Cooperative shoot and pass behaviour of mobile robots in the context of the TIGERS-Mannheim SSL Robocup-project

SEMINAR PAPER

in the course of studies: Information Technology

at the Baden-Württemberg Cooperative State University Mannheim

from

Daniel Waigand (273109)
Gunther Berthold (241013)

12/23/2010

Time period	12 Weeks
Course	TIT08AIN
Supervisor	Prof. Dr. Poller

Abstract

The student project Tigers Mannheim aims at enabling a team of autonomous robots, controlled by a central artificial intelligence, to play soccer. For this intent, a coordinated shooting and passing is essential and has therefor been designed and implemented. Here, the real-time environment and the adaption to the proceeding game has to be taken into account.

To achieve a successful implementation, the shooting skill has been divided into different phases. At first, a suitable target has to be evaluated by the analysis and the robot has to be moved to the ball. Then the robot has to aim by circling around the ball and the shooting mechanism has to be triggered.

The pass functionality is an enhancement of the shooting skill, but additional points have to be considered, since two co-dependant protagonists have to be coordinated.

Regarding these requirements, complex algorithms have been developed and tested successfully in both, the simulated and real environment. The procedure may be extended by dynamic approaches and adjustments to future versions of the used robot might be necessary.

Declaration

This is to solemnly declare that we have produced this paper all by ourselves. Ideas taken directly or indirectly from other sources are marked as such. This paper has not been shown to any other board of examiners so far and has not been published yet. We are fully aware of the legal consequences of making a false declaration.

Mannheim, December 23, 2010

Contents

Acronyms

AI	Aritficial Intelligence
PID	Proportional-Integral-Derivative
SSL	Small Size League
TIGERS	Team Interacting and Game Evolving Robots
UML	Unified Modelling Language

List of Figures

Listings

1 Robocup and team TIGERS

RoboCup, well, that is robots that compete in some kind of “cup”. But what robots and what competition? Now, to make things even more difficult, there are different domains and different types of robots.

It all started of back in 1997 as a venue for robot soccer matches, which is still the primary focus of RoboCup. The overall goal is to form a team of (possibly humanoid) robots that can win a match against the human soccer world champion by 2050. Besides soccer, there are also competitions in the domains of rescue systems and home assistants.

RoboCup aims at advancing robotics especially in the field of artificial intelligence. The competitive character propels the development.

RoboCup soccer takes place in different leagues. That is the Humanoid League, the Standard Platform League, the Simulation League, the Middle Size League and last but not least the Small Size League. Each league has a number of restriction on size and capabilities of the robots.

The TIGERS Mannheim RocoCup team plays in the Small Size League (SSL). Here, the teams actors must fit into a cylinder of 150mm height and 180mm diameter. A match takes place between two teams of 5 robots each on a pitch with a length of 6.05m and a width of 4.05m. Robots play with an orange golf ball.

A global vision system above the pitch determines the positions of all robots and the ball. Position data will then be available to the teams. Typically there is a central, off-field process computer that calculates robot behavior bases on the vision. Robot movements may then be commanded via a wireless connection.

Most teams incorporate an omnidirectional drive that can accelerate the robot to a velocity of more than 3m/s. The ball may be kicked at maximum with a speed of 10m/s. Both lead to a highly dynamic environment where only the most clever design can succeed. [3][2]

2 Introduction

Enabling robots to play soccer successfully is unimaginably complex. Apart from building the robot, which is highly difficult in itself, the main work lies in clever software algorithms which can transform a stupid machine into a “mechanical Pelé”. But a soccer team need not only one player but a whole team. Moreover, the teams player needs to interact and cooperate to manoeuvre the ball into the opponent goal.

2.1 Motivation

Now, can you imagine a soccer player that isn’t capable of shooting the ball? We neither!

The robots of team TIGERS were completely lacking adequate algorithms that control the robots kicking device. What is more, complex movement patters that are needed especially for passing were not existent. Thus neither shooting nor passing was possible. Since shooting and with it passing is crucial for an effective soccer team, particular focus was turned towards a both accurate and fast shoot and pass behaviour.

2.2 Application

Accurate shoot and pass capabilities are needed throughout various offensive plays. Usually an attack is finalized by an decisive shoot on the opponent goal. More advanced attacks may incorporate a number of passes between two robots before the ball can be shot on the goal. Even in defensive situations it might be beneficial to pass the ball to another player to quickly get the ball away from the own gaol. This would be the preferred behavior since a counterattack can then be executed.

2.3 Goals

The goal of our work is the development of straight forward, simple to use and robust algorithms that let the robots perform shoots and passes. The components should be

generic and highly reusable. Every AI developer should be able to include the shoot and pass routines into his own play.

3 The Artificial Intelligence

3.1 A black box view

The general purpose of the TIGERs Artificial Intelligence (AI) is to generate movement-commands based on input, which provides the positions of the ball as well as of all robots in play. To fulfill this task, the AI is embedded as a single module in the central software called Sumatra. Sumatra basically consists of three main modules and several optional and/or supporting parts. These modules and their interactions are shown in figure 3.1.

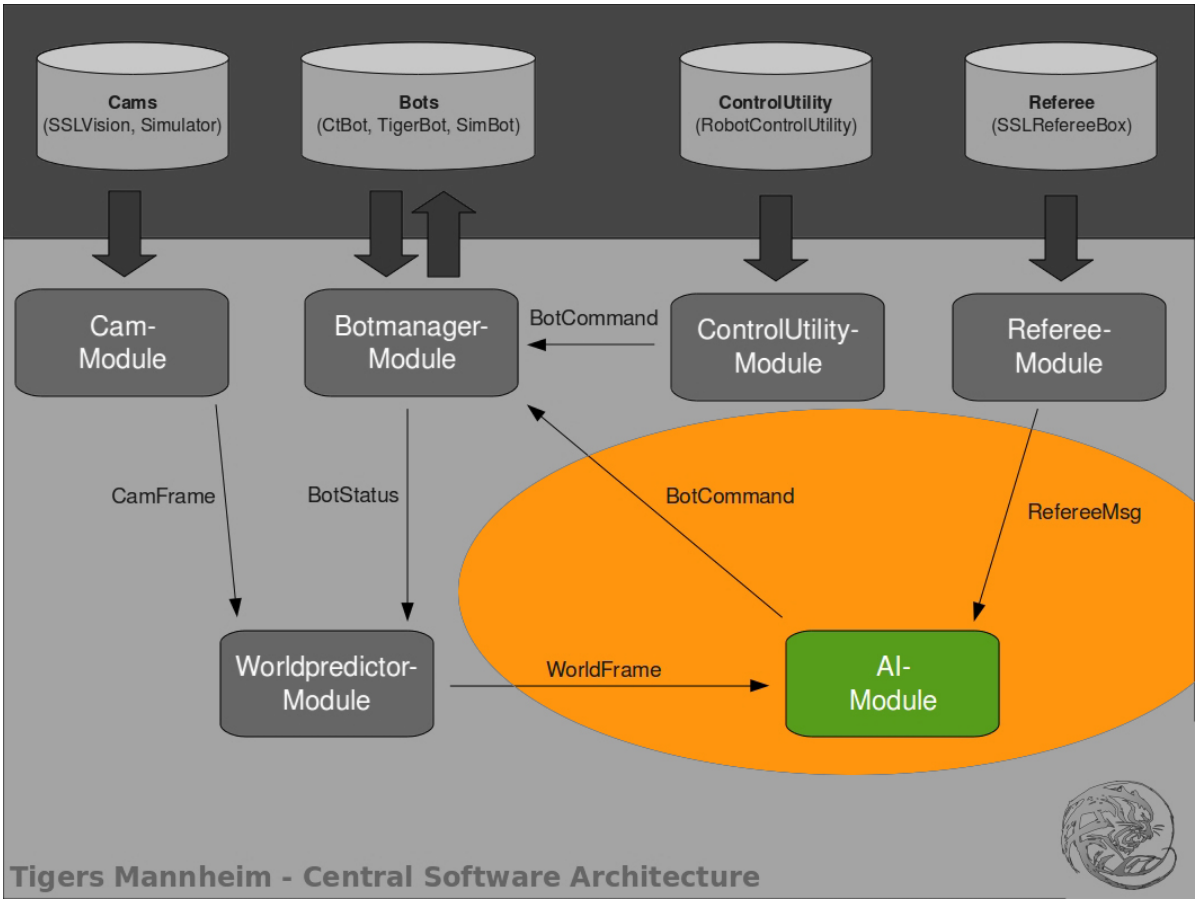

Figure 3.1: Central Software Architecture

Via two cameras and the open software SSL Vision, the positions of the ball and robots, as well as geometry-data of the field, are transmitted to each teams central software. Sumatra receives these data frames and forwards it to its first module "Worldpredictor", which analyses these data, filters noise and uses motion models to provide exact position data. Also, the transmissions and systems latency is compensated here by predicting a future state of the game using probabilistic algorithms. Subsequently, the AI is triggered with that information. Based upon this data, a purpose and corresponding commands for each single bot is found and sent to the "Botmanager", which converts and transmits them to the robots via radio communication.

3.2 The AI modules

To achieve the goal of generating appropriate actions, the AI is divided into different layers. During the incremental design and redesign of the software, these layers and their interactions have been modified over and over. The current resulting structure is shown in figure 3.2, all inner-AI-information are passed top to bottom in the AIInfoFrame.

To simplify team communication, we decided to give each module an arbitrary name. In the process of name finding, we found many Greek deities' tasks to be very similar to those of the modules, hence the Greek names.

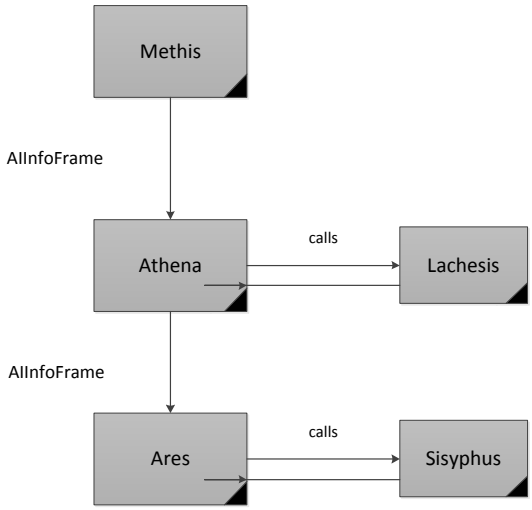

Figure 3.2: AI modules

The first module in line is "Methis". Its job is to watch and analyze the world(-frame), and gather various information that can be used in the following modules, such as identifying menacing areas or the ball possession. Then, "Athena" provides the strategy

by evaluating different plays. Pursuing the most rewarding play, it coordinates the robots to fulfill its unique role. Since the real "intelligence" is located here, Athena is the beating heart of the AI module. Also, the coordination of shooting and passing behaviour, which is the subject of this paper, occurs here.

When Athena decides for a new play to start, "Lachesis" is called to distribute the roles onto the robots. This allocation will not change throughout the execution of the play. Since one play is preferably pursued across multiple consecutive iterations, Lachesis is not always called.

To guide the real robots on the field, "Ares" collects the movement-commands and transmits them to the Botmanager, which will take care of sending the data to the robots. To find complex routing around moving obstacles, it will call Sisyphus (over and over again)to calculate a path via modern pathfinding-algorithms.

3.3 A tale of great plays, mighty roles, tricky conditions and impressive skills

In a hypothetical match without robots, the superior strategy and tactics will win over the inferior. Since "Strategy requires thought, tactics require observation" (Max Euwe), we studied many team description papers of leading RoboCup participants[1] very closely. The common thread was a segmentation of the matter into different layers. Regarding this mean, the ai team defined plays, roles, conditions and skills.

A single "play" incorporates the layer of strategy. It represents an idea or a plan of what we want to do in this particular situation if everything works well. For instance, a passing play will somehow transfer the ball from the possession of on robot into the possession of another robot. In our definition of plays, one implemented play has a fixed set of protagonists. These are embodied by one real robot on the field and by a virtual "role" in the AI. Every single role has a definite task, which it tries to fulfill. So in the example above, a single pass play, there would be two roles: a pass sender and a pass receiver.

Each role only knows its own task(s), and can therefor be combined with any other role, since there are no co-dependencies between roles. The combination of roles, the handling of all communication and strategy is implemented in the play.

The subtasks of roles, like finding a good position or facing the ball, are described by different conditions. A condition is a state, which is either true or false. The role works towards satisfying all its conditions. Just like roles, these conditions are independent of roles or other conditions and may be combined arbitrary.

[1] such as CMDragons (USA), ER-Force (Germany) and Skuba (Thailand)

Inside the AI, the lowest layer is "skill". Whenever an explicit action needs to be performed by the bot, the respective skill is used. So if a robot needs to face the ball it will use a rotation skill to do so.

Skill

*Skill*s directly represent basic robot capabilities. These include moving, spinning and operating the kicker or the dribbling bar. *Skill*s inherit from a generic *ASkill* class, which handles the execution. However, a *Skill* has to implement the calcActions() method, where so called *Command*s are created, collected and prepared for the transmission to the actual robot by the *BotCenter*. *Command*s are the most basic instruction set for a robot and represent mere the devices of the robots such as the wheel-motors, the dribbling motor or the kicking device. Each *Skill* may use a subset of them or even combine two or more. *Command*s are then executed within the robots hardware-firmware. Moreover, *Command*s expect that all parameters, such as move-vectors, are in a robot-local coordinate system whereas *Skill*s are parametrized with global coordinates. Thus, a *Skill* needs to perform the conversion. See chapter 4.2 for an example.

Additionally there is the possibility to combine two or more *Skills*. A *CombinedSkill* is a *Skill* itself but maintains a list of its sub-*Skill*s. When a *CombinedSkill* is asked to calculate its actions, it calls the calcActions() method and collects the *Command*s of all its sub-*Skill*s. Since only one *Skill* is executed at a time, only *CombinedSkill*s allow multiple *Skill*s to send *Command*s to a bot in the same cycle. A combination of *CompinedSkill*s is also possible. This suffice for all needs.

Condition

All conditions extend the same superclass: *ACondition*. In order to keep the combination of different conditions versatile, a single condition will always represent an atomic (not further dividable) status, such as "robot looks at target" or "robot stands near destination". In each frame, the implemented conditions checkCondition()-method will be called and the state of the condition determined. Attributes, such as the target position, may be updated via implemented update()-methods of an implemented condition. An example of the outline of such an implemented condition is shown in listing 3.1 on page 15.

Role

Inheriting the superclass ARole, each implementation has to fill the methods update() and calculateSkills(). In the first, all logic towards fulfilling its purpose has to be implemented.

```java

public class DestinationCon extends ACondition {

    public DestinationCon(float tolerance)
    { [...] }

    @Override
    public boolean doCheckCondition(WorldFrame worldFrame,int botID)
    {
        final TrackedTigerBot bot = worldFrame.tigerBots.get(botID);

        float dist = distancePP(bot.pos, destination);

        if (dist < tolerance)
        {
            return true;
        }
        return false;
    }

    [...]
    public void updateDestination(IVector2 newDestination)
    {
        this.destination.set(newDestination);
        resetCache();
    }

    [...]
}
```

Listing 3.1: The Destination Condition

In general, the role will update its unique combination of conditions with relevant data depending on the conditions current state and the world-information. The latter holds the generation of skills, that will make the robot fulfill its purpose, based on the states of the updated conditions.

Play

A Play holds a definite set of roles and generally contains all role-to-role communication and coordination. While it will never interfere with a roles procedure of "how" to do something, it will very much tell its protagonists "when" to do something and where other protagonists are about to go. The Play itself holds a beforeUpdate() and a afterUpdate(), in between the roles are updated automatically. The beforeUpdate() will be commonly used to determine the current state of the play and to communicate the (designated) position to (co-)dependant roles. In the afterUpdate() (that's when the roles conditions are already updated) the play will check, based on the individual conditions, whether or not an action, such as a pass or a shot at the goal, will be forced. See figure 3.3. These decisions will then affect the outcome of the roles' calculateSkills(). A Play may consist of one to five roles, several plays may be executed in parallel, making combination of multiple Plays, such as a OffensePlay and a DefensePlay, possible.

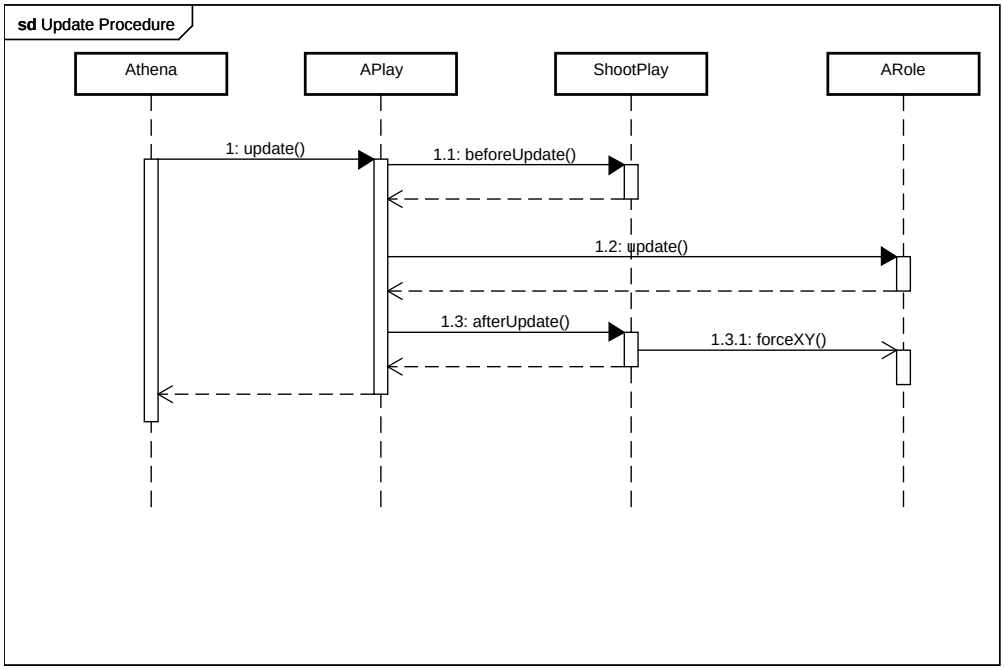

Figure 3.3: How plays get updated

4 Aiming

Aiming is a crucial ability of mobile robots to perform precise shoots and passes. Aiming is generally the process of moving the robot with the objective that it "looks" towards the target with the ball located in front of it. See Figure 4.1 for a visual overview. This is necessary since it is only possible for the robot to shoot straight ahead (apart from chipping the ball, which is not covered here).

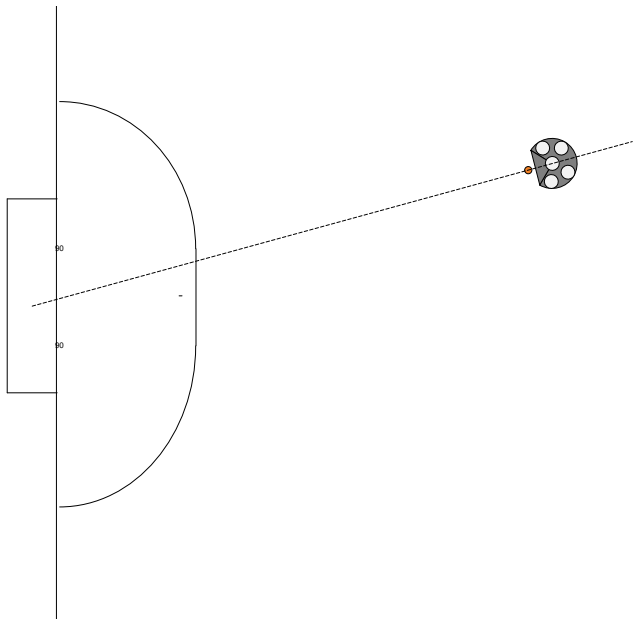

Figure 4.1: Robot and ball on a straight line indicates completed aiming

As of now the aiming algorithm assumes that the robot controls (eg. dribbles or on a set piece situation) the ball which implies that the ball rests on a constant location. See chapter 7 for ideas on aiming with a moving ball.

Unfortunately there are a few limitations in what movement the robot can perform while handling the ball which led to a relative complex movement pattern to achieve aiming. From an in-depth analysis of public video footage showing other SSL teams competing in a soccer match we know that it is impossible for a robot to neither move sideways nor spinning while it is dribbling the ball (Figure 4.2)[3]. If it would do so, the ball cannot be fully controlled and may just slip away, which, of course, needs to be prevented under all circumstances.

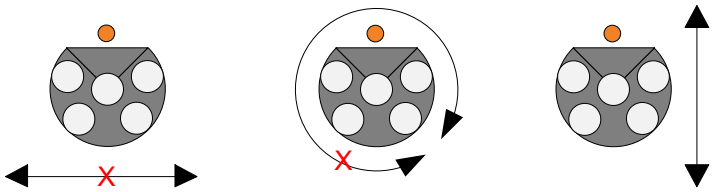

Figure 4.2: basic robot movements, X indicates impossible movement

That led to the assumption that the robot needs to move on a circular trajectory while always "facing" the ball at the same time. Thus, a complex movement that combines the circular movement of the robot with an appropriate spin is executed (Figure 4.3).

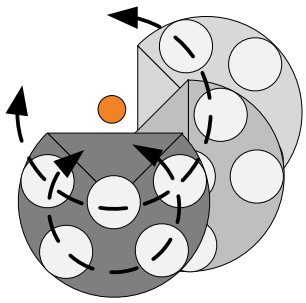

Figure 4.3: combine circular movement with spin: aiming

Repeating development patterns in other AI parts, aiming is implemented with a *Condition* that checks if aiming is already done and a *Skill* which creates the actual movement commands.

4.1 AimingCondition

The *AimingCondition* needs to check three separate constraints. The aiming procedure is complete, only if all of them are met.

1. is the robot "looking" at the ball
2. is the robot on the straight line defined by ball and target

3. is the robot "behind" the ball

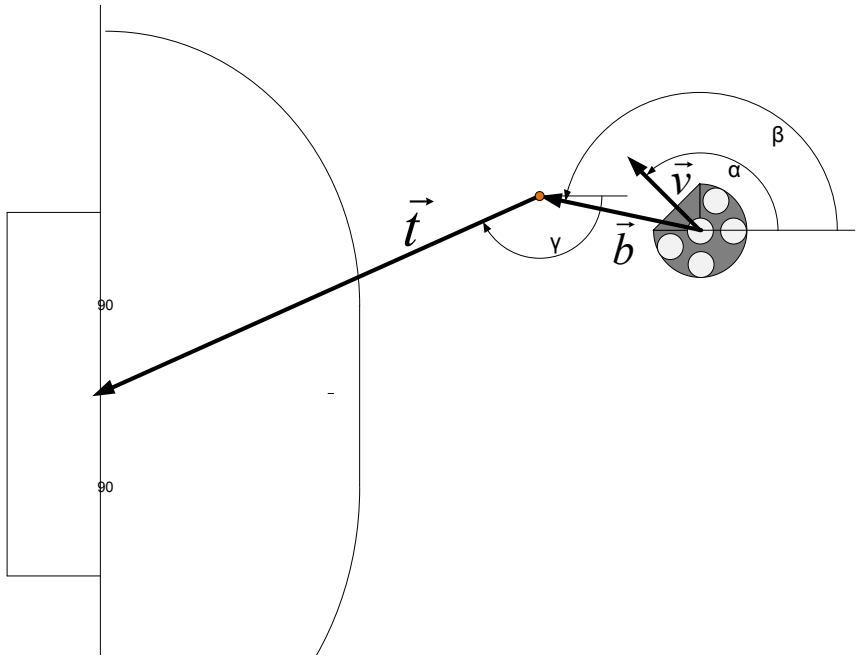

Figure 4.4: aiming constraints

However, there is no distance between robot and ball defined since this can vary. Think of a set piece situation where the robot must not touch the ball versus an in-game aiming situation with active dribbling. Thus, the distance is not tested in the *AimingCondition*.

Constraint #1 is delegated to the *LookAtCondition* that is extended by the *AimingCondition*. Here, the current orientation vector of the robot ($\vec{v}$) is compared with the vector between the robot and a target point (in this case this is the ball ($\vec{b}$)). The constraint is met, if both vectors point in the same direction with respect of some tolerance. (see figure 4.4)

In fact, only the angle between the x-axis and the respective vector is compared. That angle is per definition in $(-pi, pi]$ interval.

$$\text{tolerance} > |\alpha - \beta| \tag{4.1}$$

#2 and #3 are implemented directly in the *AimingCondition*. Again, the algorithm compares two vectors: the vector between robot and ball ($\vec{b}$) with the vector between ball and target ($\vec{t}$).

$$\text{tolerance} > |\beta - \gamma| \tag{4.2}$$

Generally, the aiming process is completed, when

$$\alpha = \beta = \gamma \tag{4.3}$$

is. But because of imprecise vision data and positioning capabilities a tolerance is introduced.

4.2 AimingSkill

The purpose of the *Aim Skill* is to calculate movement commands for the robot based on the current situation.

As seen above, the aiming process consist of a circular movement of the robot combined with a spin. Thus the *Aim Skill* itself is in fact a *CombinedSkill* of the following sub-*Skills* (see the UML class diagram in figure 4.5):

- the *LookAt Skill*, which enables the robot to rotate towards a certain target point
- the *MoveInCircle Skill*, which lets the robot move on a circular trajectory

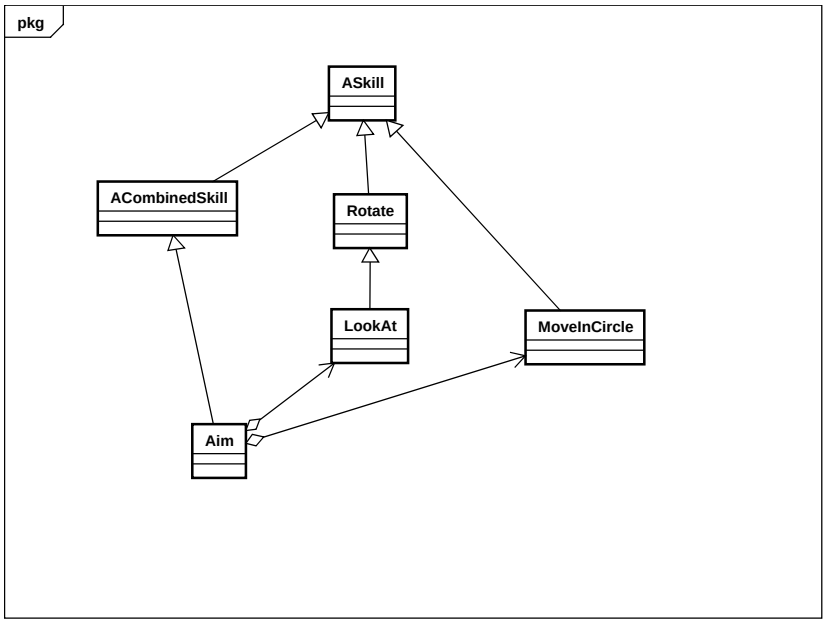

Figure 4.5: class diagram: *Skill*

The only code needed to realize the Aim Skill shows listing 4.1. This is that simple since *Aim* is *CombinedSkill* which is actually more like a container. The real work is done by the contained sub-*Skills*.

```
/**
 * @param ballCoords current ball position
 * @param radius of the circular movement
 * @param aimingTarget target the bot should aim at
 */
public Aim(Vector2 ballCoords, float radius, Vector2 aimingTarget)
{
    super(100, ECompletionType.ALL, ESkill.AIM);
    addSkill(new MoveInCircle(ballCoords, radius, AIMath.
        angleBetweenXAxisAndLine(aimingTarget, ballCoords)));
    addSkill(new LookAt(ballCoords));
}
```

Listing 4.1: the Aim Skill

With the combination and parallel execution of the two, the aiming movement pattern is achieved. The architecture of *CombinedSkill*s in general enables the AI to execute more than one *Skill* in the same cycle without the one knowing the other . This is strongly needed here, since it must be ensured that the robot always "looks" at the ball while orbiting it in order to not lose control.

Both sub-*Skill*s are full-blown *Skill*s in itself and may be used elsewhere.

Again, there is no distance between robot and ball predefined. The desired aiming distance ("radius") needs to be passed in by the role that wants to aim. The role always knows what distance is correct in a certain situation.

4.2.1 LootAt Skill

The *LookAt Skill*, as well as the *Rotate Skill* lets the robot perform a spin (rotation around its own center). The only difference between the two is that the *Rotate Skill* operates with an absolute view angle (designated orientation of the robot) whereas the other takes a target point as parameter. The *LookAt Skill* extends the *Rotate Skill* and uses its capabilities but adds the conversion from the target point to the desired view angle.

Commanding a spin to a robot is as simple as sending a *TigerMotorMove-Command* with the desired angular velocity as parameter (Listing 4.2).

```
//cmds is a list of commands
cmds.add(new TigerMotorMove(currentTurnSpeed));
```

Listing 4.2: The TigerMotorMove

The crux is now to determine the appropriate angular velocity so that the desired orientation of the robot is reached in the shortest possible time. Additionally, the robot may not overshoot and start oscillating.

The *Rotate Skill* uses a slightly modified PID-controller to compute the appropriate angular velocity in a given cycle. The controller starts with calculating the current error x_d:

$$x_d = \text{designated orientation} - \text{current orientation} \tag{4.4}$$

Since the error is still an angle value, it now needs to be normalized to ensure it is still in $(-pi, pi]$ interval. This also ensures that the robot will always rotate in the right (shortest) direction.

The output, which will then be set as angular velocity, is the sum of the proportional, integral and derivative portion:

$$\text{output} = k_p \cdot x_d + k_i \cdot \int_0^t x_d d\tau + k_d \cdot \frac{d}{dt} x_d; \tag{4.5}$$

Finding good parameters (k_p, k_i, k_d) of the PID-controller is not easy, especially since the characteristics of the robot are unknown. Those parameters are dependent on different environment conditions such as the wheel-ground-friction.

The *Skill* will finally be completed, when the designated orientation is reached (with respect to some tolerance) and the current angular velocity is below a threshold.

The *LookAt Skill* automatically compensates a moving robot as well as a changing target since the designated orientation is recalculated every cycle.

4.2.2 MoveInCircle Skill

The *MovInCircle Skill* enables the robot to move on a circular trajectory. Generally, commanding a move is again as simple as sending a *TigerMotorMove-Command*. The *Command* takes a vector as parameter. The direction of the vector determines the direction of the move, the length corresponds to the velocity.

The difficulty here is, that with the basic *TigerMotorMove* only straight moves are possible. Curved trajectories need to be approximated with a reasonable amount of straight parts. This is done on *Skill* level. Since the aiming capability requires only circular movements, the development of a generic "move-on-curve" ability which might be based upon splines, was delayed.

The *Skill* uses the following algorithm to approximate a circular trajectory. The goal is in each cycle to calculate a point on the circle (B') towards the robot (B) shall move. The circle is defined by the center (C) and the radius ($|\vec{cb}|$), which are both input parameters. See Figure 4.6 for an overview.

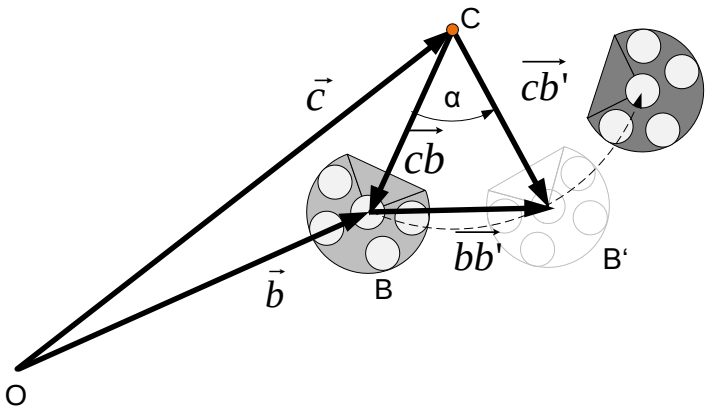

Figure 4.6: How to move on a circular trajectory

Another important input parameter is the target angle from which the arc the bot shall travel is calculated. α defines the current robot position (in reference to the center and the x-axis) whereas the target position is defined by β. (See figure 4.7)

The arc can now be calculated with

$$\text{arc} = \beta - \alpha \tag{4.6}$$

A positive value indicates a counter-clockwise movement and vice versa. The robot will always automatically take the shorter distance.

The next point on the circle the bot shall move to is now calculated with a scaling and rotation of the vector $\vec{cb}$. The vector is scaled to the constant passed in aiming distance to ensure that the robot keeps that distance. The new vector is calculated as follows (Figure 4.6):

$$\vec{cb'} = R_\alpha \cdot \vec{cb} \tag{4.7}$$

where

$$R_z, \alpha = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \tag{4.8}$$

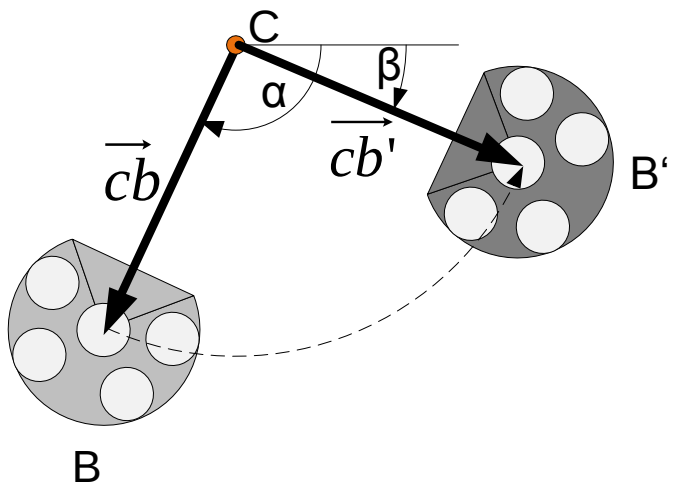

Figure 4.7: robot and target position

The rotation angle α determines the smoothness of the approximation. Of course it is also dependent on the radius, but since the skill is currently only used for aiming purpose, the angle is adapted to that scenario. The default value with good results is $\pi/18 = 10\,\text{deg}$. If the arc is smaller than that default step size, the rotation is performed with $\alpha = \text{arc}$.

Now everything is known to calculate the new move vector:

$$\vec{bb'} = \vec{cb'} - \vec{cb} \tag{4.9}$$

Now, that vector is still in global coordinates and needs to be transformed. The final move vector ($\vec{m}$) that gets commanded to the robot is obtained from a simple rotation:

$$\vec{m'} = R_\beta * \vec{m}\beta = \frac{\pi}{2} - \alpha \tag{4.10}$$

See figure 4.8 for an overview, where $\vec{m}$ is the global move vector from above and α is the current robot orientation. At first glance this is a bit mind-warping. To understand consider the following:

- A vector $\vec{m'} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ would cause the robot to move straight forward.

- A vector $\vec{m'} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ would cause the robot to move sideways to the right.

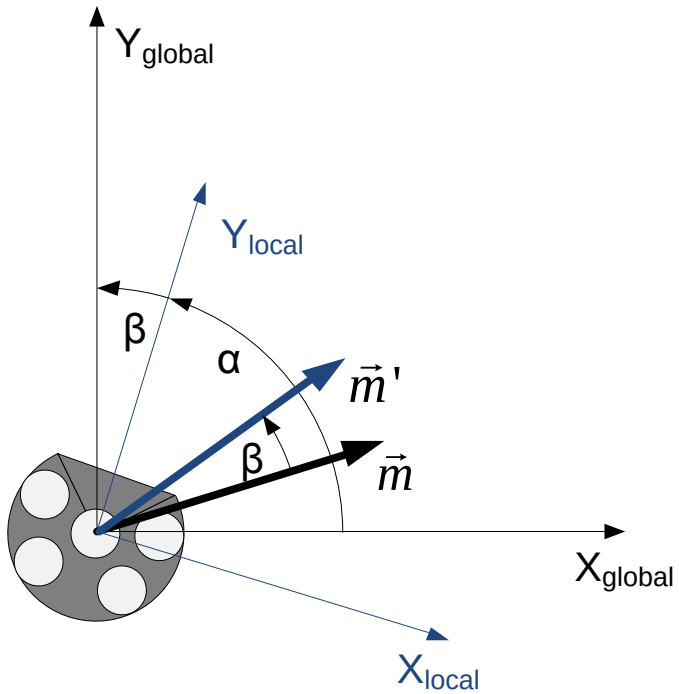

Figure 4.8: vector conversion from global to local coordinates

The final operation that needs to take place before the move vector can be transferred to the robot is the application of the desired velocity. Velocity is represented by the length of the move vector. Thus a simple scaling of the vector applies the velocity.

The velocity is a linear function of the length of the arc the bot shall travel, where the slope is determined by the maximum allowed velocity and the distance it takes to decelerate. Both parameters are variable and dependent on the environment (eg. friction).

5 Shooting

To win a game of soccer one needs to score goals. Therefore, and for the fact that a shot at a precise target involves every kind of action our robot is capable of and can be implemented as a one-bot play without any need for role-to-role communication, it was implemented first.

5.1 The Shooter role

Since shooting at a target is an atomic action, it is implemented as a single class in the layer roles. The states this role tries to achieve is to

1. reach a position near the ball to enable effective aiming
2. reaching a position and an orientation, that will make a shot hit the target position (aiming)

From these tasks, the two conditions DestinationCondition and AimingCondition derive and are part of the Shooter-role. The Destination-Condition is used to transport the robot close to the ball. Therefore, it has been initialized with a great tolerance towards its accuracy and always targets the current ball position, so that the robot will travel towards the ball, but stop moving when it's in close range of it (coming from any side), so it will not accidentally knock the ball away. Also, the position the robot tries to reach is exclusively stored in the Destination-Condition. So if that position needs to communicated anywhere, it can always be found in this container. However this feature is not used in the *Shooter*, since there are no dependencies to another role in this scenario.

Also, the Aiming-Condition must be updated in every frame, or the logic will try to match an outdated target. In this case, the current position of the ball must be forwarded to the condition. Additionally, a change in the targets position can be set, for example when the Shooter needs to target another portion of the goal because the opponents robots block the way to the old target. Regarding the aiming, there are some difficulties which hamper a "clean shot". The accuracy of a shot, especially on long distances, is limited by the fact that the cameras resolution is mediocre at best, giving us uncertain positional and orientational feedback. Also, minimal corrections are hard to accomplish

and the difference between the planned, virtual, idealistic movements and the executed, real actions makes it hard for the system to adjust perfectly.

In the calculateSkills()-method, Skills will be generated according to the state of the conditions. Firstly, the Destination-Condition will be checked. Remember that, due to the intentionally low accuracy, the robot only needs to be in medium range of the ball. If, however, the robot is farther away, Sisyphus will be called to generate a path towards the ball, and these are returned as a chain of Move-Skills, that will maneuver the bot to its destination.

If the robots is near enough (Destination-Condition is true), the state of the Aiming-Condition will be checked. If its aiming is not completed yet, Aim-Skills will be generated, else the robot will idle and wait for further commands.

5.2 The "One Bot Shooter"-Play

As might have been noticed, the Shooter doesn't actually shoot, yet. This is because we consider the shot itself to be of the interacting type, that needs to be timed by a coordinating instance, which is the Play. As being stated earlier, the shooting process itself is atomic. Getting to the ball efficiently is not part of it, as it describes another, atomic action. Therefore, another Role has been written; BallGetter. This has been designed and implemented by other members of the AI team, as well as being documented in another paper. For this review it shall be known, that the BallGetter takes velocity, acceleration, viewing direction of the robot and the ball, as well as the designated target into consideration to approach the ball efficiently[1]. The *ShootingPlay* will process the scheme displayed in figure 5.1.

The OneBotShooter-Play has two roles and one robot, and decides in every frame, whether or not the robot is close enough to the ball already, thus ruling either the BallGetter or the Shooter to be the active Role. This makes it a state machine, and it is implemented accordingly. In the beforeUpdate(), the Play decides upon the current state. The Play starts in the BallGetting-state, the BallGetter is active. If the Robot enters the immediate surrounding area of the ball, it will switch to the Shooter-state. If, on the other hand, it once has been in this close range, but the distance to the ball surpasses another, greater value, it will switch back into the BallGetting-state, since it considers the robot to be to far away to use the simple approaching-algorithm implemented in the Shooter. If the BallGetter is the active Role, the beforeUpdate() will only forward the target, at which the Shooter is going to aim, so the BallGetter can include this information. Since the BallGetter will always move towards the ball, the designated

[1]For further information see the German paper "Entwicklung von Algorithmen zur Balleroberung im Rahmen des RoboCup Projekts TIGERS Mannheim an der DHBW" from F. Schwanz and T. Kessler

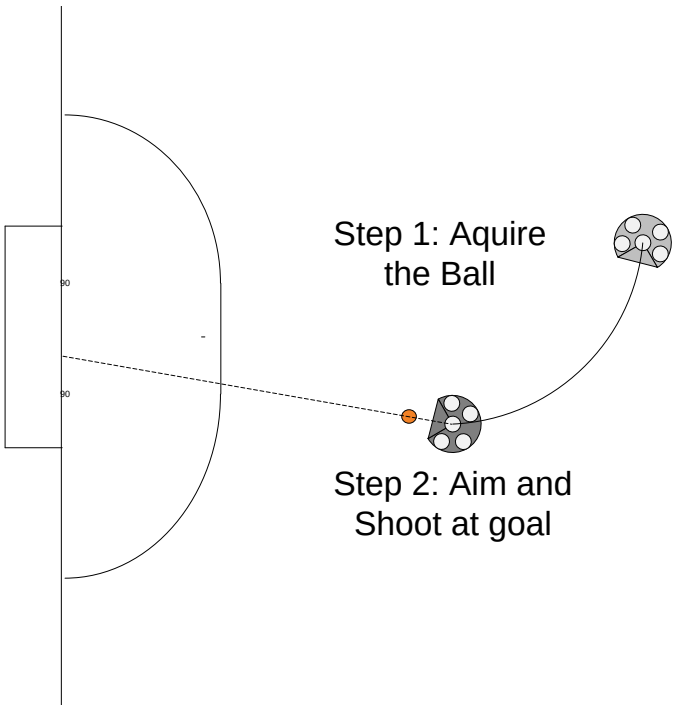

Figure 5.1: Shooting Scheme

Position of this Role needs no update. In contrary, the Shooter will need an update to its designated position, which is the ball in this Play.

While the BallGetter requires no afterUpdate(), the Shooter's afterUpdate() is vital, since here the coordination takes place. The Play confirms, that the Shooter is ready to fire. In this spot it's possible to direct multiple co-dependant Roles. Since here, we only have one Role, no further checking takes place and the command to shoot is given to the Shooter. Instead of performing the usual Move/Aiming Skills, the robot will now move forward and kick the ball with maximum strength towards the target, and if everything works out, then its one more goal for the TIGERs.

6 Passing

The next step that needs to be taken to develop an effective offensive playing is a passing algorithm. This will also be the most versatile action that our team will perform, since it must be used in various situations in a most reliable fashion in order to make a victory achievable at all. While the passing serves as the second-to-basic way to move the ball from one spot to another (the most basic way being dribbling it to the destination), it involves a great deal of indispensable advantages, which all come down to time and space efficiency. This means, the passing needs much less time and, due to the balls unlike smaller diameter (compared to a dribbling robot), much less space to move through any kind of obstacle field. These advantages come down to, but are in no way limited to the following points:

- transport the ball to another attacker, that is in a better position to shoot a goal
- transport the ball away from a dangerous position, i.e. when a shooter is closely attacked by multiple opponent defenders
- to shift the game to the other side of the field (and possibly back close after) to achieve an advantage in positioning
- to transport the ball around opponent robots in a very fast fashion (read: counter-attack after blocking an opponent push)
- to counter and surprise the opponents plotting of the game, which will result in a planned (thus ongoing) tactic on our side, but a change of game plan on the opponents side, giving a time and positioning advantage

Without an effective and reliable passing a success in soccer is sheer unthinkable. Although there are many problematic issues to tackle in this department, namely accuracy, receiving, timing and shooting force, the (offence) ai developers set sail in this direction next, since it's that essential.

6.1 The Roles in Passing

In order to achieve a pass two robots and the ball are needed. The robots obviously fill the part of *PassSender* and *PassReceiver*. On many instances, the *PassSender* is

very similar to the *Shooter*. It inherits a *DestinationCondition* and an *AimingCondition*, that will handle its positioning and aiming, just as they did for the *Shooter*. The *DestinationCondition* will be updated every frame and will determine, if the robot is within the vicinity of the ball. If not so, the *calculateAction()* will produce Skills that transport the robot closer to the ball. If, however, the robot is close to the ball, an *Aim*-Skill will be generated to put the robot behind the ball and in proper orientation to its target. In order for this to happen, the *PassSender* must be consistently updated with its Receivers designated position.

Since the *PassReceiver* is intended to shoot at the goal in this scenario, its positioning is by far more complex. It has to match multiple *VisibleConditions* and still find a rewarding area to stand in. For raw positioning, the *PassReceiver* is given a rectangle to stand in. Within this rectangle the robot needs to find a point, where the *PassSender* and the potential target are visible (in order of priority). Furthermore, the Receiver needs to face the Sender, so it can capture the ball properly. This results in the following conditions:

- VisibleCondition: the *PassSender* must be visible
- DestinationCondition: the calculated position must be reached
- LookAtCondition: the *PassSender* must be faced
- VisibleCondition: the target shall be visible

Only if the first three *Conditions* are true, the robot is ready to receive the ball. If the fourth *Condition* is also true, an optimum is achieved (as far as calculation go here). This results from the consideration, that the robot should in all instances be ready to take the ball, even if a perfect spot is not found yet (resulting from good opponent play), in case the Sender gets attacked and needs to put the ball to safety. The *PassReceiver* must be consistently updated by the designated position of its sender.

6.2 The PassingPlay

The structure of the *AI* requires a *PassingPlay* to coordinate the actions. While this one is very similar in structure to the *ShootingPlay*, it has its very own complexity. To determine the different phases of the play, we must explore the tasks closely. At first, the Sender needs to reach the ball, the Receiver needs to find a good location to be a rewarding recipient. Then, the Sender needs to aim at that location, while the Receiver continues to hold it, or finds a better one. When both robots are ready, the Sender kicks the ball in a straight line. After that, the Receiver needs to capture the ball, the Sender will find a good location to position itself, preparing subsequent plays. This leads to three stages of *Passing* and to the following roles for the robots:

1. GettingPlay: getting hold of the ball and finding position;
   - Sender: *BallGetter*; Receiver: *PassReceiver*
2. PassingPlay: Aiming at the Receiver, matching role conditions, kicking the ball
   - Sender: *PassSender*; Receiver: *PassReceiver*
3. ReceivingPlay: the Receiver captures the ball, the Sender finds a good location and positions itself for further actions
   - Sender: *PassReceiver*; Receiver: *BallGetter*

The *PassingPlay* will follow the scheme presented in figure 6.1.

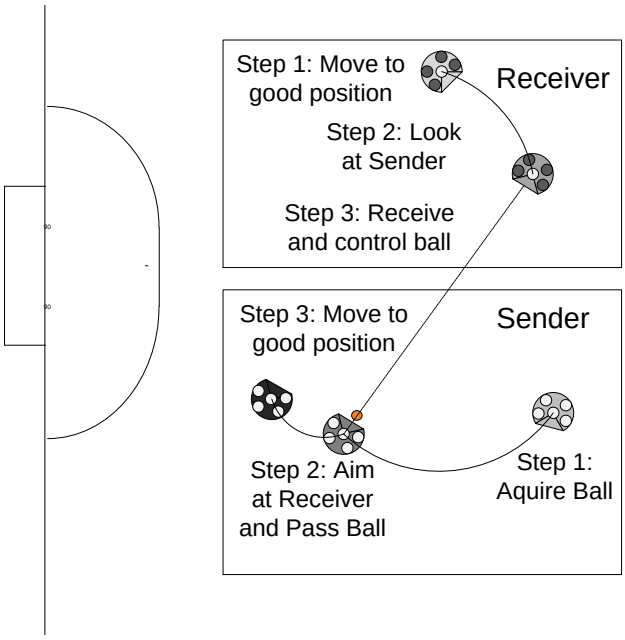

Figure 6.1: Passing Scheme

Each *PassingPlay* starts in the same stage; the GettingPlay. To initialize the Play properly, the Receiver must be given the rectangle mentioned above.

In the Plays beforeUpdate() the current stage is being checked. If the Play is in the getting-stage, and the sender is close enough to the ball, the play will switch to the passing-stage and the sender will switch from being a *BallGetter* to being a *PassSender*. The Receiver will stay being a *PassReceiver*. If the *PassSender* then leaves the ball-area again, everything will switch back. If however the *PassSender* has kicked the ball, the Play will switch to the receiving-stage, the *PassSender* becomes a *PassReceiver* (for the sake of holding the line-of-sight with the original receiver, as well as positioning itself

well), the original *PassReceiver* becomes a *BallGetter* to secure the ball. This procedure is also shown in graph 6.2.

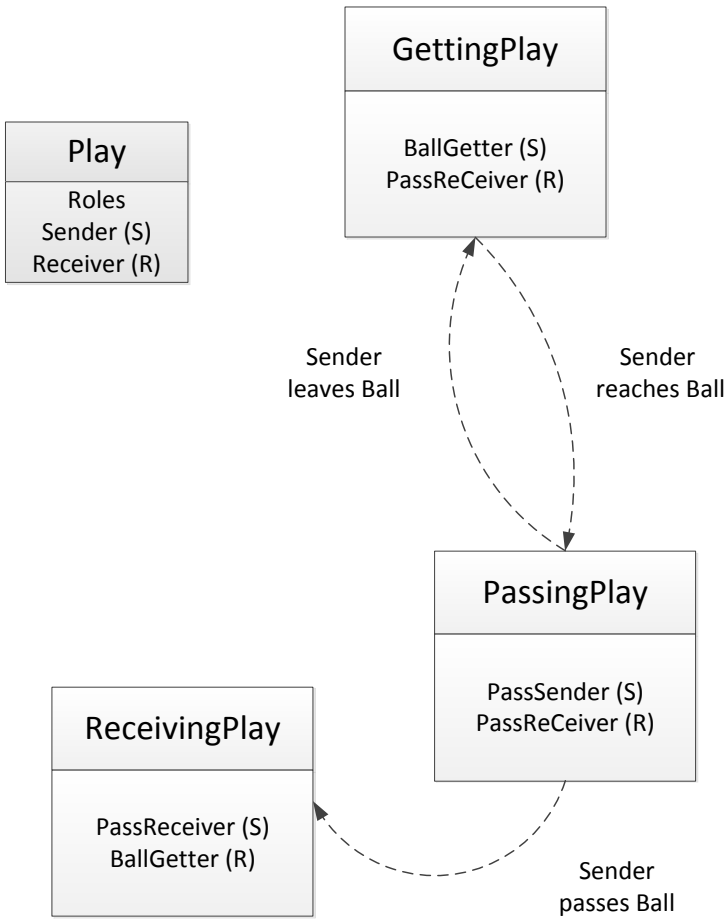

Figure 6.2: *PassingPlay* states

Since the *GettingPlay* is only preparative, not much coordination work is needed. Here, the *BallGetter* will move towards the ball in the usual fashion. Additionally, the Play will consistently give an update on the *PassReceivers* designated position, so that the Getter can approach the ball from the opposite direction to reduce the aiming efforts. The *PassReceiver* will roam its given rectangle-area to find the most rewarding spot to receive a pass at, following above mentioned Conditions and computing the designated position of the *BallGetter*, which is the balls position itself.

In the *PassingPlay*, the Play will furthermore communicate the robots position, so proper orientation can be achieved. In the afterUpdate(), the Play will check if the Conditions mentioned in section 6.1 are met. If so, the Play will command its roles to execute the pass. This flag will impact the roles procedure to compute commands, since its

high-priority will result in making the role regard only this command and forget about any other actions it would have taken. This is an important decision on the *AIs* structure: The Play will identify certain sets of conditions and force its Roles to strictly follow its orders. This resembles a one-way master-slave communication and highly simplifies coordination of protagonists. When the pass-command is given, multiple things happen simultaneously:

- The *PassSender* will move straight forward and kick its straight kicker
- The *PassReceiver* will move backwards to decelerate the ball smoothly
- The state of play will switch to *ReceivingPlay* resulting in
  - The *PassSender* becoming a *PassReciever*
  - The *PassReciever* becoming a *BallGetter*

The *ReceivingPlay* doesn't require any additional coordination by the Play besides forwarding the the position of the original Receiver to the original Sender.

6.3 Shooting force

In the passing scenario the force of the shot is not the maximum force the robot can give, since here we don't need maximum power to overcome the opponents goalkeeper. Here, the priority lies in controlling the ball, so the *BallReciever* needs to stop and control the ball safely. Sadly, the robot cannot handle a ball coming at it at full speed, but we can consider it a fact, that the robot will be able to stop the ball safely at one set arrival-speed. Which value this is exactly stays yet to be determined by trial tests. Let's put this speed at $3m/s$ for the ease of arguing, so the ball needs to be shot in a way, so its speed is $3m/s$ when arriving at the Receiver (Ve). The balls speed with a start speed of v_0 is determined by the following basic equations of motion:

$$s(t) = \frac{1}{2}at^2 + v_0 * t \tag{6.1}$$

and

$$v(t) = at + v_0 \tag{6.2}$$

Unfortunately, we don't know about the time, but we do know about the distance the ball needs to travel. Mathematical conversion will lead to

$$v(s) = \sqrt{v_0^2 + 2as} \tag{6.3}$$

The factor a is a constant, which is dependant on the floors friction. This leaves the distance between the ball and the Receiver the only left over variable, which is exactly what we want. Lastly, we transform the equation to give us the needed start speed v_0:

$$v_0 = \sqrt{v(s)^2 - 2as} \tag{6.4}$$

In our scenario, we already know the arrival speed $v(s)$, which is always our $3m/s$ ($= v_{end}$). This makes the start speed v_0 only dependant on a, which, again, is a constant, and the only left variable s, leaving us with this final equation for our calculations:

$$v_0(s) = \sqrt{v_{end}^2 - 2as} \tag{6.5}$$

This way, we are able to calculate the speed the ball needs at the start for every shot. This enables us to calculate the force of the shot the bots mechanic needs to meet. Also, the fictional arriving speed of $3m/s$ can be increased by letting the Receiver move backwards. While maintaining the same relative arrival speed of $3m/s$ between robot and ball, the total speed of the ball - and therefore the whole shot - increases and let's us act faster and less interceptable, which will boost our competitiveness in RoboCup.

7 Outlook: dynamic aspects

While basic shooting and passing capabilities of the robot are implemented, there are some advanced scenarios left untouched yet. Currently both shooting and passing rely on a resting (= not moving) ball. Additionally the pass sender expects the pass receiver to be steady. Now, in SSL soccer this allows already for good offensive play. But being able to precisely reach a moving pass receiver or shoot a goal while in motion could open great opportunities.

Passing to a moving robot is generally possible. The pass sender is able to react on a change of its target in realtime. Problematic is the timing of the shoot, since the receiver moves while the ball is travelling. A solution might be the following: First a save location that both the receiver and the ball can reach at the same time is calculated. The sender now passes the ball to that location as if it was a steady pass receiver. That calculation is complex, though. Another concern is how the receiver could control the arriving ball. The issue here is, that the ball wont arrive straight on but rather in any possible angle.

A moving pass sender or shooter is likewise complex. Remember that the robot can only move straight forward while dribbling the ball. But it still has to be possible to shoot at a certain target and not only straight forward. This implies, that the robot cannot dribble the ball in such a scenario. However it might be possible, given a rolling ball, to aim and just shoot at the right time. But the coordination effort would be immense. This also involves complex calculation on the correct shoot angle since the present ball momentum has to be taken into account.

This is also closely related to another nice-to-have feature: a direct shoot capability. Imagine a pass receiver that doesn't have the objective to gain control over the ball but will rather shoot the ball again as soon as it touches the ball. This ability can lead to an effective attack on the opponent goal. However, it also requires precise passing and a complex calculation of the best shoot direction.

All of the above is a great challenge for the engineers of team TIGERS.

8 Results

In between the development cycles and of course afterwards, excessive testing was executed with promising results. All three important scenarios (aiming, shooting and passing) were tested and validated both in a simulated and a real world environment. A bottom-up strategy was pursued in that first the skills were checked, then conditions, roles and last but not least complete plays.

However, testing was not based on statistics but mere on "try and error". That led to short development cycles.

By the time all relevant scenarios ran reasonably well in the simulation, tests proceeded in the real environment. Here, a similar procedure was carried out.

The results with real robots were comparable to those experienced in the simulation but with greater variations in terms of precision.

First part of the test series was the *Aim Skill*, which is the base of all other shot-related behaviours. The *Skill* has been successfully developed. It is now possible for a robot to aim at any target, though this might take some time. There is a trade-off between precision and speed. The *Aim Skill* might now be incorporated in any *Role* that needs to aim at some target. Moreover, it is simple to use and every AI developer could easily integrate it. Like all *Skills*, *Aim* can be combined with dribbling. Some smaller issues remain here, where the biggest is precision especially with real robots.

While the robot was able to shoot at the goal and score points even against an AI-driven goalie, the time it took to aim and the general precision need to be improved. This partly results from the behaviour of the *Aim Skill*, but other suboptimal routines play their part as well. Exemplary, the final approach right before the shot is fired is trusted to be "straight", but in fact not controlled by PID, further increasing any aiming error in its way forward. Yet, the communication between *Role* and *Play* is very satisfying and ready to be adapted by further *Plays*.

Similarly to the shooting, the passing has been tested successfully, including slight imprecisions. The detection and taking of a proper position works fast, accurate and reliable. In this particular scenario the communication between the *Play* and two co-dependant *Roles* worked splendidly. In this field many possible enhancements are identified, such as proper reaction to opponents actions and an improved reception of the ball.

In summary, the design and implementation of the *Skills*, *Roles* and *Plays* were successful. The exact values for tolerances, fractions and PID have to be found in further, extensive testing, maybe using and automated tool. Based on the findings on the implemented layers, other *Plays* can be implemented to widen then spectrum of possible moves. With the working scenarios, the qualification for RoboCup 2011 has been initiated.

- testprozedere - sim, szenarien, aiming in schuss+ pass,einzelskill/condition/playtesting - keine statistiken, sondern rapid prototyping, schnelle entwicklungs/test-zyklen - test in realer umgebung - wiederholung der tests, weniger zeit, anpassung an neue aspekte, variierende ergebnisse

- aiming skill erfolgreich entwickelt + getestet - möglich roboter zielen zu lassen, ausrichtung dauert aber funktioniert - in alle roles portierbar, für alle KI-entwickler nutzbar - mit dribble-skill kombinierbar - problem: ball weggestoßen (sim + real) - manchmal unpräzise (real) - toleranzen gefunden (präzision vs zeitaufwand), aber verbesserungen möglich

- shoot erfolgreich entwickelt + getestet - gezielter schuss auf target möglich, geschwindigkeit unbefriedigend - präzision steigerbar (wegen movestraight -¿ kein regelung) - play-rolle kommunikation funktioniert ausgezeichnet - tore (auch gegen torwart) erzielt

- passing erfolgreich entwickelt + getestet - pässe möglich, positionsfindung schnell und zuverlässig - einschränkungen siehe schuss - präzisionsproblem vervielfacht, da kleineres ziel (dribbelwalze) getroffen werden muss - abstimmungen bei der ballannahme nötig - ansätze zur reaktion auf gegnerbewegung exestieren, müssen verbessert werden

- generell: implementierung skill/roles/play erfolgreich war, aber toleranzen/pid/fraction etc ausgiebiger gestestet werden müssen, eingestellt werden müssen, zusammenspiel der konfigurationen beachten - entwicklungen führen zu nutzbaren ergebnissen - entwicklung komplexer spielzüge möglich, bewerbung für RoboCup 2011 eingeleitet

Bibliography

[1] F. Schwanz, T. K.: *Entwicklung von Algorithmen zur Balleroberung im Rahmen des RoboCup Projekts TIGERS Mannheim an der DHBW*. 2010

[2] RoboCup: *Official RoboCup Website*. Version: 2010. `http://www.robocup.org/`, Abruf: 04/12/10

[3] RoboCup: *Small Size Robot League*. Version: 2010. `http://small-size.informatik.uni-bremen.de/`, Abruf: 22/11/10