Technologie-Zentrum Informatik

Technical Report 36

Virtual Werder

Team Documentation 2006

**Andreas D. Lattner, Carsten Rachuy, Arne Stahlbock,
Ubbo Visser and Tobias Warden**

TZI-Bericht Nr. 36
2006

Universität Bremen

TZI-Berichte

Herausgeber:
Technologie-Zentrum Informatik
Universität Bremen
Am Fallturm 1
28359 Bremen
Telefon: +49-421-218-7272
Fax: +49-421-218-7820
E-Mail: info@tzi.de
http://www.tzi.de

ISSN 1613-3773

Table of Contents 3

1 Introduction

This document describes the RoboCup 3D simulation league team Virtual Werder 3D of the Intelligent Systems department of the Center for Computing Technologies (TZI) at the Universitaet Bremen. RoboCup is an international initiative to bring together researchers from different fields like artificial intelligence and robotics. The long-term goal is to "*develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team*" by the year 2050[1]. RoboCup consists of different disciplines (RoboCupSoccer, RoboCupRescue, RoboCupJunior, and RoboCup@Home). RoboCupSoccer subdivides into different leagues– the real robot leagues (small-size, mid-size, Humanoid) and the simulation leagues. The 3D simulation league has been part of a RoboCup tournament in 2004 for the first time. It is a follow-up of the elder 2D simulation league. While the 2D simulation league simulates a flat world and lacks of a third dimension and a realistic physical model the 3D league addresses these issues [KO04].

Virtual Werder was founded in the year 1999 as a 2D simulation team. The 2D team took part at various RoboCup events starting in the year 2000 [DHS+01]. Efforts in the 3D simulation league have been started from the very first beginning of this league and Virtual Werder 3D has participated in all international 3D simulation league tournaments so far (Lisbon 2004, Osaka 2005, Bremen 2006, [BKN+04, KLP+05, LPR+06]). The greatest success has been to reach the quarter finals 2006 in Bremen where Virtual Werder 3D lost its decisive game and managed to reach rank eight out of 32 participating teams at the end.

This document consists of relevant information about our simulation league team. It describes how to install and use the team and gives an insight into the architecture, configuration and formation files and many different components like the knowledge base, behaviors and skills, communication and localization aspects. The intention of this paper is to describe how the system works and to provide means for future developers working on or extending the system.

Within this document it will be referred to different server parameters in some sections (e.g., the catch radius of the keeper's catch effector). These values are based on the RoboCup 3D soccer simulation server version 0.5.2 from May 24 2006. Text passages where other values are used are explicitely marked.

The documentation describes the Virtual Werder 3D agent of August 19 2006 (SVN rev. 1961) which functionally is compliant with the binary that took part at the RoboCup 2006 in Bremen but includes a number of performance improvements.

2 Installation and Usage

This section describes the system requirements, how the agent can be compiled and gives a more detailed view on the configure script as well as the way the agent is created, linked and finally run.

2.1 System Requirements

The following components have to be installed on the system in order to compile the agent binary. Compiling and running the agent has been tested with the versions given below. Other versions may work as well but have not been tested.

- gcc 4.0.1 / gcc 4.0.2 / gcc 4.1.0
- boost library 1.3.3
- bison 1.875 / bison 2.1
- flex 2.5.4 / flex 2.5.31

[1] http://www.robocup.org

2.2 Configure and Debug Options

Configuring the agent is done by, as usual, running the configure script. Though efforts had been made to utilize the autotools (automake/ autoconf)[2] we found out that due to some specialities in our source code regarding the build-order and the automated parser and code generation, using the autotools would have caused more difficulties than benefits. This caused us to create a configure script on our own. As the development of the agent continued and the code grew more and more complex the configure script had to meet an increasing number of requirements. Most of them were handled by adding some parameters to it which will be explained in the following.

All parameters are provided using the common form --PARAMETER or if arguments have to be provided --PARAMETER=ARGUMENT.

If program locations have to be provided the syntax is PROGRAM_NAME=ABSOLUTE_PATH.

- --help If this option is provided, a listing of all available options is printed to the screen.
- --exec=NAME This option allows to change the name of the executable. As a default the executable is called `agent.$USER` (e.g., if the user is called *werder* the executable would be named *agent.werder*).
- --prefix=PREFIX Providing this option allows for the user to set a custom prefix path. The prefix path defines the location where the executable is copied to when *make install* is called. The default value for this parameter is the user's home directory `$HOME`.
- --sym_prefix=PREFIX Providing this option allows for the user to set a custom symlink prefix path. The symlink prefix path defines the location where the symbolic link to the executable is created at. The default value is the user's home directory `$HOME`.
- --builddir=BUILD_DIR This option allows the user to change the directory where the object files, the binary and the library are built in. The default location is `./agent.build`.
- FLEX=PATH, BISON=PATH Our agent uses automatically generated parsers requiring both *flex* as well as *bison* to be available on the system. If the the *flex* or *bison* executable is located in a non-standard directory (e.g. `/usr/bin` or `/usr/local/bin`) this parameter is used to specify the location.
- --flags=FLAGS This option allows the user to provide additional compiler flags. Multiple flags have to be separated by *:* tokens. For a list of available compiler flags, please refer to the gcc manual [SC06].

The following parameters define the behavior of the debug streams (cf. section 4).

- --nodebug This option completely disables any debug output, including the monitor and the statistic debug streams. This should be the default option when compiling a binary for a competition for after compilation the debug system won't need any CPU time. This option overrides all following options. That means if --nodebug is provided in combination with --nobasicdebug, --nomonitordebug or --nostatisticdebug the latter will not have any effect.
- --nobasicdebug This option disables the basic debug streams *d_err*, *d_warn*, *d_info*, *d_dbg* and *d_misc* but leaving the monitor and statistic streams *d_mon* and *d_statistic* enabled.
- --debuglevel=LEVEL This option disables all debug streams below the provided level. Allowed options are *err*, *warn*, *info* and *dbg*. Providing *info* for example will disable *dbg* and *misc*, leaving *err*, *warn* and *info* enabled. Disabled debug streams are not compiled into the code requiring no CPU time during binary execution.

[2] http://sourceware.org/autobook/

- −−nomonitordebug This option disables the monitor debug stream.
- −−nostatisticdebug This option disables the debug stream used for statistic debug information.
- −−nocolor Normally, the debug levels receive an individual coloring for better distinguishability. This option disables the coloring, setting all debug outputs to the standard console output color scheme.

2.3 Usage

The process is very similar to the standard way of configuring, building and installing software. First the configure script `configure` is executed with the desired parameters. It creates a customized `Makefile`. This makefile is then used by *make* to compile and link the executable while *make install* can be used to both compile and install the executable into either the default or a customized location. Another makefile option is *make symlink* which installs the executable and afterwards creates a symbolic link to it.

2.3.1 Binary and Library

Using the binary and the library is pretty straightforward. After successful compilation both are placed in their appropriate directories. Running the agent only requires the user to add a matching entry into the `agentdb.xml` file. If this entry exists the teamname only has to be entered into the the `rcssserver3d.cfg` and at the start of the server the binary is automatically launched.

2.3.2 agentdb.xml

The `agentdb.xml` is the *team* or *agent* library of the *rcssserver3d*. For each team the teamname, location of the binary and possible needed startup parameters have to be provided. The following entry would match if either the user is named *vw3d* or the −−exec=vw3d option was provided to the configure script. In addition the directory `/data/agents/` exists and the *prefix* option was set to −−prefix=/data/home/agents followed by a make install, setting an appropriate symbolic link into that directory.

```
<agent_type_external name="vw3d">
    <inputfd>3</inputfd>
    <outputfd>4</outputfd>
    <timer>perfctr_instr 10000</timer>
    <working_dir>/data/agents/agent.vw3d</working_dir>
    <exec_line>./vw3d vw3d</exec_line>
</agent_type_external>
```

3 Architecture

This section describes the basic architecture of the agent framework and the idea behind it. It is divided into two subsections. In the first subsection an overview of the framework and its structure is given, followed by a brief description of the more vital parts. The second subsection focuses on the framework core, the way framework classes are handled in terms of creation, access and destruction followed by a description of the agent main loop including the used timing model.

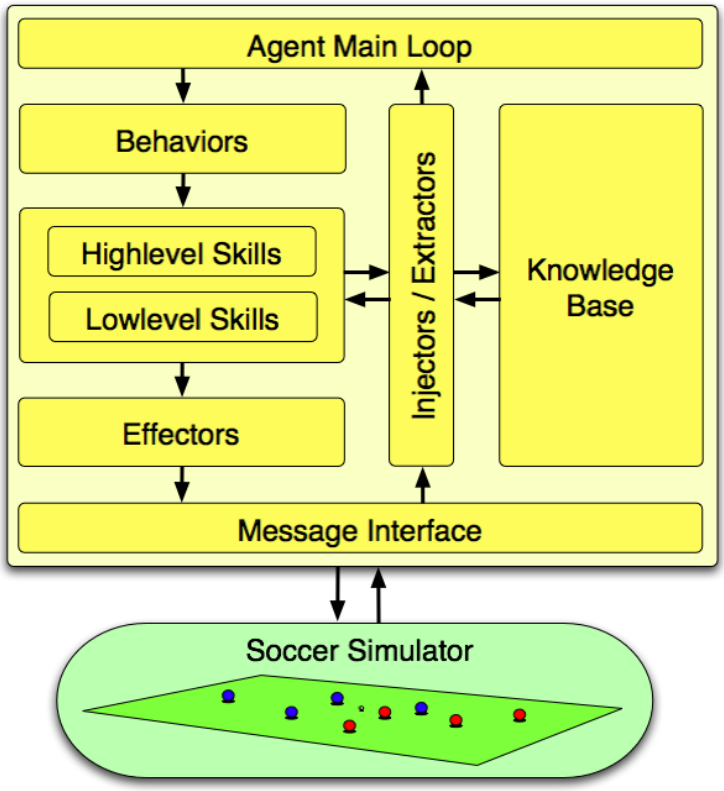


Figure 1: Overall architecture

3.1 Overall Architecture

The framework consists mainly of two parts. The first part encapsulates everything which is needed for the *percept - reason - act* cycle, namely the agent main loop, behaviors, skills, effectors and the communication interface. These parts are arranged in layers with each element in one layer only capable of utilizing elements in its own or the directly underlying level. As an example the *agent* is only able to use a behavior from the subordinated *behavior* layer. Only elements from this layer are allowed to use elements in the *skill* layer. Only skills are allowed to either use skills or elements in the *effector* layer. Finally, only effectors are able to use the *communication* layer, completing the hierarchy. Apparently, this structure does not only help to define clear responsibilities throughout the framework but it also makes the code easy to extend and maintain. The overall architecture is illustrated in Fig. 1.

The second part encapsulates all the knowledge which is needed for reasoning and performing all calculations. The knowledge base holds all information about the world, as well as all information about the agent itself. Access to the knowledge base is possible for all classes in the framework through a defined interface which supports low- as well as high-level queries for data access and prevents accidental corruption of data (cf. section 5).

3.2 Architect Paradigm

Early development discussions and experiences with the first version of our agent brought up an issue which grew more and more important as the complexity of the framework increased. The majority of classes needed pointers or references to other classes during runtime, some of them even needed information from other classes during their creation. This had a direct impact on the build order of all

framework classes which, as a direct consequence of the increasing number of objects, became more and more complex. The main development tasks were:

- Find a build order at runtime that allows all instances to be created one after another and assures all information needed to be present at that point.
- Find a way to determine an owner for each instance that is responsible for creating and destroying this particular instance.
- Assure this system to be extendible. Adding classes should not result in having to change the whole build order throughout the framework.

The solution to this problem was the so-called architect paradigm, an extendible way to handle the construction, initialization, connecting and deletion of all managed framework instances.

3.2.1 Architect

The central point of the architect-paradigm is the architect itself. The architect is a special class inside our framework. Its task is to assemble the whole framework at agent startup and disassemble it before the agent process is terminated. For better handling the processes are split up: the assembly is divided into the create-, the link- and the init-, the disassembly is divided into the finalize- and the delete-phase. To maintain clear interfaces and to allow polymorphism, all classes which are to be created/deleted in these phases have to inherit from a base class called 'constructable' which defines appropriate (pure) virtual functions which have to be implemented in order to insert this class into the framework.

- Assembly
  - *create:* In this phase all managed object instances are created by the architect by calling the respective class's basic constructor. Pointers to these instances are held afterwards in a special structure inside the architect. Upon completion, instances of all managed framework classes have been created successfully and are available for the next assembly phase.
  - *link:* In this phase every class has the possibility to link itself to other classes, e.g., get pointers to other framework classes. For this purpose the 'link' method (inherited from constructable) is called for every class, with a list with of all classes created in the 'create' phase as an argument. Though all framework objects are present, no data access or function invocation of foreign classes should take place in this phase. Upon completion, every framework class has all pointers to classes it needs during runtime.
  - *init:* In this phase all classes have the possibility to initialize themselves using either their or foreign data members. Upon completion, the framework is successfully set up and able to run.
- Disassembly
  - *finalize:* In this phase all classes have the possibility of shutting down themselves. This may include deletion of more sophisticated internal structures or saving important variables and learned values to the disk. Upon completion, all classes are ready for deletion.
  - *delete:* In this phase all framework classes are deleted by the architect by calling their destructor. Upon completion all framework classes are deleted with only the architect remaining.

3.2.2 Dynamic Loading of Framework Classes

The architect, as the instance responsible for the assembly and disassembly, has to know all classes belonging to the framework. To keep the architect code simple and prevent it from having to be altered

every time a new class is added to the framework, we decided to use dynamic class loading. To maintain this task a dynamic library is created, containing all framework classes. In addition a list of these classes is stored in a configuration file named `classes.cfg`.

During runtime the architect reads the `classes.cfg` retrieving a list of all framework classes. Afterwards the dynamic library is opened and for every entry in the classes.cfg an appropriate instance is created, linked and initialized as described in section 3.2.1.

This way of handling framework classes has the benefit that adding a new class is very simple. It only has to inherit from 'constructable', implement the link, init and finalize methods and define an extern C function returning an instance of this class (in order to allow dynamic loading). The class name has then to be put into the `classes.cfg` which will result in the architect creating, linking and initializing this class, too, when assembling the framework.

3.3 Agent

One special class inside the framework is the `agent` class. This class defines a function called *run* which encapsulates the entire agent loop - namely the *percept-reason-act* cycle. At first the main loop will be described in general, afterwards the way the timing is handled should be looked at in detail.

3.3.1 Main Loop - Basics

At the start of each cycle the server message is parsed. This means that all necessary information is put into the knowledge base and some calculations like the prediction of ball trajectories or the filtering of agent positions are made. Afterwards the appropriate behavior is selected. Selection depends on role of the agent. Currently, roles do not change during the game, but the mechanism would allow for example a defender to change into an attacker on the fly by selecting an appropriate behavior.

If a behavior is selected the behavior is executed afterwards. In the behavior an action is selected depending on the current world state, for example *intercept the ball*. The behavior then calls the appropriate skill which, after some calculations, will use one of the effectors to execute an action like *drive* or *kick*. Afterwards a *thinking done* message is sent to the server which ends this cycle.

3.3.2 Main Loop - Timing

As it is described in the server manual [ROME06] the interval between messages that contain updated world state information amounts to 20 simsteps. Normally, the agent would only be able to act as a response to these messages. But acting once every 20 simsteps would have some disadvantages in terms of reaction time and position accuracy. Fortunately, the *SPADES* system [Ril03] allows the agent to request notify messages for certain time points within these 20 simsteps, thus providing the possibility for the agent to theoretically reason/act every simstep (the length of which is currently set to 10 ms).

In our case the agent may or may not request another time notify, depending on the behavior which was executed before. If, for example, the agent is close to the ball, only acting 20 simsteps later could lead to losing the ball or an inaccurate pass. In this or similar cases the behavior sets an appropriate flag, which tells the main loop to not only finish the current *percept-reason-act* cycle, but to request another cycle for the next simstep.

4 Debugging

4.1 General

At some point in the development of our agent we needed a way to debug the agent not only by using debbugers but to actually print some information into the agent logfile, e.g., in order to see if the

intended behavior is used. Though using *cout* would be sufficient for smaller programs, the need to comment out and recompile the corresponding parts to completely get rid of the debug information makes it troublesome to use. The use of a globally defined `DEBUG` symbol enclosing all code used for debugging purpose would at least provide the possibility to compile the code without any debugging output. Unfortunately a change would require a complete recompilation of the whole code and moreover could only be switched from 'all output' to 'no output'.

We implemented the current debug system consisting of five debug streams. Each stream can be used like the known *cout* but can selectively be switched on or off. Either during runtime which only suppresses the output of the stream being written into the agent logfile or during compilation time which completely disables the stream and results in the corresponding debugging code not to require any CPU time. The five streams are used for separate levels of debug information.

- *d_err* Should be used for exceptions and errors which cause either the termination of the agent or are considered very serious.
- *d_warn* Should be used for warnings which are considered important, but will not cause the system to blow up.
- *d_info* Should be used for information messages. For example the decisions made in the behavior tree could be written to the *d_info* stream.
- *d_dbg* Should be used for debugging purpose.
- *d_misc* This stream should be used for all messages and comments which are considered least important (and least interesting).

In addition to these five streams two more are used. One stream for printing all information needed by the *vw3d logplayer* [PV07]. Another stream for printing all information needed for the statistic evaluation (cf. section 15).

4.2 Usage

The usage of the streams is similar to the well-known *cout* for example:

```
d_err << "Error: can't open file <classes.cfg> << endl;
```

To completely switch on or off debug streams so that the corresponding code will not take any CPU time the appropriate `configure` parameter `--nodebug` has to be used prior to the recompilation of the agent binary. Selective activation/deactivation of certain streams can also be realized by certain configure parameters. For more information on these parameters refer to subsection 2.2.

The debug level can also be changed during runtime to display only certain debug streams by calling `.set_debuglevel()` on one of the streams, providing DEBUG_ERR, DEBUG_WARN, DEBUG_INFO, DEBUG_DBG or DEBUG_MISC as parameter. Setting the debug level to DEBUG_WARN will result in only the streams *d_err* and *d_warn* being printed, *d_info*, *d_dbg* and *d_misc* will be ignored afterwards. The code

```
d_err.set_debuglevel(DEBUG_MISC);
d_misc << "Nothing of interest!" << endl;
d_info.set_debuglevel(DEBUG_DBG);
d_misc << "Even more nothing of interest" << endl;
d_warn << "Warning: foo.txt could not be found, using bar.txt instead"
       << endl;
```

would result in the following text being printed into the agent logfile

```
Nothing of interest!
Warning: foo.txt could not be found, using bar.txt instead
```

5 Knowledge Base

5.1 Introduction and Demands

Among the important reasons for the complete reimplementation of the Virtual Werder 3D multi-agent-system in the wake of the 2005 RoboCup world championship in Osaka, Japan, stood the realization that the development of the knowledge base used for our agents at that point had shown severe weaknesses both in its underlying design and philosophy and in the concrete implementation.

From a design point of view we decided that a new agent knowledge base needed to adhere to a set of demands that would make sure the knowledge representation was powerful enough for immediate use and would provide the possibility for future extensions.

The knowledge base currently only supplies a more or less quantitative representation of the world. However, it was designed to be extended towards a hybrid representation where a qualitative representation is available.

5.1.1 Compensation for Missing or Noisy Data

First of all the knowledge base of course needs to represent the current situation in the soccer simulation from the point of view of the acting agent. Such a representation is by nature constrained in comparison with the true baseline representation that would be available to the simulator itself. The agent's concept of the situation on the field is only an incomplete and noisy projection of the real situation.

However, this projection should approximate the real situation as good as possible even though the soccer agent for each cycle can only perceive part of the soccer field (180 degrees observability in the xy-plane) and the vision perceptor simulates real sensors by introducing random noise in the percepted sensor values.

In order to offer a decent approximation the agent's internal representation of the current situation is based upon integration of new sensory input obtained from the server by means of worldstate messages into the already available situation. In order to reduce the impact of noise in the sensor values particle filtering is used for all objects within the agent's focus of attention. Additionally, odometry information is used for a better position estimation for the agent itself. During the integration process of new worldstate information we also try to compensate for missing data due to our restricted vision by means of pre-calculation (cf. section 8).

5.1.2 Memory and Prediction

When it comes to the treatment of moving objects within the soccer simulation (the ball and the players regardless of team affiliation) we wanted the knowledge base to store not only the current *movement information*[3] for each moving object but we would like to maintain information covering a certain time interval anchored at the present whose extend can be defined in the agent's configuration (cf. section 7). What the knowledge base thus does is to store not only the current (*now*) movement for each moving object, but a series of movements which starts n steps in the past and ends m steps in the future where a step is defined by the duration of the server update cycle. Our agents thus have a configurable *short-term memory* for movements and a configurable *foresight* into the near future.

5.1.3 Agent Introspection - Logging of Actions

Another demand which needed to be satisfied was *agent introspection*. Our old knowledge base was used to store two types of information:

[3] *Movement Information* in this context means information such as absolute position of the object with respect to the the agent itself in polar coordinates, absolute coordinates on the field in Cartesian coordinates and velocity of the object. Within our framework we encapsulate these pieces of information in so-called movements.

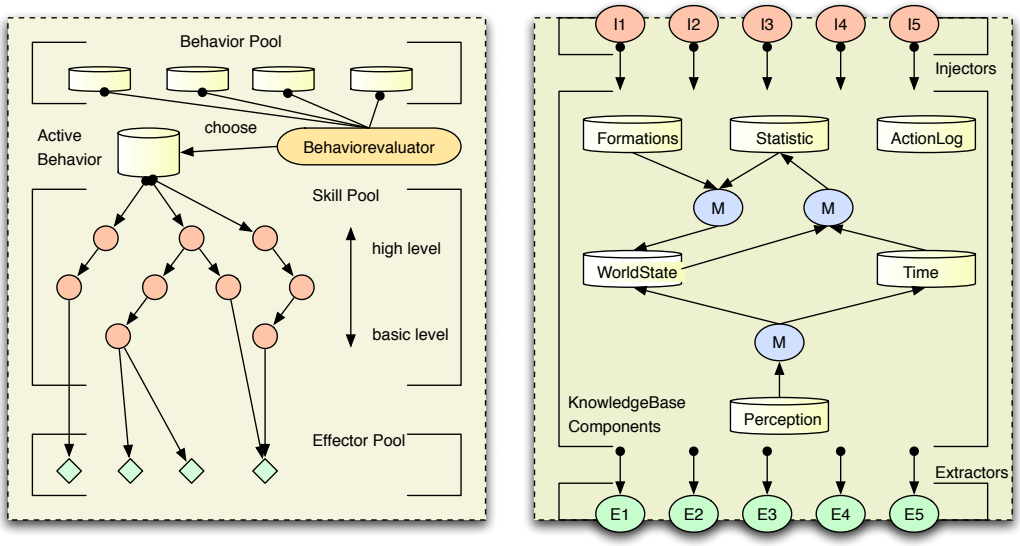


Figure 2: Schematical Overview of the Knowledge Base Structure

1. (domain-dependent) background knowledge such as information about the field layout which was either provided initially by the server or by our configuration system

2. Information about the status of the simulation, the course of the game which was provided by the server worldstate messages.

In the new agent framework, we figured out that in order to implement commitment strategies on the behavior or skill level (cf. section 11) or for debugging purposes we would need a full log of the activities of the agent over the course of the game.

Two methods for the introduction of the required information into the agent knowledge base were considered: 1) compilation of a new action perception during each *reason-act* execution that could be injected into the agent knowledge base analogous to the worldstate perception. 2.) On-the-fly injection of action related data during the *reason-act* cycle. Since the latter solution made action information available within the same *reason-act* cycle it was produced we decided to design the knowledge base accordingly. In section 5.8 the system for action logging is described in more detail.

5.2 Knowledge Base Structure - A Walk-through

After covering the major design issues from a high-level perspective, we will take a closer look at the knowledge base design. The knowledge base representation is build upon four important pillars (Figure 2):

1. Knowledge Containers which in our frame work are called Knowledge Base Modules (cf. section 5.3).

2. Means to insert data obtained through sensors or self-inspection into appropriate knowledge containers. The insertion tasks are performed by the Injectors (subsection 5.4).

3. Means to extract information stored either explicitly or implicitly in one or several available knowledge containers for further use by instances such as behaviors, skills or effectors. The extraction tasks are performed by the Extractors (subsection 5.5).

4. Means to create new information using information from several source knowledge containers as input. The mapping tasks are performed by Mappers (subsection 5.6).

5.3 Knowledge Base Modules

The core of the agent's knowledge base is not a single monolithic class but a collection of knowledge base modules which act as containers for all sorts of information the agent needs in order to reason and act during a match of simulated soccer. Currently there are six knowledge base modules. Each of these modules stores a certain kind of information such as information about time. Some modules are less differentiated then others. However, the main idea was to break up the knowledge base into conceptual pieces with clear responsibilities that can be easily recognized just by looking at the module's names:

1. *Perception_WorldState*: This perception container stores the raw data which is acquired at each sensation cycle as result parsing the worldstate message string communicated to the agent by the soccer server. Basically, the module always contains a straightened version of the most recent agent sensor readings.

2. *KnowledgeBase*: The knowledge base stores the agent's internal representation of all objects on the field, additional information about the internal state of the agent (both things like battery and temperature as well as for example current view direction), information about the layout of the world (field and goal dimensions etc.). Lately there have been additions with regard to off-side recognition and avoidance.

   The knowledge base can be understood as the agent's image of the world obtained through accumulation of single perceptions, extrapolation with respect to missing information and development of the game in the immediate future. The latter point is of special importance. The agents of the Virtual Werder team have a limited recall of four steps in the past for positions of all moving objects. They also have a preview of three steps for all these objects. The number of recall and preview values can be setup in the configuration file.

3. *SoccerTime*: This container stores all information which are related to the simulation and game time. It is the agent's internal clock.

   In order to work properly with the information which is stored in the *SoccerTime* knowledge base module it is important to to understand the agent's notion of what is to be considered the present or 'now'. The definition of the present is quite important in throughout the agent framework due to the fact that actions initiated by the agent at a moment in time will take effect within the simulation with a delay. This delay is the *action delay* which is implemented by the server. The duration of the delay was found in a series of experiments that were conducted with a dedicated test behavior. We basically found out that if the agent sends an action, the effect of this action is visible 20 simsteps later. This could mean that the action is executed 20 simsteps after the server has received them. It could also mean that visions we receive are 10 simsteps old and actions are executed 10 simsteps after the server has received them. One could construct many more possibilities, but it is not important which of these is true – the only relevant point is that one needs to predict the situation which exists 20 simsteps after the last seen situation (plus the number of simsteps that have passed since having received the last vision) in order to have the situation in which an action that the agent sends out 'now' will take effect.

   The *action delay* has strong implications for the agent and its reasoning process. Basically, the agent knows what the situation just looks like whenever he starts a *reason-act*-cycle. This is what we might call the agent's *vision reality*. However, this reality is not suitable as a basis for the reasoning process which is initiated in order to decide which action the agent should perform. If it was used anyway it would mean that the agent would choose an action that is appropriate given the *vision reality*. However, when the action is actually performed the reality at that moment might have already changed significantly. Thus, for reasoning the agent needs to consider what we might call the *action reality* which corresponds to the *Vision Reality* pre-calculated 20 ms into the future.

The agent lives in both realities at once and it is important for the developer to understand which reality should be referred to for a specific task. Within the TIME_SIMULATION enum which is located in `../knowledgebase/quantitative/include/soccertime_enums.h` the *vision reality* is associated with STIME_CURRENT and the *action reality* with STIME_CURRENT_DELAYED. Note that requests for position or velocity data are always answered with data corresponding to the *action reality* if the request was not made for a specific simtime.

4. *ActionLog*: This container stores a history of the agent's recent actions during the course of the game. In short, each action consists of a responsible behavior, the chain of skills (invoked either directly by the behavior or by other skills) with a complete parameterization and the set of invoked effectors (by the skills in the skill chain) also with complete parameterization. Thus, the agent has explicit knowledge about its past actions.

5. *GameStatistic*: This container stores information which could be used to derive online game statistics such as the play modes and residence of the ball in certain regions of the field. In order to encourage statistic evaluation the information is stored in a special form (logical predicates with temporal dilation).

6. *FormationRepository*: This container stores the possible systems of play which could be used during our matches. For each system it also stores start-up position for each agent. The information stored here is parsed from a dedicated configuration file at agent startup.

Conceptually all of the above mentioned containers are dumb. Besides the capability to store data in appropriate data structures optimized for ease of data insertion and data retrieval, there is not much functionality integrated[4].

In order to use the knowledge base modules the agent framework needs means to insert information into the knowledge base on the one hand and later retrieve information on the other hand.

Based on our experiences with the last knowledge base, we decided to design read and write interfaces to the knowledge base core which completely encapsulated the core from the rest of the agent framework and which were as modular and easy to maintain as possible. We came up with the idea of smart inserters which are called *Injectors* in our framework, and smart *Extractors* for reading access to the knowledge base.

5.4 Injectors

Definition An *injector* is a function object which is part of the knowledge base's write-access interface.

It accepts certain input data such as the perception generated by our worldstate parser. It has access to a subset of the available knowledge base modules and after processing of the raw data inserts polished or calculated information at the appropriate spots within those knowledge base modules.

5.5 Extractors

Definition An *extractor* is a function object which is part of the knowledge base's read-access interface.

It is a representation of single or a set of queries to the knowledge base. Just like the injectors introduced in the previous definition it has access to a subset of the available knowledge base modules which is relevant for performing the knowledge extraction for calculation of the query results.

For all entities outside the agent knowledge base such as skills or effectors the extractors provide the only means to access information stored in the knowledge base.

[4] Information about advanced data structures which are used in some of these modules such as in the knowledge base module can be found in [Kar05, Jos99, Str00].

5.5.1 Retrieval of Implicit and Explicit Information

Extractors are capable of two forms of information retrieval. First, they can provide information which is explicitly stored within the knowledge base. That would be a direct, simple query. Second, extractors can provide additional information which is only stored implicitly within the knowledge base. Thus, they can be used to calculate data based on information explicitly stored in the knowledge base.

5.5.2 Query Caching

Another interesting fact which is due to the extractors' incarnation as function objects, that is functions with a state, is that it is possible to use the extractors themselves as cache for repetitive queries in-between two sensations. Therefore to a certain extend extractors can be used to extend the knowledge base's storage capabilities.

An example where caching is used a lot is the extraction of movements (representations of the current motion situation of movable objects on the field). Since the same information is extracted multiple times during a single reason-act-cycle, caching is reasonable in this situation.

From a general design point of view knowledge synthesis and caching is superior to immediate pre-calculation by mappers presented in the next subsection if you have reason to believe that the information provided by the extractors probably will not be needed all the time which means that part of the time there is a real chance computation time can be saved.

5.6 Mappers

With knowledge base modules, injectors, and extractors a large part of the knowledge base has been described. Mappers can be seen as a hybrid of extractors and injectors as they have read and write access to the knowledge base.

Definition A *Mapper* is a function object just like an *extractor* or *injector*. However in contrast to these elements of the knowledge base's read & write access interfaces a mapper is an internal link between the knowledge base modules. Mappers are completely integrated into the agent's knowledge base.

The reason for their development is that there are two flavors of information. First, there is information which can be directly inserted into the knowledge base by the injectors. But second there is also information which must be derived by integrating the information stored in several knowledge base modules.

For example the knowledge about the current situation on the field which is stored in the knowledge base module and the static knowledge about possible tactic lineups stored in the formation knowledge base module and the data kept about the course of the game which is stored in the game statistic module can be used to decide which system of play the agent should use (defensive or offensive). A mapper which performs such a calculation thus integrates information from three knowledge base modules and derives new knowledge which is then inserted into an appropriate target module.

5.7 Flexibility and Extensibility

The number of injectors and extractors in the framework is in no way restricted. New additions can be introduced at any time without the need to touch the already existing injectors/extractors. This means that each functor encapsulated a clearly defined task with a manageable complexity. Maintenance is thus possible even when the number of injectors/extractors grows.

5.8 Monitoring of Agent Actions

5.8.1 The Concept of Descriptors

The knowledge base uses *Descriptors* in order to represent aspects of agent actions. There are basically three different flavors of Descriptors in the knowledge base which are all derived from a common basis. The first is the generic *Action Descriptor*[5]. It describes a complete agent action first by storing the source of the action which can be a behavior or sub-behavior (handle method within a behavior), the so-called *Skill Chain* and the *Effector Set*.

The skill chain is implemented as a list of *Skill Descriptors* which represent the second flavor of Descriptors. Skill Descriptors represent concrete invocations of certain skills in the course of an action execution. While the skill type is identified via its ID, the exact character of the skill invocation is represented by the set of parameters which were used in the call to the skill. Currently, skill descriptors are not generic and need to be implemented separately for each active Skill in the agent framework[6].

The reason why the skill chain is implemented as a list is that this type of implementation preserves the order of skill invocations in actions where multiple skills are applied. This is possible in two different ways. On the one hand the behavior itself is allowed to call more than a single skill directly (e.g., the goal keeper can approach the ball while it already tries to catch the ball). On the other hand due to their hierarchical nature skills are themselves allowed to call other subordinated skills. Using the skill chain it is possible to trace the sequence of applied skills for a single action.

The effector set is implemented as a simple vector of *Effector Descriptors* which are the third flavor of Descriptors. Analogous to Skill Descriptors Effector Descriptors represent concrete invocations of certain effectors by a skill during the execution of an action. Their internal structure resembles that of Skill Descriptors very much. The only conceptual difference is that Effector Descriptors also store the final message which is sent to the soccer server. Currently, effector descriptors are not generic and need to be implemented separately for each active effector in the agent framework[7].

5.8.2 Logging of Agent Actions

In section 5.1.3 we talked about our decision to logging agent actions on-the-fly. Now that the concept of descriptors and the three important flavors thereof have been introduced, we will describe concisely how the concrete logging mechanism works.

During the whole process three dedicated injectors come in handy: Injector_Action, Injector_Skill and Injector_Effector. The first thing every of our implemented soccer behaviors introduced in section 12 does is to create a new instance of Descriptor_Action which will be used as container for information about the anatomy of action execution within the current *reason-act*-cycle. Through instantiation the new action descriptor already knows which behavior is responsible for the action. We use this information frequently for debugging purposes in order to see whether our agent's act according to their supposed role[8]. By means of the Injector_Action the action descriptor is subsequently stored in the agent's action log within the knowledge base even though of course it is not yet complete (no skill or effector invocations have been recorded so far). The behavior engages in its top-down traversal of the behavior decision tree at whose leaves the decision for the execution for a single or a series of skills is made. The behavior calls the skills with a proper parameterization.

The first thing each skill does upon invocation is to create a Descriptor which represents this individual skill call. It uses the parameter provided by the calling behavior or skill in order to describe the skill invocation completely. After that, by means of the Injector_Skill the Descriptor is stored in the appropriate action within the agent's action log. In order to do this, no direct access to the action previously created by the behavior is required for the skill. After the injection the skill begins with its

[5] the accordant class is called *Descriptor_Action*

[6] the naming convention for Skill Descriptors is *Descriptor_Skill_<name_of_skill>*

[7] the naming convention for Effector Descriptors is *Descriptor_Effector_<name_of_effector>*

[8] If for example an agent believes to be a defender, of course the defender behavior should be used while reasoning and acting

real purpose. This of course has implications with respect to the agent introspection since currently due to convention rather than design we cannot store internals of a skill invocation in the skill descriptor. In order to do that, we would either need a mechanism to add information to the skill descriptor once it has been injected into the knowledge base or we would need to inject it right at the end of a skill invocation just before the skill returns.

Each skill has two options about what it is going to do in a certain invocation. It can delegate the work to subordinate skills or it can take action itself which in our framework means calling one or more effectors. The logging mechanism for effectors works analogous to that in skills with the exception, that the injection procedure is performed as late as possible so that besides the effector parameterization the final string command which is sent to the server can be logged as well.

5.9 Knowledge Base Support for Evaluation

The knowledge base supports offline evaluation of the agent's actions during a match of simulated soccer performed with the EvalTool presented in section 15. In order to work the EvalTool relies upon special evaluation agentstate messages stored in each agent's log file. The mechanism to create these output messages over the course of a game is based on:

1. A special mapper which controls the assembly of the evaluation messages. It is called *Mapper_EvalOutput* and differs from the other mappers within the agent knowledge base in that its purpose is not the fusion of information origination from a set of knowledge base modules in order to derive further information. Its purpose is to control the creation of a message string which can then be written into the agent's log file using a special output stream provided by the core classes. Thus the *Mapper_EvalOutput* has the character of a Reporter Class. As the agent framework matures we expect to develop the reporter concepts further in order to be able to create all sorts of reports at runtime.

2. An extension to all knowledge base modules and all objects[9] (static or moving) stored in the knowledge base. All of those classes implement a special function which upon invocation creates a string representation of the respective object which can be integrated into the evaluation message. Furthermore, all descriptors are equipped with the same functionality.

The generation of the evaluation messages is triggered by the automatic invocation of the mapper after each behavior execution (given that the system is configured to produce evaluation output). The message string itself is then constructed using a divide & conquer approach.

6 Formation

In this section, we describe how we enable our agent team to play in different formations. First, we define what the term 'formation' means to us, then we describe the different sub-aspects of a formation.

6.1 Definition

When we speak about 'formation', we essentially mean 'playing system', i.e. a formation is defined by the number of defenders, midfielders and forwards it consists of. This definition may be somewhat inaccurate when it comes to real world football (there may be more than one 4-4-2 formation, for example), but it has been sufficient for us so far. A formation also includes the kick-off positions for the players, but, as this is only relevant in kick-off situations, they are not a major feature of a formation.

[9] in this context objects refers to objects within the soccer simulation

6.2 Roles

In a given formation, each player is assigned a role. That role can currently be one of the following:

- goalkeeper
- defender
- midfielder
- forward

Naturally, each formation has one and only one goalkeeper. In contrast to that, the number of players assigned to the other roles is different in every formation, in essence, the number of defenders, midfielders and forwards is what identifies a distinct formation.

Depending on his role, a player follows a different behavior in the match, using different skills or different parameters for skills. Emphasis is put on the midfielders, as they are the connecting part between defense and offense. If the opponent attacks and forces our team deep into our own half, they may function nearly as full defenders, while they are expected to support our offense if we have the chance to go forward.

Defenders will also come somewhat forward when we attack and forwards will go back if we are under attack, but these two types of players will mainly keep their role and be ready to do their main task should the ball possession change to the other team. For more information about the roles and their behaviors see section 12.

6.3 Kick-off Positions

As mentioned in the definition, the formation includes the kick-off positions for the players. This means, whenever a kick-off comes up, players move to their assigned position and wait for the referee to start the game. The kick-off positions may be different for each formation to give the players a proper position that reflects their role. It would not harm too much however, if the players were given positions that do not fit their roles, as the repositioning mechanism would direct players to adequate positions once the ball is free. It would take some time until the players reach their proper positions, though.

6.4 Voronoi Regions

The repositioning mechanism, fully described in section 11.2.1 makes use of Voronoi cells to direct the players to a position where they could be useful for the team. However, the players are not equally distributed around the entire field, as that would mean too big distances between them, and also would most of them be too far away from the action and therefore useless.

We therefore do not use the entire field as the base area for the Voronoi algorithm, but smaller, rectangular areas that are only a part of the entire field. Where and how big such an area is depends on the player's role, the current ball position, and the playmode. Use of this mechanism will lead to the players not following a static scheme of positioning around the ball, but instead they try to be well distributed but still in range of the action.

7 Configuration Files

This chapter describes the various configuration files which are used in our framework. Actually there are three major configuration files: `values.cfg`, `formation`, and `classes.cfg`. In the following these files will be described in detail.

7.1 Configuration Values

In the first version of our agent we had all our configuration values hard-coded. This had the disadvantage that changing one single value would require the whole code to be recompiled. We soon decided to put all these values into a configuration file which would then be parsed during runtime. But we wanted our system to be as error safe as possible which means that the agent should also run without the file being available. The solution was to use the configuration file if present but use hard-coded fallback values otherwise. Because maintaining consistency between the hard-coded values and the values in the a separate configuration file would be a possible error source we decided to automatically generate both the config file as well as the hard-coded fallback values. For that reason we created a basic file called `values.cfg.definitions` in which all values are defined by type, name, value and some additional explaining comments. The most important part of this system is a script which takes these definitions, runs a syntax check on all values and afterwards creates three files:

- `vw3d.cfg` This is an automatically generated configuration file which includes all the values previously defined in the 'values.cfg.definitions'.
- `Configuration (.h/.cpp)` These two files include the hard-coded values from the `values.cfg.definitions` and an automatically generated parser which is able to read and parse the `vw3d.cfg`.

The actual parsing is done during runtime, more precisely, during the *init* phase (cf. section 3.2.1). An instance of the `Configuration` is created and all hard-coded values are initialized, afterwards the `vw3d.cfg` is parsed. All hard-coded values are overwritten with their appropriate counterpart from the `vw3d.cfg`. If the `vw3d.cfg` cannot be found, the hard-coded values are used assuring agent functionality in either case.

7.2 Formation File

The formation file is a configuration file containing all currently supported formations for the agent. We now describe its structure:

```
<formations> ::= <formations> <form>
                | /* empty */
```

The file may contain any number of formations. If it is empty or even non-existing, the team will use a standard formation.

```
<form> ::= "formation" p "value" q "system" a b c "kickoffpositions" <kickoffpos>
```

A formation begins with the keyword 'formation', followed by a positive integer value used as its ID. We continue with the keyword 'value', followed by an integer denoting the value of this formation. This value has nothing to do with quality of the formation, but is used to indicate how defensive or offensive a formation is, where more positive values indicate more offensive formations. It is used when the agent requests a more offensive or more defensive formation. Another keyword follows: 'system', followed by three positive integers. These indicate the number of defenders, midfielders and forwards, respectively. The sum of these values must be ten– otherwise a fallback formation will be used. After that, we have the last keyword 'kickoffpositions' and then a list of kick-off positions.

```
<kickoffpos> ::= <kickoffpos> unum x y
                | unum x y
```

Said list contains of entries in the form unum x y, where unum is an integer from 1 to 11, x and y are values of double type, indicating the kick-off position for unum. The list should contain one entry for

each of the eleven players. It will cause problems if players are left out as they will then all have the same standard position. Note: There is no check whether a given kick-off position is actually on the field.

We conclude this part with an example of a complete formation:

```
formation 442
value -2
system 4 4 2
kickoffpositions
1 -50 0
2 -30 22
3 -35 9
4 -35 -9
5 -30 -22
6 -15 18
7 -20 5
8 -20 -5
9 -15 -18
10 -10 0
11 -11 -10
```

7.3 classes.cfg

As described in section 3.2.1 the architect needs a list of all framework classes to be able to dynamically create them from the library. The `classes.cfg` actually supports such a list. Though the order of classes is irrelevant (due to the way the framework is constructed) the classes are ordered by namespace as well as alphabetically.

8 Localization and Ball Prediction

In this section, we explain how our agent processes the vision information in order to get reliable movement data for players and ball. This process happens every 20 simsteps, i.e., every time a vision message is received. The identified positions will later be injected into the knowledge base, where they are hold for skills and behaviors to work on.

8.1 Calculating Own Position

The base of all further calculations is the position of the agent itself, as vision data gives us all positions relative to the agent's own. If one wants to have all positions as absolute positions on the field, it is required to have one reference object with absolute position, and we use the agent itself for that purpose. We calculate the agent's position using two different approaches and then combine their results.

8.1.1 Odometry

The first one of these approaches is based on odometry, i.e., the agent uses its old position and speed and the drive commands he sent for calculating the new resulting position and speed. This of course only works if all required data is available; if not, we cannot use odometry and must use the second approach - vision - alone.

Let us assume we have the agent's correct position and velocity for the time of the last vision (i.e., the one before the newly received one). The agent also stores all drive commands he sends with a timestamp, so we can look up those drive commands that were executed by the server in the 20

simsteps between last and current vision. (Note: these drive commands are not equal to the drive commands we sent out between the two visions, as there is an intended delay in the server. The drive commands executed in the last 20 simsteps have been sent out earlier; see text about the *action delay* in section 5.3 for more details.) We then apply their effect on the known position and velocity and get new position and velocity as result.

8.1.2 Vision

Second approach is to use the vision data for calculating the agent's own position. As there are some static objects, the flags, which positions are known (as we know field width and length), we can use their relative positions included in the vision data for getting the agent's absolute position. With the implementation of the pan-tilt effector, this approach became somewhat more complicated, as all relative positions are now given relative to the current view direction of the camera, and that view direction is subject to quite heavy noise. We therefore decided to use only the distances to the flags, as they are independent of camera view directions. This introduces a new condition: we need at least two flags to be able to calculate the agent's position.

Let us assume we have at least two flags in the current vision data. We then take two flags out of them, create circles around them (with the radius being the agent's distance to the respective flag) and intersect these circles. Typically, this yields two intersection points, of which one is the agent's position. The decision which of these is the right one is made with the assumption that the agent will not leave the bounds of the field, except for kick-ins and corners, and even then he leaves the field only for some decimeters. If we have used two flags which are located on the same field border, one of the intersection points is in the field, the other usually far outside, and we then take the point inside the field as the right one. This is only problematic if the agent is close to the border, in which case we use another safeguard: We then decide to use the point to which the last known agent position is closer. This safeguard is also used if both intersection points are inside the field, which can be the case if the two flags used are not on the same side of the field.

If we have more than two flags available, we can repeat this with another pair of flags, essentially using as much combinations of flags as possible and then using the average of our selected intersection points.

8.1.3 Combining the Results

At this point, we may have two possible positions for the agent, one calculated by odometry, the other by vision. Either of them may be unavailable, as there may be some data missing for odometry calculations or we may have seen only one flag, not enough for vision calculation. In case we have only one position available, we have no choice but to use this position as the agent's position for the next steps.

But if we have both, we can combine them. If the agent's movement has been undisturbed by other objects, i.e., there were no collisions, and the agent also has not been beamed, odometry based positions have proven more accurate than vision based ones. On the other hand we can only use vision based positions to correct the odometry based ones in case one of the two disturbing factors (beaming, collision) has indeed taken effect. We therefore combine the two positions as follows. If they differ too much, we assume a disturbance has taken place and use the vision based position only. Otherwise, we combine them, giving higher weight to the odometry based one.

There is one more possibility: both position calculation approaches may have failed. In this case, we try as a last resort to see if we have only one flag seen. We can not use the vision based calculation described above under this condition, but we can use distance and angle to that flag combined with our assumed camera direction to calculate a position. But as the camera direction is suffering from noise, this turns out to be quite inaccurate. If we do not have one flag at all, we are forced to continue using an old position as the current one, which is also quite inaccurate.

8.2 Calculating Positions of Other Objects

Now that we have the agent's own position, we can calculate the camera direction, using the percepted angles to the flags and their absolute positions. Summing up our camera turn commands, we can of course calculate a close approximation of the real camera direction without using the flags, but that would not take noise into account, which is, as mentioned before, quite heavy for camera direction. We therefore prefer to use the calculated direction. Having this direction and the agent's own position, we can calculate the absolute positions of all other objects included in the vision.

These 'raw' positions, including the agent's own, then serve as input to the next step.

8.3 Position Processing

So far, we have one position for each of the percepted objects. This is not sufficient for tracking the movement of an object. For that, we require positions for different timesteps and velocities for different timesteps.

The data structure for an object's position data therefore is designed to include positions not only for the 'current' (i.e., the new percepted timestep), but also for a number of past visions and a number of future steps which we try to predict. The structure holds position and velocity for the 'current' timestep, p./v. 20 simsteps before 'current', p./v. 40 simsteps before 'current' and so on, (predicted) p./v. 20 simsteps after 'current', (predicted) p./v. 40 simsteps after 'current' and so on. The time difference between the data pairs is always 20 simsteps, as this is the frequency we receive vision data from the server. It is configurable how far into past and future the structure should reach.

For each moving object, we hold a data structure like this, and we now describe how we fill it when we receive vision data. We have two different approaches in place, one is the 'quick and dirty' method which does not much more than simply inserting the new percepted position, the other is a mechanism based on particle filtering, which takes more time to calculate but is more accurate as it is not as susceptible to noise.

8.3.1 Without Filtering

This method takes the data structure (which possibly already has values in it from the past), shifts the old values one step into the past and sets the new percepted position as the current one. For the current velocity, we take the difference between the new position and the one before. If there is no old position (which may happen if the agent sees this object for the first time at all or for the first time after a long period of not seeing the object), a velocity of zero is assumed. Prediction for future positions and velocities assume that the object keeps its current velocity. This would be highly inaccurate for the ball, but the ball is one of the objects which is always calculated by the particle filtering method.

If an object using this method is not seen in a vision, we can simply do the described shifting of old values one step into the past, using the former 'predicted position in 20 simsteps' as new 'current position'. This should however not be done too often as the position information rapidly loses quality. There exists a configurable number of steps, making it possible to deem an object without known position if not seen for this number of steps.

8.3.2 Particle Filtering

In this approach each object is represented by a 'particle cloud', consisting of a number of particles. Each of these particles holds a data structure as described above, with positions and velocities for different timesteps. It also possesses a weight. If we then need the position of an object, we use the average of all particles belonging to that object, taking into account the individual weights of the particles. In each update cycle, the weights are changed so that particles containing 'good quality' data get a high weight, those with 'low quality' data get low weight or are even reset to initial state.

Let us assume we see an object for the first time and want to use particle filtering for it. We then initialize all its particles with the seen position (and some random offset added to it, each particle gets a different offset so that average of all offsets should be close to zero) and give them an initial weight, which is equal for all particles. The initial velocity vectors are determined randomly, with mean velocity being zero. We then have a number of particles, their 'current' positions being almost equal, but with very different velocity vectors. As the predicted positions and velocities are created using the current velocity (same as above in the other approach), the future positions will widely differ.

When we see the object again in the next step, we check all particles whether the percepted position matches (or comes close) to the previously predicted one. If this is true, we can assume that this particle has a velocity close to the real one, so the weight of this particle is increased. If it is of medium quality, we set weight to a medium value, and if it is not fitting at all, we reset this particle, meaning we delete all its data and initialize it again with the new position. If a particle survives, we also draw the position and velocity of the particle somewhat closer to the percepted values, as this increases the chance that the particle will survive the next cycle as well. (If we did not do this, we would need a much higher number of particles in order to have some matches, which would in turn need a high computation time.)

If an object changes direction, the particles that were good before (as they had velocity pointing to the correct direction) will decrease in quality and lose weight, while others, being of low weight before, now become more important. It is therefore a requirement for the particle cloud to always have particles for all possible directions the object could move, with the particles having the correct direction having so high weights that the other particles do not influence the average of all particles too much. This consideration is a factor of importance when selecting the number of particles for a particle cloud. On the other hand, one wants to reduce the number of particles as far as possible as this yields calculation speed.

The particle update function is tolerant to some level of noise, so that a predicted position not exactly matching the percepted one will not lead to reinitialization of the particle too early. By drawing particle positions closer to percepted ones (but not setting them on the percepted positions at once), we dampen the effect of the noise.

Some notes: The ball particle cloud uses two different particle types at the same time, *BallHeightParticle* and *BallPosParticle*. This is done because the movement on the xy-plane and the movement on the z-axis can be seen as independent movements. This way, we can reduce the total number of particles needed. For the agent itself, a different type of particle, *MyselfParticle* is used. It differs from the normal *PlayerParticle* by having a different function for future position prediction. As the agent knows its own drive commands, he can employ odometry here again, which works the same as described at the beginning of this section.

If an object is not seen, we can do the same as we do in the non-filtering approach, i.e., predicted positions become current positions, and if we do not see the object for too long, it is declared as having unknown position. If the object is seen again after a period of not seeing it, all particles for this object are reinitialized. For further information on particle filtering refer to [Rek02, FTBD01].

9 Communication

When we talk about communication, we have to distinguish between two different kinds of communication. One is the communication between player and server, the other is between different players.

9.1 Server Communication

To communicate with the server, the agent must be capable of following the communication protocol defined by the server and its component SPADES [Ril03]. Messages coming from the server are length-prefixed, followed by a letter indicating the message type. Depending on what type of message it is, further data follows. The actual receiving of the messages is done in the Messenger class, detecting the message type is done in the Parser_MessageType class. The latter class is also responsible for injecting

all data contained in the message into the knowledge base, for which it uses some sub-injectors. Further actions, depending on the message type, is then initiated in the *Agent* class.

Messages in the other direction have one purpose: an agent wants to perform an action. The message declaring that the agent wants to do a specific action is constructed in the effector class responsible for that action. It is then again the job of the Messenger class to perform the actual sending. The message must be length-prefixed, which is done by the Messenger.

9.2 Agent Communication

Agents can communicate with each other using a "say" command provided by the agent-server interface. All communications between agents must use this command; it is against the rules to use other means of agent communication.

The messages will be heard by other agents in the vicinity of the talking agent up to a threshold distance beyond which it is not heard. There is also a limit on how long a message can be, and how often an agent can hear something. A team must therefore make sure that it is not jamming its own ears by talking too much, because when there is more than one message that could possibly be heard by an agent, the server decides which of these the agent gets. We solve this by letting only the ball-controlling agent talk and limiting its frequency of talking to that which can safely be understood by the others. Other solutions, such as a rotating "right to speak" have been thought of, but not implemented at the moment.

A team does not only hear its own talking, but also that of the opponent - it is however not possible to jam the opponent's hearing, as a player hears his own team and the opponent at the same time. The important thing about this is that the server does not provide the information which player (by unique ID) did send a certain message, it merely gives the direction where the message came from. It is therefore required to have a mechanism which lets a player identify his team mate having sent the current hear message and at the same time make sure that the received communication does not come from an opponent imitating messages he may have heard from our team mates in order to disturb our behavior. We currently use message signatures for this purpose. Each message one of our players sends contains a unique signature that lets all his team mates recognize the source. Moreover, the signatures are made for one time use, so that an opponent can not simply echo a message. Each player has a pool of signatures, which can each be used once. Each player also knows all signatures of his team mates and will accept each heard signature only once. Messages containing wrong, already used or no signatures at all are discarded and their content is not used.

For creating a complete set of signatures, we use the gen_signatures tool contained in the bin directory of the agent source tree. The resulting file must be placed along with other configuration files.

10 Effectors

This section describes the effectors of the agent. Effectors are the low level interface between the agent and the soccer simulation. They take the arguments given to them by the skills, transform them into a message string and send this string to the server, using the *messenger* class.

10.1 Beam

The *beam* effector is used to instantly teleport an agent to a designated position. The coordinates have to be provided to this effector as a *Vector3d* parameter. This effector can only be used in the playmode 'before kick off'. Using this effector in any other playmode will have no effect. With the input *Vector3d(x,y,z)* the message to the server is *"A(beam x y z)"*.

10.2 Catch

The *catch* effector is used to catch the ball and is called with no parameters. It is usable in all playmodes but limited to the agent with the unum 1 (the goalkeeper). Using this effector will only have an effect when the distance between the goalkeeper and the ball is below a certain catch margin which is currently set to 2.0 meters. If the distance is below this threshold and the effector is called, the ball's speed is set to zero, it is teleported to the goalkeeper and all agents which are closer to the goalkeeper than 2.0 meters are moved 5.0 meters away (current server settings). The message to the server is *"A(catch)"*.

10.3 Drive

The *drive* effector is used to move the agent and is called with a *Vector3d* as a parameter. It is usable in all playmodes. The x and y parameters define the direction in which the agent is accelerated. The z component is ignored because agents can only move in the x/y-plane. The amount of the vector, including the z component, defines the percentage of the desired speed. Therefore the length of the vector should lie between 0 and 100. The vector $(100, 0, 0)$ would cause in the agent to accelerate into positive x direction until it reaches maximum speed. The vector $(0, -50, 0)$ would cause the agent to accelerate into negative y direction until it reaches half its maximum speed. The vector $(0, 0, 100)$ would cause the agent to stop as there is no way to accelerate 'up'. With the input *Vector3d(x,y,z)* the message to the server is *"A(drive x y z)"*.

10.4 Kick

The *kick* effector is used to kick the ball by applying a force to it. This effector is called with two parameters, defining the kick power, ranging from 0 to 100 and the (horizontal) kick angle ranging from 0 to 50 degrees and is usable in all playmodes but only will cause an effect if the distance between the agent and the ball is below a certain kick margin, currently set to 0.04 meters. If the distance is below this threshold the ball will be accelerated with the provided power and into the desired angle. The direction in which the ball is accelerated on the x/y plane depends on the position of the agent relative to the ball for the force will always be applied in a direction from the agent's center to the ball's center. This means that correct agent positioning is vital for accurate shooting. With the input *double(x),double(y)* the message to the server is *"A(kick x y)"*.

10.5 Pan-Tilt

The *pan-tilt* effector is used to move the camera which is hinged to the agent's center and is called with two parameters defining the angle-delta for both the pan (vertical) as well as the tilt (horizontal) parameter. It is usable in all playmodes. Using this effector will cause the camera to be turned the appropriate 'pan'- and 'tilt'-value. The 'pan' angle has a scope of only 10 degrees while the 'tilt' angle has a scope of 360 degrees with a maximum turning angle of 90 degrees per simstep. With the input *double(x),double(y)* the message to the server is *"A(pantilt x y)"*.

10.6 Say

The *say* effector is used to send a message to other agents. It is called with a single parameter, namely the message which is to be transmitted. This effector is usable in all playmodes. The message length is limited by the server to 512 characters. If the sent message exceeds this size it is truncated. In addition to the length limitation there are also some constraints regarding the allowed characters. Only the printable characters without space and '(' and ')' are allowed. With the input *string(x)* the message to the server is *"A(say x)"*. For more information about communication see section 9.

10.7 Spades

The *spades* effector is a special kind of effector which is used for all low level, SPADES specific communication. For this reason it capsules a variety of functions both with as well as without parameters. For more details on how the communication with the server works, especially regarding the SPADES system, please refer to the spades manual [Ril03].

- `send_create_message()` This function is called after the agent is created and sends back the first life sign to the server. The message to the server is *A(create)*.
- `send_init_message()` This function is called after the the create message has been sent. It tells the server the name of the agent and its uniform number. With the input *string(x),uint32_t(y)* the message to the server is *"A(init (unum x) (teamname y))"*.
- `send_initialisation_done_message()` This function is called after the agent is fully initialized and ready. It is send as a response to a initialisation message. The message to the server is *"I"*.
- `send_done_thinking_message()` This function sends a 'done thinking' message which tells the server that the agent is done for that thinking cycle. If this message is not sent, the agent will be terminated by the server with a 'too long think time' error. The message to the server is *"D"*.
- `send_request_current_think_time()` This function requests the server to send a message back, telling the agent how many simsteps he has already used in the actual thinking cycle. The message to the server is *"C"*.
- `send_request_time_message()` This function is used to request a time notify message from the server at a certain time. The parameter defines the delta from the current simtime to the requested time. The time notify message will start a new thinking cycle. This function is used to act more often than every 20 simsteps as a reaction to the sense messages. For more detail refer to section 3.3.1 and 3.3.2. With the input *uint32_t(x)* the message to the server is *"Rx"*.
- `send_migration_message()` This function is the appropriate answer to a migration request which can be send by the server. It tells the agent to shut down and to put all its internal data into a string which then should be used to initialize a new agent process on another system. This would cause something like a 'migration' of the current agent. Currently this process is neither supported by the server nor by our agent, For more detail please refer to the *SPADES* manual and the rcsssever3d manual [Ril03, ROME06]. With the input *string(x)* the message to the server is *"Mx"*.
- `send_exit_message()` This function is used to tell the server that the agent is going to shut itself down. The message to the server is *"X"*.

11 Skills

This section describes all the agent's skills. It is divided into two subsections. The first subsection deals with the basic, so called, low-level skills which lie only one level above the effectors. The second subsection deals with the so called high-level skills which utilize low level skills to perform higher level tasks.

11.1 Basic Skills

The basic skills which are going to be described here are low level skills, or (the other way round) high level effectors. Most of them only utilize one effector by just passing the arguments directly to it.

11.1.1 Beam

The skill is used to teleport the agent to the designated position and is called with a *Vector3d* as parameter which defines the point the agent should be teleported to. Although this skill can be used in any playmode, it will only show an effect when used in the *before kick off* playmode. The skill directly passes the provided *Vector3d* to the *beam* effector.

11.1.2 Move

This skill is used to move the agent to a designated position by driving toward this point. It is called with two arguments - a *Vector3d* defining the absolute position the agent should move to and an enum value defining the speed the agent should move with. Currently three velocities are distinguished MOVE_SLOW, MOVE_MEDIUM and MOVE_FAST which equal to 33, 66 and 100 percent. Given the absolute destination and the agent position the relative destination is calculated. This vector is normalized, scaled with the desired speed percentage and passed to the 'drive' effector.

11.1.3 Kick

This skill is used to kick the ball to a designated position. There are two ways to use the skill. The first way is to provide two enum values, the desired ball speed and the desired ball height. Currently three velocities are distinguished RECEIVABLE, DRIBBLING_SPEED and FULL_POWER as well as four heights LOW, MED, HIGH and SCORE with the last one being a special height only used for the *score* skill. If the skill is invoked with only these parameters the skill will use the *kick* effector to kick the ball when possible. If the distance to the ball is too great, the *drive* effector is used to approach the ball.

The second way is to provide the ball speed and ball height in combination with a designated target and an enum value, defining the tolerated position error for the ball. Currently five error granularities are distinguished KICK_ERROR_HIGH, KICK_ERROR_LOW, KICK_ERROR_DRIBBLE, KICK_ERROR_GOALKICK and KICK_ERROR_DEFAULT which can be adapted in the configuration file (see section 7). If the skill is invoked with these four parameters it checks whether the agent is in a good shooting position. If the agent is not properly positioned it repositions itself to correct the alignment towards the ball. Otherwise the *kick* effector is used.

11.1.4 Catch

This skill is used to catch the ball and is called without any argument. It only invokes the *catch* effector without further checking neither whether the agent calling this skill is the goalkeeper nor whether the distance to the ball is below the catch margin.

11.1.5 Say

This skill is used to communicate a message to other agents of our team (which can also be perceived by opponent players). There are two ways of using the 'say' skill. The first way is to provide the message to be sent as a string parameter. All limitations regarding messages which apply to the *say* effector also apply to the 'say' skill (length / allowed characters). If there are no 'pending' messages - messages that are still in the queue for later transmission - the message string is prefixed with a unique signature and sent directly to the *say* effector. Otherwise it is stored for later sending. The second way is to provide no argument. This will cause the skill to send any messages which were stored earlier to the *say* effector and to flush its 'pending message queue'.

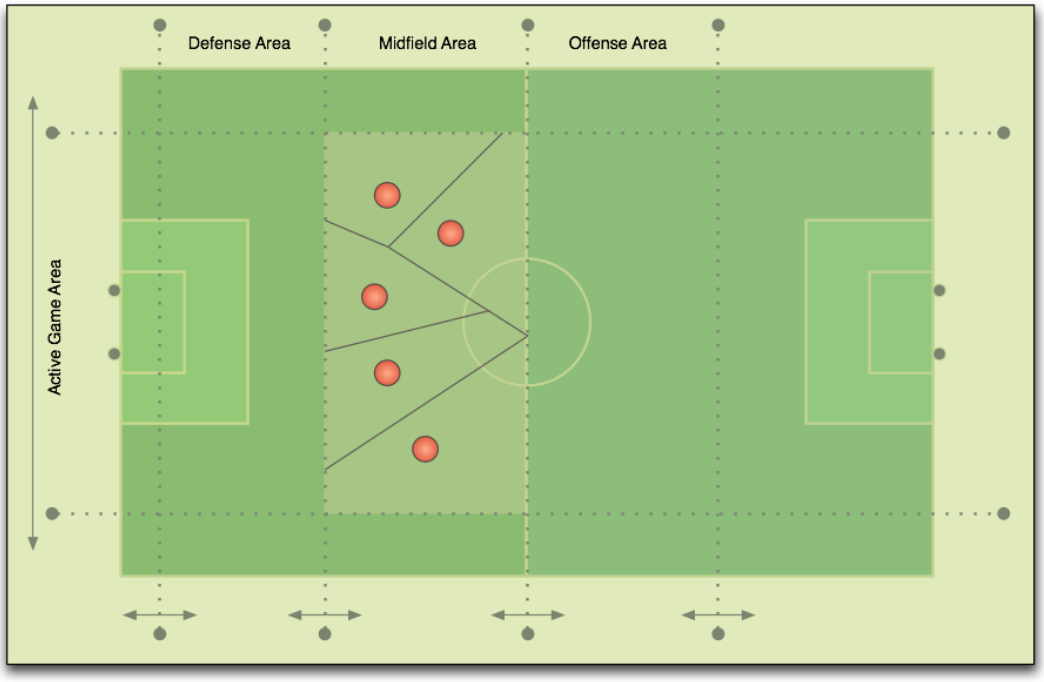


Figure 3: Voronoi regions for midfielders (schematic)

11.2 High-level Skills

High level skills are used to accomplish more complicated tasks which require more sophisticated calculations and/or the use of some low level skills.

11.2.1 Reposition

The *reposition* skill is responsible for calculating a 'good' position for an agent that is not responsible for going to the ball and then moving the agent to that position. Positions are considered good if they are not too far away from the ball, not too close to a team mate and not too close to opponents, so that an agent at a good position can receive the ball and has some time to decide on the next move until an opponent can pose a threat to him.

Distribution of our agents is done by using Voronoi diagrams, assigning each agent one of the polygons in the diagram and moving it to the centroid of the polygon. This way, we can keep our agents from moving too close together. We were inspired to use this technique by [ATDAA+05] and then added some ideas of our own.

We do not use the entire field as the base area, as that would move too many of our agents too far away from the action. Instead, we use a smaller rectangle as base, the dimensions of which are dependant on the current playmode, the current ball position and the role of the agent. For example, the base area for the attackers is always in front of the ball if the ball is not yet close to the opponents' goal, or the base area for all players if equal to our own half if we have a kick-off situation. These situation-dependant base areas for the Voronoi polygon calculation enable us to distribute the players in an area somewhat close to the ball and at the same time maintaining a position that is ideal for their role. We use this mechanism also to keep our field players out of the penalty box if there is a goal-kick to be performed. The Voronoi regions are illustrated in Fig. 3.

Note: the calculation of the polygons currently only takes our own players into account. We did experiment with also using the opponents in order to keep our players away from them, but this

resulted in our players moving to positions that the spectator did not consider to be good. In essence, the opposing team too often forced our players into bad positions with their own positioning and we therefore disabled the option of taking opponents into account.

The *reposition* skill also takes the offside rule into account - if a player was to move to a position making him offside, it will simply go as far as possible without overstepping the current offside line.

11.2.2 Score

The scoring skill is used if the agent has ball possession in a position that will allow a shot at the goal (both distance- and angle-wise). It then searches for any objects that could obstruct the path of the ball from its actual position to the goal and determines which parts of the goal line are reachable and which parts are blocked. Let us assume there are some free parts. The skill then selects the longest free part and executes the shot to the center of this part. If the free part ends at one of the posts, the shot is targeted close to the post (as this is the target as far away as possible from the obstructing object). For the height of the shot, the distance from ball to goal is taken into account: the shot is performed as high as possible. If the ball is too far away from the goal, the shot must be of low height as the distance covered by the shot is higher that way. On the other hand, if the ball is very close to the goal, the agent must be careful not to shoot over the goal.

The scoring skill can remember its last target and be instructed to use that target again or to select a new target. This way, we can keep the agent from selecting a new target every step, which would lead to the agent taking a very long time to shoot, as it will probably have to reposition itself over and over again to aim at the possibly changing target.

11.2.3 Pass

The *pass* skill is used to shoot the ball to a point on the field that is deemed good. There are some criteria that make a point good, among these are presence of team mates, absence of opponents and distance to the opponents' goal. For deciding on the actual target, a field evaluation function is employed, which works as follows: The field and the area around the field is partitioned in small squares of 2m * 2m. Each of these cells is assigned a basic value which depends on its position on the field: the closer the opponents' goal is, the higher the value. Cells that are out of bounds get a very low value, as well as cells close to our own goal. The function then iterates all players and changes the values of the cells around the player: the cells around a team mate are increased in value, those around an opponent are decreased. The amount the cell value is changed depends on the relative position of the players close to the position.

The skill can, like the score skill, remember its target to avoid jumping between different targets. When instructed to select a new target, it uses the aforementioned field evaluation to select the best possible target in reach. Points that have an opponent closer than an own player will not be selected, and passes to the back have a very low value so that they are used only as last resort.

As the execution of a pass takes time, the selected target is rechecked in every step, but to avoid jumping targets, a new target is only selected if the old one has become drastically worse (defined by: no own player present in its vicinity, but an opponent). Also, this new target will typically not be chosen with a complete new selection (and possibly requiring the agent to rotate a long way around the ball), but by the get_immediate_target function instead. This function does also iterate all possible targets, but does so with the targets ordered by the distance the agent has to rotate around the ball in order to aim at that target. It does not select the best of all targets, but stops if a target is 'good enough', thereby preferring targets which do not require the agent to take a long way around the ball.

As a final resort, if a pass must be made in the instance the skill is called, it also has a panic function which only chooses among targets that do not need any agent rotation around the ball, so that the shot can be executed as fast as possible.

11.2.4 Cover

The *cover* skill is used in the game without ball. As the name indicates it is used to cover a player of the opponent team thus making it harder if not impossible for the opponent to participate in coordinated attacks of the opponent team. Covering is invoked with a single flag which indicates whether or not the *cover* action should be manually reset or not.

The first thing the *cover* skill does is to perform a check whether or not the current choice of cover target needs a revision. This situation can happen either upon manual request indicated by the reset flag. It also happens when no cover target could be found in the last call to the *cover* skill. The last situation where the cover target needs to be verified is due to the agent's limited commitment. Due to the dynamic changes in the situation on the soccer field while the opponent is controlling the ball and actively tries to push the ball forward in order to score a goal blind commitment is not feasible for covering agents. Thus, after a configurable amount of simsteps the commitment is revised.

If the agent finds that a new cover target needs to be chosen an evaluation of the current situation on the field is performed which as a result returns the new cover target (if such a target can be found) or a dummy (no one to be covered).

First a map of potentially threatening agents is compiled which is ordered in decreasing threat magnitude[10]. Only opponent agents which are closer to the agent's own goal than a certain configurable distance (cover_consideration_distance) and whose x-coordinate is smaller than that of the ball plus a certain configurable distance (cover_ignore_distance) are considered. Agents which are located very close to the field borders or even beyond those borders are not considered as well.

Next a list of potentially covering agents is compiled as well. With these two lists it is possible to calculate with reasonable certainty which of the possible covering agents is supposedly responsible for which attacking opponent. If an agent is closest to an opponent player it is considered responsible. When the ID of a responsible player first matches that of the player just performing the *cover* skill a new cover target has been identified.

It is possible that during iteration of the opponent map no opponent can be found which should be immediately covered due to direct responsibility assignment. In earlier versions of the *cover* skill the evaluation would have been aborted without success.

However during the preparation for the RoboCup 2006 we found situations where using only the system described so far let to severe problems. In those dangerous situations an opponent agent managed to dribble the ball with considerable speed with one of our agents in close pursuit. The problem with such a situation is that as long as the attacker does not slow down for a *pass* action there is often no chance for our agent to overtake the attacker in order to stop him.

Thus, the attacker needs to be stopped by a supporting cover agent which closes in on the attacker from the direction of the goal. This concept is really promising and given a good positioning strategy can stop solo runs by single attackers most of the time. In the implementation a check similar to that for direct cover responsibility is performed where the agent checks whether or not it is responsible supporting cover player.

In the *cover* action itself the agent checks whether an acceptable cover target is currently used. If that is not the case the skill throws an Exception since it is unable to perform right now. It is thus not prepared to handle a situation where covering is requested but technically impossible on its own but leaves this task to the calling behavior.

If however an acceptable cover target can be found the agent recalculates the optimal cover position based on the ID of the object to be covered (which due to commitment is somewhat stable) and the current position of the associated agent (which is of course changing rapidly). The general idea which is applied for the calculation of the cover point is simple. It lies a certain again configurable distance away from the current position of the agent to cover in the direction of the goal center. Thus, the covering agent blocks the way of the covered agent to the goal.

[10] The threat an agent poses is associated with its distance to the goal (with respect to its teammates)

11.2.5 Intercept

The *intercept* skill is used by the agent to intercept a moving ball for the purpose of being able to kick it afterwards. It is called without any parameters. The calculation of the interception point is dependant on the current ball velocity and the position of the agent relative to the ball. The whole calculation is based on the predicted ball positions which are calculated every 20 simsteps by the appropriate mapper. In an interactive method the point the ball will be at after a number of simsteps is compared with the points the agent could reach in that amount of time. If both – the ball and the agent – can reach the same absolute position on the field at the same simtime, the iteration stops and the 'move' skill is used to move the agent to this point. If no such point exists - this can happen if either the ball is too far away or too fast - the *move* skill is used to move the agent to the point the ball is expected to finally come to a rest.

An additional ability of the *intercept* skill is the ability to evade the ball in certain situations. If the ball rolls into the direction of the own goal, intercepting the ball and colliding with it makes sense to prevent it from moving further into the own half. But if the ball moves towards the opponent goal, colliding with it might not be the desired action for it takes a lot of unnecessary time to stop the ball, walk around it and kick it. Instead evading the ball and moving directly behind it enables the agent to much faster pass it to a teammate or score a goal.

11.2.6 Pan-Tilt Handling Skills

The Virtual Werder agent currently has three skills for handling camera turns. These are:

1. View Turn
2. View Rotate
3. View Follow

Of these, *view turn* is the most basic skill. It takes the pan and tilt values as parameters and executes the turn. If the requested turn cannot be executed in one step, the camera will be turned as far as possible and the remaining turn distance will be stored. The skill can then later be executed without parameters which triggers completion of any remaining turns.

View rotate is the skill used for giving the player the best possible knowledge about the positions of all objects on the field. It simply rotates the camera in steps of 180 degrees, so that (without taking movements into account) each part of the field is seen in every second vision.

While *view rotate* is good enough for seeing all objects regularly, it may not be good enough if the position of one special object is to be tracked as accurately as possible. For that case, *view follow* exists. It takes one object as parameter and then tries to turn the camera in a way that this special object is seen in every vision while still moving the camera around as far as possible so that most other objects are seen, too. In essence, it performs turns so that the tracked object is seen on the right border of the field of vision in one step, on the left border in the next step and so on. This way we hope to see the tracked object in every vision and the other objects in every second vision. This way of camera turning has one disadvantage: there is always a small area 'behind' the agent that is never seen, and so are objects in that area. Yet we have not experienced that to be a bigger problem, as the movements of the objects sooner or later will cause the agent to see any possible 'lost' objects again.

11.2.7 Goalkick

The *goalkick* is a special skill which is only used by the goal keeper in a situation where an undisturbed kickoff of the ball needs to be performed. The skill was designed to completely fill the role of the handle method in the keeper behavior which is active while PLAYMODE_GOALKICK_OWN.

Whenever a goalkeeper receives the opportunity to perform the *goalkick* the server beams the ball automatically to a position approximately five meters in front of the goal center. For as long as the ball does not leave the penalty area no opponent player is allowed to enter the penalty area in order to interrupt the goalkick process.

The goalkick process is divided in two phases. In the first phase the keeper approaches the ball which is still resting at its initial beam position in order to perform a supporting pass which is used to move the ball to a better goalkick position closer to the penalty line while still behind the penalty line. If no decision has yet been made as to which of the three possible more advantageous kickoff positions the ball should be shot first, this selection is made once for the immediately following *goalkick* action.

When the ball has been shot towards its better position phase two of the goalkick is started. Again it starts with an approach towards the new ball position. Once the keeper is close enough the *pass* skill takes over and the keeper selects a promising target for the goalkick pass action. Since the goalkick is a special situation and there are often quite a lot of opponent agents still waiting just outside the penalty area the pass is to be performed using a high kick in order to minimize dangerous situations that arise due to deflections of the ball by opponent attackers or midfielders which manage to move into the shooting line.

11.2.8 Kickin

The *kickin* skill is a skill which is currently used by midfielders and defenders in order to perform a normal kickin during a game. Due to our current tactical setup forwards never perform kickins themselves since they are supposed to find a good position on the field in order to receive a the ball and drive the game forward. The *kickin* skill bears some similarities to the *goalkick* skill described above. The skill was also designed to fill the role of the handle method in the defender or midfielder behavior which is active while PLAYMODE_KICKIN_OWN. However, it cannot be a complete substitution since it remains the behavior's task to decide whether or not an agent is responsible for the execution of the *kickin* due to its position on the field relative to the kickin point where the ball has been beamed to by the soccer server. If an agent is responsible the skill takes over.

When a player has the opportunity to perform the kickin as mentioned above the server beams the ball to the kickin point. The kickin is divided in two phases just like the *goalkick*: an approach phase and a pass phase.

When a new kickin is initialized the agent first chooses an intermediate kickin point out of two possible points which are located about half a meter to the left and to the right of the resting ball along the field border and 25 cm outside the field. The agent chooses the closer intermediate point for the fast approach phase. The reason why an intermediate point was introduced in the first place is that in our experiments we noticed a better ball handling with the indirect approach than with the direct approach. That is not meant to say, that a direct approach will not work. Actually it does pretty well most of the time due to the orbiting feature built right into the agent's regular approach of kick positions. However, there are special situations where it does not (i.e., the player touches the ball on the way to the kickin position) and we did not want to take chances.

The agent approaches the intermediate kickin point with maximum speed. Since the agent does not break it drives through the intermediate point and recognizes it has passed a certain threshold distance between the border of the field and its own position (measured using only the length along the y-axis) and thus switches to phase two where immediately the pass skill takes over. It handles the last approach towards the ball, chooses a suitable pass target and kicks the ball.

In the current implementation of our agent framework we also use the *kickin* skill to perform corner kicks. However, it seems like the method does not work too well under these circumstances. We are investigating the problem and hope to fix it soon.

12 Behaviors

This section describes the agent's behaviors. First an introduction to the general design of the available concrete soccer behaviors is given. After that each behavior is treated in a separate subsection.

12.1 The General Design of Behaviors

In the Virtual Werder framework we distinguish three different kinds of *agent behaviors*. First of all we use a set of game behaviors for normal game-play in tournaments and test games. The second group of behaviors are *test behaviors* which are used to debug and evaluate the performance of the skills which are implemented in the framework. Typically, those test behaviors are meant to be used in certain test scenarios rather than in normal games. Actually most of the test behaviors currently implemented such as the Behavior_Test_Kick are written for a scenario with only a single agent acting on the field. Finally, the third kind are *learning behaviors* as well. In order to learn more about the whole learning complex, please refer to section 14.

Whenever the `behave()` method is called by the agent control loop the appropriate behavior for the current reason-act cycle is determined. This is performed by the *Behavior Evaluator* which controls the behavior pool where all behavior instances are located. The actual process that is used in order to select the concrete behavior to be executed by the agent in each `behave()` invocation involves two selection phases. In the first phase the desired modus of play which is indicated in the configuration (gaming, testing, learning) is evaluated for the current cycle and thus the *behavior domain* can be identified. In the second phase further differentiation takes place which means that out of the set of possible behaviors associated with the active *behavior domain* the concrete behavior the be executed is chosen. The character of the selection procedure is different for each of the three possible *behavior domains*.

The easiest case is learning since only a single learning behavior exists at the moment. Therefore of course, selection is trivial. For test behaviors the agent configuration specifies besides the fact that the agent should use a test behavior the skill which is to be tested. The Behavior Evaluator then simply chooses the matching test behavior. For game behaviors the Behavior Evaluator chooses the concrete behavior according to the role the agent currently occupies. There are four possible roles: ROLE_GOALIE, ROLE_DEFENDER, ROLE_MIDFIELDER and ROLE_FORWARD. For each of these roles an especially adjusted concrete behavior is provided. In the unlikely case that an agent for some reason occupies an invalid role instead of one of the behaviors above a simple fall-back behavior is used which still allows the agent to play decently most of the time. At this point it is worthwhile to mention that the Behavior Evaluator is not responsible for an adaption of agent roles. This task belongs to the mapper formation.

12.2 Game Behaviors

The behaviors which are used in normal simulated soccer games adhere to a certain internal design which is enforced by the inheritance hierarchy for *game behaviors*. The first game behavior to be implemented historically and probably the most important in terms of design considerations is the *Behavior_Soccer*. This behavior has two functions. First of all it implements a complete simplified soccer behavior for field players. It is *complete* in the sense that it is able to handle all play-mode situation that can possibly occur in a game of simulated soccer. It has thus a 100% coverage over the course of a game. The *Behavior_Soccer* is also parent class for all specialized game behaviors.

The design of the game behaviors was influenced by the fact that at each moment the agent can tell with certainty which play-mode is currently active. Therefore the *Behavior_Soccer* introduces an design where a control method first determines the current play-mode and then delegates the task to act according to that play-mode to a specialized handler method. Since the *Behavior_Soccer* can actually be used in games each of these handler methods is fully implemented which brings a lot of flexibility to the development of the specialized classes.

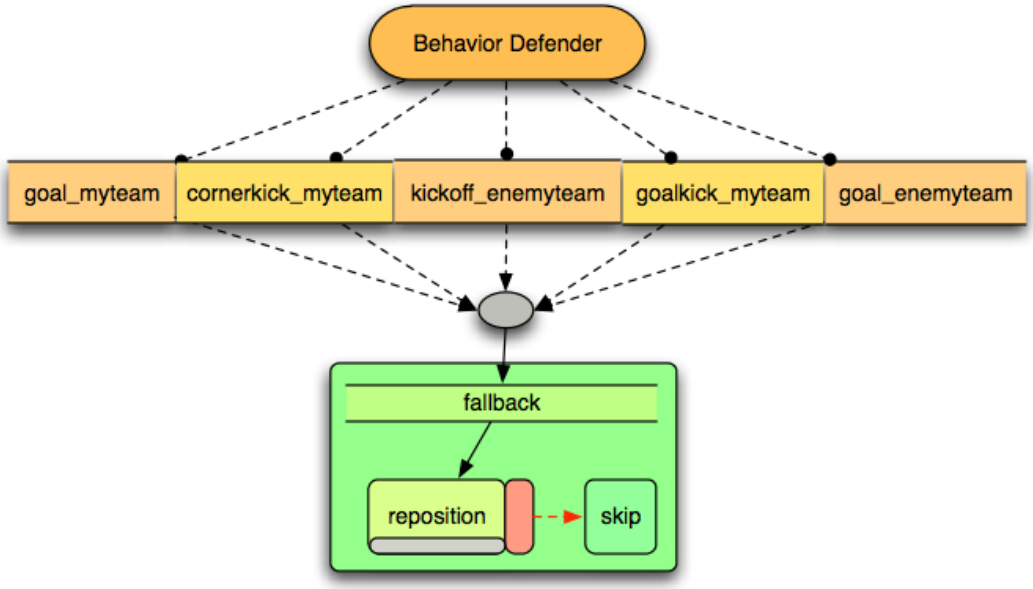
Since the play-mode control method which manages the task assignments to the handle methods is generic it is used by all derived behaviors as well. The handler methods may or may not be reused as well. In fact due to inheritance it is possible to specify only parts of derived behaviors by reimplementation of only a subset of the handle methods. In our experience gathered during the implementation of the Virtual Werder agent framework especially during the first development phases of new behaviors this mechanism is quite comfortable, since development can be done on a per-method basis.

As mentioned earlier we implemented four special game behaviors one of which is the behavior for our goal keeper while the others are used for defenders, midfielders and forwards. We considered to specialize the behaviors even further which would go together with more detailed role descriptions but during an inspection of the decision trees used for our behaviors we found out that these are rather simplistic. This fact stresses the importance and the relative independence of our skill implementations.

Subsequently we will describe the behaviors in a graphical way since it is probably the best way to comprehend and follow the decision process in our behavior trees. Note that so far all of our behaviors are basically based on decision trees. So there is no use of other mechanisms such as Bayesian or neural networks involved. Furthermore, we do not use planning yet which means that we basically implemented reactive soccer behavior.

12.2.1 Defense Behavior

This section describes the *Defense Behavior* in terms of a graphical representation of the internal structure (decision tree based) of the handle methods implemented for the various play-modes. For those play-modes which are not mentioned in this section please refer to the 'Fall-back Behavior' on page 50.

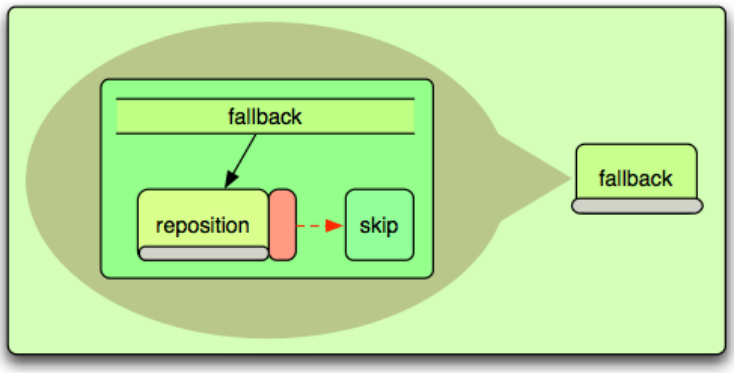


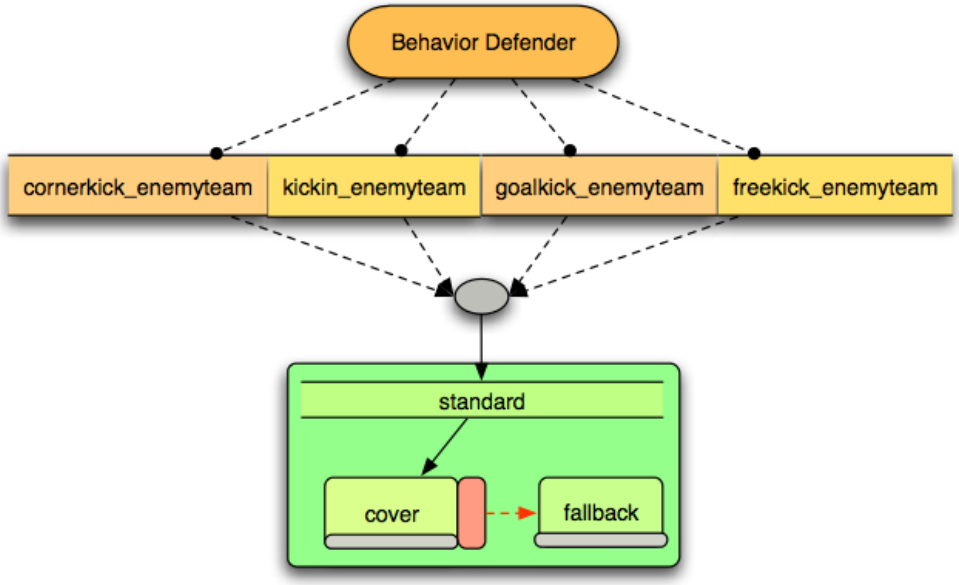
The graphic above shows the simple decision tree for

- PLAYMODE_GOAL_MYTEAM,
- PLAYMODE_CORNERKICK_MYTEAM,
- PLAYMODE_KICKOFF_ENEMYTEAM,
- PLAYMODE_GOALKICK_MYTEAM and

- PLAYMODE_GOAL_ENEMYTEAM.

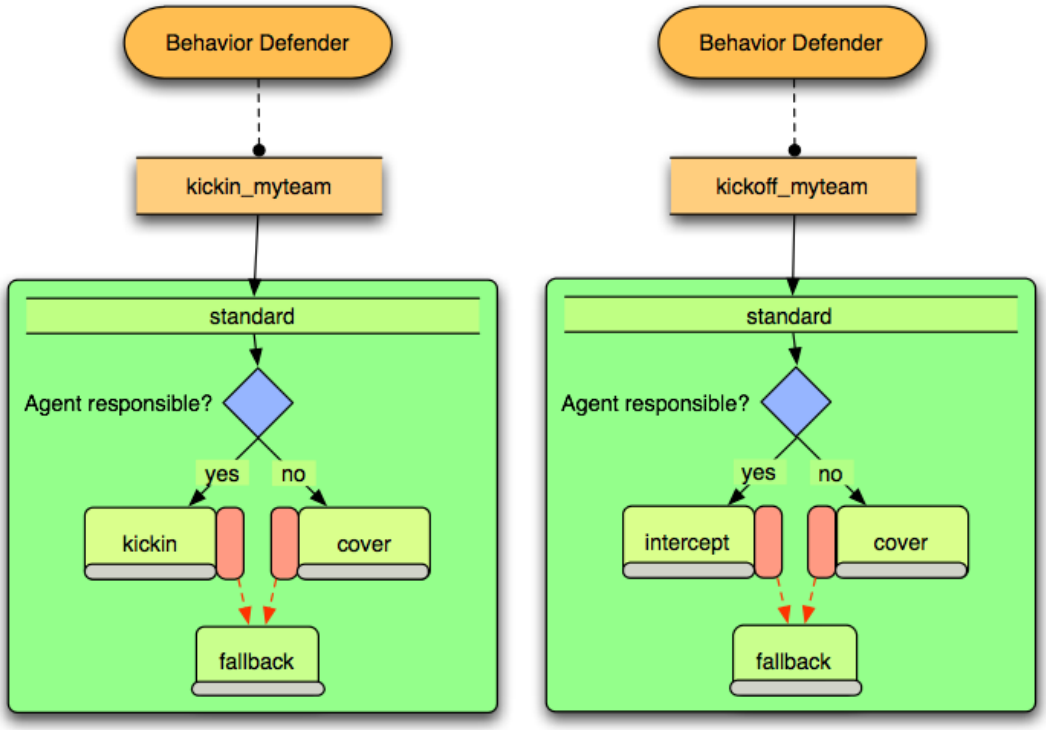
Note that the decision tree presented here corresponds to the general fall-back behavior for a defender (simple repositioning). In order to save space in the graphics presented below the fall-back is collapsed as shown below.



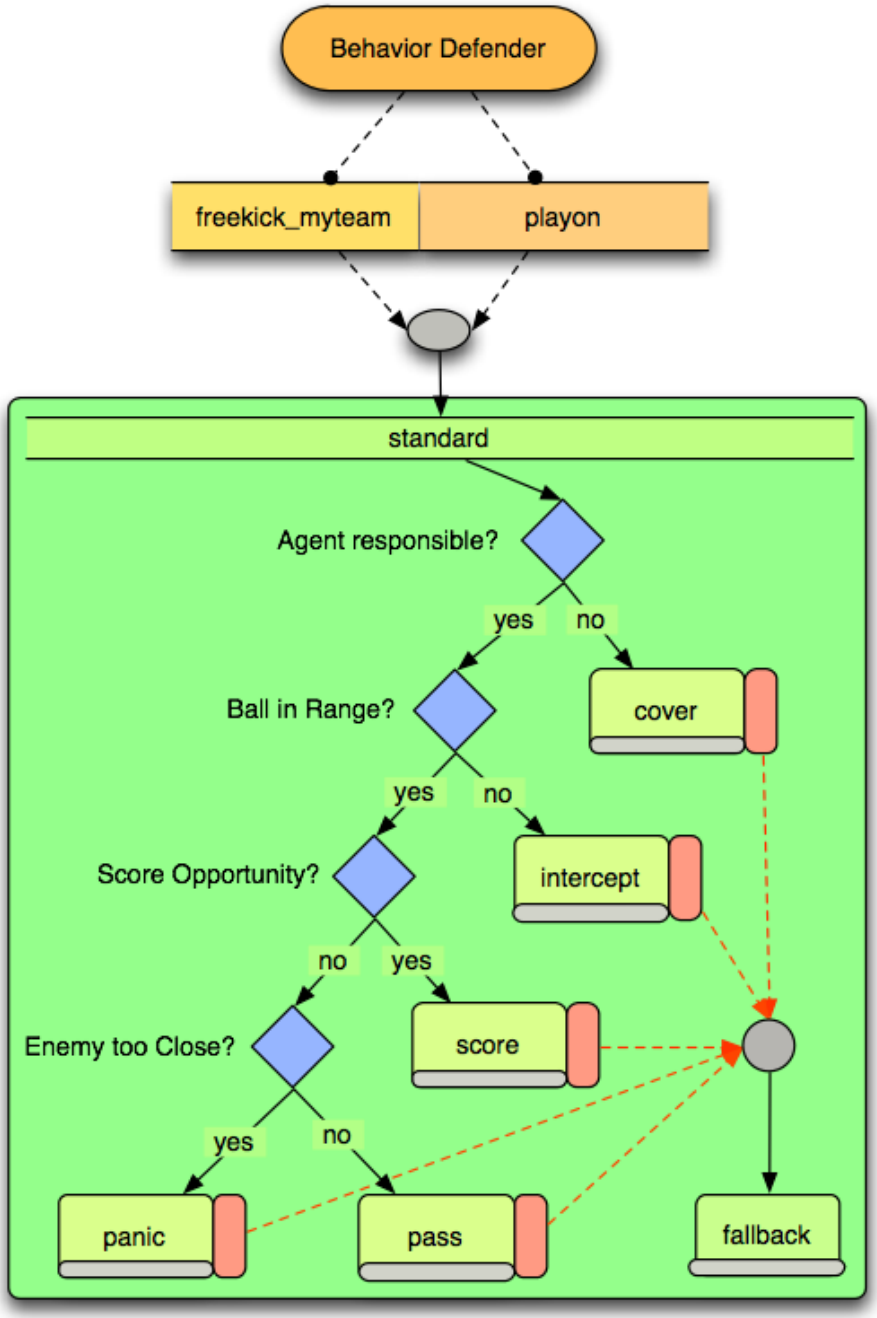


The graphic above shows the simple decision tree for PLAYMODE_CORNERKICK_ENEMYTEAM, PLAYMODE_KICKIN_ENEMYTEAM, PLAYMODE_GOALKICK_ENEMYTEAM and PLAYMODE_FREEKICK_ENEMYTEAM.

On the left side of the graphic below the decision tree for PLAYMODE_KICKIN_MYTEAM is shown while on the right side the decision tree for PLAYMODE_KICKOFF_MYTEAM is shown.

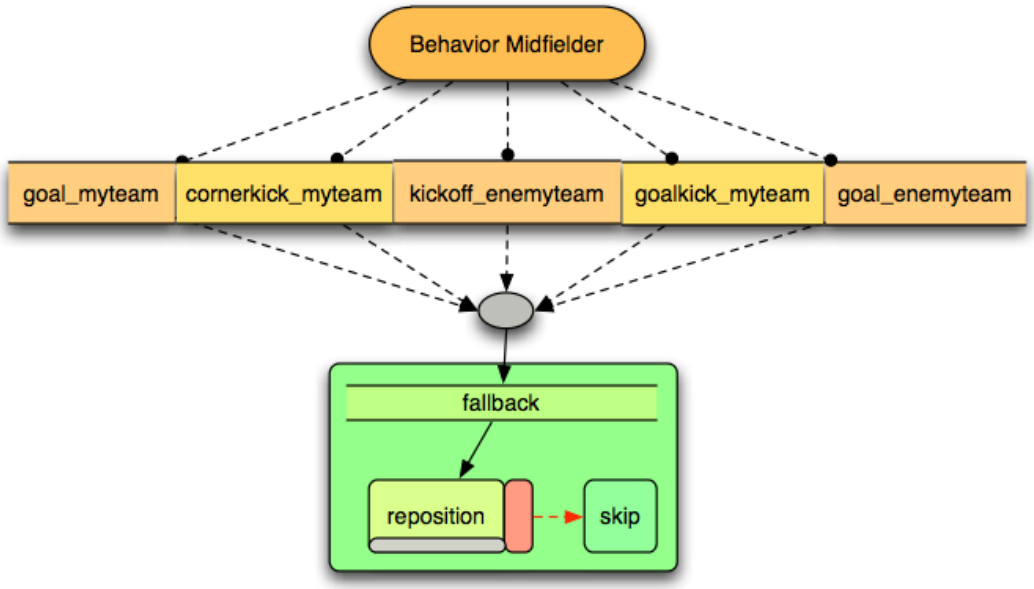

Behavior Defender
kickin_myteam
standard
Agent responsible?
yes
no
kickin
cover
fallback
Behavior Defender
kickoff_myteam
standard
Agent responsible?
yes
no
intercept
cover
fallback

The graphic below shows the decision tree which is used in the handle methods for PLAYMODE_PLAYON and PLAYMODE_FREEKICK_MYTEAM. The handle method for the latter play-mode just calls that for PLAYMODE_PLAYON.



12.2.2 Midfield Behavior

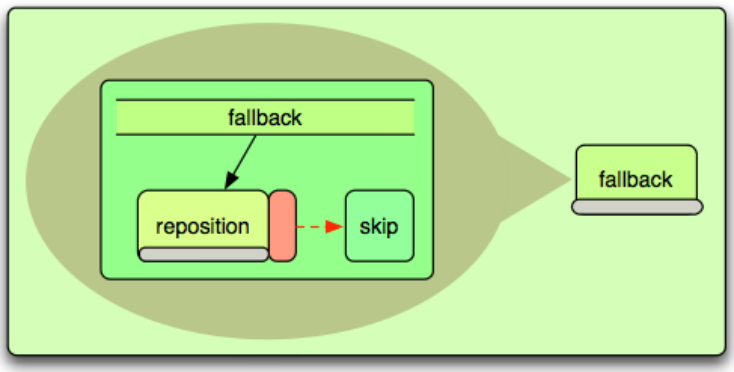
This section describes the *Midfield Behavior* in terms of a graphical representation of the internal structure (decision tree based) of the handle methods implemented for the various play-modes. For those play-modes which are not mentioned in this section please refer to the 'Fall-back Behavior' on page 50.

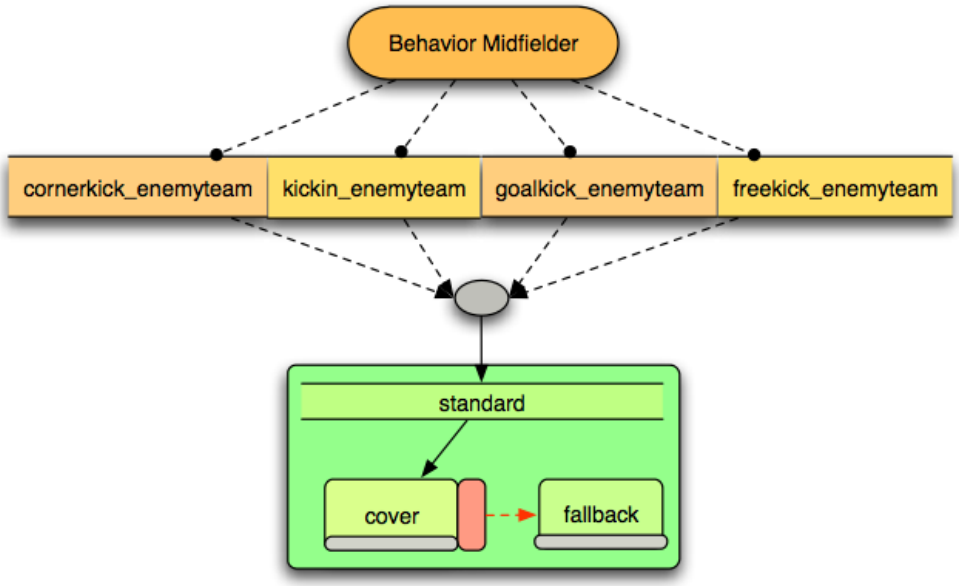


The graphic above shows the simple decision tree for

- PLAYMODE_GOAL_MYTEAM,
- PLAYMODE_CORNERKICK_MYTEAM,
- PLAYMODE_KICKOFF_ENEMYTEAM,
- PLAYMODE_GOALKICK_MYTEAM and
- PLAYMODE_GOAL_ENEMYTEAM.

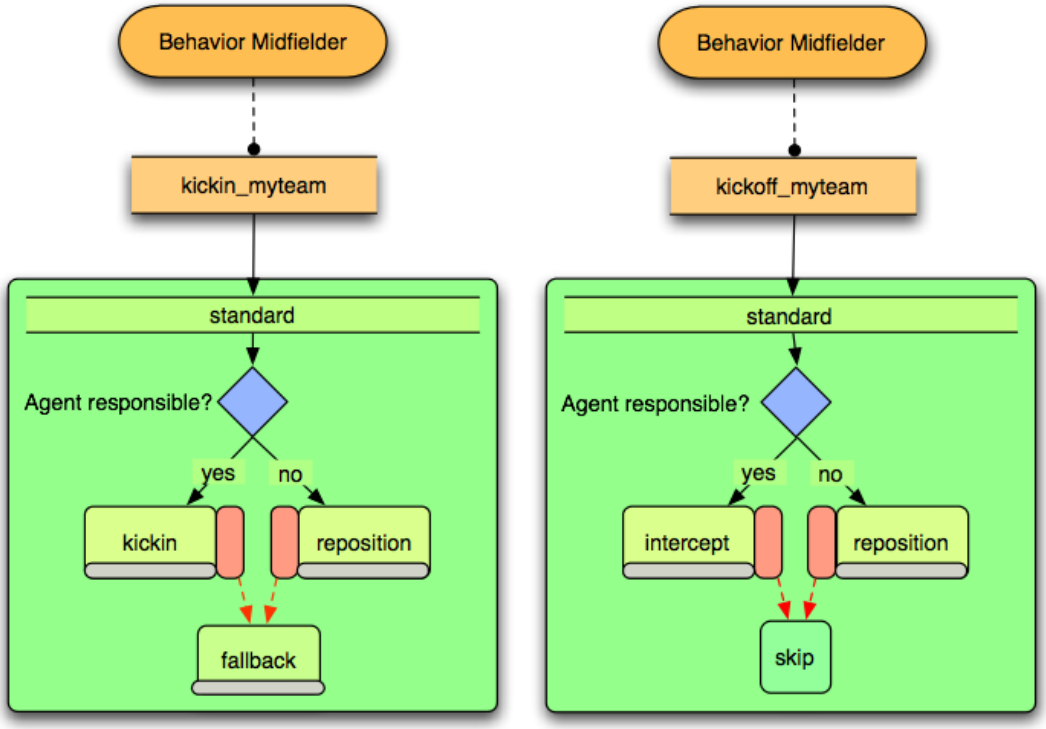
Note that the decision tree presented here corresponds to the general fall-back behavior for a midfielder (simple repositioning). In order to save space in the graphics below the fall-back is collapsed as shown below.



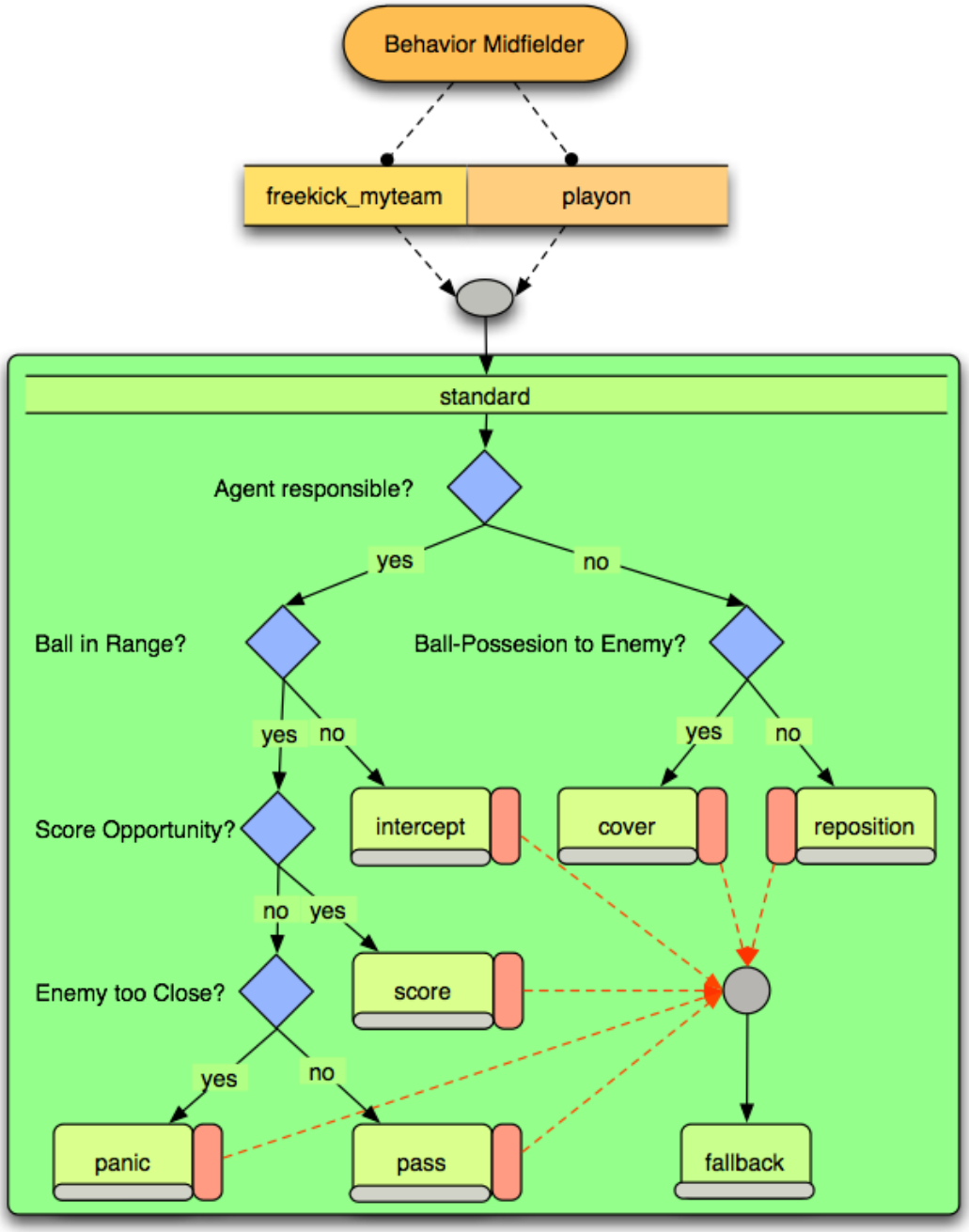


The graphic above shows the simple decision tree for PLAYMODE_CORNERKICK_ENEMYTEAM, PLAYMODE_KICKIN_ENEMYTEAM, PLAYMODE_GOALKICK_ENEMYTEAM and PLAYMODE_FREEKICK_ENEMYTEAM.

On the left side of the graphic below the decision tree for PLAYMODE_KICKIN_MYTEAM is shown while on the right side the decision tree for PLAYMODE_KICKOFF_MYTEAM is shown.



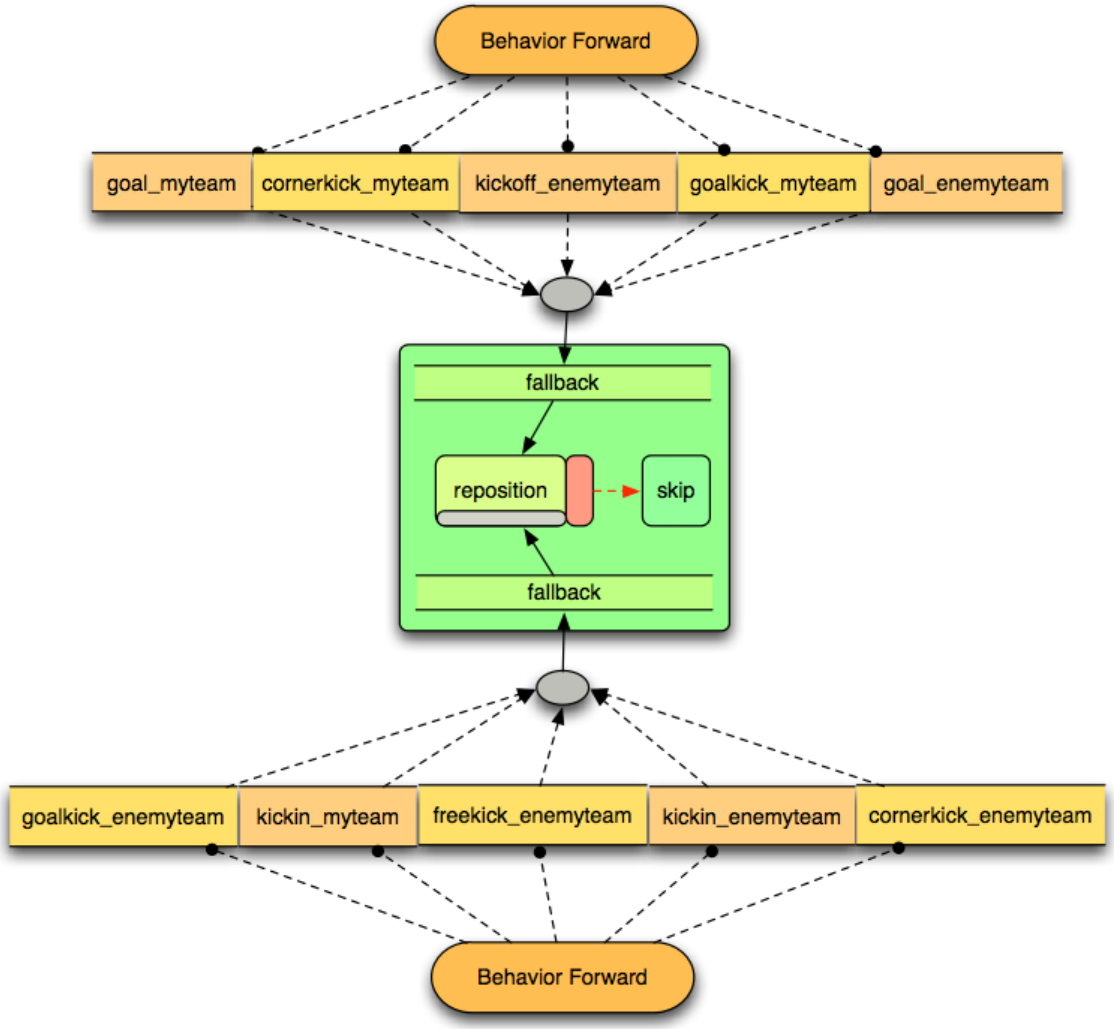
The graphic below shows the decision tree which is used in the handle methods for PLAYMODE_PLAYON and PLAYMODE_FREEKICK_MYTEAM. The handle method for the latter playmode just calls that for PLAYMODE_PLAYON.



The decision tree is a slightly modified version of that already shown in 'Defense Behavior' on page 36. In situations where a midfielder is not responsible for the ball it does not insist upon covering the enemy but in the case of a advantageous situation for the own team the agent tries to find a good position to drive forward the game of its own team.

12.2.3 Forward Behavior

This section describes the *Forward Behavior* in terms of a graphical representation of the internal structure (decision tree based) of the handle methods implemented for the various play-modes. For those play-modes which are not mentioned in this section please refer to the 'Fall-back Behavior' on page 50.

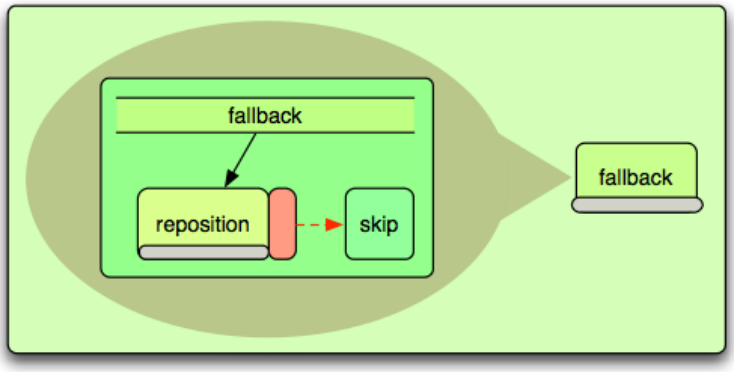


The graphic above shows the simple decision tree for

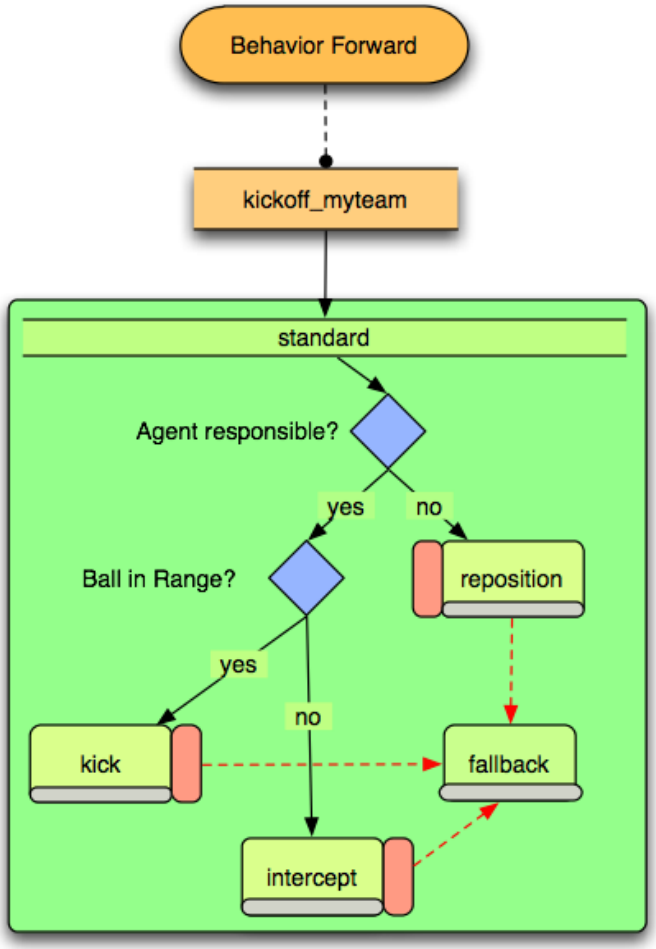
- PLAYMODE_GOAL_MYTEAM,
- PLAYMODE_CORNERKICK_MYTEAM,
- PLAYMODE_KICKOFF_ENEMYTEAM,
- PLAYMODE_GOALKICK_MYTEAM,
- PLAYMODE_GOAL_ENEMYTEAM,
- PLAYMODE_GOALKICK_ENEMYTEAM,
- PLAYMODE_KICKIN_MYTEAM,

- PLAYMODE_FREEKICK_ENEMYTEAM,
- PLAYMODE_KICKIN_ENEMYTEAM and
- PLAYMODE_CORNERKICK_ENEMYTEAM.

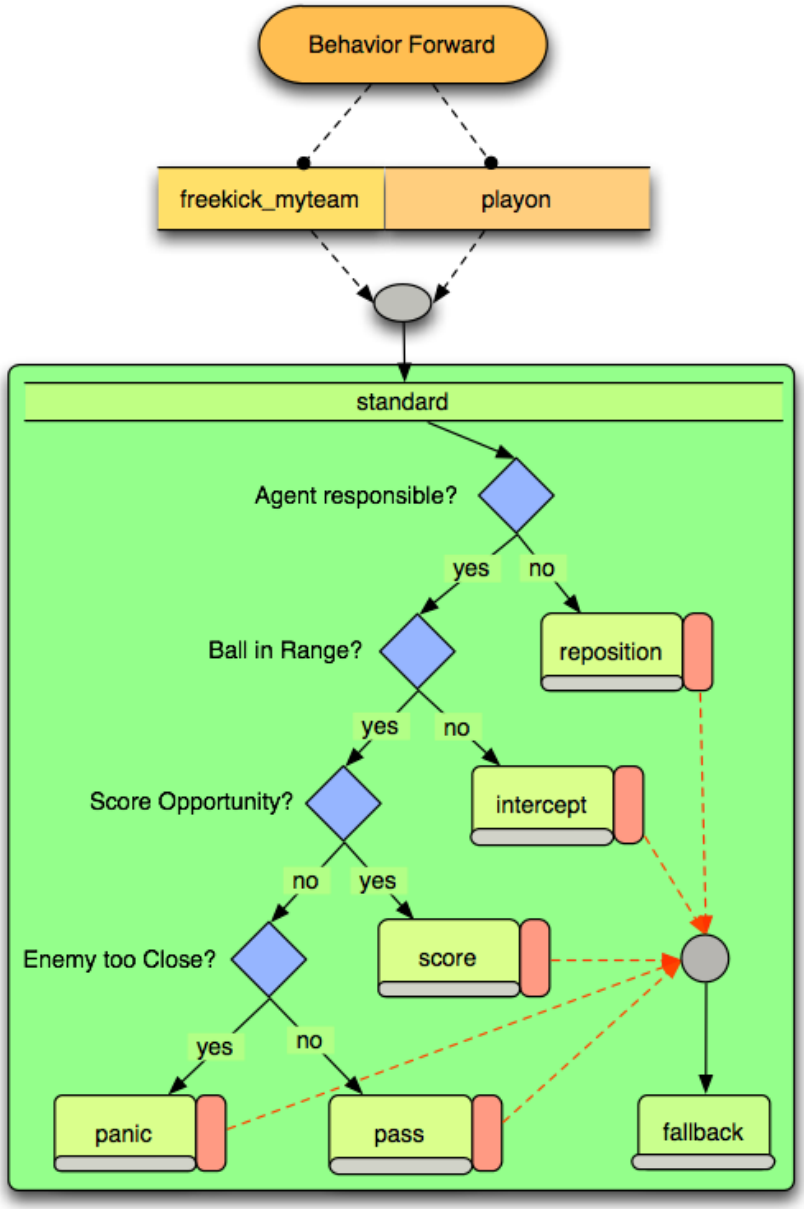
Note that the decision tree presented here corresponds to the general fallback behavior for a forward (simple repositioning). In order to save space in the graphics below the fallback is collapsed as shown below. As you can see from the multitude of covered play-modes good repositioning is quite important for attackers.



The graphic below shows the decision tree which is used for PLAYMODE_KICKOFF_MYTEAM. It is more articulated than the equivalent in the 'Defense Behavior' on page 36 or 'Midfield Behavior' on page 40 since in our team the closest attacker is supposed to perform the kickoff action.



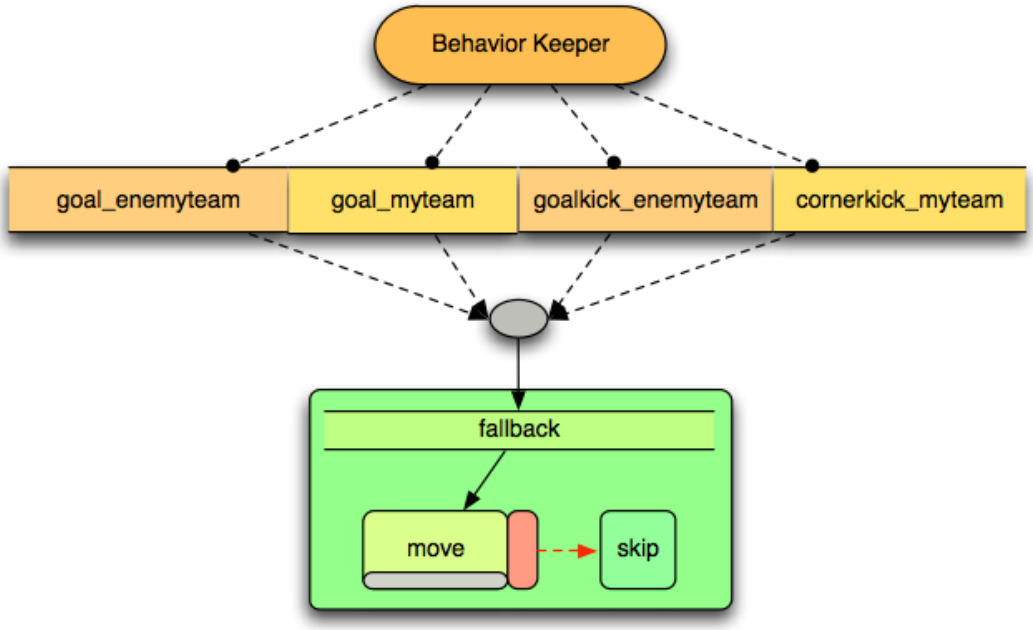
The graphic below shows the decision tree which is used in the handle methods for PLAYMODE_PLAYON and PLAYMODE_FREEKICK_MYTEAM. The handle method for the latter playmode just calls that for PLAYMODE_PLAYON.



The structure of the decision tree for an attacker matches that of a defender (cf. 'Defense Behavior' on page 36).

12.2.4 Keeper Behavior

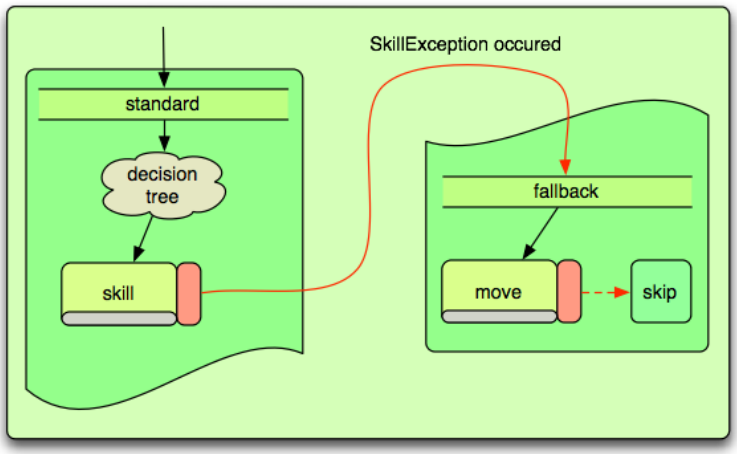
This section describes the *Keeper Behavior* in terms of a graphical representation of the internal structure (decision tree based) of the handle methods implemented for the various play-modes. For those play-modes which are not mentioned in this section please refer to the 'Fall-back Behavior' on page 50.



The graphic above shows the simple decision tree for

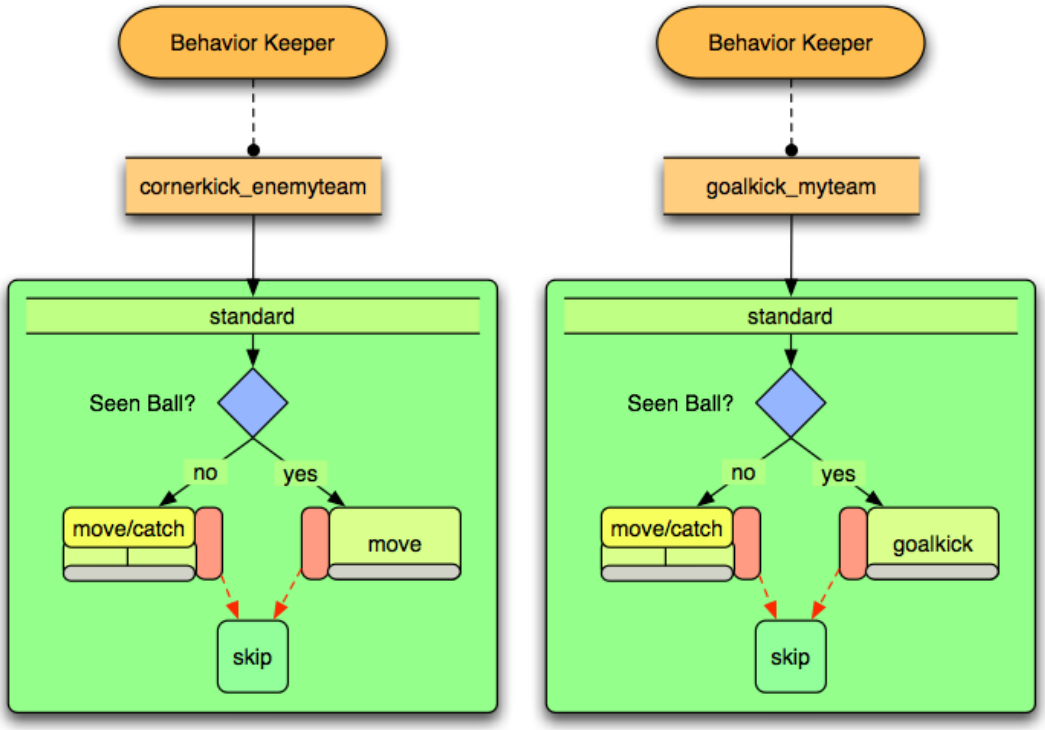
- PLAYMODE_GOAL_ENEMYTEAM,
- PLAYMODE_GOAL_MYTEAM,
- PLAYMODE_GOALKICK_ENEMYTEAM,
- PLAYMODE_CORNERKICK_MYTEAM.

Note that the decision tree presented here corresponds to the general fall-back behavior for the keeper (simple repositioning in terms of move to a certain point in the goalie area). In order to save space in the graphics below the fall-back fallback is made implicit.



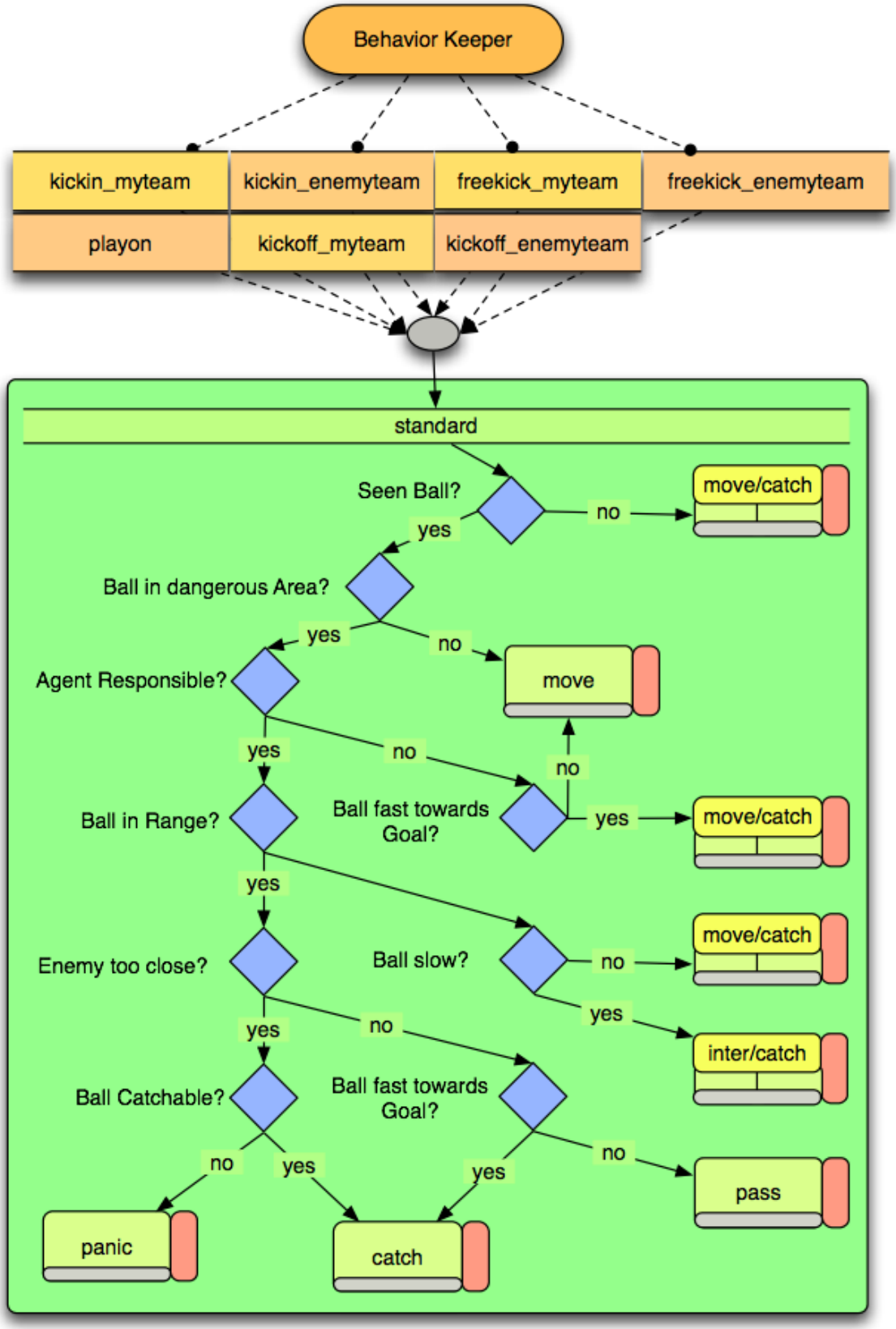
On the left side of the following graphic the decision tree for PLAYMODE_CORNERKICK_ENEMYTEAM is shown. On the right side the decision tree for PLAYMODE_GOALKICK_MYTEAM is shown.

Two remarks are required at this point in order to understand the following graphics for the keeper. First, even though within the decision trees the move skill is used in different situations that does not imply that the keeper actually does the same thing. Please refer to the comments in the code in order to find out where the keeper is moving in each situation. Second, the keeper is the only player in our team that uses multi-actions (besides view control skills which are used concurrently). Even though this is also possible for field players due to the hierarchical structure of the skills the keeper behavior is the only place where multi-actions are actively triggered directly within the handle methods.



The graphic on the following page shows the decision tree which is used in the handle methods for

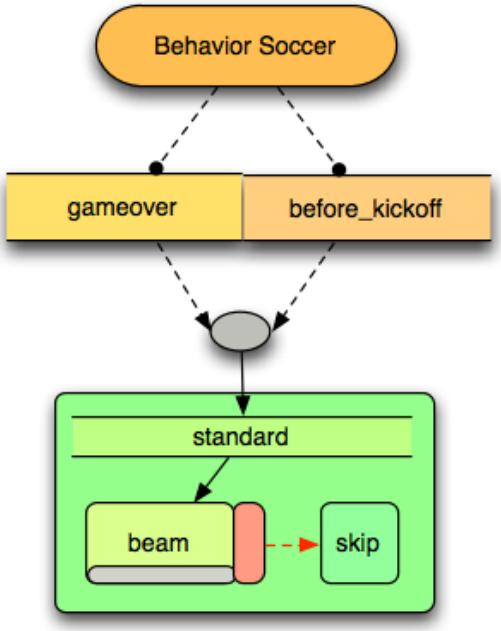
- PLAYMODE_KICKIN_MYTEAM,
- PLAYMODE_KICKIN_ENEMYTEAM,
- PLAYMODE_FREEKICK_MYTEAM,
- PLAYMODE_FREEKICK_ENEMYTEAM,
- PLAYMODE_KICKOFF_MYTEAM,
- PLAYMODE_KICKOFF_ENEMYTEAM and
- PLAYMODE_PLAYON.


Behavior Keeper
kickin_myteam
kickin_enemyteam
freekick_myteam
freekick_enemyteam
playon
kickoff_myteam
kickoff_enemyteam
standard
Seen Ball?
yes
no
move/catch
Ball in dangerous Area?
yes
no
move
Agent Responsible?
yes
no
no
Ball in Range?
Ball fast towards Goal?
yes
move/catch
yes
Enemy too close?
Ball slow?
no
move/catch
yes
yes
no
inter/catch
Ball Catchable?
Ball fast towards Goal?
no
yes
yes
no
pass
panic
catch

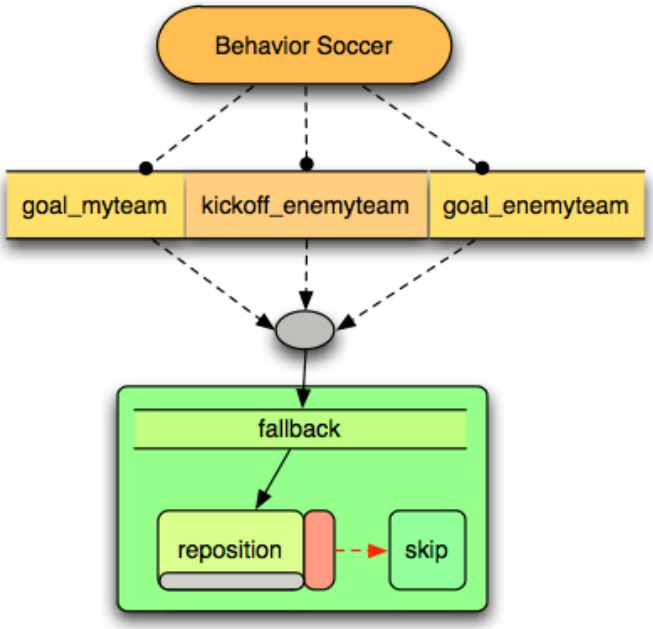
12.2.5 Fall-back Behavior

This section describes the *Fall-back Behavior* in terms of a graphical representation of the internal structure (decision tree based) of the handle methods implemented for the various play-modes.

This behavior is the base for all other specialized behaviors that have already been introduced so far. This behavior features a full set of handler methods. However, it is not really a competitive behavior. It basically makes sure the agents act like dumb standard field players

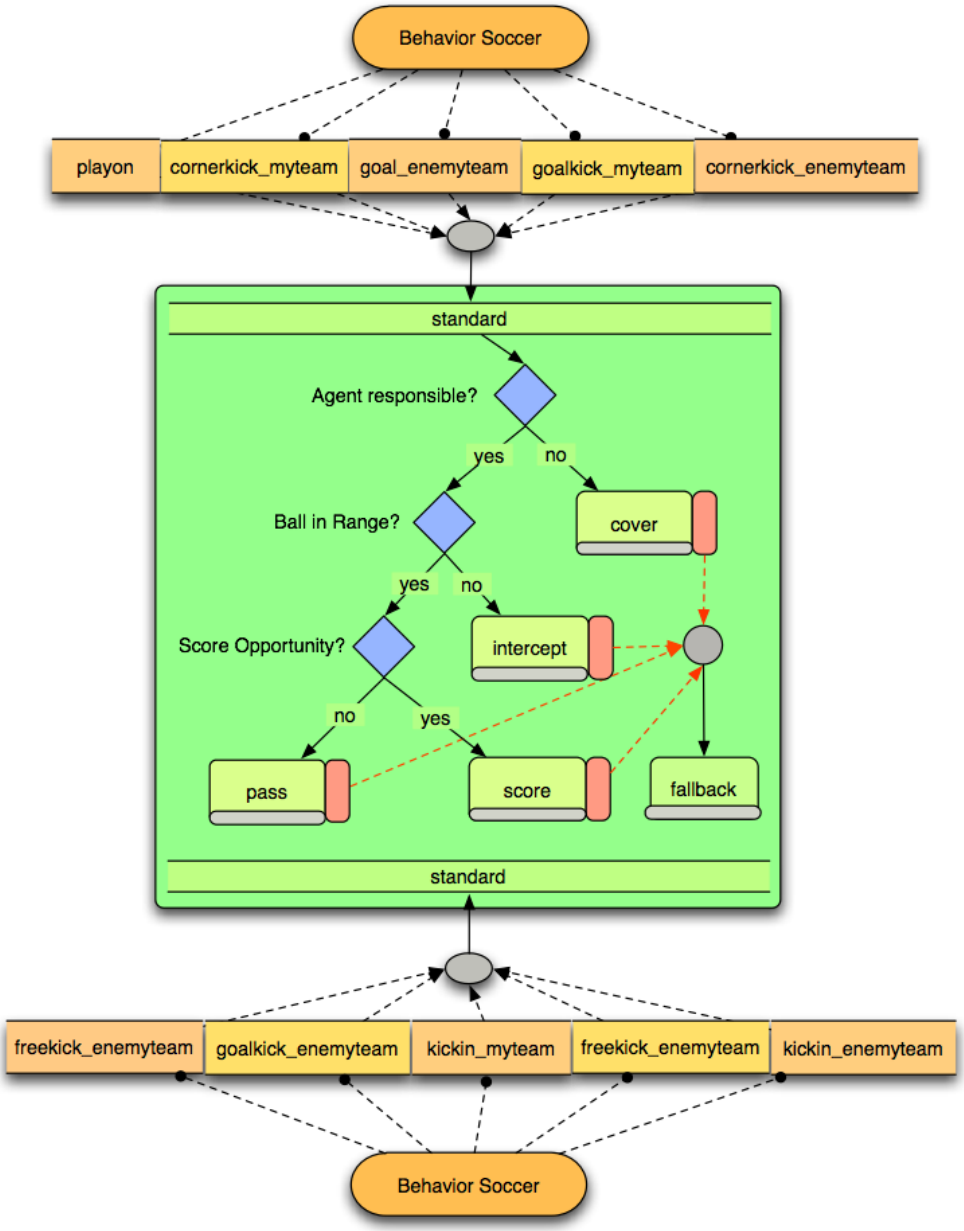


The graphic above shows the simple decision tree for PLAYMODE_BEFORE_KICKOFF and PLAYMODE_GAMEOVER.



The graphic above shows the simple decision tree for PLAYMODE_GOAL_MYTEAM, PLAYMODE_KICKOFF_ENEMYTEAM and PLAYMODE_GOAL_ENEMYTEAM.

The graphic below shows the decision tree which is used for all remaining play-modes.



12.3 Test Behaviors

Just like the game behaviors that have already been described in the previous pages the test behaviors adhere to a certain internal design which however is much simpler than that of the fully-grown game behaviors. That is due to the fact that test behaviors are used in special test scenarios which means that the fine playmode distinction is usually not necessary.

The base class for test behaviors is *Behavior_Test*. Just like *Behavior_Soccer* it defines and implements a control method which delegates the task to behave to adequate handler methods. However, *Behavior_Test* implements only the handler methods for the time before a game (preparation of tests), active

game and the time after the end of a game. If any test behavior needs a finer granularity in playmode handling this can still be done by reimplementation of the control method.

Behavior_Test is not meant to be used directly since only by specialization it is possible to define the test behaviors for the agent's skills.

13 Math Classes

The Virtual Werder code contains several classes used for frequently needed mathematical calculations. In particular, these are:

- Polar
- Polygon
- Vector2d
- Vector3d

13.1 Polar

The Polar class represents a relative position on the field, given as polar coordinates in relation to oneself. We use it as container for position related data the agent receives from the server as that comes in polar format. Apart from constructors and assignment operator, the class also has equality and inequality operators.

13.2 Polygon

The Polygon class implements convex polygons for use in calculating a player's Voronoi cell, which is used in the repositioning skill. It offers a function for cutting an initial polygon to a player's Voronoi cell, given the positions of other relevant players and getter functions to extract information about this reshaped polygon.

13.3 Vector2d

The Vector2d class represents a two-dimensional position (i.e., without elevation) on the field given in cartesian coordinates. It can be used for both relative and absolute positions. As almost all position related calculations in the skills and behaviours use cartesian coordinates, this class and its 'brother' Vector3d are frequently used. Operations available include addition, subtraction, length scaling (with the special case of scaling to length 1 – normalizing), equality and inequality checks, distance between two points, scalar product, angle between two vectors and rotation of the vector.

13.4 Vector3d

This class is almost the same as the Vector2d class with the difference that it includes the third dimension. It has the same operations as its 2D counterpart with the exception of rotating the vector, which is not available for 3D.

14 Reinforcement Learning

Manually implemented skills often lead to suboptimal solutions and time consuming parameter tuning efforts. Automated optimization techniques are thus a way to create (hopefully) better solutions and

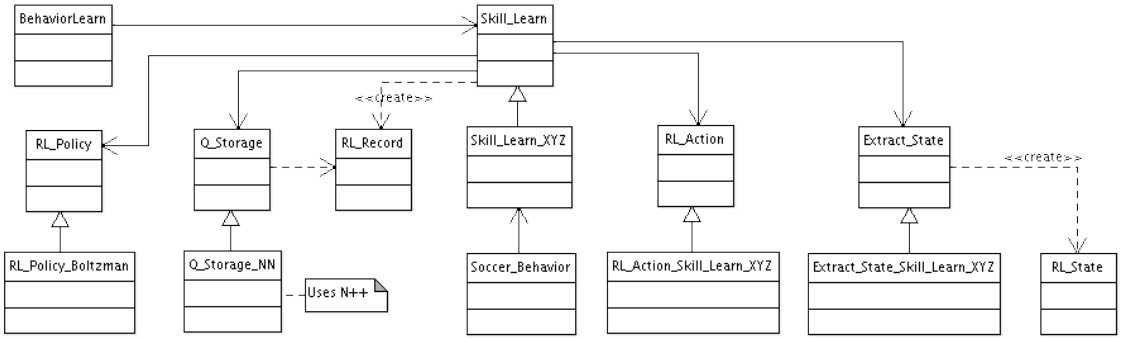


Figure 4: Learning

to be able to easily adapt to changes in the environment. Reinforcement learning has been successfully applied to RoboCup (e.g., [MR02, KS04]). A reinforcement learning framework has also been integrated to our agent. The following subsection describe how to use this framework. For more information about reinforcement learning see, e.g., [KLM96, SB98].

The architecture of the learning framework is outlined in Fig. 4. The central classes are *BehaviorLearn* for initiating the learning process and *Skill_Learn* including its subclasses for actually control the learning process and executing the actions (either random actions for exploration or the best action for using the skill). In the following subsections it is described how to use the framework for learning and using learned skills, how to map states and actions, how the Q update is performed and where the status is stored, as well as how the policy performs the action selection.

14.1 Learning and Using Skills

In order to learn a skill some new classes must be implemented which are derived from existing ones in the framework. The subclasses of the abstract class *Skill_Learn* are the central components for learning. In the learning mode a special behavior – *BehaviorLearn* – is used. This behavior has a reference to the current skill to be learned. The active learning skill is selected in the configuration file (see section 7). The behavior initiates the learning process in this skill. There are different learning modes which determine if learning is performed online or offline, and if in the current situation data for learning should be created or processed in order to update the skill. If data is collected a new learning record (*RL_Record*) is created. This record consists of all relevant information for learning, i.e., previous state, current state, executed action, immediate reward. This information can be used directly (online) for updating the Q function or later in a batch mode if the update should be performed offline.

Learned skills can be used in the same way as regular skills are used. They also have an operator, possibly with parameters, which can be called from the different soccer behavior classes. In the skill execution mode the best action for the current state is selected and executed. If the skill has been learned correctly this should lead to the best action (currently known due to learning).

14.2 State and Action Mappings

The representation of the current agent's belief of the world can become quite complex. Taking all this information into account for learning would lead to far too complex learning tasks. Thus, it is important to reduce the world state to the parts relevant for the learning task. Here, it is advisable to use a concise representation independent of the actual position or direction of objects (or object constellations), e.g., a translation and rotation invariant representation of the problem. For the creation of these learning problem adequate state representations the concept of the extractors are used. For each learning task a subclass of *Extractor_RL_State* must be implemented. The operator of this class creates *RL_State*

Parameter	Default	Description
RL__LEARN_MODE_ACTIVE	false	Learning mode is active (i.e., the learning behavior is used) if this is set to true.
RL__LEARN_OFFLINE	true	The actual update of the Q table/function is done after collecting the learning records if this parameter is set to true.
RL__LEARN_OFFLINE_EVALUATION	false	If this value is set to true the batch processing of collected records is performed.
RL__LEARN_SKILL	skill_learn_kick	Determines which learning skill is used by the learning behavior.
RL__LEARN_Q_STORAGE	q_storage_nn	Determines which q storage implementation is used.
RL__POLICY	rl_policy_boltzman	Determines which policy is used for action selection.
RL__Q_TEMPERATURE	0.03	The initial temperature value for the Boltzmann distribution
RL__Q_GAMMA	0.9	The gamma value for the Q update.
RL__Q_BETA	0.3	The beta value for the Q update.
RL__Q_USE_SINKING_TEMPERATURE	false	If this value is set to true the temperature for the Boltzmann policy is decreased while learning.
RL__Q_TEMPERATURE_DOWN_VALUE	0.01	This indicates the step size for decreasing the temperature.
RL__Q_TEMPERATURE_DOWN_TRIALS	100	The number of trials determine after how many learning steps the temperature is decreased.

Table 1: Learning parameters in the configuration file

instances for the learning task.

Similarly to the states the actions that can be executed must be defined for the learning task. In the case of Q learning a discrete set of actions must be known. Each action is represented by a unique identifier (the action ID). The learned skill holds the information which action ID is the best for a certain situation (or has been selected randomly). In order to execute the action in the agent framework it must be mapped to a concrete action, e.g., by using the effectors of the agent. For instance, it would be possible to set up a number of fixed directions and power values for the drive effector and to use this as the set of possible actions.

14.3 Q_Storage

The Q storage is responsible for holding the information how good actions are in certain situations (the Q values). Usually, this information is kept in a so called Q table where for each state-action pair a Q value is stored indicating the expected performance when executing the action in this situation. A disadvantage of the table representation is that the table can get quite big and that for continuous state attributes it would be impossible to create an entry for all situations. In order to address this problem

function approximators are used. In this case a function approximator represents the information that would usually be hold by the Q table, i.e., the function returns a Q value for a given state-action pair. The difference to the Q table is, that the function approximator can have a concise representation of the Q function and that it is no problem to use continuous values if these can be handled by the approximator.

In the current implementation the class *Q_Storage_NN* is derived from *Q_Storage*. This Q storage uses a neural network as function approximator, more precisely it uses the *n++* library [Rie97] for representing, updating, and forward propagating neural networks which approximate the Q function.

The update functionality for computing new Q values is also implemented in the subclasses of the Q storage. The different results of learning steps are stored as *RL_Record* objects which hold all relevant information for performing an Q update. If desired, these records can also be collected for batch processing at a later time.

14.4 RL_Policy

The policy is responsible for action selection. Currently, there exists one one policy implementation: *RL_Policy_Boltzmann*. Here, the Boltzmann distribution is used for random selection of the action to perform. It depends on a parameter – the so called temperature – to what degree randomness should be used. In the beginning of exploration different actions should be selected randomly. After some time the learning process can be accelerated by decreasing randomness and selecting the best action (known so far) more often.

14.5 Configuration Settings

There is a number of configuration settings in the `vw3d.cfg` configuration file which are relevant for learning. Table 1 summarizes these parameters.

15 Evaluation Tool

After our framework and the vital behaviors and skills had been implemented the next step was to evaluate and fine-tune them. Amongst other things we were interested in answering questions like:

- How fast can our agent move around the ball and kick it to a designated position?
- How good is our kick-accuracy? How good is it dependent on the distance to the target?
- How accurate is our self-localization? What is the average position error, both for the agent itself as well as for other objects?

In addition we wanted to rate different sets of configuration values or different ways of handling certain tasks against each other. In order to be able to do so we implemented a tool which is able to read the server-created logfiles and create certain statistics about the game.

Though the monitor logfile holds a lot of information, the data contained only describes the game on a global level. With only the monitor.log taken into account no agent specific statistics can be created. As a consequence we needed the agent information, too, where the only source for these were the agent logfiles. We implemented an appropriate debug stream and a mapper which allowed us to write all needed data into the corresponding agent logfile. Using both the monitor logfile as well as the agent logfiles allows us to combine and compare these date and create the requested statistics.

Figure 5: Screenshots of the Eval Tool GUI

15.1 Statistics

- *Position accuracy* This statistic evaluates the position of the various objects. At first the average position error for all objects is calculated. In addition a histogram is created for every object giving the possibility to depict whether a suboptimal average position error may only be caused by a few large imprecise values or if the majority of the data is incorrect.

- *Shooting accuracy* This statistic evaluates the shooting accuracy as well as the repositioning function. A game is partitioned up into a number of segments. Each segment starts with an agent approaching the ball followed by a repositioning phase, the actual kicking and ends with the ball reaching close to zero speed. At the end of the game the average positioning error and the average reposition time is calculated from all recognized segments.

15.2 Visualization

Though the creation of the statistics was the main reason for developing the evaluation tool, we found a simple visualization of the game to be useful for both find parsing errors as well as reviewing games on other platforms than Linux. The visualization consists of a simple 2D plan view implemented using the *Simple DirectMedia Layer - SDL*[11].

[11] http://www.libsdl.org

Acknowledgment

The work presented here was partially funded by the *Senator für Bildung und Wissenschaft, Freie Hansestadt Bremen* ("FIP RoboCup", Forschungsinfrastrukturplan). We would also like to express our gratitude to Marc Halbrügge and Martin Riedmiller at the University of Osnabrück for helpful discussions addressing reinforcement learning and for providing the neural network toolbox *n++*.

References

[ATDAA+05] Hesam Addin Torabi Dashti, Nima Aghaeepour, Sahar Asadi, Meysam Bastani, Zahra Delafkar, Fatemeh Miri Disfani, Serveh Mam Ghaderi, Shahin Kamali, Sepideh Pashami, and Alireza Fotuhi Siahpirani. Ututd2005-3d team description paper. Team Description Paper for the RoboCup-2005 in Osaka, Japan, 2005. 30

[BKN+04] Tjorben Bogon, Mirco Kuhlmann, Cord Niehaus, Steffen Planthaber, Carsten Rachuy, Arne Stahlbock, Ubbo Visser, and Tobias Warden. Description of the team virtual werder 3d 2004. Team Description Paper for the RoboCup-2004 in Lisbon, Portugal, 2004. 6

[DHS+01] Christian Drücker, Sebastian Hübner, Esko Schmidt, Ubbo Visser, and Hans-Georg Weland. Virtual Werder. In Peter Stone, Tucker R. Balch, and Gerhard K. Kraetzschmar, editors, *RoboCup 2000: Robot Soccer World Cup IV*, volume 2019 of *Lecture Notes in Computer Science*, pages 421–424. Springer, 2001. Team description paper. 6

[FTBD01] D. Fox, S. Thrun, W. Burgard, and F. Dellaert. Particle filters for mobile robot localization. In A. Doucet, N. de Freitas, and N. Gordon, editors, *Sequential Monte Carlo Methods in Practice*. Springer-Verlag, New York, 2001. 25

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library*. Addison Wesley, 1999. 16

[Kar05] Björn Karlsson. *Beyond the C++ Standard Library - An Introduction to Boost*. Addison Wesley, 1 edition, 2005. 16

[KLM96] Leslie P. Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. 53

[KLP+05] Mirco Kuhlmann, Andreas D. Lattner, Steffen Planthaber, Carsten Rachuy, Arne Stahlbock, Ubbo Visser, and Tobias Warden. Virtual werder 3d 2005 team description. Team Description Paper for the RoboCup-2005 in Osaka, Japan, 2005. 6

[KO04] Marco Kögler and Oliver Obst. Simulation league: The next generation. In Daniel Polani, Andrea Bonarini, Brett Browning, and Kazuo Yoshida, editors, *RoboCup 2003: Robot Soccer World Cup VII, LNAI 3020*, pages 458–469. Springer, 2004. 6

[KS04] G. Kuhlmann and P. Stone. Progress in 3 vs. 2 keepaway. In *RoboCup-2003: Robot Soccer World Cup VII*, pages 694 – 702. Springer Verlag, Berlin, 2004. 53

[LPR+06] Andreas D. Lattner, Steffen Planthaber, Carsten Rachuy, Arne Stahlbock, Ubbo Visser, and Tobias Warden. Virtual Werder 3D 2006 team description. Team Description Paper for the RoboCup-2006 in Bremen, Germany, 2006. 6

[MR02] A. Merke and M. Riedmiller. Karlsruhe Brainstormers - a reinforcement learning way to robotic soccer. In *RoboCup 2001: Robot Soccer World Cup V*, pages 435–440. Springer, Berlin, 2002. 53

[PV07] Steffen Planthaber and Ubbo Visser. Logfile player and analyzer for RoboCup 3D simulation. In Gerhard Lakemeyer, Elizabeth Sklar, Domenico G. Sorrenti, and Tomoichi Takahashi, editors, *RoboCup-2006: Robot Soccer World Cup IX*. Springer Verlag, Berlin, 2007. To appear. 12

[Rek02] Ioannis M. Rekleitis. A particle filter tutorial for mobile robot localization tr-cim-04-02. Technical report, Centre for Intelligent Machines, McGill University, Montreal, 2002. 25

[Rie97] Martin Riedmiller. Dokumentation zu n++. n++ library documentation, October 1997. 55

[Ril03] Patrick Riley. *SPADES - System for Parallel Agent Discrete Event Simulation, User's Guide and Reference Manual for Version 0.91*, 2003. 11, 25, 28

[ROME06] Markus Rollmann, Oliver Obst, Jan Murray, and Hesham Ebrahimi. *RoboCup Soccer Server 3D Manual*, July 2006. 11, 28

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. Also available online at http://www.cs.ualberta.ca/~sutton/book/the-book.html. 53

[SC06] Richard M. Stallman and GCC Developer Community. *Using the GNU Compiler Collection (For Version 4.1.1)*, 2006. 7

[Str00] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison Wesley, 4 edition, 2000. 16