The magmaOffenburg RoboCup 3D Simulation Framework
Manual 2011

Klaus Dorer Stefan Glaser
Ingo Schindler

{Klaus.Dorer, Stefan.Glaser, Ingo.Schindler}@hs-offenburg.de

Hochschule Offenburg, Elektrotechnik-Informationstechnik, Germany

October 13, 2011

Contents

1 Introduction **1**
  1.1 Who is magmaOffenburg? . . . 1
  1.2 About this Manual . . . 1

2 System Overview **3**
  2.1 Framework Architecture . . . 3
  2.2 Project Structure . . . 5
  2.3 Package Structure . . . 5

3 Getting Started **7**
  3.1 Download and Setup Instructions . . . 7
  3.2 Run the Client . . . 7
  3.3 Run Unit Tests . . . 8

4 Meta Model **9**
  4.1 Agent Meta Model . . . 9
  4.2 RCServer Meta Model . . . 10

5 Communication Layer **11**

6 Protocol Layer **12**
  6.1 Perception . . . 12
  6.2 Action . . . 12

7 Model Layer **13**
  7.1 Agent Model . . . 13
  7.2 World Model . . . 14
  7.3 Localizer . . . 15

8 Control Layer **18**
  8.1 Beliefs . . . 18
  8.2 Behaviors . . . 19

9 Decision Making Layer **22**
  9.1 Simple Decision Maker . . . 22
  9.2 Extended Behavior Networks . . . 22

10 Future Work **23**

List of Figures

1 Layered, component-based architecture of the magma-AF. 4
2 The components of the Communication Layer. 11
3 The components and dependencies of the Protocol Layer. 12
4 The components and dependencies of the Model Layer. 13
5 The components and dependencies of the Control Layer. 18
6 The components and dependencies of the DecisionMaking Layer. 22

List of Tables

1 Comparison of the available 3D localization methods. 16

1 Introduction

The RoboCup competition is a great playground for AI researchers all over the world. They provide a challenging environment in many different areas and - much more important - one common goal! Will science really be able to beat human football champions by 2050? All we know for sure is that there is a lot to do till then.

There are many obstacles to deal with before competing in the RoboCup competition. Of course, there is nothing more fascinating than a real robot team, but dealing with real robots is usually quite expensive and needs a lot of effort on maintenance and supervision of the hardware. The simulation leagues therefore became a popular alternative to the real robot leagues. Apart from not having the disadvantages of a real robot team, the simulations are very powerful tools for research purposes. For years the 2D Simulation league was pushing the research of multi-agent cooperation techniques and provided a reliable and cheap environment for a lot of experimentation. Its successor, the 3D Simulation, addresses the gap of the lacking third dimension while opening up a lot of new research possibilities.

Some of the biggest challenges when starting a team in the 3D Simulation league is the agent architecture and the math included in model layers to setup a basic awareness of the agent and its environment. A lot of teams publish their source code, or parts of it, to support new developers in that area to overcome these challenges. As our team initially benefited enormously from the source code release of the Little Green Bats team [1], the magmaOffenburg agent framework was aimed to be an open source development as well.

The magmaOffenburg agent framework is a complete agent framework for the SimSpark 3D Soccer Simulation. It provides basic connectivity to the server, an implementation of the current protocol, a modeling of the agent and its environment, several localization methods and finally a set of basic behaviors. Developers are free to introduce, extend, exchange or remove each component they like or need to fulfill their purpose. It is just important that the usage of source code is properly acknowledged especially when competing in a RoboCup competition.

1.1 Who is magmaOffenburg?

magmaOffenburg [2] is a team of students competing in the RoboCup 3D Simulation league, supervised by Professor Klaus Dorer at the University of Applied science Offenburg, Germany. While the magmaOffenburg project was started late 2008, the magma (Motivation Action control and Goal Management in Agents [3]) project has its roots in 1999 with the magmaFreiburg RoboCup 2D Simulation team [4].

One of the main purposes of magmaOffenburg is to simplify the creation of new teams using Java. As we have shown on several competitions, Java based agents are well competitive, easy in competitions and provide high stability.

1.2 About this Manual

This manual describes the structure and concepts behind the magmaOffenburg RoboCup 3D Simulation agent framework (magma-AF). It is aimed at developers who plan to implement their own agent upon the magma-AF. As most software projects, this framework is subject of constant change and may therefore differ slightly from the given descriptions of this manual. The content of this manual is structured as follows:

The second section provides an overview of the project structure and the framework components and introduces the general concepts behind the framework.

The third section describes how to set up and run the framework.

The forth to eighth sections describe the different layers of the framework and the components in each layer in detail.

The last section summarizes further approaches and gives an outlook on future work.

2 System Overview

The magma-AF is a stand alone SimSpark client, fully implemented in Java. It provides everything from server connectivity to decision making required to run one's own agent and should be able to play soccer out of the box.

The magma-AF makes use of the apache-commons-math library, mainly because of its immutable 3D vectors. Further it makes excessive use of the vectors and matrices provided by the javax.vecmath library. We have to admit, that the use of two different vector libraries may be confusing and result in extra work for conversion into each other. But while the apache-commons-math provides immutable objects, it lacks simple matrix representations, which is on the other hand the strength of the not-immutable javax.vecmath library. We plan to restrict on just one math library in future releases most likely a mixture of both. Apart from the Java Runtime Environment, these are the only dependencies of the magma-AF.

2.1 Framework Architecture

Even though the magma-AF reached a considerable size, it follows well known software engineering concepts and is therefore still very easy to handle. It is organized in a component-based five layer architecture. Each layer is designed and implemented to avoid dependencies from lower to higher layers. Moreover, Interfaces are used to establish loose coupling between the components of each layer and the corresponding higher layer. This modular design provides other teams with the choice of creating their own agent upon any layer they prefer and exchanging specific components with their own implementations. Figure 1 shows the five layer architecture and the main components of the magma-AF.

Dependency injection is used to setup the components and their dependencies. This means that the overall structure of the system can be generated by creating the desired components and passing them to dependent components. This task is part of the initialization of the runtime component.

Since we have a synchronous client-server situation, the main application flow is cycle based and triggered by an incoming perception message from the server. The AgentRuntime is the central component in the background which manages the process of a cycle. It listens to the ServerConnection for incoming messages and triggers the different framework components in sequence. While Perception, AgentModel, WorldModel, DecisionMaker and Action are always triggered directly from the AgentRuntime, the belief- and behavior components are passive and just called on demand by the DecisionMaker.

The sequence in which the different components are triggered by the AgentRuntime is very important! As indicated before by the passive Control layer it is not important that all components are triggered once. In fact it is more like a logical sequence of steps that follow an incomming message to procude a response:

1. Parse the new message (Perception)
2. Update internal state (AgentModel)
3. Update state of environment (WorldModel)
4. Decide for and perform next behavior (DecisionMaker)
5. Reflect the executed actions deposited in the AgentModel to the effectors in the Action (AgentModel)
6. Encode and send the action message to the server (Action)

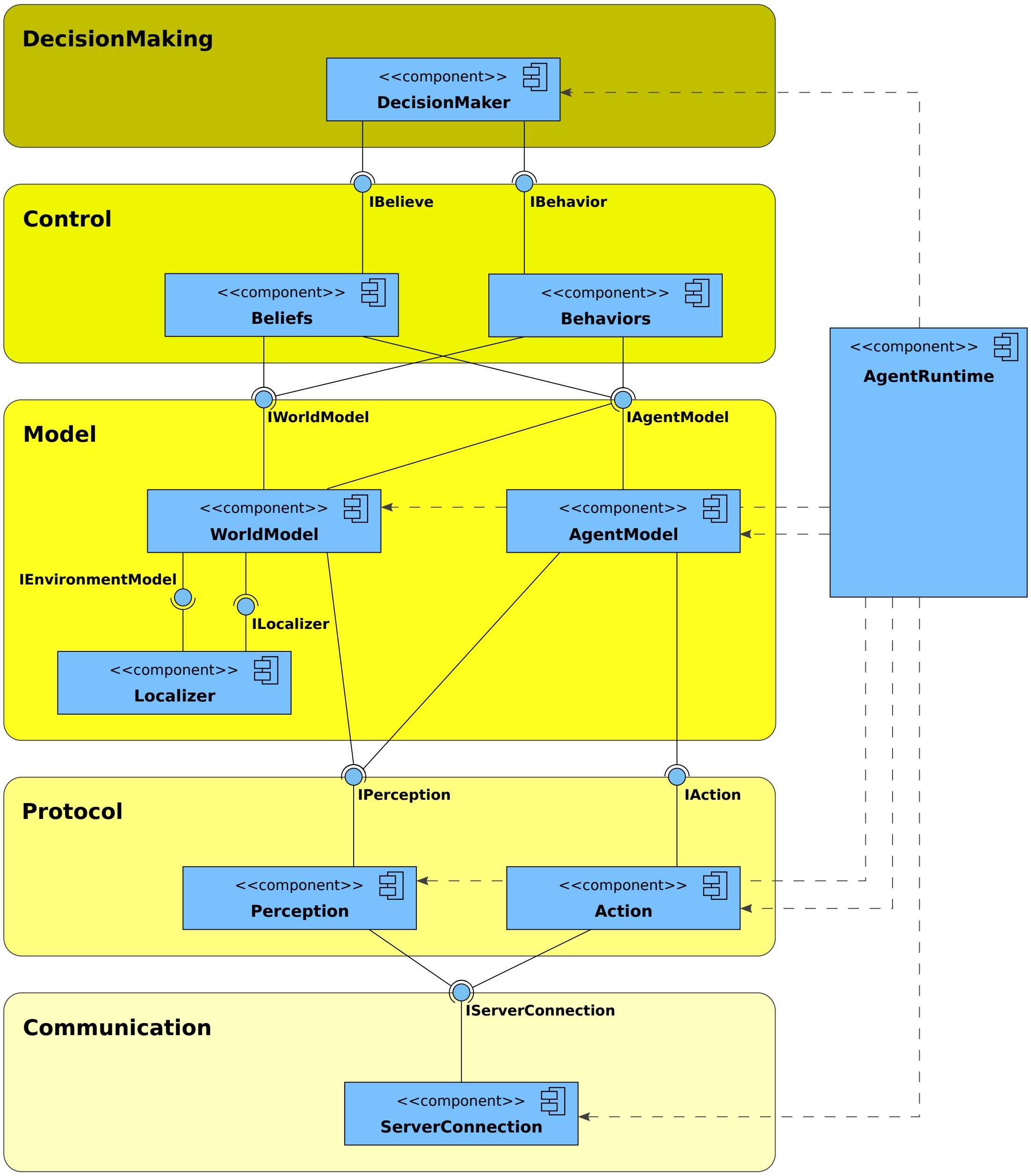

Figure 1: Layered, component-based architecture of the magma-AF.

2.2 Project Structure

The project structure corresponds to an eclipse project and therefore consists of the following directories:

- `src`: contains all source files
- `config`: contains configuration files
- `srcTest`: contains all test source files
- `bin`: contains general start scripts
- `lib`: contains runtime and test libraries
- `log`: log files go here

2.3 Package Structure

The magma-AF is organized below the top level package `magma`. Since the SimSpark simulation environment is highly flexible with respect to the final simulation itself, it is sensible to split the client into two parts - a generic and a robot-model-specific part. Together with a collection of utilities, this results in three further main packages:

magma.agent

The `agent` package contains the general, robot model independent agent framework components. This is the largest part of the framework. It defines the components shown in the architecture and provides flexible implementations to every component. Since it defines the architecture, its sub-packages are named appropriately:

- `general` - the runtime component
- `meta` - the meta model of the environment and the robot (physically)
- `connection` - establishes the connection to the server
- `perception` - parses incoming messages
- `action` - encodes outgoing messages to the server
- `agentmodel` - models the information the agent has about itself
- `worldmodel` - models the information the agent has about its environment
- `belief` - a set of (fuzzy value) beliefs indicating situations
- `behavior` - a set of behaviors carrying out diverse actions
- `decision` - decides for and performs a behavior

magma.robot

The `robot` package contains all robot specific components and definitions. Each robot model has its own package with the corresponding definitions and specific implementations. The structure of each robot package corresponds in turn to the `magma.agent` package, although usually not all components need specific implementations.

magma.util

The `util` package contains a collection of powerful utilities, used all over the magma-AF. Examples are domain independent classes for design patterns or collections of geometric calculations, which are neither dependent on a specific robot model, nor on the agent framework.

3 Getting Started

This section describes how to download and setup the magma-AF and how to run the client and the JUnit tests.

3.1 Download and Setup Instructions

The magma-AF is provided as an eclipse project (version 3.6 or higher). To setup your own project in eclipse use the following steps:

1. Download the framework sources from

   `http://robocup.fh-offenburg.de/html/downloads.htm`

2. Extract the archive to your preferred workspace folder
3. Import the extracted folder as an "existing project" into eclipse
4. Manage dependent libraries (if necessary)

3.2 Run the Client

The main-class is located under `magma.robots.RoboCupClient`. The following start parameters are possible:

1. `Team name`: Specifies the name of the team (default: magmaOffenburg)
2. `PlayerID`: Player's ID (default: 0)
3. `Host`: Host where the server is running (default: localhost)
4. `Port`: Port to connect to the server (default: 3100)
5. `Log Level`: The level of logging (default: 'severe')
   Possible log levels: 'fine', 'finer', 'finest', 'warning', 'severe' and 'info'
6. `ServerVersion`: The server version (default: 64)
   Supported server versions are currently 62, 63 and 64. Since the server version is only used to determine the correct field dimensions, the magma-AF supports also higher server versions as long the field dimensions stay the same.
7. `DecisionMakerID`: DecisionMaker of the Agent (default: 0)

   0: to decide based on player number (player number `=1 ->` goalie, `>1 ->` field player)
   1: for goalie
   2: for field player
   3: for simple decision maker (used for testing)
   4: for penalty shoot decision maker
   5: for do nothing decision maker

There are two options to run a client:

- Run from eclipse
  An example launch configuration is given in the project directory called:

  ```
  RoboCupClient (9) 2011.launch
  ```

- Run from shell script
  Start scripts are located in directory `bin`. When executing a start script the generated jar file as well as the dependent libraries (from `lib/runtime`) have to be in the same directory.

3.3 Run Unit Tests

To run all JUnit tests, a launch configuration is given in:

```
AllMagmaTests.launch
```

4 Meta Model

One of the basic concepts behind the magma-AF is the separation of the definitions and implementations of the basic agent framework from the robot model specific implementation. This enables high reusability of most of the components and significantly reduces the effort in developing different robot models.

As natural this sounds, as problematic is the implementation. Even very low layers depend on the robot model, e.g. the Action component needs to know with which effectors the current robot model is equipped, in oder to setup the effector structure. This dependency does not get better in higher layers, if anything, it gets worse. Furthermore the robot model is not the only dynamic part, also the environment definition changed over the last few years several times.

It turns out that most of the dependencies on the environment and the robot models are not very hard to resolve. This is achieved with a meta model containing a set of definitions which enable several components to accomplish their tasks independent of specific conditions. The WorldModel, for example, does not know the size of the soccer pitch directly, but obtains it via dependency injection from the server meta model. This allowes the same WorldModel class to be used to handle all server versions, without the need of class hierarchies and specific class instantiation.

4.1 Agent Meta Model

The Agent meta model defines all information relevant to the robot model in use. This encompasses the scene string that has to be sent to the server initially to choose this robot model as well as the physical definition of the robot itself. The physical description in turn is given by a hierarchical layout of body parts. Each body part is defined relative to its parent and contains the following information:

- name - name of the body part
- parent - name of the parent body part
- translation - the position of the body part relative to its parent
- mass - the mass of the body part
- geometry - the geometry of the body part
- jointAnchor - the anchor of the joint
- jointConfiguration - the configuration of the joint that connects this body part to its parent (name, perceptor name, effector name, min/max angles, max-speed)
- gyroRateConfiguration - the configuration of the gyro rate perceptor (name, perceptor name)
- accelerometerConfiguration - the configuration of the accelerometer (name, perceptor name)
- forceResistanceConfiguration - the configuration of the force resistance perceptors (name, perceptor name)

Components that do not exist are simply not defined, e.g. the root body has no parent and not every body part contains a gyro rate perceptor or an accelerometer, etc.

4.2 RCServer Meta Model

The RCServer meta model is the equivalent to the Agent meta model and contains the basic environment description of the RC soccer simulation. It includes the server version, the dimensions of the soccer pitch and its special areas, a set of visible landmarks and a set of visible field lines. To all recent RCServer versions, there exist corresponding implementations.

5 Communication Layer

The lowest layer is the Communication Layer. Its only purpose is to provide server connectivity and message exchange. The communication is not restricted to a specific medium or protocol but is abstracted to be connection and message based. As shown in figure 2, the ServerConnection is the only component of this layer.

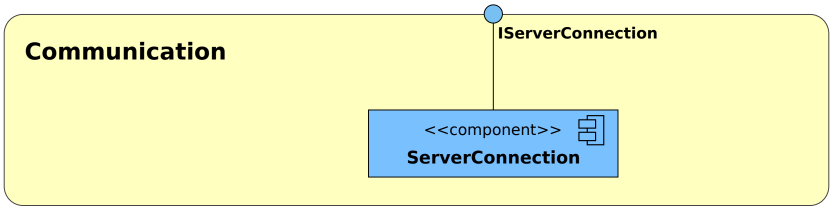

Figure 2: The components of the Communication Layer.

The ServerConnection is quite a simple component. It waits in a so called "receiveLoop" for incoming messages from the server, publishes the availability of new server messages to its listeners and provides access to the latest server message.

The ServerConnection also offers simple functionality for sending messages to the server. Outgoing messages are completely independent from the receive loop and are thought as "fire and forget" messages (We do not care if the "send" is successful or not).

Currently there is just one implementation which is based on the TCP/IP network protocol.

6 Protocol Layer

The Protocol Layer encapsulates the common protocol with the SimSpark simulation server. It makes use of the Communication Layer and consists of two components, Perception and Action. These two components are responsible for decoding incoming messages and encoding outgoing messages to the SimSpark server. The components and dependencies of this layer are depicted in figure 3.

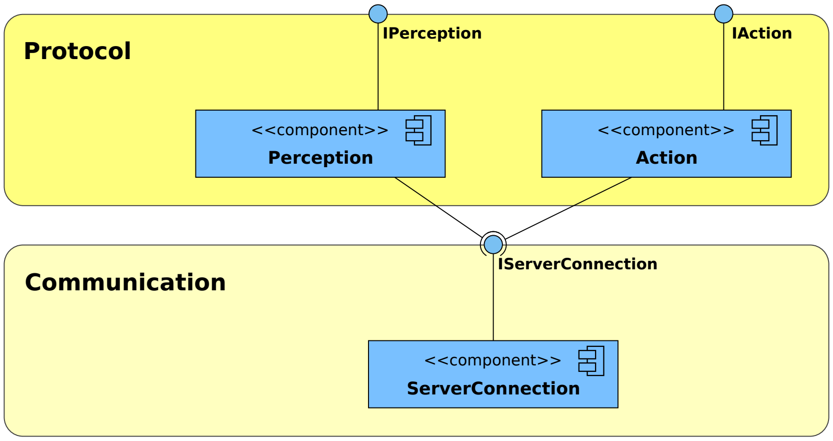

Figure 3: The components and dependencies of the Protocol Layer.

6.1 Perception

The Perception is the component decoding incoming messages from the server. Similar to the ServerConnection component, the Perception just represents one perceived state at any point in time. A specific message parser is used to extract the different perceptor information contained in the received message, which is then assigned to the corresponding perceptor instances. To each available perceptor type of the SimSpark soccer simulation, there exists a corresponding representation for holding the information of a perceptor message. These perceptors are then buffered in the Perception component until the next message is parsed.

6.2 Action

The Action is the counterpart of the Perception. It encodes outgoing messages to the server. Similar to the perceptors, possible actions are encapsulated in so called effectors. There is a corresponding representation to each available effector type of the SimSpark soccer simulation in the Action component. These effector representations buffer all relevant information to encode a corresponding effector message. This way upper layers can deposit actions which are then automatically encoded on demand. A reply to the server is created by iterating over all available effectors and concatenating their encoded states.

7 Model Layer

The Model Layer represents the foundation for the software agent. While the lower layers are responsible for establishing a connection to the RC soccer domain and provide uniform internal access to it, the Model Layer is the first abstraction layer of the software agent. Its main purpose is the definition and representation of the agent's state.

As shown in Figure 4, the Model Layer consists of three components and depends just on the Protocol Layer. The state of the agent is distributed over two components, WorldModel and AgentModel, both of which make use of the Perception to update their internal state. During an update the WorldModel provides a description of the environment based on the used RCServer meta model to the localizer to determine the position and orientation of the agent. This way, the Localizer is almost completely independent from the domain. The AgentModel uses the Action component to deposit its actions.

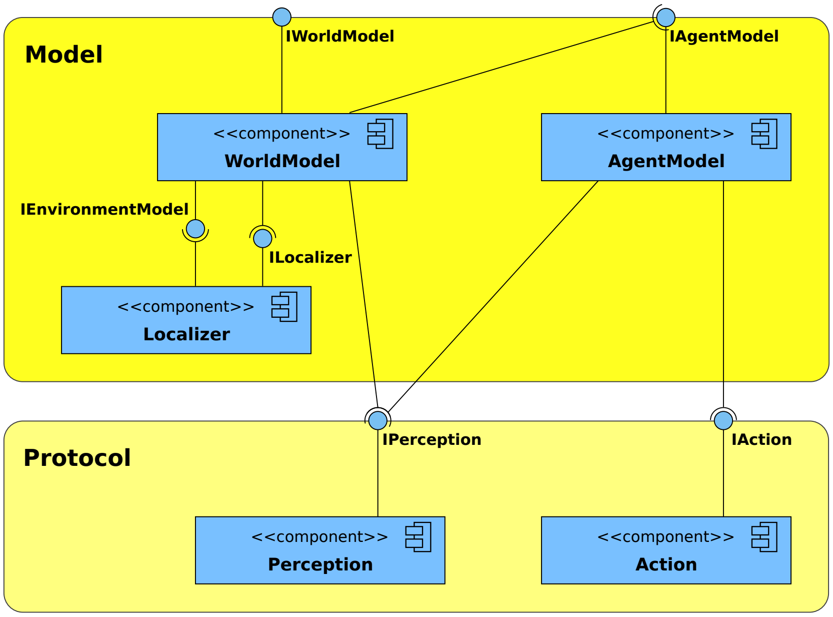

Figure 4: The components and dependencies of the Model Layer.

The separation of the overall state into the state of the robot (AgentModel) and the state of the environment (WorldModel) has many benefits, but also some drawbacks. The logical separation allowes easier maintainability and provides more flexibility, but some extra functionality is required to avoid direct dependencies. Since the extra functionality has just to be added once, the benefit is significant.

7.1 Agent Model

The Agent Model component combines interfaces for accessing the information the agent has about itself and for performing actions. The information comprises of the physical representation of the robot's body, its sensors and its actuators. Although these information is highly dependent on the robot model, by using the Agent meta model, the Agent Model component is able to dynamically provide a powerful set of calculations even to complex tasks like the kinematics model. The only requirement to the robot model is an acyclic body part structure.

Perceptions in the real world usually originate from sensors attached to the robot. Since the Agent Model is designed to organize all information the agent has about itself, it provides models for all different kind of sensors available in the RC soccer simulation. In the case of Joints, the sensor and actuator is modeled within a single component.

Due to the design of the simulation loop of the SimSpark simulator, the result of performed actions is not perceived immediatelly, but two cycles later. With respect to the Agent Model this results in three simultaneous states - the current state, the expected next state and the future state. While the current state represents the perceived information, the expected state is a combination of the current state and the previously performed actions. Finally the future state represents how the next state should look like (with respect to the physical body configuration). Later in the cycle the future state is reflected to the Action component and sent to the server. The actions are dynamically calculated on demand from the difference between the expected state and the future state. These three states are divided into three different Body Model instances. The main benefit of the third state, the future state, is not the dynamic determination of actions on demand, but the possibility to think and adjust a movement step by step before its final execution. This not only enables the easy use of inverse kinematics methods, but also provides robot model specific physical boundaries like the maximum joint angles to these methods.

The Body Model itself is unaware of what state it represents (current, expected, future). The context of the Body Model is therefore very important and other components using a Body Model need to make sure to work on the correct instance.

The design of the simulation loop is unfortunately the reason why we can not ensure the conditions of a Markov chain within the magma-AF. To deal with this issue is definitely part of the future work of this project.

Body Model

The actual information to an agent's state is held and processed in a Body Model. A Body Model is organized as a tree of body parts, typically having the torso as its root body part. During initialization of a Body Model, the Agent meta model is used to dynamically setup the body part and sensor/actuator structure.

This hierarchical representation has many benefits. Sensors and Actuators can be attached to their corresponding body parts and are thereby directly related to its local position, orientation, etc. Furthermore, forward and backward kinematics can be dynamically calculated along the hierarchical structure.

All positions, orientations, forces, etc. described in a Body Model are relative to a coordinate system facing the y-axis. This coordinate system is also called the “internal (local) system”.

7.2 World Model

All information the agent has about its environment is organized within the World Model component. Its central definitions are the local and global coordinate systems of the environment. The local coordinate system has its origin at the localized position of the agent and the system axes correspond to its internal orientation, except that this local system faces the x axis (the internal orientation is rotated by 90 deg around the z-axis). The global system in turn is abstracted in such a way, that the origin is always located in the center of the field and the x-axis points towards the opponent's goal. This is easily possible because of the point symmetry of the landmarks and field lines and results in the huge benefit of the elimination of the side-dependency within the World Model so that no other component has to deal with the question if we play left to right or right to lest. Objects in the environment, like the ball or the players around, are represented by specific interfaces and their corresponding implementations. While static object definitions of the RCServer meta model are almost directly encapsulated by the World Model component, dynamic objects like the players are not necessarily known during initialization. These objects are therefore created on demand and buffered afterwards.

Besides the basic coordinate systems and the representation of the objects in the environment, the World Model provides a wide variety of information and functionality related to the environment in general. For example, it specifies the home position of the agent (start position), the positions of the own and the opponent's goal, the general time, game time and game state, the scores of the teams, the size of the goal and penalty areas, etc. and dynamically provides a list of closest players/team mates/opponents to the ball or to the agent itself (indexical functional objects). Moreover, the World Model is the central component holding further calculation models to the environment. Currently there are approaches to calculate an area of influence of the agent or a preferred kick direction. Although, these approaches are not deployed in practice yet.

During the update process of the World Model component, the first step is the determination of the current position and orientation of the agent, in order to be able to relate the relative player positions to global field positions. This is done by calling the Localizer component with the current sensor information. If the localization is not possible, the position estimation of the last cycle is simply kept. After the localization process, all perceived visual objects are updated with their new local and global positions. Since sensors are usually distributed in different body parts, all visual information are related to the root body part in order to define the local coordinate system independent of the camera position on the used robot model. The corresponding transformation is generated by the body model of the Agent Model component.

7.3 Localizer

The Localizer is responsible for calculating the position and orientation of the agent on the soccer field. Furthermore it also tries to match the unlabeled field lines. The Localizer component defines its own interfaces for reference points and lines and is therefore even independent of the magma-AF and the soccer domain in general. Nevertheless, it should be noted that domain knowledge is used in some of the actual localizer implementations. The magma-AF provides a small collection of straight forward localization techniques. No probabilistic modeling or filtering is used yet. Table 1 shows a comparison of the available localization methods. The different localization methods are combined in a composite structure, as the localizers use different amount of reference points.

There are several different sensors available in the SimSpark simulation, which can be used for localization. The key sensor is of course the camera, which provides vectors to known/unknown objects in the environment. Besides the camera, there are also gyroscope and accelerometer sensors which can be used for localization purposes. The more the information that is combined, the more accurate the localization gets. But one has to be aware of the noise and precision of the sensors.

Method	Used information (without line assignment)	Disadvantages
LocalizerGyro	- min one point - an orientation estimation based on the gyro changes	- drifts over time (gyro based) - uses just points
LocalizerUmeyama	- min three points	- uses just points - orientation flickers
LocalizerFieldNormal	- min two points and two lines	- orientation flickers slightly

Table 1: Comparison of the available 3D localization methods.

LocalizerGyro

The LocalizerGyro was the first 3D localization method of the magma-AF. It was initially designed to use the gyroscope of the robot to transform all local vision vectors into the global system and then use them directly for position calculation. Meanwhile, the gyroscope information is preprocessed in the World Model component and provided as an orientation estimation to the Localizer. This simplifies the access to the needed transformation, but the basic concept keeps the same. The estimated orientation is used to transform the vision vectors from the local into the global system. The resulting vectors are then subtracted from their corresponding known position and the resulting position is the position we are looking for. This way even one flag is enough to calculate the own position. If more reference points are available, the average of the resulting positions is taken.

Even though this localization method just needs one flag, it is highly dependent on the gyroscope and its drifting nature. Experiences show, that the gyro lasts quite long before the drift gets too massive (approximately 2 to 3 minutes, highly dependent on the amount of movements of the agent). But since a re-initialization can be performed after every beam action and thereby after each goal, the drifting is usually not a big problem in the beginning.

LocalizerUmeyama

The Umeyama method is a well known method in 3D image analysis to register two corresponding 3D point-sets in a least square sense. While the translational component is simply defined by the data's centroid, the key calculation of the Umeyama method is the determination of the orientation. In the context of localization, this method can also be applied upon the perceived and known positions of reference points to calculate the orientation. This orientation calculation is independent of the position of the agent, which is still unknown after the application of the Umeyama method. The position of the agent is calculated afterwards in the same way as the LocalizerGyro does, except that the estimated orientation is not given by the gyroscope, but is determined earlier by the Umeyama method.

This method uses just the visual information and is independent of the drifting gyroscope. Since the field lines can not be easily assigned and there is no assignment implementation available yet, a lot of potential reference points are unavaiable, wherefore this method is not very stable. It tends to flicker in orientation as well as in the position estimation. The extent of this flicker is highly dependent on the distribution of the reference points.

LocalizerFieldNormal

The LocalizerFieldNormal is a result of exploring linear algebra. It is domain dependent but very flexible with respect to the given information. The idea behind is the fact that all field lines represent one plane, the ground. Even though they are not labeled and can't be easily related to their known positions and even worse, they could be only partially visible, all perceived points lie on one plane. A Singular Value Decomposition provides the three eigenvectors to the point set of the field lines and since they form a plane, the eigenvector to the smallest eigenvalue is a normal vector to that plane. This in turn implies, that we need at least three non-colinear points on the plane for a reliable result. Since math does not care where the sky and where the ground is, both normal vectors describe the plane identically, we have to make sure that the resulting normal vector points towards the sky. But this can easily be done by letting the normal vector always point towards the camera, which is physically bound between the ground and the sky.

The availability of the vector to the sky already defines the x- and y-rotation of the orientation (like if you know where up and down is). This partial orientation can then be used to transform reference points in such a way, that just the z-rotation is wrong. Since we are still unaware of our position, we need to relate two reference points to each other in order to calculate the missing z-rotation. It corresponds to the simple 2D top-view case. Two vectors from one reference point to another, one in the vision system and one in the known system. The angular difference of these two vectors is the z-rotation. With more than two reference points, this process can be applied several times, to average out sensor errors. After the calculation of the z-rotation, the orientation is completely determined and can in turn be used to transform the vision vectors of the reference points and to calculate the agent's position, just similar to the LocalizerGyro method.

This method turns out to be quite stable - it uses just visual information and is therefore not affected by the drifting gyroscope. It makes use of as much as possible information to determine the orientation, which also flickers a bit because of the noise in the vision system, but much less than in the above described Umeyama method. The use of the Singular Value Decomposition to determine the eigenvector to the smallest eigenvalue is clearly overkill. There exist several methods to quickly converge to just this eigenvector, which may speed up this method drastically.

8 Control Layer

Due to the high complexity and breadth of the Model Layer, it gets increasingly difficult to query and interact with it directly. To simplify decision making, the Control Layer was introduced as an abstraction layer. Within this layer, Beliefs are used to indicate different situations and Behaviors encapsulate possible interactions. As shown in Figure 5, both components make use of the World Model and the Agent Model.

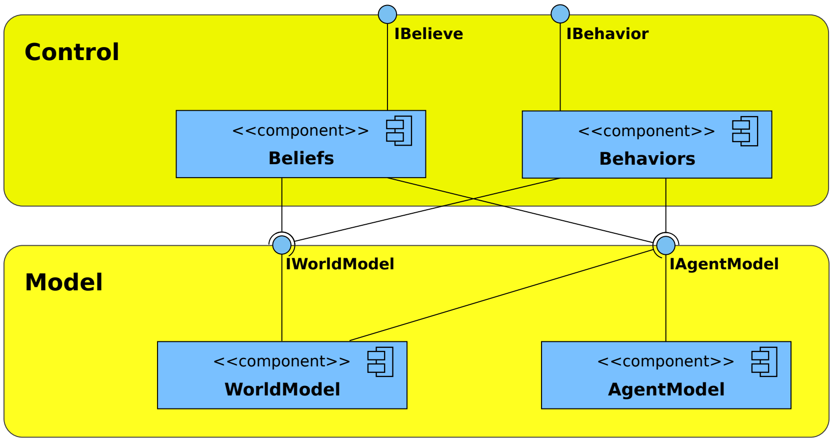

Figure 5: The components and dependencies of the Control Layer.

The introduction of this layer may be confusing in the first place. It restricts the extent to which the decision making is able to take decisions, since all interactions with the Model Layer are encapsulated in the Control Layer and there is no direct access to detailed information like position data. But it is our opinion, that this is a huge benefit in the long term. Of course, the Control Layer somehow defines the extent to which the decision making can take place. Even though this may sound harmful, it forces the decision making to be high level and abstract. This clearly separates the responsibilities and organizes the code.

8.1 Beliefs

A Belief is similar to a query to the Model Layer. It represents a fuzzy truth value between 0 and 1, indicating how much this Belief is fulfilled in the current state. The truth values are then used in the decision making, either directly or in a combined way, to take a decision for the next behavior to perform.

The magma-AF already provides a basic set of Beliefs, indicating general situation information like if the game is running or in which state the game currently is (before kickoff, playOn, etc.). Also more specific information like if the agent is lying on the front/back/side or if an opponent is near the ball are provided.

8.2 Behaviors

The coordinated, continuous control of the joints of a robot in the 3D space is one of the key challenging tasks in the RoboCup 3D Simulation. As simple the control of movements seem to be for humans, as difficult it is to teach it a robot or even let it learn by itself. The Behavior component of the magma-AF encapsulates the possible actions, that can be performed by the agent. The scope of the behaviors is hereby not restricted to the control of the joints, they are also used to control other actuators of the Agent Model like the say-actuator or the beam-actuator or may even use other behaviors.

One possibility to define movements are static movement sequences. Static movement sequences are simple collections of functions representing joint positions over time. This kind of representation is usually quite simple but only partially adaptable to the current situation. The Behavior component of the magam-AF offers several different ways of defining static movement sequences.

Another possibility is to define movements dynamically, based on the current situation. While there exist many different approaches to implement dynamic movement control procedures, the magma-AF mainly focused on the concept of inverse kinematics so far.

In general, behaviors have an internal state. For example in the case of static movement sequences, a counter is needed to indicate the current index in the movement sequence. This implies, the state of the behaviors has to be maintained when switching between them. In the normal case, behaviors are performed till they indicate that they are finished before switching to the next behavior. Besides the possibility to abort performing a behavior and directly switch to another, some kind of behavior morphing is reached by asking the current behavior to switch to another if possible.

The Behavior component is subdivided into five kind of behaviors:

Basic Behavior

A Basic Behavior represents the fundamental form of a behavior. It is typically static, hard-coded and meant to be simple and straight forward. Basic Behaviors are used to provide fundamental actions like doing nothing, beaming, emergency stop, etc.

Motor Behavior

A Motor Behavior defines a static movement sequence. It consists of a set of functions, representing joint positions over time. Since functions over time can be easily represented, either as a discrete table or as a set of parameters to construct a continuous function, Motor Behaviors are stored in separate files. All available Motor Behaviors are dynamically loaded during startup. Special file loaders scan different folders for Motor Behavior files and try to load them. Please see */config/behaviors/nao* for some example behavior definitions.

The magma-AF currently supports two different behavior file formats:

- **CSV behavior file format:**
  This format is organized as a table of joint positions over discrete time steps. The different values are separated by commas. The first line specifies the name, the version and the set of joints used by this behavior. The subsequent rows are called poses and contain the corresponding time step, the pose name and a set of joint positions (in radians) for the specified joints.

File format:

```
<behavior-name>,<version>[, joint-name]
<time-step>,<pose-name>[, joint-value]
...
```

`<behavior-name>`: the name of the behavior

`<version>`: the version of the file format

`[, joint-name]`: a list of joint names, which are controlled by this behavior

`<time-step>`: a discrete timestep in the form of MM:SS:mmm (M=minutes, S=seconds, m=milliseconds)

`<pose-name>`: the name of the pose (usually not used)

`[, joint-value]`: a list of joint positions, corresponding to the specified joint names (in radians)

- **Function behavior file format:**
  This format defines continuous functions to a set of joints by different parameter sets. The first line specifies the name of the behavior, the file format version and the period (the overall duration of the behavior). The subsequent lines each relate a specific parameterized function to a joint. Currently there are three different function definitions available: A spline-, a sinus- and a piecewise linear function.
  File format:

```
<behavior-name>,<version>,<period>,1.0,1.0
<joint-name>,<function-type>[, parameter]
...
```

`<behavior-name>`: the name of the behavior

`<version>`: the version of the used format

`<period>`: the duration of the described behavior in cycles

`<joint-name>`: the name of the joint

`<function-type>`: the type of the movement function to a joint (currently there are representation for spline-, sinus- and linear functions)

`[, parameter]`: the parameters to the specified function:

spline: a list of support points

sinus: period, amplitude, phase, offset

linear: a list of support points for a piecewise linear function

Movement Behavior

A Movement Behavior defines semi-static movement sequences within Java code. It represents a state machine and consists of one or more movement phases. Each movement phase contains a set of target positions to some joints with a corresponding maximum speed and the overall duration of the phase. The current movement phase is also used as the state of the Behavior. The transitions between the available movement phases can be designed dynamically based on further checks of the Model Layer. For example, if there is a movement phase which stabilizes the robot before kicking the ball and after stabilizing, the ball is not there anymore, the behavior can switch to “finished” immediately without performing the actual kick.

Dynamic Behavior

As the name already implies, Dynamic Behaviors adapt their actions dynamically to the actual situation. In order to do so, they usually use different information of the Model Layer to calculate the next action. A classical Dynamic Behavior is the behavior to focus on the ball. It uses the model to calculate angle differences and moves the relevant joints accordingly. Another example is the walk behavior. It uses spline functions to define body part positions and orientations to model the basic movement in an abstract way. These splines are then used together with internal state information to calculate the next set of target positions for joints involved in the current cycle.

Complex Behavior

In contrast to its name, a Complex Behavior does not necessarily need to be very complex. In fact, some Dynamic Behaviors are fairly more complex than some Complex Behaviors. Complex Behaviors are composites in a software engineering sense built from possibly many Simple or other Complex Behaviors. The term Complex Behavior describes a kind of intermediate decision behavior, calling other behaviors instead of directly performing actions. A Complex Behavior is usually initialized with a list of all other behaviors. Since behaviors have direct access to the Model Layer, they are able to perform specific checks on the Model Layer which can not be done in the decision making. This direct access allowes Complex Behaviors to orchestrate and parameterize other behaviors more precisely than decision making can do to fulfill an abstract task. Complex Behaviors can be nested, to keep the different behaviors as flexible and simple as possible. A simple example is a Keep behavior which uses other behaviors to jump to the left or the the right as soon as the ball seems to pass by the player. Another example is a RunToBall behavior that is based on a RunToPosition behavior which itself is a Complex Behavior built on the Walk behavior.

9 Decision Making Layer

The top most layer in the magma-AF is the Decision Making Layer. It is responsible for taking high level decisions which result in the choice of a specific behavior to perform. The DecisionMaker is the only component in this layer. Its dependencies are depicted in Figure 6.

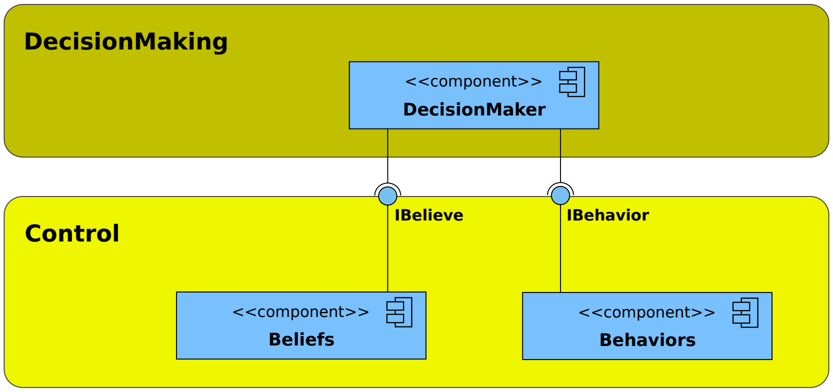

Figure 6: The components and dependencies of the DecisionMaking Layer.

The key functionality of the Decision Maker component is the mapping from belief values to a certain behavior. There are many different approaches to represent such a decision component like decision trees, logic programming languages, neural networks, extended behavior networks, etc. With the belief and behavior abstraction available, it is relatively easy in the magma-AF to switch between different decision making implementations.

9.1 Simple Decision Maker

The simplest and also the only method currently supported by the magma-AF is a hard-coded, decision tree like, hierarchical sequence of belief checks resulting in the choice of a specific behavior. The fuzzy-valued nature of the beliefs is hereby not very important. Most decisions are represented by simple if-then checks. This rudimentary decision making soon becomes relatively hard to maintain. But as long as the number of beliefs and behaviors is not very high, it is still simple and sufficient to define basic decisions.

9.2 Extended Behavior Networks

Extended Behavior Networks (EBNs) [5, 6, 7] can be used as a means to balance reactive and proactive behavior. They have been used as only decision making implementation in the 2D predecessor magmaFreiburg. The Java implementation has not been used until now in the 3D magmaOffenburg team since the focus has been on improving skills so far. That is why the existing implementation is not provided with the magma-AF.

10 Future Work

This section provides an overview on open tasks, that we plan to address in future releases:

- **Guarantee of Markov assumption.**
  Due to the design of the SimSpark simulation server, where the result of performed actions is perceived two cycles later, the Markov assumption is slightly violated. As the Markov assumption is a ground concept of many modern learning systems, it is desirable to ensure it.
- **Further implementations for inverse kinematics calculations.**
  We think that inverse kinematics (IK) is a key concept in dynamic robot movement, independent of the used robot model. The exploration of IK methods will definitely be part of future developments.
- **Richer model functionality.**
  Although the actual models already provide a wide variety of calculations and information, there are still some calculations missing, like the calculations of the center of gravity / stability. Further development will also focus to include more functionality in the model components.
- **Just one vector library.**
  To avoid confusion, we plan to restrict to a single vector library in future interface definitions.
- **Flexible positioning.**
  Currently each player has a fixed role and position relative to its home position. Players should be able to place themselves more dynamically and to switch roles in certain situations.
- **Probabilistic reasoning.**
  Currently there is some simple position filter averaging the last five position estimations for visible players and the ball. This helps to smooth position flickering, but using a probabilistic reasoning would result in a much more powerful representation.

It is our hope that the release of this framework helps others in getting into RoboCup. We are convinced that also the work of others will then fertilize our development again.

References

[1] Veenstra A. Neijt B. Vermeulen F. Veenstra G. Prins J. Kuypers J. Stollenga M. vd Sanden M. Klomp M. Platje M. van Dijk S. (2008) The Little Green Bats at http://www.littlegreenbats.nl/

[2] The magmaOffenburg Team Homepage http://robocup.hs-offenburg.de/

[3] Dorer K (2000) Motivation, Handlungskontrolle und Zielmanagement in autonomen Agenten. PhD thesis, Albert-Ludwigs University

[4] Dorer K (2000) The magmaFreiburg Soccer Team. In: Veloso M, Pagello E, Kitano H (Hrsg.), RoboCup-99: Robot Soccer World Cup III, Springer, Berlin

[5] Dorer K (2009) Modeling Human Decision Making using Extended Behavior Networks. To appear in: RoboCup Symposium, Graz, Austria

[6] Dorer K (2004) Extended Behavior Networks for Behavior Selection in Dynamic and Continuous Domains. In: U. Visser, et al. (Eds.) Proceedings of the ECAI workshop Agents in dynamic domains, Valencia, Spain

[7] Dorer K (1999) Behavior Networks for Continuous Domains using Situation–Dependent Motivations. Proceedings of the Sixteenth International Conference of Artificial Intelligence 1233–1238