TIGERS Mannheim

Artificial Intelligence - Overall Documentation

Christian König
Daniel Waigand
Florian Schwanz
Gunther Berthold
Malte Mauelshagen
Tobias Kessler

Mannheim, August 4th, 2011

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Tutor:
Prof. Dr. Jochem Poller

Contents

List of Figures **5**

List of Listings **6**

1 Introduction **7**

2 AI Structure **8**
2.1 AI Submodules . . . 8
2.2 Agent . . . 8

3 Pandora **10**
3.1 Purpose of Pandora . . . 10
3.2 Naming of Pandora . . . 10
3.3 Plays . . . 12
3.3.1 Playbook (List of Plays) . . . 12
3.3.1.1 List of Defense-Plays . . . 12
3.3.1.2 List of Offense-Plays . . . 12
3.3.1.3 List of Standard-Plays . . . 13
3.3.2 APlay . . . 13
3.4 Roles . . . 14
3.4.1 List of Roles . . . 14
3.4.1.1 List of Defense-Roles . . . 14
3.4.1.2 List of Offense-Roles . . . 14
3.4.1.3 List of Standard-Roles . . . 15
3.4.1.4 List of other Roles . . . 15
3.5 Conditions . . . 15
3.5.1 List of Conditions . . . 15
3.6 Criteria . . . 16
3.6.1 List of Criteria . . . 16

4 Skills and Skillsystem **17**

5 Precalculator: Metis **18**
5.1 Purpose of Metis . . . 18
5.2 Naming of Metis . . . 18
5.3 Implementation of Metis . . . 20
5.4 Calculators . . . 20
5.4.1 Defense Calculators . . . 20
5.4.2 Offense Calculators . . . 21
5.4.3 Other Calculators . . . 21
5.4.4 Example-Calculator in Detail . . . 21

6 PlayFinder: Athena **23**
6.1 Purpose of Athena . . . 23
6.2 Naming of Athena . . . 23
6.3 PlayableScore . . . 25
6.4 Implementation of Athena . . . 25
6.4.1 Athena process() . . . 25
6.4.2 Play-factory . . . 27
6.5 PlayFinder . . . 28
6.5.1 Interface IPlayFinder . . . 28
6.5.2 MatchPlayFinder . . . 28

7 RoleAssigner: Lachesis **30**
7.1 Purpose of Lachesis . . . 30
7.2 Naming of Lachesis . . . 30
7.3 Implementation of Lachesis . . . 32
7.3.1 Interface IRoleAssigner . . . 32
7.3.2 OptimizedRoleAssigner . . . 33
7.3.3 PermutationGenerator . . . 33

8 RoleExecuter: Ares **34**
8.1 Purpose of Ares . . . 34
8.2 Naming of Ares . . . 34
8.3 Implementation of Ares . . . 36

9 PathFinder: Sisyphus **38**
9.1 Purpose of Sisyphus . . . 38
9.2 Naming of Sisyphus . . . 38
9.3 Implementation of Sisyphus . . . 40
9.4 Global pathfinder: Extended Rapdidly Exploring Random Tree . . . 40
9.4.1 Basic RRT algorithm . . . 40
9.4.1.1 Basic-RRT: chooseTarget() . . . 41
9.4.1.2 Basic-RRT: createRandomNode() . . . 41

9.4.1.3 Basic-RRT: getNearest() . . . 42
9.4.1.4 Basic-RRT: extendTree() . . . 42
9.4.1.5 Basic-RRT: isWayOK() . . . 42
9.4.1.6 Basic-RRT: growTree() . . . 43
9.4.2 Extension 1: Abbreviation . . . 43
9.4.3 Extension 2: Waypoint cache . . . 44
9.4.4 Extension 3: KD-Trees . . . 47
9.4.5 Path-postprocessing . . . 48
9.5 Pathfinding coordinator: Sisyphus . . . 49
9.6 Internal pathfinding-datastructures . . . 49
9.6.1 Node . . . 49
9.6.2 Path (for internal pathfinding-use . . . 49
9.7 Oberserver in Sisyphus . . . 50

10 AIMath **51**
10.1 Basic Functions . . . 51
10.1.1 Conversions . . . 51
10.1.2 Other Basic Functions . . . 51
10.2 Geometrical Functions . . . 52
10.2.1 Trigonometrical Functions . . . 52
10.2.2 Other Geometrical Functions . . . 52
10.3 Other Functions . . . 53

Bibliography **54**

List of Figures

2.1 Application flow of the Agent 9

3.1 Pandora with her box painted by Jules Joseph Lefebvre [16] 11

5.1 An ancient depiction of the goddess Metis [15] 19

6.1 Athena [13] 24

7.1 Three Fates: Clotho, Atropos and Lachesis [9] 31

8.1 Ares [12] 35

9.1 Sisyphus painted by Vecellio Tiziano [17] 39
9.2 The spreading basic RRT algorithm 44
9.3 The spreading RRT algorithm with Waypoint Cache-Extension 45
9.4 Representation of the KD-tree 47

List of Listings

5.1 process() in Metis . . . 20
5.2 Calculator TeamClosestToBall . . . 21
6.1 calcPlayableScore . . . 25
6.2 process() in Athena (part 1) . . . 26
6.3 process() in Athena (part 2) . . . 27
6.4 Play-factory . . . 27
6.5 MatchPlayFinder: method *choosePlaysFree* . . . 29
7.1 Call of the Role-assigner . . . 32
8.1 Roleexecuter Ares . . . 36
8.2 Roleexecuter Ares . . . 36
9.1 Basic-RRT function: chooseTarget() . . . 41
9.2 Basic-RRT function: createRandomNode() . . . 41
9.3 Basic-RRT function: getNearest() . . . 42
9.4 Basic-RRT function: extendTree() . . . 42
9.5 Main-loop in Sisyphus: growTree() . . . 43
9.6 RRT-Extension 1: Abbreviation . . . 44
9.7 RRT-Extension 2: Waypoint cache, method chooseTarget() . . . 44
9.8 RRT-Extension 2: Waypoint cache, method getNodeFromWPC() . . . 46
9.9 RRT-Extension 2: Waypoint cache, method insertNodeToWPC() . . . 46

Chapter 1

Introduction

This seminar paper shall give a widespread overview of the implementation of the artificial intelligence of the RoboCup Small-Size-League Tigers Mannheim.
Due to the fact that the mentioned artificial intelligence is a work in progress this seminar paper describes the software at a fixed moment. This moment is defined by the SVN commit #3753. If a specific part is based on another SVN commit it is stated explicitly.

Chapter 2

AI Structure

2.1 AI Submodules

There are a couple of submodules in the AI of the TIGERS RoboCup team. These are *Metis*, the Precalculator (see section 5), *Athena*, the PlayFinder (see section 6), *Lachesis*, the RoleAssigner (see section 7), *Ares*, the RoleExecuter (see section 8) and *Sisyphus*, the PathFinder (see section 9).
All the modules fulfill a specific purpose that is explained in detail in the particular section.

2.2 Agent

In the AI-module there is the Agent that supervises all submodules. The thread of the Agent is called *Nathan*, according to a very big computer in the Perry-Rhodan-universe.
When starting the software Sumatra, the Agent creates one object of each submodule class.
While running the Agent calls every AI-cycle the *process()*-method in the submodules *Metis*, *Athena* and *Ares*. In this method the logic of these submodules is executed. The remaining submodules are called within one of the others. For example, *Sisyphus* is called within *Ares*.
The described behaviour is shown in figure 2.1.

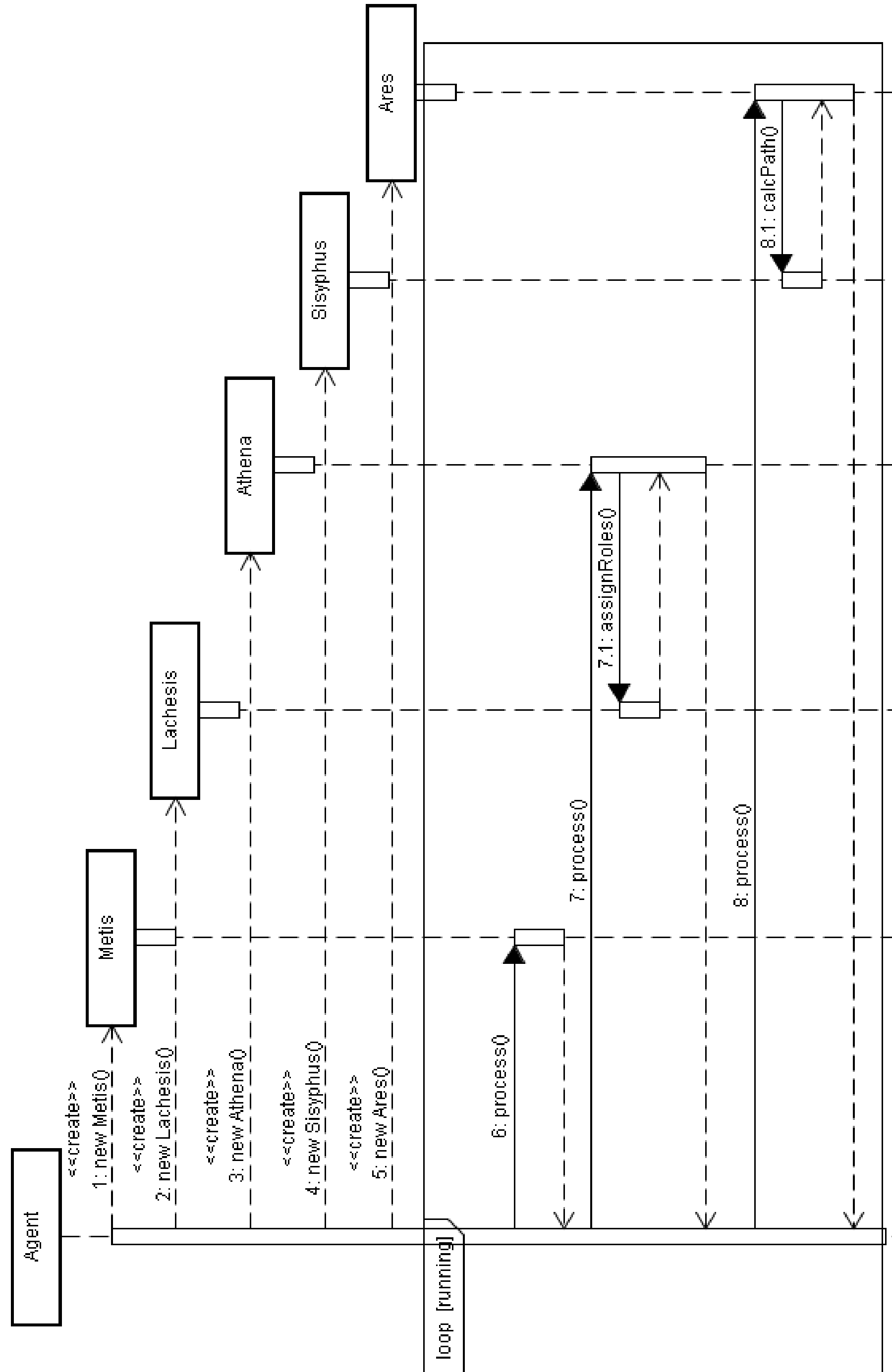

Figure 2.1: Application flow of the Agent

Chapter 3

Pandora

3.1 Purpose of Pandora

In Pandora the logic of the AI is stored in. This logic is subdivided into Plays, Roles, Conditions and Criteria.

3.2 Naming of Pandora

> In Greek mythology, Pandora was the first woman. As Hesiod related it, each god helped create her by giving her unique gifts. Zeus ordered Hephaestus to mold her out of earth as part of the punishment of mankind for Prometheus' theft of the secret of fire, and all the gods joined in offering her "seductive gifts". (...)
> According to the myth, Pandora opened a jar, in modern accounts sometimes mistranslated as "Pandora's box", releasing all the evils of mankind - although the particular evils, aside from plagues and diseases, are not specified in detail by Hesiod - leaving only Hope inside once she had closed it again. She opened the jar out of simple curiosity and not as a malicious act. [16]

In this context, Pandora is a symbol for Pandora's box, where all the good and bad things are stored. The Plays, Roles, Conditions and Criteria are these good and bad things - good things for us, bad things for our opponents.

Figure 3.1: Pandora with her box painted by Jules Joseph Lefebvre [16]

3.3 Plays

A Play defines what the overall-plan for the next few seconds is. This may be an indirect shot or defending the own goal with two defenders and the goalie. It is not necessary, that a Play has a task for each robot in store, since there can be more than one play simultaneously. Typically there is one offense Play and one defense Play parallel. A set of Plays is chosen by Athena, the PlayFinder (see section 6). A Play may but don't need to have one ore more following-Plays. These are chosen more likely by the Playfinder after completing the current Play.
The mentioned task for a single robot in a Play is called Role. For more information please refer to section 3.4.

3.3.1 Playbook (List of Plays)

This list is based on SVN commit #4439. For more details please refer to [6].

3.3.1.1 List of Defense-Plays

- KeeperPlus1DefenderPlay
- KeeperPlus2DefenderPlay
- KeeperSoloPlay
- ManToManMarker

3.3.1.2 List of Offense-Plays

- Attack
  - With Ball
    - ChipForwardPlay
    - DirectShotPlay
    - GameOffensePrepareWithThreeBotsPlay
    - GameOffensePrepareWithTwoBotsPlay
    - IndirectShotPlay
    - PassForwardPlay
    - PassToKeeperPlay
  - Without Ball
    - PositioningImprovingNoBallWithOnePlay
    - PositioningImprovingNoBallWithTwoPlay

    * SupportWithOneBlockerPlay
    * SupportWithOnePassBlockerPlay

- Ball Winning
  - BallCapturingWithDoublingPlay
  - BallCapturingWithOnePassBlockerPlay
  - BallWinningWithOneBlockerPlay
  - BallWinningWithOnePassBlockerPlay
  - PullBackPlay
- PassTwoBotsPlay

3.3.1.3 List of Standard-Plays

- Freekick
  - FreekickMarkerPlay
  - FreekickOffensePrepareWithThreeBots
- Kickoff
  - KickOffSymmetryPositioningPlay
  - PositioningOnKickOffThemPlay
- Penalty
  - PenaltyThemPlay
  - PenaltyUsPlay
- Stop
  - PositioningOnStoppedPlayWithThreePlay
  - PositioningOnStoppedPlayWithTwoPlay

3.3.2 APlay

Every Play inherits from the abstract class APlay.
Basic methods like *calcPlayableScore()* (see section 6.3) are declared in here.

3.4 Roles

A Role is a specific task within a Play (see section 3.3). Roles are mapped 1:1 to our robots. They are assigned within Lachesis, the RoleAssigner.

3.4.1 List of Roles

This list is based on SVN commit #4439.

3.4.1.1 List of Defense-Roles

- DefenderK1DRole
- DefenderK2DRole
- KeeperK1DRole
- KeeperK2DRole
- KeeperSoloRole
- ManToManMarkerRole
- PassiveDefenderRole

3.4.1.2 List of Offense-Roles

- AttackerSetPiece
- BallDribbler
- BallGetterRole
- ChipSender
- FreeBlocker
- IndirectShooter
- PassReceiver
- PassSender
- PositioningRole
- PullBackRole
- Shooter

3.4.1.3 List of Standard-Roles

- KeeperPenaltyThemRole
- PenaltyUsShooterRole
- DirectFreekicker
- KickerSetPiece

3.4.1.4 List of other Roles

- AimingRole
- MoveRole
- PassBlockerRole

3.5 Conditions

Each Role (see section 3.4) is defined by a set of Conditions. Conditions define in which state a Role is.

3.5.1 List of Conditions

This list is based on SVN commit #4439.

- AimingCon
- BallVisibleCon
- DestinationCon
- LookAtCon
- TargetVisibleCon
- ViewAngleCon
- VisibleCon

3.6 Criteria

Criteria are used to calculate the PlayableScore (see section 6.3), therefore they are the most important indicator the decision of Athena is based on. The Criteria are divided into global and local Criteria. Global Criteria only need to be calculated once each ai-cycle, e.g. which team is in ball-possession. Local Criteria may be different for each robot, so they need to be calculated for each one of them, e.g. if they can shoot at the goal.
Each Criterion analyses the current situation and compares it to the ideal situation for this Play. This ideal situation is stored in the variable `wish`. If the criterion is not matched, it returns its `penalty factor`, a value greater than zero.

3.6.1 List of Criteria

- global
  - BallPossessionCrit
  - OpponentApproximateScoringChanceCrit
  - OpponentScoringChanceCrit
  - TigersApproximateScoringChanceCrit
  - TigersScoringChanceCrit
  - TeamClosestToBallCrit
- local
  - DynamicFieldRasterCrit
  - ObjectPositionCrit
  - OpponentPassReceiverCrit
  - TigersPassReceiverCrit

Chapter 4

Skills and Skillsystem

The Skills and the Skillsystem are an important part of the AI. Nevertheless they are not part of this work, because they are descriped in detail in another seminar paper (see [4]).

Chapter 5

Precalculator: Metis

5.1 Purpose of Metis

In Metis situation and field analysis is done. There is a bunch of Calculators (see section 5.4) that are coordinated within Metis.
The gathered tactical information is added to the AIInfoFrame.

5.2 Naming of Metis

> Metis was the first great spouse of Zeus, indeed his equal and the mother of Athena, Zeus' first daughter, the goddess of war and wisdom. By the era of Greek philosophy in the fifth century BCE, Metis had become the goddess of wisdom and deep thought.[15]

The name Metis was chosen for calculations because she is the goddess of deep thought, which is necessary to analyse the situation thoroughly. Metis being the mother of Athena is a gimmick, because the information retrieved by Metis are handed to Athena in the next step.

Figure 5.1: An ancient depiction of the goddess Metis [15]

5.3 Implementation of Metis

Each ai-cycle in Metis the method `process()` is called. This method is shown in listing 5.1.

```
1  public AIInfoFrame process(AIInfoFrame currentFrame, AIInfoFrame previousFrame)
2  {
3     // check whether there is a worldFrame
4     if (currentFrame.worldFrame != null)
5     {
6        currentFrame.tacticalInfo.setDefGoalPoints(defPointCalc
              .calculate(currentFrame.worldFrame));
7        //...
8     } else
9     {
10       log.warn("Metis receives null frame");
11
12       // create default list
13       Vector2f fieldCenter = AIConfig.getGeometry().getCenter();
14       List<ValuePoint> defaultList = new ArrayList<ValuePoint>();
15       defaultList.add(new ValuePoint(fieldCenter.x, fieldCenter.y));
16
17       // take default values for tactical field
18       currentFrame.tacticalInfo.setDefGoalPoints(defaultList);
19       currentFrame.tacticalInfo.setBallPossesion(null);
20       //...
21    }
22
23    return currentFrame;
24 }
```

Listing 5.1: process() in Metis

It is differentiated between two cases: If there is a WorldFrame (line 4-7) or if there is no WorldFrame (line 8-21).
If in fact there is a WorldFrame (which should normally be the case) the Calculators are called and the result is stored in the WorldFrame. In this listing there is one calculator symbolic for the bunch of them (line 6).
If there is no such thing as a WorldFrame first a warn-message is invoked. Then a default list is created (line 12-15) that is stored within the WorldFrame, if there is a list required (line 18). If there is no list required, there is either `null` (line 19) or a default-value stored. These two called Calculators are symbolic for the other ones.

5.4 Calculators

5.4.1 Defense Calculators

DefensePoints A Calculator for dangerous points. The result can be used for placing defenders (see [7]).

5.4.2 Offense Calculators

OffensePointsCarrier The OffensePointsCarrier-Calculator returns positions on the field, that are suitable for placing the ball-carrier.

OffensePointsReciever The OffensePointsReciever-Calculator returns positions on the field, that are suitable for placing robots without the ball. The positions distinguish themself by having a clear line to the ball-carrier and, ideally, a clear line towards the goal

5.4.3 Other Calculators

ApproximateScoringChance This calculator determines whether the TIGERS or the opponents have an *direct* or *indirect* chance to score a goal. This means a bot may has to get the ball and aim before a shot would be successful. Wether this Calculator returns the scoring chance for the TIGERS or for the opponent is defined by a parameter of the constructor.

ScoringChance This calculator determines whether we or our opponents have a *direct* chance to score a goal. This means the ball carrier is only a kick move away from scoring a goal. Therefore it only calculates for the ball-carrier. If this Calculator returns the scoring chance for the TIGERS or for the opponent is defined by a parameter of the constructor.

BallPossession Indicates which team has the ball. Besides the two teams there are also the options `BOTH` and `NO_ONE`.

TeamClosestToBall This Calculator determines which team is closer to the ball. This information is useful when the BallPossession-Calculator states, that no one is in ball-possession.

5.4.4 Example-Calculator in Detail

For clarification the Calculator *TeamClosestToBall* (see section is described in detail. It is shown in listing 5.2.

```
1  public ETeam calculate(WorldFrame worldFrame)
2  {
3     float distance;
4     float closestDistance = UNINITIALIZED_ID;
5     Vector2f ballPos = worldFrame.ball.pos;
6
7     ETeam teamClosestToBall = ETeam.UNKNOWN;
8
9     // identify the Tiger bot closest to ball
```

```
10      for (TrackedBot currentBot : worldFrame.tigerBots.values())
11      {
12         distance = AIMath.distancePP(currentBot, ballPos);
13         if (closestDistance == UNINITIALIZED_ID || distance < closestDistance)
14         {
15            closestDistance = distance;
16            teamClosestToBall = ETeam.TIGERS;
17         }
18      }
19
20      // identify the opponent bot closest to ball
21      for (TrackedBot currentBot : worldFrame.foeBots.values())
22      {
23         distance = AIMath.distancePP(currentBot, ballPos);
24         if (closestDistance == UNINITIALIZED_ID || distance < closestDistance)
25         {
26            closestDistance = distance;
27            teamClosestToBall = ETeam.OPPONENTS;
28         }
29      }
30      return teamClosestToBall;
31   }
```

Listing 5.2: Calculator TeamClosestToBall

The return-value of this method is an enum (line 1). It has the states `UNKNOWN`, `TIGERS`, `OPPONENTS` and `EQUAL`. First the necessary variables are declared (line 3-7). The variable `ballPos` is initialized with the actual ball position (line 5), other are initalized with default-values (line 4,7).
Afterwards, the same procedure is executed first for the TIGERS (line 9-18) and then for the opponents (line 20-29). Since the algorithm for both teams is the same, it is only described once. For each bot of a team (line 10) the distance to the ball is calculated (line 12). If it is either the first bot to be checked or its distance is closer than the formerly closest (line 12), the varialble `closestDistance` is updated (line 15) and the enum `ETeam` is set accordingly (line 16).
When this procedure is executed for each robot of both teams the enum `ETeam` is returned. This is most likely either `TIGERS` or `OPPONENTS`. Only if the WorldFrame does not contain a single robot it is `UNKNOWN`.

Chapter 6

PlayFinder: Athena

6.1 Purpose of Athena

Athena supervises the *PlayFinder*. In this module the decision is made, which Plays are performed. For this purpose the given situation is evaluated as well as messages from the referee.
Also *Lachesis*, the Roleassigner, is called in this module.

6.2 Naming of Athena

In the ancient Greek mythology *Athena* is the goddess of wisdom and strategy. Therefore *Athena* is privileged to decide which strategy, i.e. which *Plays*, are chosen.

Figure 6.1: Athena [13]

6.3 PlayableScore

Each Play calculates its PlayableScore. This score is a value between `0` and `100`. A high value indicates that a Play is suitable for the current situation. The Playable-Score is calculated based on the Criteria (see section 3.6).
Although `calcPlayableScore()` is a method of the class `APlay` and hence of the Plays, it is described in this section because it is called by Athena. Listing 6.1 shows this method.

```
1  public int calcPlayableScore(AIInfoFrame currentFrame)
2  {
3     float result = this.basicPlayableScore;
4     // mulitply all criteria values
5     for (ICriterion crit : this.criteria)
6     {
7        result *= crit.doCheckCriterion(currentFrame);
8     }
9
10    // 0 < result < 100 ?
11    if (result > maxPlayableScore)
12    {
13       result = maxPlayableScore;
14    } else if (result < minPlayableScore)
15    {
16       result = minPlayableScore;
17    }
18
19    return (int) result;
20 }
```

Listing 6.1: calcPlayableScore

First the basic PlayableScore is stored in the variable `result` (line 3). This way, it is possible to privilege some Plays by giving them a higher basic score. Also the basic score can be reduced if a Play has failed several times during the current match. Then every Criterion of the Play is calculated and multiplied with the current result (line 5-8). Since the return-value of the Criteria is a value between `0` and `1` the basic score is as well the maximum score. Because of the multiplication, even one KO-Criterion, e.g. the Play is `directShot` and the opponent is in ball possession, can set the PlayableScore to zero.
Afterwards it is ensured that the PlayableScore is between `0` and `100` (line 11-17). An initial value above `100` may be caused by a basic score above `100`. Finally, the result is returned (line 19).

6.4 Implementation of Athena

6.4.1 Athena process()

The most important method in *Athena* is *process()*. This method is logically divided into two parts. The first part handles the PlayFinding, the second one deals with

calling the Role-assignment.
Aside from a couple of organization-structures, the first part of this method is as follows.

```
1  public AIInfoFrame process(AIInfoFrame currentFrame, AIInfoFrame previousFrame)
2  {
3     //...
4
5     // Whether any circumstances changed and a new play seems appropiate
6     boolean anythingChanged = false;
7
8     final boolean tigersChanged = tigersRemoved | tigersAdded |
            tigerDisconnected | tigerConnected;
9
10    anythingChanged |= tigersChanged;
11
12    if (anythingChanged)
13    {
14       frame.playStrategy.setForceNewDecision();
15    }
16
17    // If switched from auto-control to manual (GUI) or the other way round
18    boolean switched = false;
19
20    // Should GUI override Athena?
21    if (athenaAdapter.overridePlayFinding())
22    {
23       switched = !lastControlState; // Switched from auto to GUI
24
25       athenaAdapter.choosePlays(frame, preFrame);
26
27       lastControlState = true; // Remember state for next cycle
28    } else
29    {
30       //...
31
32       // Choose plays
33       playFinder.choosePlays(frame.playStrategy.getActivePlays(), frame,
               preFrame);
34    }
35
36 //...
37 }
```

Listing 6.2: process() in Athena (part 1)

First of all a boolean is declared and initialized with the default-value `false` (line 6). This boolean indicates to the PlayFinder, if the situation underwent a radical change. One reason to set this boolean to `true` may be, if the amount of our bots has been changed, e.g. if a bot timed out or if the SSL-Vision-Software doesn't detect it for a few seconds (line 8-10). If in fact anything basically has changed, a new Play decision will be forced (line 12-15).
Afterwards a second flag is declared and initialized with the value `false` (line 14). This one states, if the used Playfinder has been changed by a user input.
The remaining lines of this listing contain an `if-else` statement. If the flag to override Athena is set within the GUI, the `athenaAdapter` chooses the set of Plays

accordingly to the user input(line 21-27). If elsewise this flag is not set, the normal PlayFinder, the `matchPlayFinder` (see section 6.5.2), is called (line 32-33).
The second part of `process` deals with the Role-assignment. This part is shown in listing 6.3.

```
1  if (!athenaAdapter.overrideRoleAssignment())
2  {
3     // Role assignment
4     if (switched || tigersChanged || frame.playStrategy.hasPlayChanged())
5     {
6        lachesis.assignRoles(frame);
7     } else
8     {
9        frame.assignedRoles.putAll(preFrame.assignedRoles);
10    }
11 } else
12 {
13    athenaAdapter.assignRoles(frame, preFrame);
14 }
```

Listing 6.3: process() in Athena (part 2)

Like the PlayFinder, the RoleAssigner can be overwriten by the GUI. If this is not the case (line 1) and if anything has changed (line 4), Lachesis is being called to assign the Roles (line 6). If nothing has changed the Role-assignment of the last frame will be chosen again (line 7-10).
If otherwise the GUI overwrites the Role-assignment, the `athenaAdapter` is called (line 11-14). The `athenaAdapter` is an object of the class `AthenaGuiAdapter` that allows the GUI to influence the behavior of Athena.

6.4.2 Play-factory

The Play-factory is used to create new Plays with the help of their names, that are stored within enums. It is shown in listing 6.4.

```
1  public APlay createPlay(EPlay play, AIInfoFrame currentFrame)
2  {
3     //...
4
5     WorldFrame wf = currentFrame.worldFrame;
6
7     switch (play)
8     {
9        case ONE_BOT_TEST:
10          return new OneBotTestPlay(wf);
11
12       case AROUND_THE_BALL:
13          return new AroundTheBallPlay(wf);
14
15       //...
16
17       case CHIP_FORWARD:
18          return new ChipForwardPlay(wf);
19
20       default:
```

```
21            throw new IllegalArgumentException("Play type could not be handled by
                  play factory! Play = "
22                + play.toString());
23        }
24    }
```

Listing 6.4: Play-factory

First of all, the WorldFrame is extracted from the AIInfoFrame (line 5). It is later on used to create the new Plays (line 10, 13, 18). Then the new Play is created within a `switch-case` statement (line 7-23). If the used enum is not registered in the Play-factory, an `IllegalArgumentException` is thrown (line 20-22).

6.5 PlayFinder

6.5.1 Interface IPlayFinder

Accordingly to the interface design-pattern, the used PlayFinder can easily be replaced by another one. The method that needs to be implemented is called `choosePlays()`.

6.5.2 MatchPlayFinder

The most important method in the MatchPlayFinder is called `choosePlays`. In here a set of Plays is chosen and therefore the strategy as well.
The possible situations are divided into five alternatives. They are evaluated in the following order:

1. *first frame*: since the current frame is the very first one, the init-Play is chosen
2. *no ball*: if there is no ball, the init-Play is chosen as well
3. *referee message available*: so a decision according to this message is made
4. *new decision has been forced*: therefore the Plays are chosen freely
5. *no finished Plays*: since everything is normal (otherwise one of the alternatives above would fit) and no Plays have finished, the old Plays are kept
6. *finished Plays*: one or more Plays have ended, so one or more new Plays are chosen. There are the following alternatives:
   (a) *finished Play failed*: the new Plays are chosen freely
   (b) *no follow-Plays exist*: the finished Play succeeded and there are no possible follow-Plays, so the new Plays are chosen freely as well

(c) *otherwise*: the Play-tuple with the highest PlayableScore is chosen

In *4.*, *6.a))* and *6.b)*, the new Plays are chosen freely from all existing Plays, i.e. the method `choosePlaysFree()` is called. This method is shown in listing 6.5.

```
1  private void choosePlaysFree(List<EPlay> choices, AIInfoFrame frame,
           AIInfoFrame preFrame, List<APlay> result)
2  {
3     PlayTuple bestTuple = playMap.getTuples().get(0);
4     int numberOfBots = frame.worldFrame.tigerBots.size();
5     for(PlayTuple tuple : playMap.getTuples())
6     {
7        if(tuple.calcPlayableScore(frame,numberOfBots) >
                 bestTuple.calcPlayableScore(frame,numberOfBots))
8        {
9           bestTuple = tuple;
10       }
11    }
12    result.clear();
13    result.addAll(factory.createPlays(bestTuple.getPlays(), frame));
14 }
```

Listing 6.5: MatchPlayFinder: method *choosePlaysFree*

The possible Play-tuples are stored within `playMap`. The first element of this list is stored as the current best one (line 3). Then the PlayableScore for each tuple is calculated and the best one is used to get a set of Plays from the Play-factory (see section 6.4.2), which are stored in the variable `result` (line 12-13).

Chapter 7

RoleAssigner: Lachesis

7.1 Purpose of Lachesis

In Lachesis the Roles are mapped to the robots.

7.2 Naming of Lachesis

> In Greek mythology, Lachesis was the second of the Three Fates, or Moirae.(...)
> Lachesis was the apportioner, deciding how much time for life was to be allowed for each person or being. She measured the thread of life with her rod. She is also said to choose a person's destiny after a thread was measured. In mythology, it is said that she appears with her sisters within three days of a baby's birth to decide its fate.[14]

Since Lachesis decides a persons destiny, her name is used for the Role-assigner that chooses the Role of a robot.

Figure 7.1: Three Fates: Clotho, Atropos and Lachesis [9]

7.3 Implementation of Lachesis

First of all, the Roles that need to be assigned are divided into three groups: *aggressive Roles*, *creative Roles* and *defensive Roles*. This way, the Role-assigner can set a priority to one of this groups, e.g. the defensive Roles if the opponent is in ball possession. Also the keeper-Role is assigned in here, because the same robot has to be keeper the whole game.
After checking against some special cases such as if there is only one bot that needs a Role, the RoleAssigner is called for each group of Roles. This is shown in listing 7.1.

```
1  if (numFreeTigers > 0 && numRolesToAssign > 0)
2  {
3     switch (frame.playStrategy.getPlayBehavior())
4     {
5        case AGGRESSIVE:
6           roleAssigner.assignRoles(assignees.values(), aggRoles, assignments,
                     frame);
7           roleAssigner.assignRoles(assignees.values(), defRoles, assignments,
                     frame);
8           roleAssigner.assignRoles(assignees.values(), creRoles, assignments,
                     frame);
9           break;
10
11       case NOT_DEFINED:
12       case DEFENSIVE:
13          roleAssigner.assignRoles(assignees.values(), defRoles, assignments,
                     frame);
14          roleAssigner.assignRoles(assignees.values(), aggRoles, assignments,
                     frame);
15          roleAssigner.assignRoles(assignees.values(), creRoles, assignments,
                     frame);
16          break;
17    }
18 }
```

Listing 7.1: Call of the Role-assigner

If there are both TIGERS without a Role and Roles that need to be assigned (line 1), the Play-behaviour is evaluated (line 3). If the behaviour is `AGGRESSIVE` (line 5), first the aggressive Roles are assigned, than the defensive and at last the creative Roles (line 6-9).
If the Play-behaviour is not defined (line 11) or `DEFENSIVE` (line 12), the defensive Roles are assigned before the aggresive Roles.

7.3.1 Interface IRoleAssigner

As in our other modules, the logic is implemented on an interface. This way, the Role-assigner can easily be replaced.

7.3.2 OptimizedRoleAssigner

The Role-assigner called *OptmizedRoleAssigner* calculates every possible robot-to-Role combination and returns the one with the shortest added distance. Therefore a 2-dimensional array is created in the constructor with all possible permutations.

7.3.3 PermutationGenerator

The implemented PermutationGenerator is explained and implemented in [3] and only slightly modified. Now it returns a copy of the internal buffer instead of the object itself.

Chapter 8

RoleExecuter: Ares

8.1 Purpose of Ares

Ares is the RolExecuter. In here the behaviour of the Roles is calculated.

8.2 Naming of Ares

> Ares is the Greek god of war. He is one of the Twelve Olympians, and the son of Zeus and Hera. In Greek literature, he often represents the physical or violent aspect of war, in contrast to the armored Athena, whose functions as a goddess of intelligence include military strategy and generalship.[12]

Since Ares is responsible for the physical part of war he has been chosen to lead our troops (robots) on the battle ground (field).

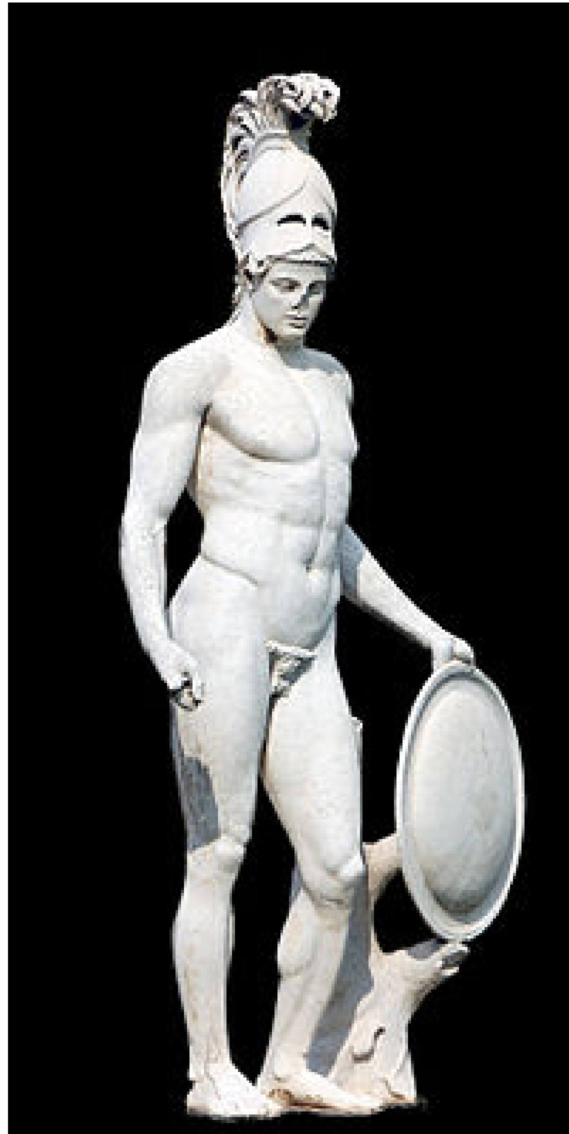

Figure 8.1: Ares [12]

8.3 Implementation of Ares

Since Ares is the RoleExecuter the `calculateSkills`-method of each robot is called in here. This is shown in listing 8.1.

```
1  for (Entry<Integer, TrackedTigerBot> entry : wFrame.tigerBots.entrySet())
2  {
3     Integer botId = entry.getKey();
4     ARole role = frame.assignedRoles.get(botId);
5
6     // # Calc skills
7     final SkillFacade facade = new SkillFacade();
8     role.calculateSkills(wFrame, facade);
9
10    // # Execute skills!
11    skillSystem.execute(role.getBotID(), facade);
12 }
```

Listing 8.1: Roleexecuter Ares

In the happy-day scenario this method would look like this. First the current `botID` (line 3) and the assigned Role are extracted (line 4). Then a *SkillFacade* is declared and initiated (line 7) and the Skills are actually calculated (line 8). The SkillFacade is a kind of container for the Skills. For an explanation in more detail please have a look at [4]. Finally, the SkillFacade and the botID are given to the SkillSystem.
As mentioned the described listing is a happy-day scenario. To make sure everything is save there are a few more lines in the real method. First of all it is very unlikely but possible that a Role has changed and the old Skills regarding dribbling and kicking are accidently still active. Therefore the code shown in 8.2 has been added after calculating the Skills (listing 8.1, line 8).

```
1  if (roleChanged != null && roleChanged.contains(botId))
2  {
3     if (facade.isSlotFree(ESkillGroup.DRIBBLE))
4     {
5        facade.dribble(false);
6     }
7
8     if (facade.isSlotFree(ESkillGroup.KICK))
9     {
10       facade.disarm();
11    }
12 }
```

Listing 8.2: Roleexecuter Ares

First it is checked if the Role in fact has changed (line 1). If this is the case, the dribbling device is turned off (line 5) and the kicker is disarmed (line 10) if the new Role has not created Skills itself for these devices (line 3,8).
Furthermore it is theoretically possible that there are less Roles than robots. In this case one ore more robots would no longer be controlled by our software but would still have active Skills. Therefore a bot is stopped, disarmed (kicker is activated until its inductors are empty) and the dribbling device is stopped.

With this two extensions it is impossible that a robot accidently kicks or goes postal.

Chapter 9

PathFinder: Sisyphus

9.1 Purpose of Sisyphus

Pathfinding is as old as computer science itself. Thereby pathfinding always refers to the same problem: An object (in this case a robot) has to be navigated from one point of a map to another avoiding obstacles. Obstacles can be static or dynamic. In the RoboCup SSL environment these obstacles are goalposts (static) and other robots or the ball (dynamic).[8]

9.2 Naming of Sisyphus

> *As a punishment from the gods for his trickery, Sisyphus was made to roll a huge boulder up a steep hill, but before he could reach the top of the hill, the rock would always roll back down, forcing him to begin again. The maddening nature of the punishment was reserved for Sisyphus due to his hubristic belief that his cleverness surpassed that of Zeus. As a result when Sisyphus was condemned to his punishment, Zeus displayed his own cleverness by binding Sisyphus to an eternity of frustration with the boulder rolling away from Sisyphus when he neared the top of the hill. Accordingly, pointless or interminable activities are often described as Sisyphean.*[17]

Finding a path stands for rolling the boulder. The rock rolling back before *Sisyphus* can reach the top is represented by the ever changing environment in the RoboCup SSL, causing the necessity to find a new path. Therefore and because pathfinding is in fact an interminable activity (the calculation-method is called on every *WorldFrame* for each robot all game long) this module is called *Sisyphus*.

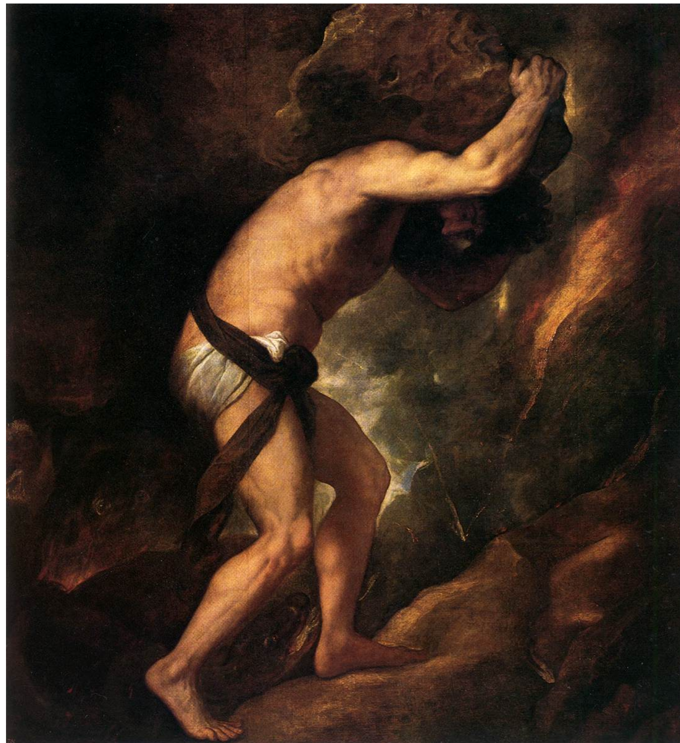

Figure 9.1: Sisyphus painted by Vecellio Tiziano [17]

9.3 Implementation of Sisyphus

To fulfill the requirements of pathfinding in the RoboCup SSL there is a need for a fast global pathfinder.

9.4 Global pathfinder: Extended Rapdidly Exploring Random Tree

For the global pathfinding the ERRT has been chosen. For a detailed explanation of the reasons please refer to section 3.1 of [8].
In the following, the term *goal* describes the endpoint of the path that needs to be searched. The term *target* always refers to a point that is generated during the pathfinding.
To understand the explanations, it is necessary to know about the internal used datastructures (see section 9.6).
The basic RRT-algorithm is described in section 9.4.1. The extensions that have been added to the basic algorithm are described in the sections 9.4.2, 9.4.3 and 9.4.4.

9.4.1 Basic RRT algorithm

The basic Rapidly-Exploring Random Tree (RRT) planner searches, as every pathfinder, for a path from a starting point to a goal. As stated by its name, it is based on a tree structure. Furthermore, important elements are random numbers. Hereafter the term node is used for states, given that they are organized in a tree structure. Since this RRT is designed for RoboCup, a suited terminology is chosen: The environment is referred to as field. To perform its search, there is a need for the following functions:

- `Node chooseTarget()`: Decides if a random node or the goal is the next target. (see section 9.4.1.1)
- `Node createRandomNode()`: returns a node, uniformly drawn from the field. (see section 9.4.1.2)
- `Node getNearest(List<Node>, Node)`: does a nearest neighbor search on all known nodes to find the nearest one.(see section 9.4.1.3)
- `Node extendTree(Node, Node)`: takes a step along the line from the node nearest to the target towards it. In this implementation the step size is constant (in the following refered to as `stepSize`).(see section 9.4.1.4)
- `boolean isWayOK(IVector2, IVector2, Float)`: checks if a collision would happen if a robot moves from one node to another.(see section 9.4.1.5)

9.4.1.1 Basic-RRT: chooseTarget()

To shorten calculation time, the goal is chosen more likely as next target than any other node. Which target is chosen is decided stochastically:

- with probability p the next target is the goal
- with probability $1 - p$ a random node is chosen by `createRandomNode()` (see section 9.4.1.2)

This is done by the function *chooseTarget()* that is shown in listing 9.1.

```
1  Random generator = new Random();
2
3  private Node chooseTarget()
4  {
5     p = generator.nextFloat();
6
7     if (p <= P_DESTINATION)
8     {
9        return goalNode;
10    }
11    else
12    {
13       return createRandomNode();
14    }
15 }
```

Listing 9.1: Basic-RRT function: chooseTarget()

In line 1 an instance of the class `Random` is created. This object is capable of creating pseudo random numbers. Its method *nextFloat()* is used to create a pseudo random number between `0` and `1` (line 5). According to this number is smaller than `P_DESTINATION` or not, the goal is chosen as next target respectively a random node (line 7-14).
`P_DESTINATION` is a constant defined in `AIConfig`.

9.4.1.2 Basic-RRT: createRandomNode()

The function *createRandomNode()* creates, as stated by its name, a new node, uniformly drawn from the field. This function is shown in listing 9.2.

```
1  private Node createRandomNode()
2  {
3     float x = (generator.nextFloat() * FIELD_LENGTH) - FIELD_LENGTH / 2;
4     float y = (generator.nextFloat() * FIELD_WIDTH) - FIELD_WIDTH / 2;
5
6     return (new Node(x, y));
7  }
```

Listing 9.2: Basic-RRT function: createRandomNode()

The x- and the y-value of the new node is chosen stochastically (line 3-4). Finally, a node is created with these values and is returned (line 6).

9.4.1.3 Basic-RRT: getNearest()

The function *getNearest()* performs a nearest neighbor search on all known nodes. This function is shown in listing 9.3.

```
1  for(Node currentNode : nodeStorage)
2  {
3     currentSquareDistance = AIMath.distancePPSqr(nextTarget, currentNode);
4
5     //found a better one
6     if (currentSquareDistance < minSquareDistance)
7     {
8        nearestNode = currentNode;
9        minSquareDistance = currentSquareDistance;
10    }
11 }
```

Listing 9.3: Basic-RRT function: getNearest()

To speed up the search the square of the distance is calculated instead of the real distance (line 3). This is possible because of

$$a^2 > b^2 \Leftrightarrow |a| > |b|$$

9.4.1.4 Basic-RRT: extendTree()

The function *extendTree()* creates a new node that is `stepSize` away from a given node. The direction in which the new node is created is given by a target node. This function is shown in listing 9.4.

```
1  private Node extendTree(Node target, Node nearest, float step)
2  {
3     Node extended = new Node(AIMath.stepAlongLine(nearest, target, step));
4     return extended;
5  }
```

Listing 9.4: Basic-RRT function: extendTree()

The AIMath-function returns the new node (line 3). The function *stepAlongLine* is described in section 10.2.2.
Unlike seen in other RRT-implementations, extendTree() does *not* add the new node to the tree. Rather it just returns it to the calling function (line 4), where a set of conditions can be checked.

9.4.1.5 Basic-RRT: isWayOK()

The function *isWayOK* checks, if any obstacle would hinder the bot from moving on the direct connection between two points. Obstacles that have to be checked are other bots and the goal posts. The ball is an optional obstacle. The algorithm used for collision detection is described in [1].
Additionally, a float value is given to this method as parameter: the safety-distance. This value sets the space that has to remain at least between two bots.

9.4.1.6 Basic-RRT: growTree()

The functions described in the sections 9.4.1.1, 9.4.1.2, 9.4.1.3, 9.4.1.4 and 9.4.1.5 run within a loop. This loop is provided by the method `growTree()`. Reduced to its essence this method is shown in listing 9.5.

```
1  private ArrayList<Node> growTree()
2  {
3     ArrayList<Node> tree = new ArrayList<Node>();
4     tree.add(new Node(thisBot.pos));
5
6     for(int iter = 0; iter < MAX_ITERATIONS && !isGoalReached; ++iter)
7     {
8        target = chooseTarget();
9        nearest = getNearest(tree,target);
10       extended = extendTree(target, nearest);
11
12       if(isWayOK(nearest, extended)
13       {
14          nearest.addChild(extended);
15          tree.add(extended);
16
17          if(goalReached(extended)
18          {
19             isGoalReached = true;
20          }
21       }
22    }
23 }
```

Listing 9.5: Main-loop in Sisyphus: growTree()

First, the current position of the bot, the path has to be calculated for, must be added to the storage (line 3-4). This storage is called *tree*.
The loop iterates either until the goal is reached, i.e. a path has been found, or until a set amount of iterations has been done. This is due to restrict calculation time and to avoid a stack-overflow, if there is no possible path (line 6).
Each iteration is as follows: First a target is chosen by the function `choseTarget()` (line 8). Next, the node with the shortest distance to the mentioned target is determined, using the `getNearest()` method (line 9). Then, the `extendTree()` method generates a Node between the nearest node and the target node, `stepSize` away from the nearest (line 10). If the path between the nearest node and the extended one is free, this node is added to the tree (line 12-16). Afterwards it is checked, if the goal is reached (line 17). If this is the case the boolean `isGoalReached` becomes `true`, so that the loop finishes (line 19).
The application flow is shown in figure 9.2. [8]

9.4.2 Extension 1: Abbreviation

The first extension to the basic-RRT is called *Abbreviation*. Right at the beginning, even before the method `growTree()` has been started, the direct path between the start and the goal is checked using the method `isWayOK()` (see section 9.4.1.5).

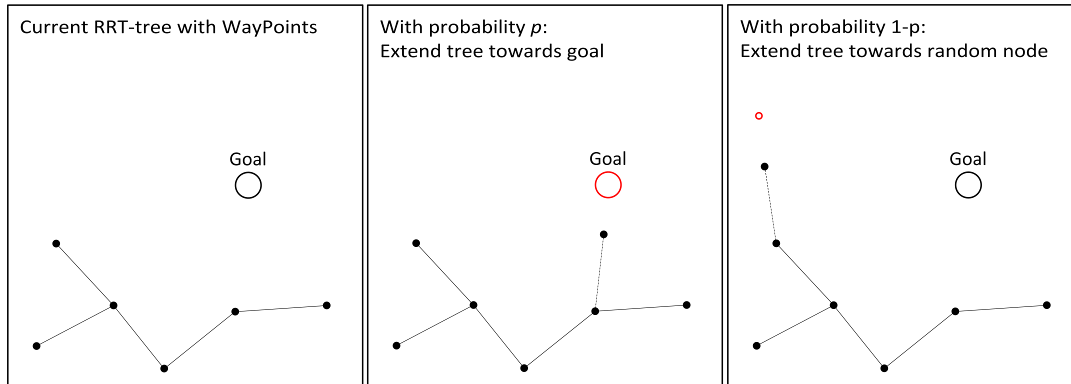

Figure 9.2: The spreading basic RRT algorithm

Furthermore, the direct path between a new node and the goal is checked each time a new node is created. This is done by a little modification to the method `growTree()`. Therefor, the lines 14-17 of the function described in section 9.4.1.6 are replaced by the lines shown in listing 9.6.

```
1  if (isWayOK(extended, goalNode,SAFETY_DISTANCE))
2  {
3     extended.addChild(goal);
4     tree.add(goal);
5  }
```

Listing 9.6: RRT-Extension 1: Abbreviation

The goal is added as child of the newest node (line 3-4).
This version still is not the implemented one. This is due to the path-postprocessing. See section 9.4.5 for the used one.

9.4.3 Extension 2: Waypoint cache

The extension *Waypoint Cache* has the purpose to speed up the pathfinding. Although the environment is changing very quickly, it doesn't alter completly. If the old path is no longer valid, it is likely, that a valid one is pretty similar to the old one. Therefor the waypoint cache has been implemented. While using the basic RRT, there are two options, which target will be the next one: The goal or a random node. With this extension, there is an additional option: A waypoint. This waypoint is a node from a previous valid path and may help finding a new path.
The described behavior is shown in figure 9.3.
To implement the waypoint cache it is necessary to make changes to the method *chooseTarget* (see section 9.4.1.1). The updated version of this method is shown in listing 9.7.

```
1  private Node chooseTarget(){
2     p = generator.nextFloat();
```

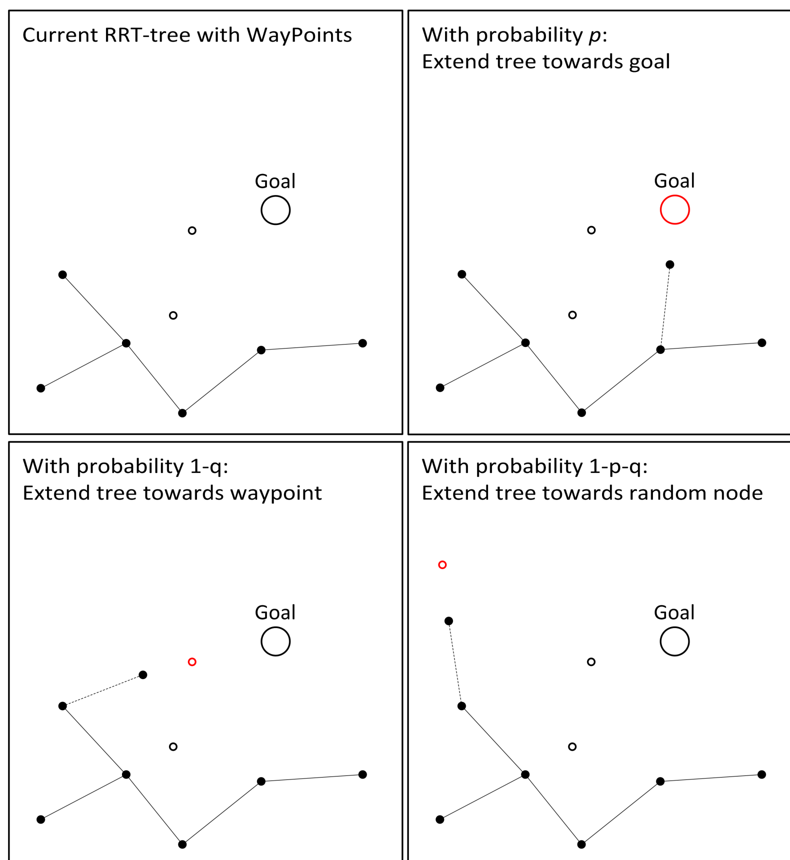

Figure 9.3: The spreading RRT algorithm with Waypoint Cache-Extension

```
3       if (p <= P_DESTINATION){
4           return goalNode;
5       }
6       else if (p <= P_DESTINATION + P_WAYPOINT && WPCFillLvl > 0){
7           return getNodeFromWPC();
8       }
9       else{
10          return createRandomNode();
11      }
12  }
```

Listing 9.7: RRT-Extension 2: Waypoint cache, method chooseTarget()

To integrate the waypoint cache the `if-then-else` has been enlarged by one option (line 8-10). Besides checking if the probability `p` is in the right range, also it must be checked if the waypoint cache is not empty (if it is empty, no waypoint can be returned, because there is none). The called method `getNodeFromWPC()` is shown in listing 9.8.

```
1   private Node getNodeFromWPC()
2   {
3       return waypoints[(int) (generator.nextFloat() * WPCFillLvl)];
4   }
```

Listing 9.8: RRT-Extension 2: Waypoint cache, method getNodeFromWPC()

This method randomly chooses one node from the waypoint cache and returns it (line 3).
As well, there was the need for the following methods:

- `fillWPC()`
- `insertNodeToWPC(Node)`

The method `fillWPC()` takes the list with all the waypoints of a found path and calls the method `insertNodeToWPC(Node)` for each one. `insertNodeToWPC(Node)` is shown in listing 9.9.

```
1   private void insertNodeToWPC(Node node)
2       {
3           if (WPCFillLvl < WPC_SIZE)
4           {
5               waypoints[WPCFillLvl] = new Node(node);
6               WPCFillLvl++;
7               return;
8           }
9           else
10          {
11              waypoints[(int) (generator.nextFloat() * WPC_SIZE)] = new Node(node);
12              return;
13          }
14      }
```

Listing 9.9: RRT-Extension 2: Waypoint cache, method insertNodeToWPC()

The method `insertNodeToWPC` can be devided into two phases: Filling the waypoint cache before it is completely full (line 3-8) and afterwards (line 10-13).
Before, the new waypoint is added at the end of the already stored waypoints (line 5). If it is full, a randomly chosen waypoint is replaced by the new one (line 11).

9.4.4 Extension 3: KD-Trees

The extension *KD-Tree* is not implemented at the moment. Therefor not the used implementation is described below but the theory.
Using a kd-tree (short for k-dimensional tree) has the purpose to fasten the RRT algorithm. The kd-tree-approach attacks at an important part of the basic algorithm, the nearest neighbor search. Its importance is revealed by a thought experiment: In each cycle of the RRT, the nearest neighbor to a given point is searched. Let the number of iterations be n. This means, nearest neighbor search is executed n times during pathfinding If every node is evaluated each time, the NodeDistance function is $\sum_{i=0}^{N} i$ called times. With $n = 500$, this means that the NodeDistance function runs 125.250 times. Even if a very high-performance NodeDistance function is used, it is instrumental to reduce the amount of calls of this function. A kd-tree is a form of binary-tree. Each node divides the space into two subspaces. Accordingly to a nodes relative position to the splitting object, it is added as left or right child. The geometrical representation of the splitting object is a hyperplane. In the context of RoboCup and a 2-dimensional environment, it is a line. Therefore, those lines are either parallel to the x- or to the y-axis and the decision, if a node is left or right child is made simply by comparing the x- respectively y-value. A graph and geometrical representation are shown in figure 9.4. [8]

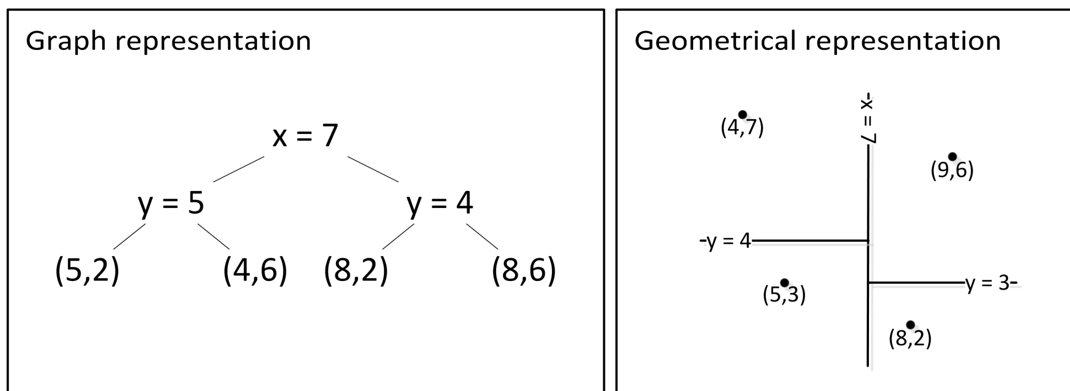

Figure 9.4: Representation of the KD-tree

The reduction of calls of NodeDistance is due to the geometric segmentation of the kd-tree. While performing the nearest neighbor search, the algorithm first traverses

the tree the same way, a node is inserted. This way, the rectangle containing the target is found. The node representing the rectangle may not be the nearest one, but is definitly a good approximation. The following algorithm describes the exact nearest neighbor search: Having found a leaf node near the target, now the tree is traversed backwards, starting with the found leave node. At every node passed while traversing backwards, the following consideration is made: Is there the mere possibility, that a node from the other subtree may be nearer to the target? If it is, that subtree will be traversed the same way as the original tree. If a node is found that is nearer than the current best one, this one will be stored as new comparison element. The complexity of the primitive approach, checking every node, is $O(n)$. According to [10], the average complexity of the nearest neighbor search on a balanced kd-tree is $O(logn)$. In accordance with [5], the worst case complexity is $O(k * N^{1-1/k})$. With $k = 2$ a worst case complexity of $O(2 * \sqrt{N})$ is given. Therefore, even the worst case complexity of a nearest neighbor search on a kd-tree is much better than the complexity of the primitive approach. Anyway, this method for the nearest neighbor search is not suitable for an RRT-algorithm, if speed is very important. Due to the fact that the new node will be added as a child of the nearest node, it is possible that the complete kd-tree must be rebuilt in each iteration. Furthermore, the complexity mentioned above is only for a balanced kd-tree. Balancing a given imbalanced kd-tree is very complex due to the multiple dimensions. To improve speed, a simplification was made: Although the leaf node representing the targe rectangle is not necessarily the nearest node, this one is assumed to be it. The inaccuracy has only a little influence and is not of importance after smoothing the kd-tree in the end. [8]

9.4.5 Path-postprocessing

Before path-postprocessing the path consists of many pathpoints near to each other and one long, straight line towards the goal. Because this is pretty unfunctional, there are a few modifications to the path to be made:

- subdividePath
- smoothPath
- reduceAmountOfPoints

`subdividePath` is called to subdivide the long, straight line at the end of the path. This helps to smooth the path, which is the next step.
The method `smoothPath` tries to find abbreviations in the path. It iteratively checks the way how the crow flies between two points and connects them if the path is free. These new connections may be longer than the normal `stepSize`. Therefore the method `subdividePath` is called on this new section of the path.

The method `subdividePath` may cause straight sections of the path with more points on it than necessary. These additional and unnecessary points complicate the calculation of splines, so they are removed by the method `reduceAmountOfPoints`.

9.5 Pathfinding coordinator: Sisyphus

The pathfinding coordinator has the purpose to coordinate the global and a possible (but not implemented) local pathfinding as well as creating splines.
First, the global pathfinding, the RRT is called. The returned path-object contains a path that is most-likely valid. For the improbable case of beeing not valid, the path may be handed to a local pathfinder. Thus the local pathfinder is not implemented at the moment, this step is not executed.
For a better moving behaviour a spline is created based on the calculated path. The algorithm for creating splines is explained in [11].

9.6 Internal pathfinding-datastructures

9.6.1 Node

The datastructure `Node` inherits from `Vector2`, a 2D-datastructure. Additionally, it has the fields `parent` (the Node before this one in the path), `children` (a list of all its successors) and `successor`. The successor is that child among the list of children, that leads to the goal. Since not every Node is part of the resulting Path, this may be `null`.

9.6.2 Path (for internal pathfinding-use

A `Path`-object contains of the following fields:

- `boolean changed`, indicates if the pathfinding made any changes to the path calculated the last circle
- `final int botID`, the bot the path is associated with
- `final List<IVector2> path`, the list of pathpoints
- `List<Node> wayPointCache`, a list of nodes that are likly to be on or near a valid path
- `XYSpline spline`, a spline based on the pathpoints

9.7 Oberserver in Sisyphus

There is just one observer in the module Sisyphus: `onNewPath`. This notifies every observer when a new path has been calculated.

Chapter 10

AIMath

10.1 Basic Functions

10.1.1 Conversions

rad returns the parameter given in degree as radiant.

deg returns the parameter given in radiant as degree.

vector2ToSlope calculates the slope of a given `Vector2`

10.1.2 Other Basic Functions

exp returns the result of `Math.exp(double)` casted to `float`

isZero returns a boolean that indicates if a float value is nearly `0`. Due to the maschine epsilon, a `0` may be stored as a value close to `0`.
The epsilon may be given as optional parameter.

faculty returns the faculty to a given `integer`. This is calculated beforehand and stored within an array. So this function is limited to the interval it has been calculated for

square returns the square of a parameter

cubic returns the cubic of a parameter

10.2 Geometrical Functions

10.2.1 Trigonometrical Functions

To speed up the often used trigonometrical functions the results of 360000 values are calculated beforehand and stored within an array.

sin returns the correspondening `float`-entry of the array mentioned above.

cos returns the correspondening `float`-entry of the array mentioned above.

tan returns the correspondening `float`-entry of the array mentioned above.

acos returns the result of `Math.acos(double)` casted to `float`.

10.2.2 Other Geometrical Functions

distancePP returns the euclidean distance between two given points

distancePPSqr returns the euclidean distance between two given points to the square. This method is faster than `distancePP`

distancePL returns the euclidean distance between given point and line

leadPointOnLine returns the closest point on a line to a given point

angleBetweenXAxisAndLine returns the angle between the x-axis and a given line

getNextPointOnCircle takes a step along a circle and returns the new point

normalizeAngle makes sure the given angle is in the interval $[\Pi; \Pi]$

calculateBisector returns the point where the bisector of an angle within an triangle intercepts the opposite site

intersectionPoint returns the intersection point of two lines

yInterceptOfLine returns the intersection point of a given line and the y-axis

isLineInterceptingCircle returns if a line has any intersection points with a circle

stepAlongLine takes a step along a given line and returns the new point. The step-size is given as parameter as well

10.3 Other Functions

getNearestBot returns the nearest robot to a given point

p2pVisibility returns if the beam between two points is blocked. Individual robots can be excluded by this calculation

getKickerPosFromBot returns the position of the kicker of a given robot. No statement is made about the orientation of that kicker

min returns the minimal value of a given set

Bibliography

[1] Bourke, Paul: *Minimum Distance Between A Point And A Line*. http://paulbourke.net/geometry/pointline/. – [Online: 11.05.2011]

[2] Bruce, James: *Real-Time Motion Planning and Safe Navigation in Dynamic*. http://reports-archive.adm.cs.cmu.edu/anon/2006/CMU-CS-06-181.pdf. – [Online: 15.12.2010]

[3] Gilleland, Michael: *Permutation Generator*. http://www.merriampark.com/perm.htm. – [Online: 30.06.2011]

[4] Gräser, Lukas ; Teichmann, Clemens: *Design and Implementation of the Botmanager and the Skillsystem of the Robocup Team Tigers Mannheim*, Baden-Wuerttemberg Cooperative State University Mannheim, Seminar Paper, 2011

[5] Lee, D. T. ; Wong, C. K.: Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees. In: *ACTA INFORMATIKA* Volume 9 (1978), Nr. 2, S. 23–29

[6] Mannheim, Tigers: *Plays*. http://tigers-mannheim.de/trac/wiki/KI/Pandora/Plays. – [Online: 23.06.2011]

[7] Nakajima, Makoto ; Kimura, Takaumi ; Masutani, Yasuhiro: *ODENS 2010 Team Description*, Osaka Electro-Communication University, TDP, 2010

[8] Perun, Bernhard ; König, Christian: *Design and Implementation of a real-time pathplanning module for holonomic RoboCup Small Size League robots in JAVA*, Baden-Wuerttemberg Cooperative State University Mannheim, Seminar Paper, 2010

[9] Raia, Prof. A.: *Lachesis*. http://www.vroma.org/images/raia_images/fates.jpg. – [Online: 30.06.2011]

[10] Renals, Prof. S.: *Nearest neighbours and kD-trees*. http://www.inf.ed.ac.uk/teaching/courses/inf2b/learnnotes/inf2b-learn06-lec.pdf. – [Online: 15.12.2010]

[11] Universität der Bundeswehr - München: *Spline-Interpolation*. http://www.unibw.de/bauv1/lehre/Ingenieurmathematik/Skriptum/getFILE?tid=Skriptum&fid=1985716. – [Online: 21.06.2011]

[12] Wikipedia: *Ares*. http://en.wikipedia.org/wiki/Ares. – [Online: 30.06.2011]

[13] Wikipedia: *Athena*. http://en.wikipedia.org/wiki/Athena. – [Online: 30.06.2011]

[14] Wikipedia: *Lachesis*. http://en.wikipedia.org/wiki/Lachesis_%28mythology%29. – [Online: 30.06.2011]

[15] Wikipedia: *Metis*. http://en.wikipedia.org/wiki/Metis_%28mythology%29. – [Online: 30.06.2011]

[16] Wikipedia: *Pandora*. http://en.wikipedia.org/wiki/Pandora. – [Online: 21.06.2011]

[17] Wikipedia: *Sisyphus*. http://en.wikipedia.org/wiki/Sisyphus. – [Online: 14.04.2011]