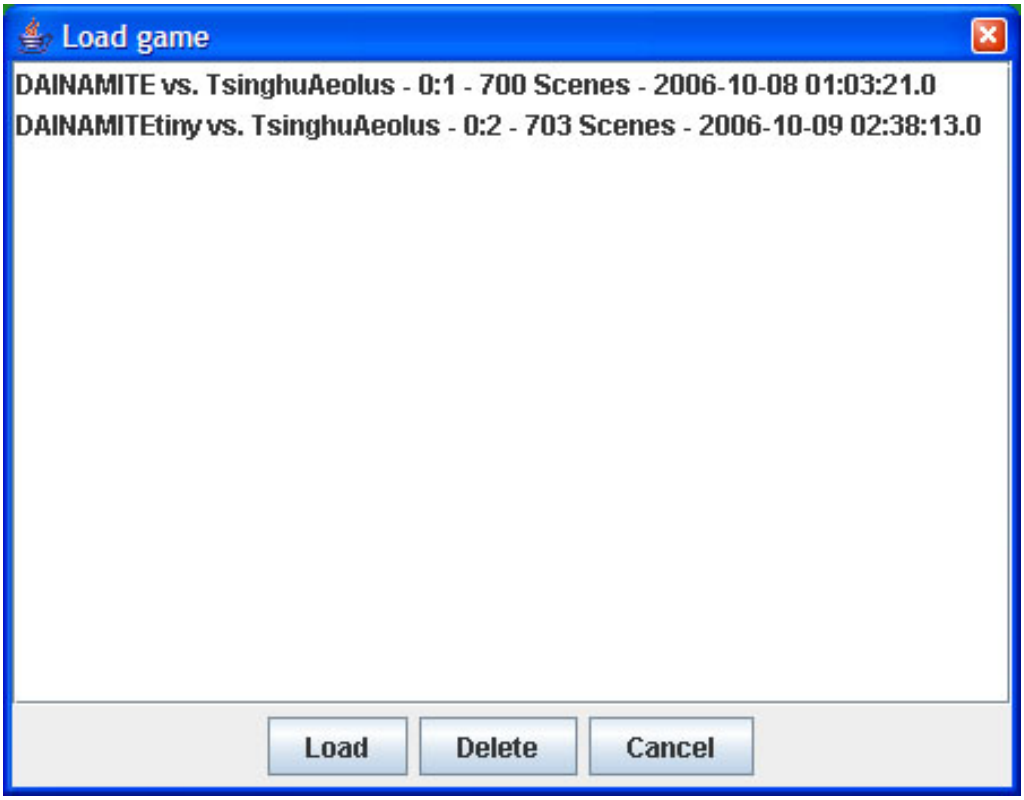


Figure 12.19: Load game

Record a game

A game is simply a sequence of several scenes. One can start, pause and stop recording at any time and as many times as desired.

By click on *New game* a new game is started, but scenes are not recorded until click on *Record scenes*. By click once more time on *Record scenes* it is paused. A game is saved with team names, score, number of scenes and the moment, at which the game was started, e.g. "DAINAMITE vs. TsinghuAeoplus - 0:1 - 700 Scenes - 2006-10-08 01:03:21.0".

The check box *Record scenes* on the menu Dai is checked indicates that scenes are recorded.

Load a game

A dialog box for opening saved games (Fig. 12.19) appears when one clicks on *Load game...* shows a list of recorded games with extra information like number of scenes, score...

View statistics

Statistic can only be showed for saved games. To view statistics click on *Statistics...*, a window (Fig. 12.20) will open.

On the top of this window is a list of saved games. Click on a game and statistics for that game will be showed at the bottom. The number in brackets show when the event happended.

12.4.4 Structure and Implementation

The program's data model is depicted on Figure 12.21

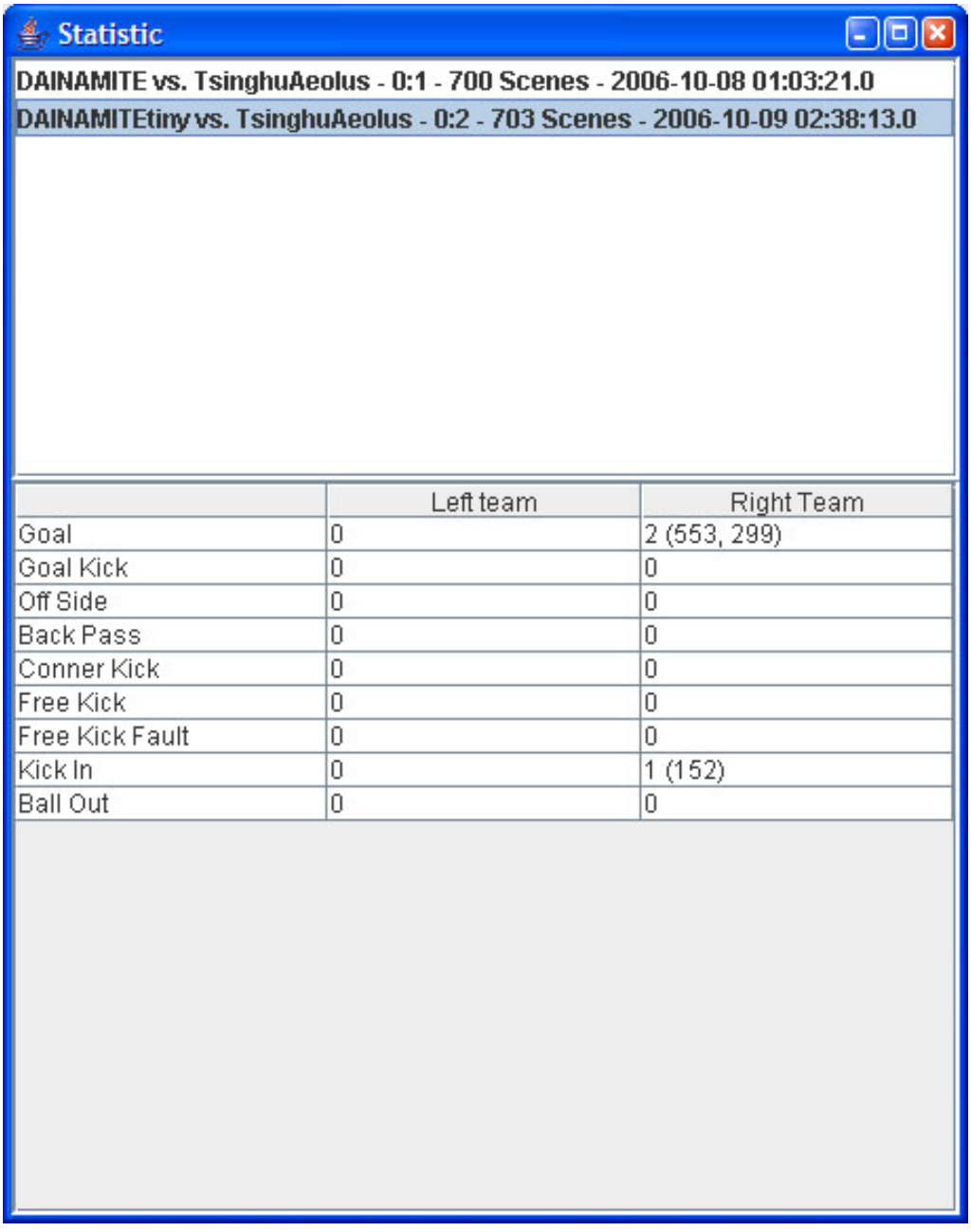
Colored classes are saved in respective tables. We look at the E-R model (12.22) of the database now

Classes for database functions:

- soccerscope.db.DatabaseConfig: database setting
- soccerscope.db.RobocupDatabase: scene saving methods
- soccerscope.db.DatabaseReader: methods for loading a game
- soccerscope.view.LoadGameDialog: dialog box for choosing a game to load
- soccerscope.view.StatisticFrame: window to show statistics
- soccerscope.util.analyze.DBAnalyzer: statistic methods

12.4.5 Remarks

The database is currently in a beta status. It is mainly tested under windows, and there might be still some bugs in its implementation. However, we think it is a good thing to have the database, and future work will, beside bugfixing, concentrate on analysis-queries and their integration into the monitor.



	Left team	Right Team
Goal	0	2 (553, 299)
Goal Kick	0	0
Off Side	0	0
Back Pass	0	0
Conner Kick	0	0
Free Kick	0	0
Free Kick Fault	0	0
Kick In	0	1 (152)
Ball Out	0	0

Figure 12.20: Statistics

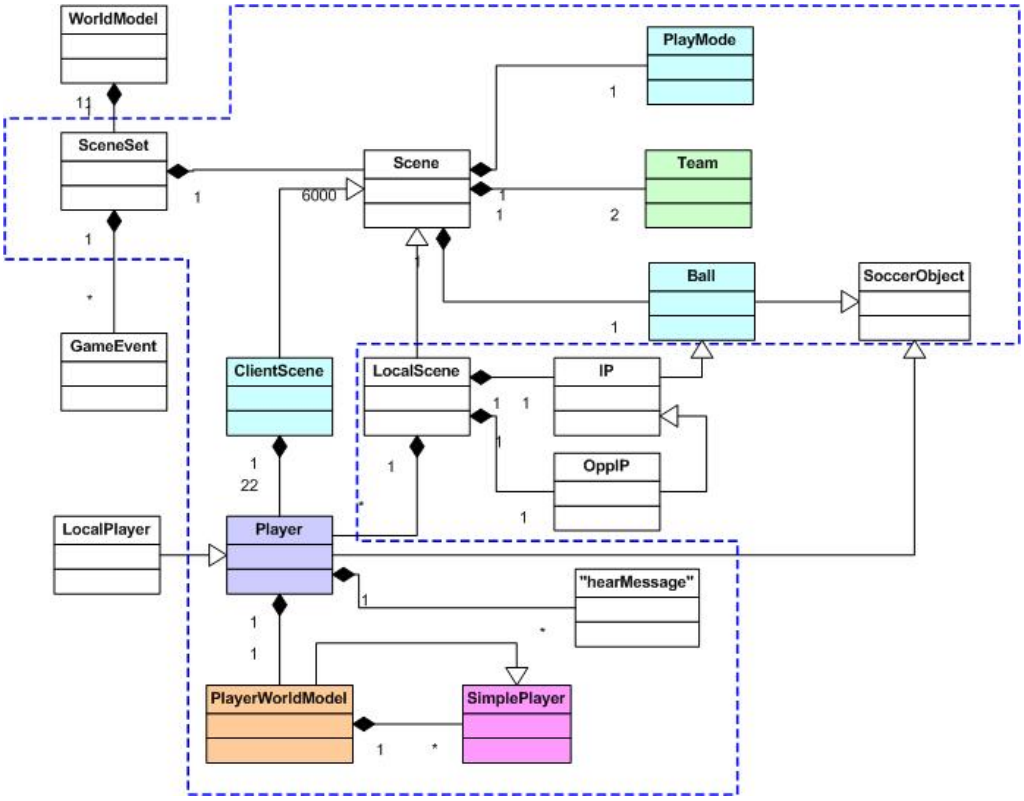


Figure 12.21: Data model

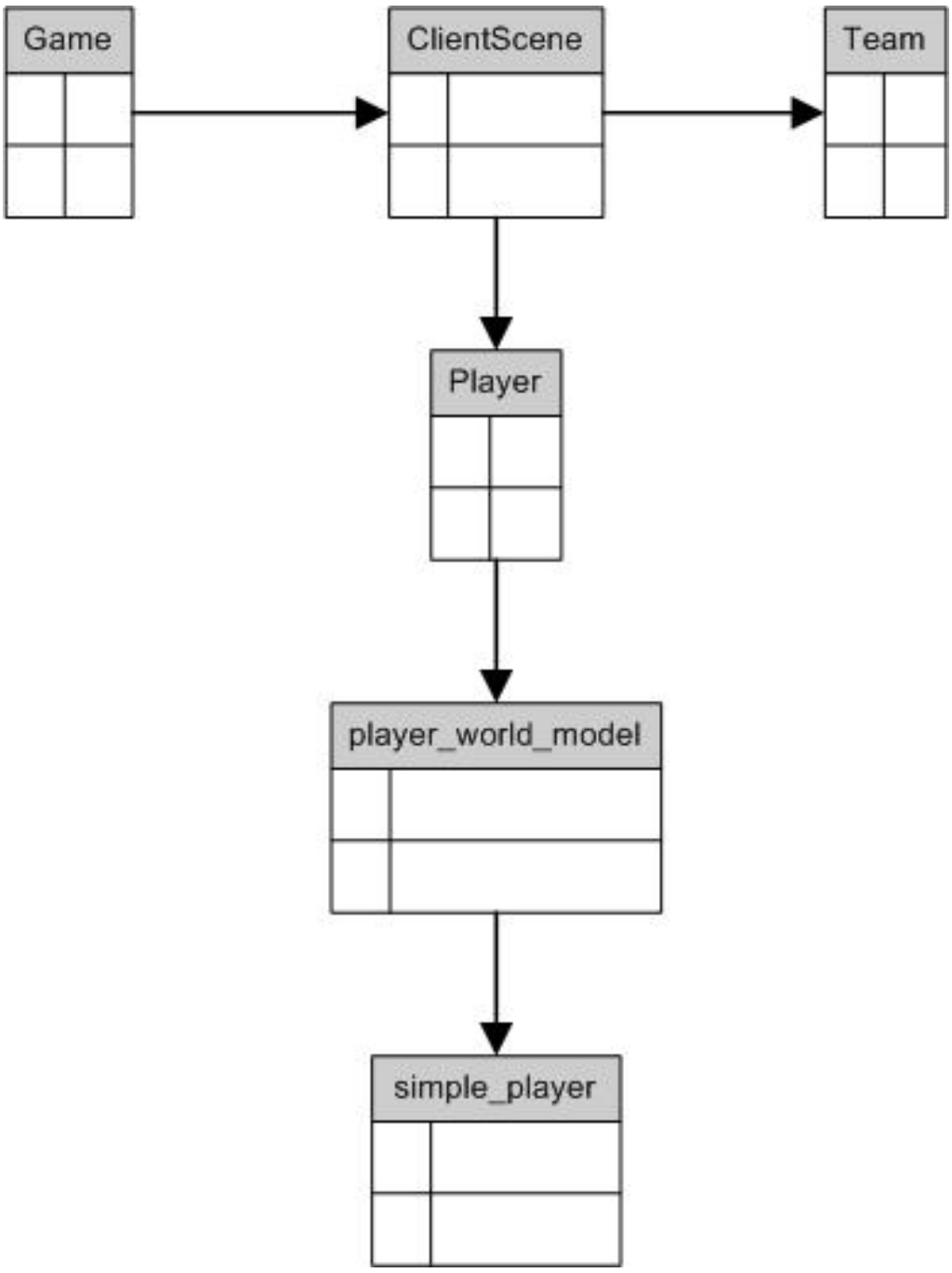


Figure 12.22: ER-Model

13 Acknowledgement

First of all we like to thank Prof. S. Albayrak, the head of the DAI-Labor, for supporting us in the development of the dainamite team. Next, we like to thank the YowAI team for allowing us to publish an extended version of their SoccerScope implementation, which enriches this release significantly. For their great help at the Robocup Competition 2006 in Bremen special thanks go to Jan Murray and Tomomi Kawarabayashi. Without them, we wouldn't be able to participate in the tournament (even with their help, we had to pass the qualification without our coach). Further thanks go to all students, which helped to developed the team and the tools. These are Rodin von Georgi, Paul Häder, Aubrey Schmidt, Karl Bartel, Maximilian Kern, Paul André, Luzian Wild, Grzegorz Lehmann, Tomasz Olszewski, Felix Brossmann, Janis Danisevskis, Wai-Lung Lee, Sebastian Peters, Dirk Roscher, Oliver Thiel, Winfried Umbrath, Till Klister, Sebastian Boelter, Judith Rohloff, Jürgen Widiker, Peter Sander, Andreas Windisch, Moritz Hilger, Jakob Uszkoreit, Stefan Harke, Carsten Wirth, Martin Eismann, Marco Lützenberger, Michael Waller, Daniel Bicher, Florian Gödde, Andreas Baginski, Dennis Hamerla, Dennis von Ferenczy, Thomas Helm, Matthias Runge, Tian Kuan, Benjamin Böttcher and Linh Phong Le. Finally we like to thank Hiroaki Kitano and all others of the Robocup community to establish and maintain this research initiative.

Team Dainamite

14 Appendix

14.1 The CLang Grammar

`Package: robocup.parser9.CLangParser`

The following Grammar specifies the standard coach language and complies with version 7 and 8. The implementation can be found in CLangParser.jj in package robocup.parser9.

```
<MESSAGE> :  <FREEFORM_MESS> | <DEFINE_MESS> | <RULE_MESS> | <DEL_MESS>
           | <INFO_MESS> | <ADVICE_MESS> | <META_MESS>

<RULE_MESS> :  (rule <ACTIVATION_LIST>)

<DEL_MESS> :  (delete <ID_LIST>)

<DEFINE_MESS> :  (define <DEFINE_TOKEN_LIST>)

<FREEFORM_MESS> :  (freeform <CLANG_STR>)

<INFO_MESS> :  (info <TOKEN_LIST>)

<ADVICE_MESS> :  (advice <TOKEN_LIST>)

<TOKEN_LIST> :  <TOKEN_LIST> <TOKEN> | <TOKEN>

<TOKEN> :  (<TIME> <CONDITION> <DIRECTIVE_LIST>) | (clear)

<META_MESS> :  (meta <META_TOKEN_LIST>)

<META_TOKEN_LIST> :  <META_TOKEN_LIST> <META_TOKEN> | <META_TOKEN>

<META_TOKEN> :  (ver [int])

<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_]*
```

Bibliography

[1] The eclipse project http://www.eclipse.org.

[2] Gnu general public license, version 2, june 1991
http://www.fsf.org/licensing/licenses/gpl.txt.

[3] Javacc: The java compiler compiler https://javacc.dev.java.net/.

[4] Mysql website http://www.mysql.com/.

[5] Ssj: Stochastic simulation in java http://www.iro.umontreal.ca/ lecuyer/ssj/.

[6] Xstream website http://xstream.codehaus.org/license.html.

[7] Mao Cheny, Klaus Dorer, Ehsan Foroughi, Fredrik Heintz, ZhanXiang Huangy, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Jan Murray, Itsuki Noda, Oliver Obst, Pat Riley, Timo Stevens, Yi Wangy, and Xiang Yiny. Robocup soccer server. 2003.

[8] R. de Boer and J. Kok. The incremental development of a synthetic multi-agent system: The uva trilearn 2001 robotic soccer simulation team. Master's thesis, University of Amsterdam, 2002.

[9] Jfree website: http://www.jfree.org/jfreechart/index.php, Last visited 23 Nov 2006.

[10] 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.

[11] S. Takahashi. Soccerscope website:
http://ne.cs.uec.ac.jp/~newone/soccerscope2003/, Last visited 23 Nov 2006.