Cooperative State University Mannheim

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Study Report

Development of an Autonomous Referee Software for the Small Size League

by **Lukas Magel**

- Matriculation.: 9273080 -

- Study Course: TINF13ITIN -

TIGERs Mannheim
Small Size League
Robot Soccer Team

Supervisor: Prof. Dr. J. Poller

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from used sources.

Mannheim, July 16, 2016

Contents

List of Figures

List of Tables

List of Abbreviations

SSL	**S**mall **S**ize **L**eague
AI	**A**rtificial **I**ntelligence
TIGERs	**T**eam **I**nteracting and **G**ame **E**volving **R**obot**s**
DHBW	**D**uale **H**ochschule **B**aden-**W**ürttemberg
GUI	**G**raphical **U**ser **I**nterface

1 Introduction

This chapter of the report shall give a brief introduction of the RoboCup in general as well as the Small Size League (SSL).

1.1 RoboCup

The RoboCup is an international scientific initiative to promote robotics and artificial intelligence (AI) research. It was originally founded with the goal of developing a humanoid robotic soccer team that will be capable of competing against the most recent World Cup champion in 2050 [1]. The RoboCup comprises multiple leagues which target different areas of research. The TIGERs Mannheim participate in the Small Size League which focuses on the problem of intelligent multi-agent cooperation and control in a highly dynamic environment [2].

1.2 Small Size League

In the SSL two teams of 6 robots each compete against each other on a rectangular field of 9 m by 6 m using a regular golf ball. All robots must be of cylindrical shape and must not exceed a diameter of 180 mm and a height of 150 mm. The interior of the robots however can be designed freely to the likings of each team as long as the exterior complies with the official rules. The robots are equipped with a kicking device that is capable of accelerating the ball in a straight line or a vertical chip.

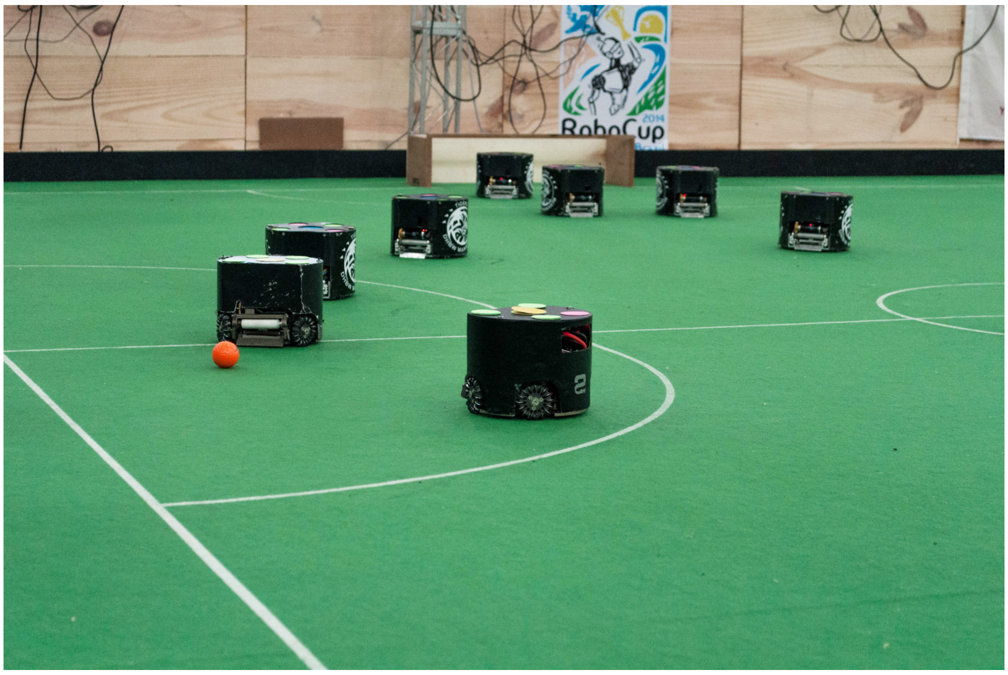

Figure 1.1: SSL robots of different teams with ball

1.3 TIGERs Mannheim

The TIGERs Mannheim is a group of students from the Cooperative State University (DHBW) Mannheim that participates in the RoboCup SSL. The team was founded in 2009 and consists of 30 students from different fields of research. The team members design the mechanical and electronic hardware and develop the artificial intelligence called Sumatra. Sumatra is written in Java and controls the robots during the games. It consists of several modules that fulfill different tasks like processing the vision and referee data, planning the strategy of the team and the paths of each robot and visualizing the decisions of the AI in a graphical user interface (GUI). The source code of Sumatra can be downloaded freely from the team website.

1.4 AutoRef Source Code

As shortcut for anyone interested in the source code of the AutoRef application: It can be downloaded freely from the Gitlab repository on the team website [3].

2 Motivation

Currently all games in the RoboCup Small Size League are controlled by a human referee. He is in charge of enforcing the rules of the game and prosecuting all infringements that occur on the field. Due to the multiplicity and complexity of rules this task can be quite demanding for a single person as he must supervise the actions and positions of all robots on the field simultaneously. One example for this is the regular free-kick scenario that is performed after rule infringements. The referee must ensure that at the time of the kick no robot of the team that performs the kick is positioned too close to the defense area of the defending team while at the same time the defending team must not approach the ball before it has been kicked. Monitoring both conditions at the same time can be quite challenging for a human referee but would be easily accomplished by a computer based referee system that is aware of the locations of the robots on the field. Contrary to other leagues that do not rely on a vision system in the SSL the positions of all entities on the field are available for all parties over the game network. This puts the league in a unique position to use this information for the automated monitoring of rule infringements on the field. A first attempt to pursue this goal was made in 2014 with a technical challenge to implement an autonomous referee that was capable of passively detecting a subset of all possible rule infringements [4].

For the 2016 RoboCup a new technical challenge has been devised to reward the development of an autonomous referee software (AutoRef) that is capable of actively controlling a game [5]. It shall start and stop the game, award goals and detect a specified set of rule infringements. The goal of this work is to develop an autonomous referee software for submission to the RoboCup Technical Challenge. The software should meet all requirements stated in the challenge description and support the human referee in its task of controlling a game. In the following sections each of the infringements listed in the challenge is explained in greater detail. Additionally, this report shall give a detailed insight into the technical details and structure of the implemented software in chapter

4.

Number of Players

Each team is allowed to play with up to six robots during a regular match [6, p. 5]. The autonomous referee must ensure that no team places more than six robots on the field and reduces the number of bots if shown a yellow card.

Pushing/Substantial contact

A robot must not make substantial contact with an opponent. The official rules define substantial contact as follows:

> Substantial contact is contact sufficient to dislodge the robot from its current orientation, position, or motion in the case where it is moving [6, p. 26].

This rule is meant to protect slow moving or stationary robots from being damaged by a fast moving opponent.

Multiple Defender

A robot other than the goalkeeper must not touch the ball while being positioned inside his own defense area. If at the time the ball touches the robot it is positioned partially inside the defense area the team is shown a yellow card [6, p. 25]. If it is positioned entirely within the defense area the opposing team is awarded a penalty kick [6, p. 23].

Attacker in Defense Area

A robot must not touch the ball while being at least partially positioned inside the opponent's defense area or an indirect free kick is awarded to the opposing team [6, p. 24].

Icing

An indirect free kick is awarded to the opposing team if a robot which is located inside his own half of the field kicks the ball such that it crosses the midline and the goal line without touching another robot or entering the goal [6, p. 24].

This rule was added the the official rules to discourage “unintelligent” gameplay by teams which try to directly score goals from a great distance.

Ball Speed

The ball velocity must not exceed 8 m/s when kicked by a robot or an indirect free kick is awarded to the opposing team [6, p. 24]. This rule was added to reduce the advantage of a superior kicking device and to avoid kicks that could potentially harm spectators.

Robot Speed during Stop

During game stoppage the robots are not allowed to move faster than 1.5 m/s or the team will be shown a yellow card [6, p. 25]. This rule was added to prevent robot collisions at high speeds which can occur if robots travel great distances across the field to reposition themselves after the game has been stopped.

Maximum Dribbling Distance

A robot must not dribble the ball for more than 1 m measured linearly from the position where it first touched the ball or an indirect free kick is awarded to the opposing team [6, p. 24]. A robot is considered to dribble the ball when there is no observable separation between the ball and the robot [6, p. 26]. The rule was added in order to not give robots with a superior dribbling device and unfair advantage. The rule does however not prevent a robot from moving greater distances with the ball as long as it periodically separates itself from the ball to give other robots the chance to gain possession of the ball.

Touching the opponent goalkeeper

An indirect free kick is awarded to the opposing team if a robot touches the opponent goalkeeper inside its defense area [6, p. 24]. This rule is meant to protect the goalkeeper from aggressive attackers.

Double Touch

The robot that takes a free kick is not allowed to touch the ball a second time until the ball has touched another robot [6, p. 21]. The official rules further specify the criteria for a double touch as follows:

> It is understood that the ball may remain in contact with the robot or be bumped by the robot multiple times over a short distance while the kick is being taken, but under no circumstances should the robot remain in contact or touch the ball after it has traveled 50 mm, unless the ball has previously touched another robot [6, p. 22].

If it does not abide to the rule the game is stopped and a free kick awarded to the opponent team.

Ball Out of Play

When the ball leaves the field a free kick is awarded to the opposing team of the robot that last touched the ball. If the ball exits over the touch line a Throw-In (Indirect free kick) is performed 100 mm from the point where it left the field. However, if the ball exits over the goal line a goal kick or a corner kick is awarded to the opposing team depending on whether the ball was last touched by the attacking or the defending team.

Attacker to Defense Area Distance

When a free kick is performed the robots of the attacking team must not be positioned too close to the defense area of the defending team at the time the ball is kicked. This rule was added to prevent robots of the attacking teams to position themselves at the defense area and block the defending robots of the other team before the free kick has been performed.

Defender to Free Kick Point Distance

After a free kick command has been issued by the referee only the robots of the team taking the kick are allowed to approach the ball. All robots of the opponent team must remain at least 500 mm from the ball until the kick has been taken [6, p. 29].

Kick Timeout

After the referee signals a command to commence play through a kick the addressed team is given 10 seconds to perform the kick. If the kick is not executed within this time window the referee stops the game and issues a *Force Start* command to allow both teams to approach and touch the ball [6, p. 22].

This rule gives the referee the option to cancel a command if the team is not capable of performing the requested action due to hardware or software issues.

3 Technical Background

3.1 Game infrastructure

The following section shall give a more detailed description of the required infrastructure that is used to conduct the games.

3.1.1 Vision System

Contrary to other leagues where each individual robot is fitted with a vision system to track its surroundings the SSL uses a standardized camera system to locate the robots and the ball on the field. For this purpose each robot is assigned a unique color pattern that is installed on its top. During the game four cameras which are mounted above the field perimeter take periodic images of each quadrant. The resulting stream is then processed by a central computer that detects the position of the robots as well as the ball on each image. The position data of all detected entities is afterwards forwarded to the teams which participate in the game over a wired network. Each team runs its own control computer which further processes the position data, plans the team's strategy and controls their own robots over a wireless link.

3.1.2 Referee

A regular game is led by two human referees which run the game and control the robots by the means of special commands. These commands are pronounced verbally by the referees and entered into a computer program (referee box) by the referee assistant. The current command is then continuously broadcasted over the wired game network by the referee box program. Figure 3.1 depicts the different referee commands as transitions between multiple logical states. These states are a non-binding interpretation of the referee commands as well as the current situation on the field and serve as a means to better explain and name the current state of the game. In every logical state only a subset of all possible commands represents a valid transition to the next state.

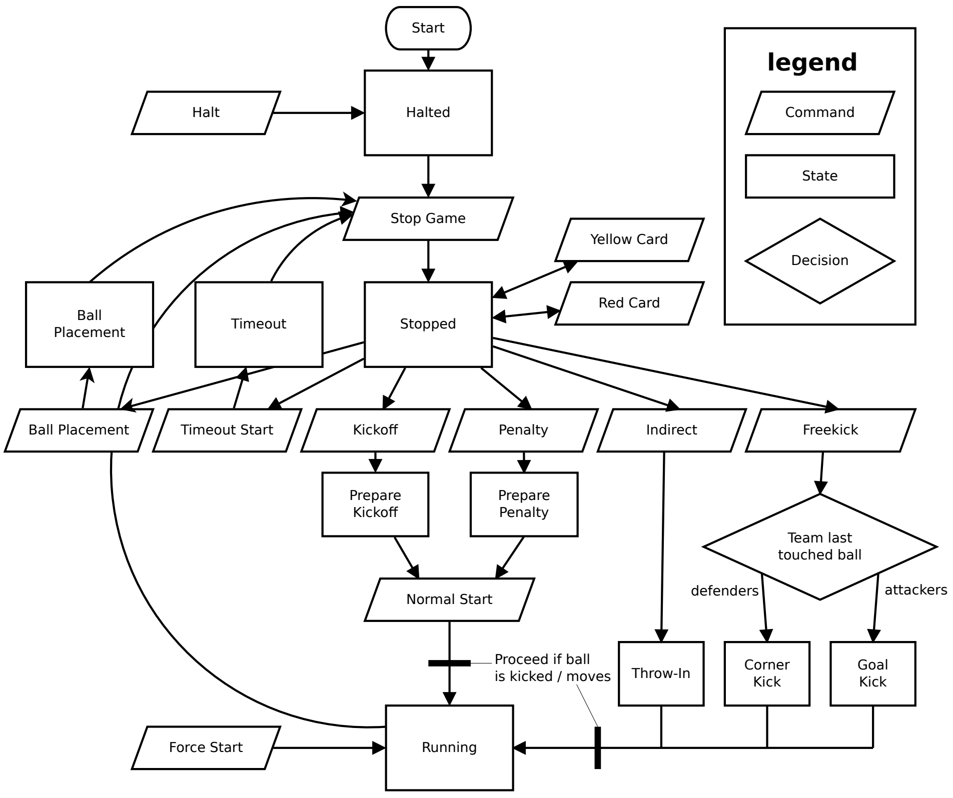

Figure 3.1: Referee commands and game states [7]

A regular game lasts for two halftimes of ten minutes each and always starts in the *Halted* state in which the bots are not allowed to move. When both teams are ready the referee signals a **Stop** command and the game state transitions to the corresponding *Stopped* state. The *Stopped* state serves as starting point for all actions that initiate a play sequence. The first action in a half time is always a kick-off. To execute a kick-off the referee first issues a **Prepare Kickoff** command for the team that will perform the action. After a short wait period the referee signals the start of the kick-off by sending a **Normal Start** command. The game then transitions into the *Running* state as soon as the ball is kicked by the team executing the kick-off. In *Running* state both teams are allowed to score a goal. If the referee detects a legal goal the game is stopped through a **Stop** command and a kick-off is initiated. If however a rule infringement is committed by one of the teams while the game is *Running* the referee stops the game and signals an appropriate action. This action can either be a direct free kick, an indirect free kick or a penalty kick depending on the infringement. This procedure is repeated until the end of the halftime. If no team was able to take the lead and score more goals than the other team the referee can extend the game by two additional halftimes of 5 minutes each or demand a penalty shootout.

3.1.3 Summary

The overall flow of data through all essential systems involved in the game is depicted as pipeline in figure 3.2.

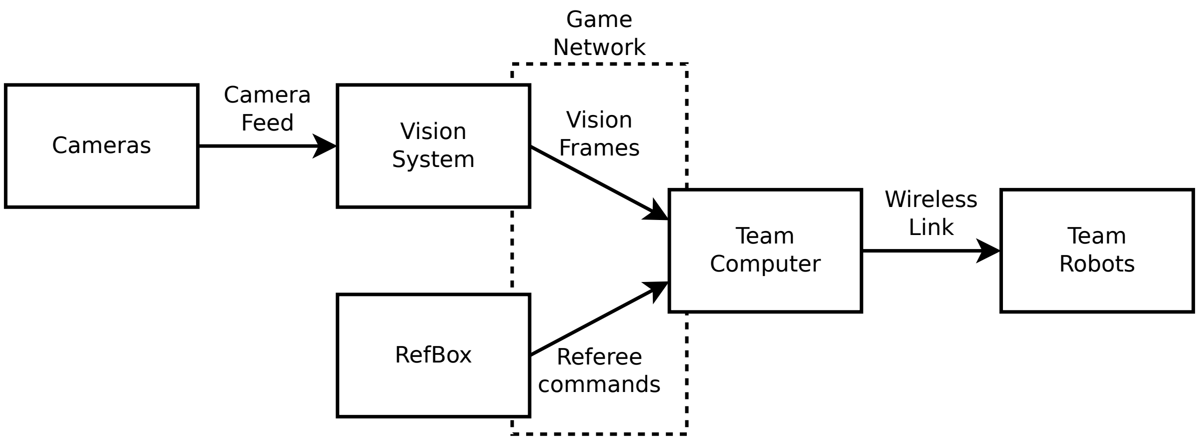

Figure 3.2: Flow of data through all game systems

Both the vision frames and the referee commands use a standardized message format based on the Google Protocol Buffers language. They are broadcasted into the network to be received by the teams. The communication between team computer and robots on the other hand is not standardized and must be implemented by the teams themselves.

4 Implementation of the AutoRef

The following chapter explains the internal structure of the TIGERs AutoRef software.

4.1 TIGERs AI software

The AutoRef software requires several basic building blocks which are already implemented as part of the TIGERs AI software (Sumatra). Because of this the software is not built from scratch but rather shares a common set of functionality with the AI software. For this reason, the Sumatra project is divided into several subprojects which are also used by the AutoRef. Figure 4.1 depicts the AI software components in red which are used as foundation for the AutoRef software.

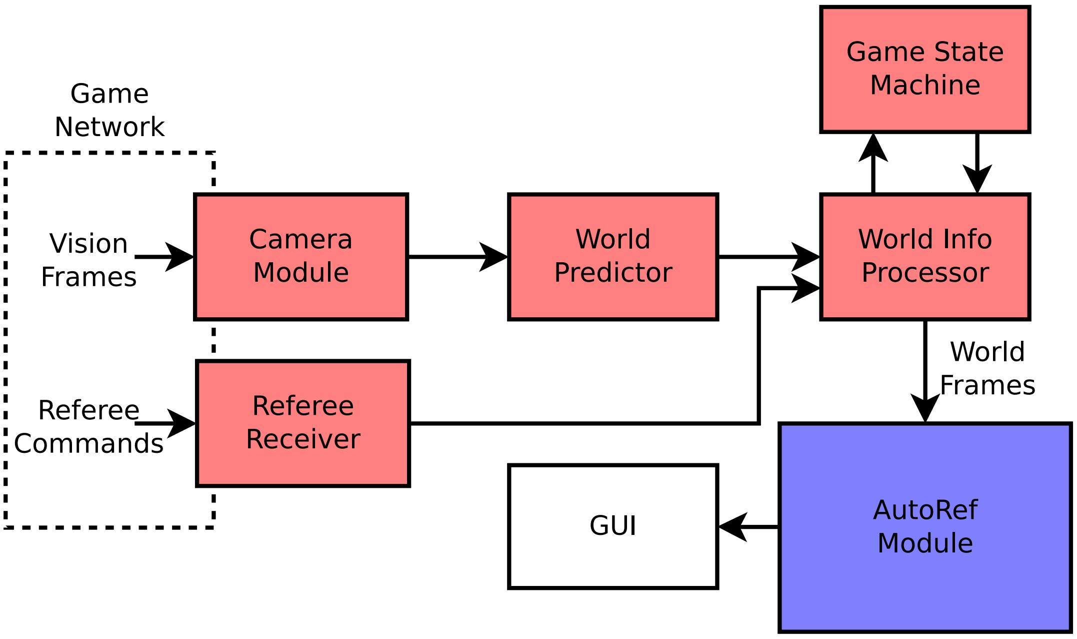

Figure 4.1: AutoRef components shared with the TIGERs AI software

The external vision frames are received by the *Camera Module* and parsed into an internal data structure. They are then passed into the *World Predictor* module which employs a Kalman Filter to clear up the vision data and calculate velocity and acceleration of the moving objects in the frames. The resulting frame is afterwards forwarded to the *World Info Processor* that injects the current referee command into the frame. The *Game State Machine* implements the state graph as shown in figure 3.1 to track the game and derive the current state. The state is also stored in the frame. The resulting *WorldFrames* are used by the AutoRef module which shall be explained in greater detail in the next section.

Besides the processing pipeline the AutoRef also uses the GUI framework from Sumatra. It simplifies the creation and reuse of individual views that are displayed in a modular frame.

4.2 Overall structure

Figure 4.2 depicts the internal structure of the AutoRef software.

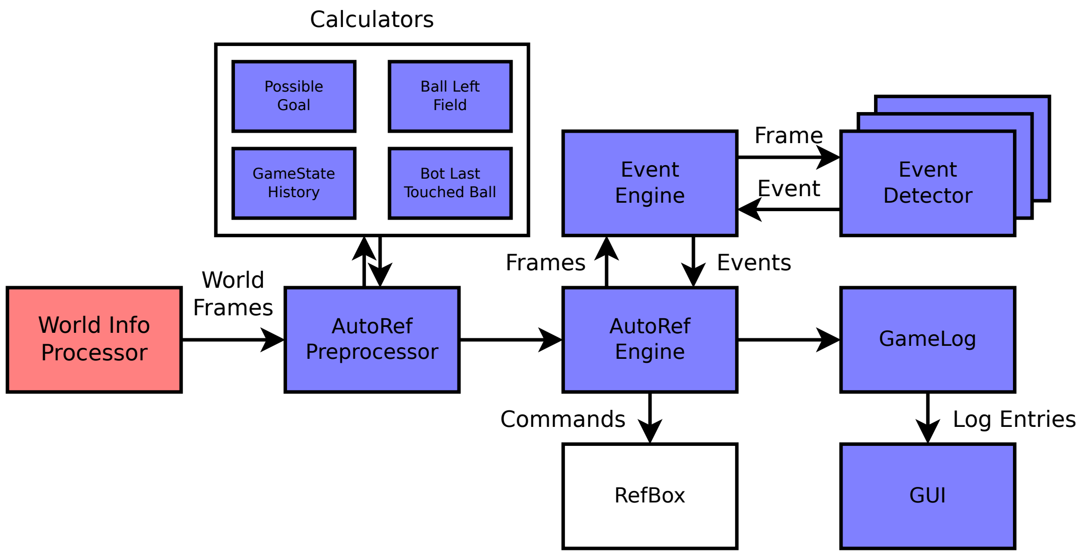

Figure 4.2: AutoRef Structure

In the *AutoRef Preprocessor* the *WorldFrames* which are received from the *World Info Processor* are enriched with additional data that is calculated in *Calculator* instances. The purpose of each of the calculators is explained in further detail in section 4.3.

The *WorldFrame* is then passed into the *AutoRefEngine* which itself feeds the frame into the *EventEngine* for processing. The *EventEngine* detects events like goals or rule infringements that occur on the field and returns them to the *AutoRefEngine*. The AutoRef Engine knows an active as well as a passive mode. The passive mode can be used if the game should be led by a human referee with passive support by the AutoRef. The AutoRef will then merely log the detected events to the *GameLog* that is displayed as part of the GUI. The active engine however serves as a partial replacement for the human referee and actively reacts to the detected events by sending referee commands to the referee box in order to alter the game state accordingly.

In the following sections each of the subcomponents of the AutoRef will be explained in greater detail in the order in which they appear in the processing pipeline.

4.3 Calculators

Calculators are mostly stateless components that enrich the *WorldFrame* with additional data which is calculated from the *WorldFrame* itself. They are set as the first step in the processing pipeline because the calculated data is needed in several subsequent components like the *EventEngine* or the *AutoRefEngine*.

Possible Goal Calculator

As part of its regular operation the AutoRef must be able to detect goals. The simplest approach would be to simply award a goal whenever the ball is located inside the rectangle that constitutes a team's goal. However, since the ball can also be chip kicked over the goal the position itself does not provide a sufficient condition to award a goal. For this reason, the *Possible Goal Calculator* uses a different approach to detect goals and inform later components about a goal by setting the detected goal as value in the *WorldFrame*. In order for it to detect a goal one of two sufficient conditions must be fulfilled:

- The ball is located inside the goal and its velocity is equal to zero
- The ball changes its heading by at least α_{min} while being located inside the goal

If one of both conditions evaluates to true, the calculator stores the team color of the goal as well as the current timestamp in each *WorldFrame* until the ball exits the goal perimeter. The detection logic is implemented as a calculator to ensure that all components which use this information are informed synchronously about a detected goal.

Ball Left Field Calculator

The *Ball Left Field Calculator* tracks the ball and whenever the ball leaves the field it calculates the position on the boundary line where the ball left the field.

The calculator keeps a history of the last ball positions. When the ball leaves the field it uses the current position as well as the last position of the ball inside the field to interpolate the intersection of field boundary and ball heading. This information is stored as attribute in the *WorldFrame* and used by other components to determine what action to take depending on whether the ball exited the field through the touch or the goal line.

Game State History Calculator

The *Game State History Calculator* maintains a list which contains the last game states. This information helps other components to react differently depending on which game states the game was in before the current state.

Last Ball Contact Calculator

For several rule infringements, like *Ball Out Of Play*, it is necessary to determine the robot that last touched the ball in order to correctly detect the infringements and derive the proper next action for the AutoRef. For this purpose, the AutoRef features a *Last Ball Contact Calculator* which determines the robot that last touched the ball and stores its ID along with the timestamp at which the contact occurred in the *WorldFrame*.

To detect ball contacts the calculator monitors changes in the ball heading $\vec{v}_b(t)$ between two consecutive frames f_i and f_{i+1} with time difference Δt. It assumes that a contact has occurred if the change is greater than a certain threshold α_{min}:

$$\vec{v}_b(t) \sphericalangle \vec{v}_b(t + \Delta t) > \alpha_{min}$$

It then tries to find candidates in the vicinity of the position of the ball $\vec{p}_b$ and picks

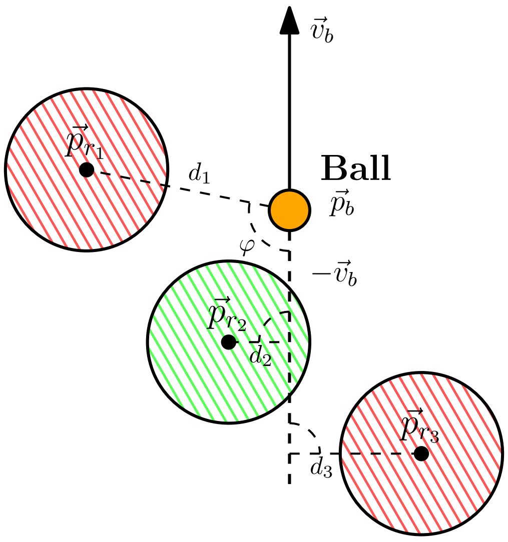

Figure 4.3: Selection of candidates

the most suited robot i with position $\vec{p}_{r_i}$. It only considers robots that are positioned "behind" the ball i.e. fulfill the following condition:

$$(\vec{p}_{r_i} - \vec{p}_b) \sphericalangle (-\vec{v}_b) < 90°$$

Figure 4.3 outlines an exemplary situation with three robots. Robot 1 is not positioned behind the ball ($\varphi > 90°$) and is therefore not considered as candidate.

As second criteria the robots must be positioned close enough to the imaginary line $\vec{l}_b = \vec{p}_b + \lambda \cdot \vec{v}_b$:

$$distance(\vec{p}_{r_i}, \vec{l}_b) < r_{bot}$$

Of the robots 2 and 3 in figure 4.3 only robot 2 fulfills the second criteria. If more than one robot fulfills all criteria then the algorithm selects the robot that is positioned closest to the ball.

The proposed algorithm delivers good results in a simulation as well as during a regular game. It does however detect false positives when the ball is chip kicked as the flight curve follows an arc when projected as 2D vision data. The algorithm interprets this arc as change in ball heading and detects false ball contacts for all robots that the ball passes by.

4.4 Event Engine

The *EventEngine* is responsible for detecting events that occur during a game. These events can either be rule infringements as the ones stated in chapter 2 or events like a goal that demand a change of game state. The AutoRef decouples the detection of different events in the game from the components that run the game to facilitate an easier reuse. The event engine is used for both the active as well as the passive mode. During regular operation the engine is continuously feed with *WorldFrames* and returns a list of all events that it detected in each frame. The logic to detect one, or in some cases, multiple different event types are encapsulated in one *EventDetector* component. The *EventEngine* serves as container for all detector instances and processes each one of them with every *WorldFrame* it receives.

In order to give an insight into the workings of the event engine the following sections explain what an event is comprised of, outline how the detector instances are invoked by the event engine, and finally illustrates how each of the detector components is implemented.

4.4.1 Events

An event is a generic data type to describe a situation or incident that occurred on the field. It contains the following attributes:

Attribute	**Optional**	**Description**
Type	Required	Type of the event like *Goal* or *Ball Left Field*
Responsible Team	Required	Team which was responsible for the event
Responsible Robot	Optional	Robot responsible for the event
Next Action	Optional	Action that should be taken to handle the event
Card Penalty	Optional	Card that should be shown to the responsible bot

Table 4.1: Attributes of an event

Each event is given a unique global type to distinguish it from other events and describe the detected condition. Appendix A lists all possible event types. As second attribute an event must always be assigned a team that was responsible for triggering it. For some events it can make sense to also specify the exact robot that triggered it. In the case of a *Ball Left Field* event the responsible robot would be the robot that last touched the ball before it exited the field. For other events like the *Bot Count* event it does not make sense to specify a robot. The Next Action attribute describes the action that the AutoRef should take in case it accepts the event. A *Ball Left Field* event would specify an indirect free kick as next action. The Card Penalty attribute optionally specifies a yellow or red card that should be shown to the responsible robot if present. The logic to detect each event type is encapsulated in multiple detector components.

The *Next Action* attribute of each event is modeled through another data type called *FollowUpAction*. It specifies what action to take next and contains the following attributes:

Attribute	**Optional**	**Description**
Action Type	Required	Type of the action like *Kickoff* or *Indirect Free Kick*
Team in Favor	Required	Team which will perform the action
New Ball Pos	Optional	Where to place the ball before initiating the action

Table 4.2: Attributes of a FollowUpAction

The new ball position is set as optional attribute as the ball position does not need to be specified for some action types like the penalty kick or the kickoff.

4.4.2 Detector Processing

Each detector component is assigned one or multiple game states in which it is active and is only processed when the game is in one of these states. This simplifies the implementation of each detector component as it must not track the state of the game itself. For example, the detector type which monitors the ball velocity for an infringement does only need to be active in the *Running* state of the game. Additionally, all detector components must implement a reset method that is invoked by the *EventEngine* when the game transitions from a game state in which the detector component should be inactive to one of its active game states. Upon invocation the detector component should discard any state that it has stored. As third requirement each detector component is assigned an ascending priority to define which detector should be processed first.

When the *EventEngine* receives a frame for processing, it first determines all detector components which are active in the game state of the frame and sorts them by their priority. It then invokes the reset method on all detector components that have become active with the current frame. Finally, the frame is forwarded to each detector component for processing. The component may optionally return an event that it detected in the current frame. All detected events are stored and returned to the caller. The entire process is outlined as pseudocode in listing 4.1.

```
processFrame(frame, lastFrame):
    detectors := getActiveDetectors(frame.gameState)
    sortByPriority(detectors)
    events := empty list
    for detector in detectors:
        if not detector.isActiveIn(lastFrame.gameState):
            detector.reset()
        result := detector.process(frame)
        if result exists:
            events.append(result)
    return events
```

Listing 4.1: Event Engine Loop

4.4.3 Implementation of the Detector Components

In the following sections each of the detector components is explained in greater detail where required along with the difficulties associated with detecting each infringement.

Ball Left Field Detector

The *Ball Left Field Detector* component fires an event whenever the ball exits the field while the game is in the *Running* state. It covers throw-ins, goal kicks, corner kicks as well as the Icing rule. Figure 4.4 shows the chain of decisions that the detector makes with every frame to determine the event that it emits. It uses the result of the *Last Ball Contact Calculator* to determine which team should be granted the free kick. If the blue team was the last team to make contact the yellow team will be awarded the free kick and vice versa.

One problem that must be solved for all detector components is their inherent statelessness regarding the continuous emission of new events. In the case of the *Ball Left Field Detector* this statelessness would cause it to continuously emit a new event for every frame in which the ball is located outside the field. This problem is solved for each detector component individually however they all use some kind of cool down mechanism to prevent this behavior. The *Ball Left Field Detector* component for example emits a

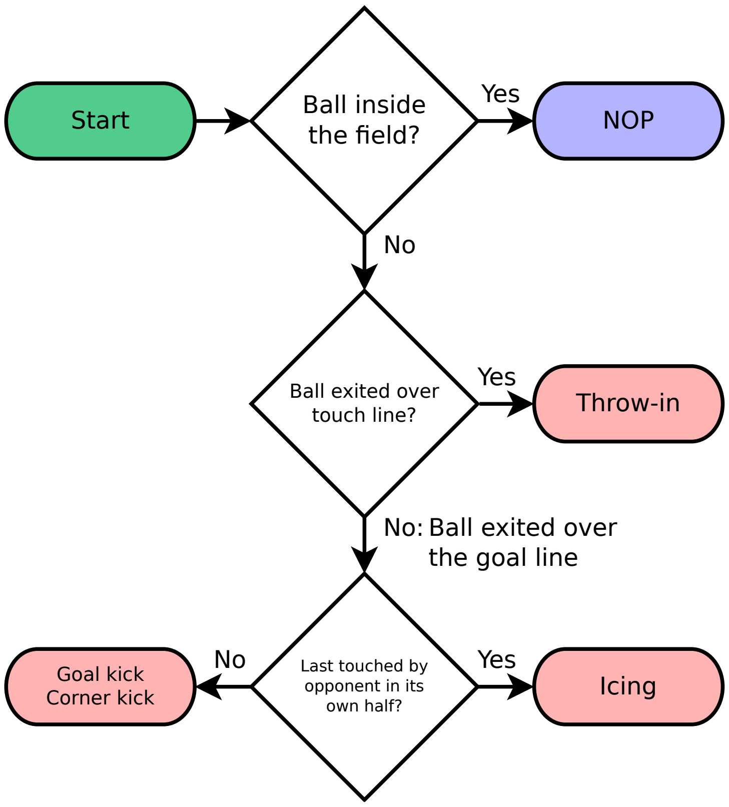
Start
Ball inside the field?
Yes
NOP
No
Ball exited over touch line?
Yes
Throw-in
No: Ball exited over the goal line
Last touched by opponent in its own half?
No
Goal kick
Corner kick
Yes
Icing

Figure 4.4: Chain of decisions for the Ball Left Field Detector

single event whenever the ball crosses the field boundary but does not report another event until the ball has reentered the field.

Detecting when the ball has left the field perimeter is rather easy to accomplish. It is however difficult to correctly determine which team last touched the ball. This information is necessary to determine if the rule regarding Icing has been violated and which team will be awarded the free kick. The *Last Ball Contact Calculator* can correctly determine the robot which last touched the ball in most cases but does not work reliably when the ball is chip kicked due to the reasons described in section 4.3.

Attacker to Defense Area Distance Detector

This detector component tries to find all robots of the attacking team which are located too close to the defense area of the defending team during a free kick as described in chapter 2.

To detect robots which violate this rule the detector triggers on the change of game state from a an *indirect kick* or *direct kick* state to the *Running* state as at this point the robot which performs the free kick has touched the ball and therefore taken the kick. It then determines all robots which are located in a 200 mm perimeter around the defense area and returns an event for the first violator that it detects. The detection of violations regarding this rule does not pose any greater difficulties as the position of each of the bots can be determined directly from the vision frames returned by the vision system.

Attacker Touches Keeper Detector

The *Attacker Touches Keeper Detector* implements the corresponding rule infringement described in chapter 2. It monitors the distance of all robots of one team to the goalkeeper of the other team and fires an event if the distance drops below a certain threshold while the goalkeeper is positioned inside its defense area. Violators are stored in a cool down list along with the timestamp of the event. The detector does not report another event for the same robot until the cool down timeout has expired in order to reduce the noise caused by duplicate events.

Ball Velocity Detector

This detector component monitors the ball for a speed violation. If the ball velocity exceeds a maximum threshold for multiple frames in a row the detector emits an event for the robot that last touched the ball. The ball velocity in each frame is automatically calculated by the world predictor. The detector triggers only once until the ball velocity has dropped below the critical threshold.

The detector implementation is rather simple since the velocity is automatically calculated by the world predictor. It does however sometimes emit false positives if the ball jumps between multiple locations due to inaccurate vision data as the world predictor calculates an abnormally high velocity to match these jumps.

Bot Collision Detector

The official set of rules penalizes substantial contact between robots of opposing teams but only gives a vague definition of the term itself (see chapter 2 for more information) that is more suited for interpretation by a human referee than by an algorithm. Due to the lack of a more precise definition this AutoRef uses a rather simple approach which should be considered a first attempt at implementing an algorithm to detect the required rule infringement. It is deemed in no way complete nor entirely correct. The implementation is described in the following paragraphs.

For every possible pair of robots of both teams $(b_i, y_j) \in R_b \times R_y$ the detector calculates the distance between them:

$$d = |\vec{p}_{b_i} - \vec{p}_{y_j}|$$

If the distance is smaller than a certain threshold d_{min} the detector assumes that the robots will collide. For the correct calculation of the crash velocity it is crucial that the collision is detected before the physical impact occurs as the collision would falsify the velocities of both robots.

To determine if the collision satisfies the requirements of a substantial contact the detector then calculates the absolute impact velocity v_c of both robots. This velocity must exceed a certain maximum threshold $v_{c_{max}}$ for the contact to count as collision. Figure 4.5 outlines such a scenario where robot b_i with velocity $\vec{v}_{b_i}$ is about to collide with robot y_j with velocity $\vec{v}_{y_j}$.

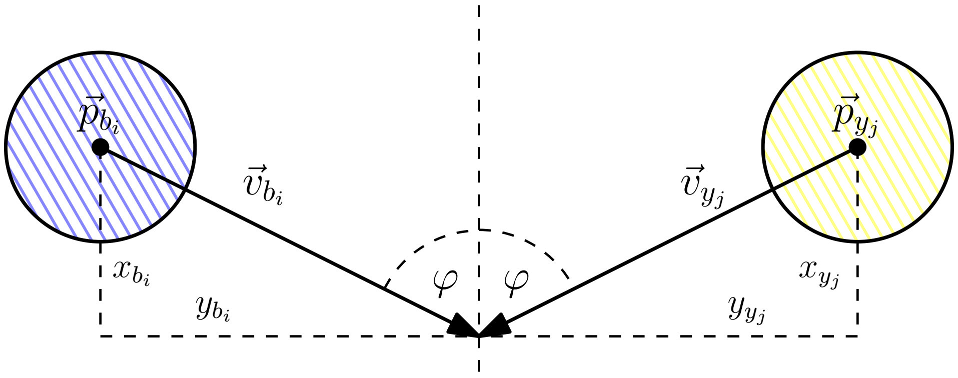

Figure 4.5: Bot Collision Speed Calculation

In order to calculate the impact velocity both vectors are split into their perpendicular components (x_{b_i}, y_{b_i}) for b_i and (x_{y_j}, y_{y_j}) for y_j. The impact velocity is then calculated as:

$$\begin{aligned} v_c &= |x_{b_i} - x_{y_j}| + (y_{b_i} + y_{y_i}) \\ &= |\cos(\varphi) \cdot |\vec{v}_{b_i}| - \cos(\varphi) \cdot |\vec{v}_{y_j}|| + (\sin(\varphi) \cdot |\vec{v}_{b_i}| + \sin(\varphi) \cdot |\vec{v}_{y_j}|) \\ &= \cos(\varphi) \cdot ||\vec{v}_{b_i}| - |\vec{v}_{y_j}|| + \sin(\varphi) \cdot (|\vec{v}_{b_i}| + |\vec{v}_{y_j}|) \end{aligned}$$

As second condition the difference in absolute robot velocity must be greater than a certain threshold Δv_{min}:

$$|\vec{v}_{b_i} - \vec{v}_{y_j}| > \Delta v_{min}$$

If both conditions evaluate to true, the *Bot Collision Detector* fires a collision event and sets the robot with the higher velocity as responsible robot.

The algorithm described above has proven to correctly detect robot collisions in multiple field tests. It is however difficult to correctly tune the parameters $v_{c_{max}}$ and Δv_{min} as there is no clear definition of when a contact between two robots qualifies as collision.

Bot In Defense Area Detector

This detector component covers the Multiple Defender rule as well as the rule regarding the presence of attackers in the defense area of the defending team as the logic to detect both infringements are similar. The detector triggers on the value of the last ball contact field in the *WorldFrame* that is calculated by the *Last Ball Contact Calculator*. Whenever the value changes it evaluates if the contact position of the robot lies inside the defense area. The detector differentiates full and partial penetration of the defense area as shown in figure 4.6 where r_b constitutes the robot radius.

A robot is considered to be positioned fully inside the defense area if its center point is located inside the red area as shown by robot a with position $\vec{p}_a$. A robot is considered to be positioned partially inside the defense area if its center point is located inside the yellow area but not inside the red area as shown by robot b with position $\vec{p}_b$.

The detector punishes each contact only once. After an event has been reported for a robot the detector waits a certain cool down time before reporting another event for the same robot in case two ball contacts are detected for the same robot in fast succession.

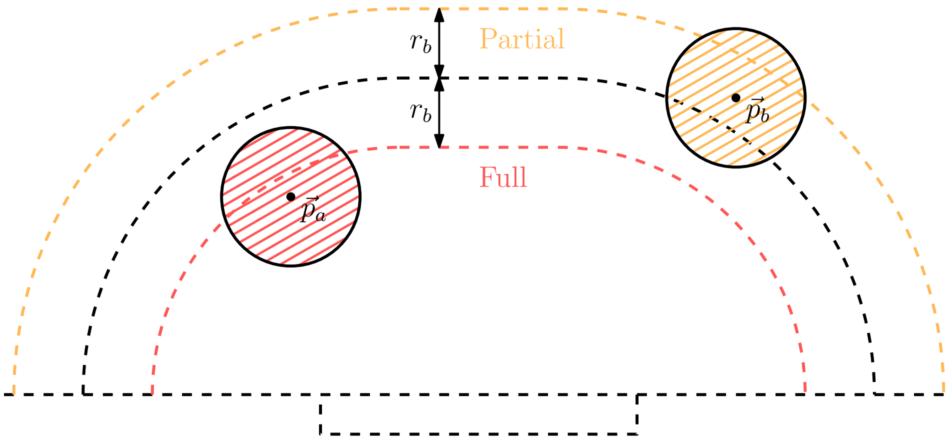

Figure 4.6: Robots located fully (a) and partially (b) inside the defense area

The detector knows three different infringements that are listed in table 4.3.

Infringement	**Defense Area**	**Position**	**Punishment**
Multiple Defender Partial	Own	Partial	Yellow Card
Multiple Defender Full	Own	Full	Penalty Kick
Attacker In Defense Area	Opponent	Partial/Full	Indirect Free Kick

Table 4.3: Bot In Defense Area Punishments

The exact implementation as stated above reacts very sensitive to situations where a robot partially enters the defense area by only a few millimeters. While the decision of the detector is technically correct it unnecessarily interrupts the game and impairs the viewing experience. For this reason, the detector implementation uses a slightly smaller yellow margin in order to avoid too sensitive decisions.

Bot Count Detector

The *Bot Count Detector* ensures that the number of robots on the field matches the maximum allowed robot count. During a regular game each team is allowed to play with up to 6 robots. However, if a team is shown a yellow card the number of robots must be reduced by 1 for a duration of 2 minutes. The number of active yellow cards is published by the referee box along with the referee commands.

The *Bot Count Detector* continuously counts the number of robots on the field in every frame and compares the result with the allowed number of robots. The allowed number of robots is calculated by subtracting the number of active yellow cards from the default number of 6 robots. In case the robot count for one team is higher than the maximum allowed value the detector emits a single event. It does not report another event for the same team until the difference between allowed and actual bot count has changed again.

Bot Stop Speed Detector

The *Bot Stop Speed Detector* component must ensure that no robot exceeds the speed limit during game stoppage. For this reason, it continuously tracks the velocity of each robot on the field as calculated by the *WorldPredictor*. A robot is considered to violate the rule if it maintains a velocity greater than the speed limit for a certain amount of time t_{max} (300 ms in the implementation). Every robot is given t_{max} as quota q. Whenever the velocity of a robot exceeds the speed limit in a given *WorldFrame* f_i the time delta between the current and the last frame $\Delta t = t_{f_i} - t_{f_{i-1}}$ is subtracted from the robot's quota $q = q - \Delta t$. However, if the velocity of a given robot does not exceed the speed limit Δt is added to its quota $q = min\{q + \Delta t | t_{max}\}$. If the quota reaches 0 for a certain robot an event is emitted and the robot is put on a cool down list. The detector does not report another event for the same robot until its quota has been restored and it has subsequently been taken off the cool down list.

The rule regarding the maximum velocity during game stoppage is not strictly enforced by the human referee in general unless the robots of a team clearly and visibly disregard the maximum velocity. Subsequently teams do not strictly abide by the rule and might temporarily exceed the speed limit for a short amount of time. If the AutoRef were to punish all infringements of this rule a team might be forced to remove all robots from the field due to the large amount of yellow cards issued over a short period of time. For this reason, the events emitted by this detector are currently not acted upon by the active AutoRef engine and are merely logged to the Game Log. It remains to be shown in future games if the detector will ever be actively used.

Defender To Free Kick Point Distance Detector

During a free kick or kick-off situation the *Defender To Free Kick Point Distance Detector* monitors the direct vicinity around the ball position in order to issue an event if a robot of the opponent team enters the circular area 500 mm from the ball. This can in some cases lead to an interrupted gameplay as some teams do not strictly follow this rule and robots partially cross the circular area on their path without actually intending to touch the ball. This behavior is normally not punished by a human referee unless a robot actually touches the ball or impairs the robot taking the kick. For this reason, the detector divides the area into an inner and an outer circle with a radius of 250 mm and 500 mm. A robot of the opponent team may linger in the outer circle for a total time of 3 seconds before an event is emitted. However, if a robot of the opponent team enters the inner circle an event is reported right away.

Double Touch Detector

The official rules state that a robot may not touch the ball a second time after the free kick has been taken and the ball has moved 50 mm. As this very strict definition may lead to an overly sensitive detector the implementation of the rule as described below uses a slightly different approach to detect a double touch that is shown in figure 4.7.

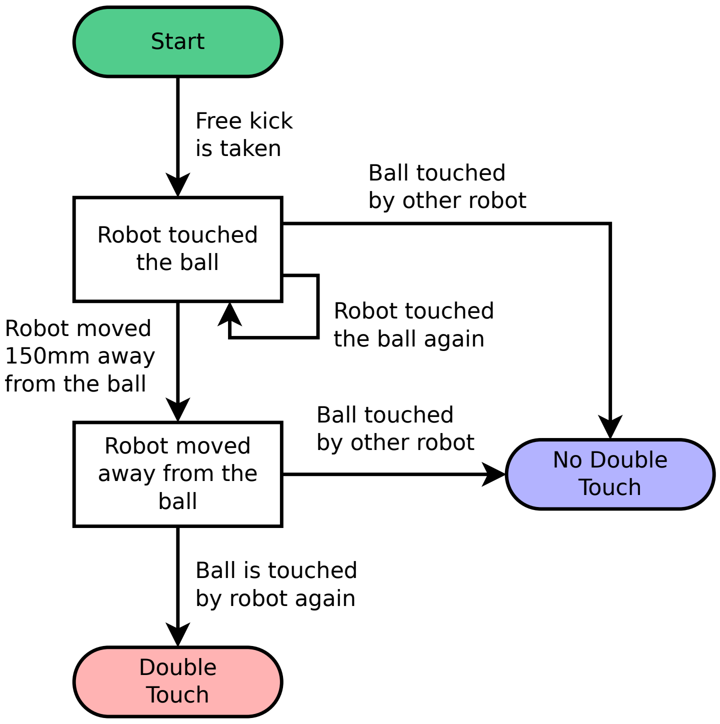
Start
Free kick is taken
Ball touched by other robot
Robot touched the ball
Robot touched the ball again
Robot moved 150mm away from the ball
Robot moved away from the ball
Ball touched by other robot
No Double Touch
Ball is touched by robot again
Double Touch

Figure 4.7: State machine of the Double Touch Detector

The detector triggers on the game state transition to *Running* and stores the ID of the robot that last touched the ball in that frame as it considers that robot to have taken the free kick. It then constantly monitors the distance between the robot and the ball and assumes that the robot has separated from the ball if there is a measurable distance of 150 mm between both. If the robot then touches the ball a second time, an event is emitted to signal a double touch. However, if a different robot comes in contact with the ball the detector becomes inactive.

Dribbling Detector

The *Dribbling Detector* works in a similar fashion as the *Double Touch Detector*. It also employs a state machine as shown in figure 4.8 to detect dribbling when the game is running. Its initial transition is triggered by a change of the Last Ball Contact value calculated by the *Last Ball Contact Calculator*. Whenever a new robot touches the ball the detector stores the position of the contact and continuously evaluates the following two conditions which must evaluate to true in order for the detector to signal a *Dribbling* infringement.

1. Necessary condition: Robot remains in close contact to the ball
2. Sufficient condition: Robot travels 1000 mm from the first contact position

The first condition must evaluate to true until the second condition is also true. The robot is considered to stay in close contact to the ball as long as the distance between both remains below a certain threshold d_{min} which is set to 100 mm in the implementation. However, if the ball is touched by a different robot or the distance does not remain below the threshold the detector is reset. The total travel distance for condition two is determined by calculating the distance between the current position of the robot and the stored position of the first ball contact. If the total travel distance is greater than 1000 mm the detector emits a *Dribbling* event to signal the infringement. The detector is then reset to its initial state.

Goal Detector

The *Goal Detector* is responsible for signaling the detection of valid or invalid goals. A goal is considered invalid if it was shot directly from an indirect free kick [6, p. 28]. The actual goal detection mechanism is implemented in the *Possible Goal Calculator* as explained in section 4.3. The detector uses the value of the *Possible Goal* field in the

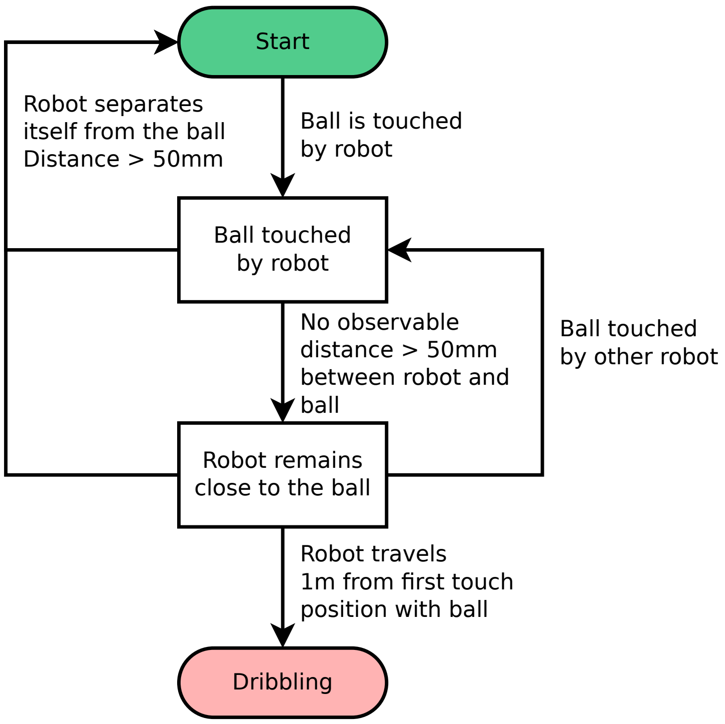
Start
Ball is touched by robot
Ball touched by robot
Robot separates itself from the ball Distance > 50mm
No observable distance > 50mm between robot and ball
Ball touched by other robot
Robot remains close to the ball
Robot travels 1m from first touch position with ball
Dribbling

Figure 4.8: State machine of the Dribbling Detector

World Frame as trigger to emit a *Goal* event if the goal was valid or an *Indirect Goal* event if the goal was not valid. Figure 4.9 outlines the different states of the detector.

In order for a goal to be considered an indirect goal the following two conditions must apply:

1. The play sequence was started with an Indirect Free Kick
2. The ball has only been in contact with the robot that executed the free kick when it enters the goal

To monitor if the ball was touched by a different robot the detector stores the ID of the robot that executed the free kick and continuously compares it to the IDs of all robots that touch the ball before it enters the goal. If at some point the ball is touched by a robot with a different ID the detector sets an internal flag which indicates that the necessary conditions do not apply anymore and any future goal must therefore be valid. When the ball finally enters the goal it consults the internal flag and only emits an *Indirect Goal* event if the flag is not set. However, if the flag is set the detector emits a valid *Goal* event.

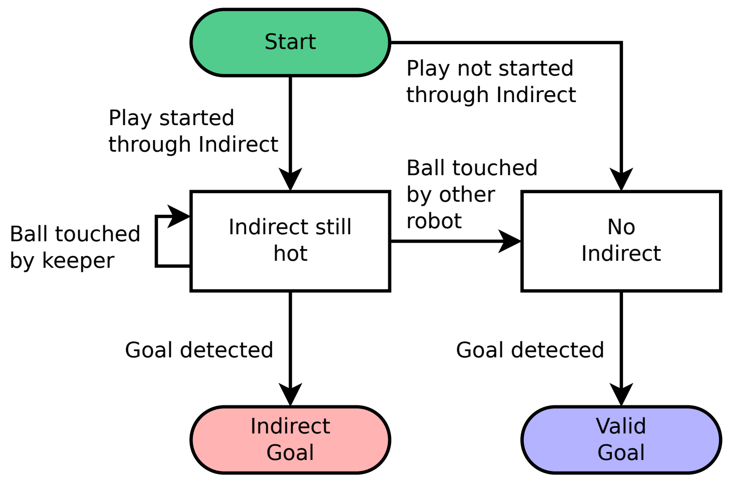

Figure 4.9: State machine of the Goal Detector

Kick Timeout Detector

The *Kick Timeout Detector* emits a *Kick Timeout* event if a team takes more than 10 seconds to execute a kick. It becomes active after the referee signals a *Normal Start*, an *Indirect Free Kick* or a *Direct Free Kick* command and stores the timestamp of the first frame that it receives after its reset. It uses the timestamp to calculate the absolute amount of time spent by the team to perform the kick. If the team takes more than 10 seconds before it touches the ball the detector signals a timeout by emitting a *Kick Timeout* event.

4.5 AutoRef Engine

The AutoRef engine contains the brain of the autonomous referee software. It knows two different implementations to provide a passive as well as an active mode. In the following two sections each of the two implementations is outline and explained in greater detail where applicable.

4.5.1 Active AutoRef Engine

The active AutoRef engine is in its basic building blocks very similar to the event engine but has been decoupled from it in order to better reuse the event engine for the passive implementation. Its main purpose is to react to game events and send appropriate commands to the referee box. In order to accomplish this goal the engine is divided into multiple states that govern different parts of the game. The AutoRef states are directly linked to the state of the game as calculated by the *Game State Machine* in a one to many relationship. This means that one AutoRef state can be active in multiple game states but not more than one AutoRef state can be active at the same time. If the selection process for the current state was modeled by a function $f : g \to a$ where g is the current game state and a represents the selected AutoRef state then f would be surjective. This ensures that the AutoRef is solely dependent upon the game state and can react to unexpected game state changes since the different parts are decoupled from one another and store as minimal state as possible.

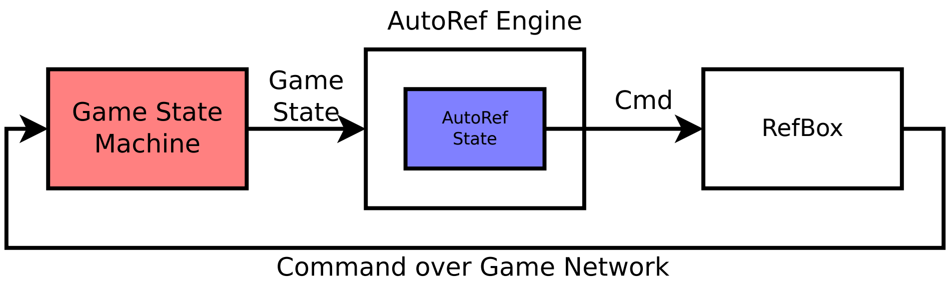

Figure 4.10: AutoRef feedback loop

The AutoRef changes its state by sending commands to the referee box. Figure 4.10 depicts this feedback loop between AutoRef and referee box. The current active AutoRef state sends a command to the referee box and subsequently triggers a change of game state.

Listing 4.2 outlines how the engine processes a *WorldFrame*. First the engine determines the currently active AutoRef state from the game state. It then forwards the frame to the event engine for processing and stores the events in a list. It is the responsibility of the AutoRef state to handle the events that were detected. This way events can be handled differently depending on the state the game is currently in.

```
processFrame(frame, lastFrame):
    autoRefState = getActiveState(frame.gameState)
    events = eventEngine.processFrame(frame, lastFrame)

    autoRefState.handleGameEvents(events)
    autoRefState.processFrame(frame, lastFrame)
```

Listing 4.2: AutoRef Engine loop

The AutoRef state handles the events by first selecting the event with the highest priority if the event engine detected more than one event. It then sends a *STOP* command if the game is not already in the *Stopped* state and sets the *FollowUpAction* of the event as current pending action. The AutoRef Stop state will then try to fulfill the pending action by sending corresponding commands to the referee box in subsequent process runs as soon as the game state has switched to Stop and it becomes active. The pending action is reset when the game changes to *Running*.

In the following sections each of the different AutoRef states is described in greater detail.

AutoRef Stop State

The Stop state performs all necessary actions to reinitiate a play sequence whenever the game transitions into the *Stopped* state under the condition that a *FollowUpAction* is set. Figure 4.11 depicts the checks the state performs to determine which command to send.

The Stop state requires a set *FollowUpAction* to perform an action. If none is set it simply idles until the game state changes due to an external referee command. If an action is set it always waits a certain amount of time before performing an action to give all robots time to settle after the game transitioned into the *Stopped* state. It then determines the target position of the ball for the next action by consulting the *New Ball Pos* field of the *FollowUpAction* and waits until the ball has come to a stop. If the ball is not positioned correctly it either sends a Placement command (see [8] for more information on the ball placement) or waits for the referee to place the ball at the desired position. When the ball is placed correctly and all robots maintain the correct distance from the ball it sends the referee command to initiate the next action.

AutoRef Prepare Kickoff State

The Prepare Kickoff state has the responsibility to send the *Normal Start* command to initiate a kick-off when the robots as well as the ball are positioned correctly. Figure 4.12 depicts the different conditions the state evaluates before it sends the command.

Like the Stop state it also waits a certain amount of time (3.5 s in the implementation) before performing any checks. It then checks if the ball is still placed on the center point

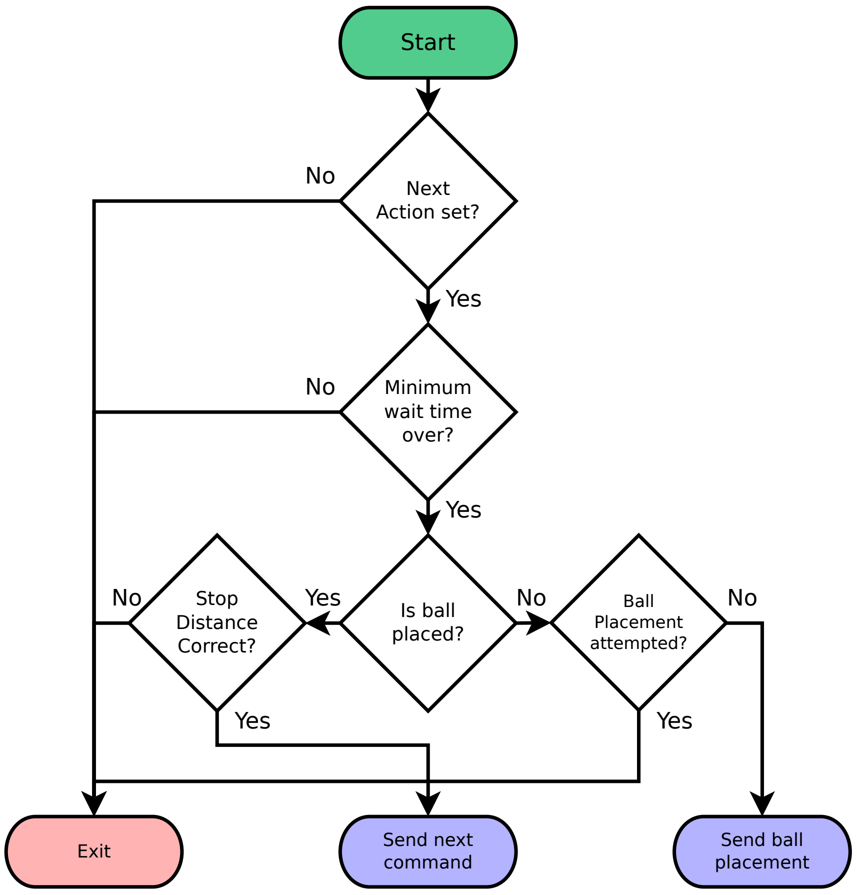

Figure 4.11: Chain of decisions for AutoRef Stop State

and waits until all robots have taken up positions in their own half of the field before sending the *Normal Start* command.

AutoRef Prepare Penalty State

The Prepare Penalty state works in a fashion very similar to the Prepare Kickoff state. It also sends a *Normal Start* command as soon as all robots have cleared the penalty kick area and a shooter has taken up position behind the ball.

AutoRef Ball Placement State

The Ball Placement state is responsible for monitoring the progress of a team during ball placement. It becomes active whenever the Stop state issues a ball placement command

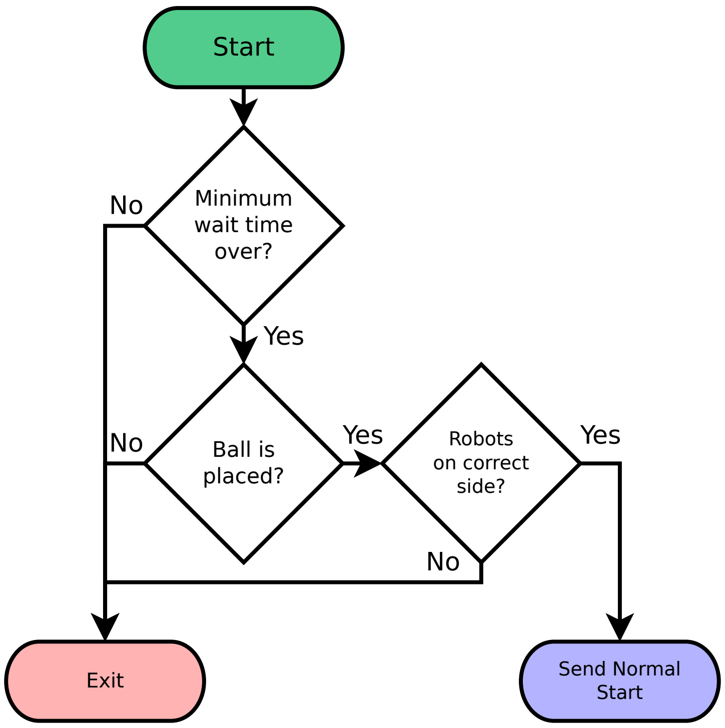

Figure 4.12: Chain of decisions for AutoRef Prepare Kickoff State

and the game transitions into the corresponding state. During ball placement the team responsible for the placement is given a certain amount of time (15 s according to the challenge) to place the ball. If the ball is placed correctly within the specified time window the state issues a *Stop* command in order to transition back into the Stop state to start the next action. If however the ball was not placed within the specified time, the Ball Placement State issues another placement command for the other team to also give it a chance to place the ball. If both teams fail at placing the ball the Ball Placement state returns to the Stop state to wait for the human referee to place the ball.

4.5.2 Passive AutoRef Engine

The passive engine serves as support for the human referee. It does not attempt to control the game but merely logs all events that are detected by the event engine to the game log. The human referee can then review the detected events in the GUI in case he missed a rule infringement or needs a second opinion.

5 Conclusion

The proposed implementation represents the first attempt at creating an autonomous referee software for the Small Size League. The software has been tested in a simulation environment as well as during regular games in passive and active mode.

In a simulation the AutoRef can control a game fully autonomously and makes accurate decisions. This is useful for testing full gameplay of an AI software or performing fully automated tests of single scenarios like a kick-off or a free kick where the referee software verifies the correct execution.

Tests during real games yielded a mixed result. Contrary to a simulation environment the vision data provided by the field camera system is not entirely accurate which can have a significant impact on the accuracy of the ball and robot position prediction. Also, chip kicks are difficult to detect and handle due to the inherent loss of information that is caused by the 2D projection which can only be partially compensated for. Under these circumstances some of the event detector classes fire false positives which can lead to incorrect decisions. Also, some events simply cannot be properly detected from vision data only and require a human referee to make a final decision.

However, if the unstable detector types are disabled and some incorrect decisions are acceptable the AutoRef can also be used in active mode in conjunction with the ball placement to run a fully autonomous game. In such a scenario it is crucial that the AutoRef communicates its decisions to the spectators as well as the human referee. For this reason, the league has proposed to introduce large display screens which can be used by the AutoRef software to display information about the game as well as its decisions.

Future work on the AutoRef will need to focus on its stability in real world environments. The robustness of each of the detector types shall be verified using footage of recorded SSL games from which different scenes shall be selected as benchmark for test scenarios. Also the AutoRef will require a visualization panel to display its current state as well as its decisions.

A Event Types

The following table lists all possible event types that can be emitted by the event engine. For most of the event types a corresponding rule infringement is explained in chapter 2. The *state* column specifies in which state the event can occur.

Event Type	**State**	**Description**
Ball Left Field	Running	The ball has exited the field and an indirect free kick needs to be taken
Ball Speed	Running	The ball was kicked above the speed limit
Double Touch	Running	The ball was touched twice by the same robot during a free kick scenario
Attacker to Defense Area	Running	The attacking team did not respect the required distance around the defense area of the defending team
Bot Collision	Running	Two robots made substantial contact
Indirect Goal	Running	The ball was kicked directly into the goal after an indirect free kick
Icing	Running	The ball was kicked over the midline and exited the field over the touch line
Ball Dribbling	Running	The ball was dribbled for more than the maximum dribbling distance
Bot Count	Running	The team exceeded the maximum number of robots allowed on the field
Bot Stop Speed	Stopped	A robot violated the speed limit during game stoppage
Attacker in Defense Area	Running	A robot touched the ball while being located in the defense area of the other team

Defender To Kick Point Distance	Free kick / kick-off	A robot of the defending team entered the off limits area around the ball during a free kick
Kick Timeout	Free kick / kick-off	The team taking a kick (free kick, kick-off) took more than the allowed time to kick the ball
Multiple Defender	Running	A robot touched the ball while being located **entirely** inside its own defense area
Multiple Defender Partially	Running	A robot touched the ball while being located **partially** inside its own defense area
Attacker Touch Keeper	Running	A robot touched the goal keeper of the opponent team inside the defense area of the keeper
Goal	Running	The ball entered the goal

Table A.1: Event types

References

[1] www.robocup.org. *Objective*. URL: `http://www.robocup.org/about-robocup/objective/` (visited on 05/05/2016).

[2] www.robocup.org. *Small Size League*. URL: `http://wiki.robocup.org/wiki/Small_Size_League` (visited on 05/06/2016).

[3] TIGERs Mannheim. *AutoReferee Git Repository*. URL: `http://gitlab.tigers-mannheim.de/open-source/AutoReferee` (visited on 06/10/2016).

[4] www.robocup.org. *AutoRef Challenge 2014*. URL: `http://wiki.robocup.org/wiki/Small_Size_League/RoboCup_2014/Technical_Challenges` (visited on 05/14/2016).

[5] www.robocup.org. *AutoRef Challenge 2016*. URL: `http://wiki.robocup.org/wiki/Small_Size_League/RoboCup_2016/Autoref_Challenge` (visited on 05/14/2016).

[6] Small Size League Technical Committee. *Laws of the RoboCup Small Size League 2016*. URL: `http://wiki.robocup.org/images/1/18/Small_Size_League_-_Rules_2016.pdf` (visited on 04/13/2016).

[7] Nicolai Ommer. *Internal document*. 2015.

[8] www.robocup.org. *Autonomous Ball Placement Callenge 2016*. URL: `http://wiki.robocup.org/wiki/Small_Size_League/RoboCup_2016/Autonomous_Ball_Placement` (visited on 06/22/2016).