TIGERs Mannheim

(Team Interacting and Game Evolving Robots)

Extended Team Description for RoboCup 2018

Andre Ryll, Mark Geiger, Chris Carstensen, Nicolai Ommer

Department of Information Technology
Baden-Wuerttemberg Cooperative State University,
Coblitzallee 1-9, 68163 Mannheim, Germany
management@tigers-mannheim.de
`https://tigers-mannheim.de`

Abstract. This paper presents a brief overview of the main systems of TIGERs Mannheim, a Small Size League (SSL) team intending to participate in RoboCup 2018 in Montréal, Canada. This year's ETDP mechanical and electrical section focuses on the development of a new dribbling device which combines good dribbling and damping characteristics and a new wireless base station. The AI section introduces a new system for self learning offensive strategies and the use of random search algorithms to find good pass and scoring positions.

1 Mechanical and Electrical System

Robot version	v2016
Dimension	Ø179 x 146mm
Total weight	2.65kg
Max. ball coverage	19.7%
Driving motors	Maxon EC-45 flat 50W
Gear	18 : 60
Gear type	Internal Spur
Wheel diameter	57mm
Encoder	US Digital E8P, 2048 PPR [1]
Dribbling motor	Maxon EC-max 22, 25W
Dribbling gear	50 : 30
Dribbling bar diameter	14mm
Kicker charge topology	Flyback Converter (up to 230V)
Chip kick distance	approx. 2.5m
Straight kick speed	max. 8m/s
Microcontroller	STM32F746 [2]
Used sensors	Encoders, Gyroscope, Accelerometer
Communication link	Semtech SX1280 @1.3MBit/s, 2.300 - 2.555GHz [3]

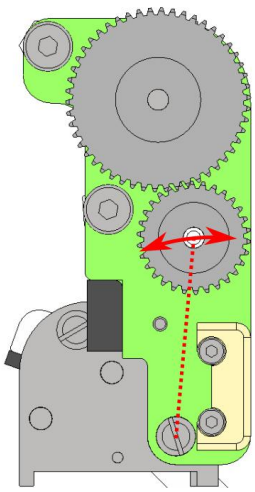
Table 1: Robot Specifications

Compared to last year, most parts of our robots have remained unchanged. Details can be found in [4] and [5]. For RoboCup 2017 we invented a new dribbler damping mechanism which is described in 1.1. Furthermore, we exchanged our primary wireless module due to persistent and severe connection problems of the nRF24L01+ modules. The new Semtech SX1280 module and our new infrastructure is described in 1.2. The complete mechanical specifications of our robots can be found in table 1.

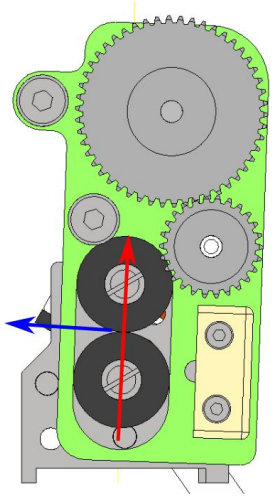
1.1 New Dribbler Damping Mechanism

Just in time for RoboCup 2017 we equipped all our robots with a new dribbler damping mechanism. The requirements for a dribbler mechanism in the SSL are twofold. Firstly, the dribbler should be able to exert a backspin on the ball and constantly maintain the ball at the dribbling bar. Secondly, it should absorb impact energy when receiving a strong pass or intercepting an opponents goal kick. This ensures that the ball remains at the dribbling bar and does not bounce off.

Our previous design had only one rotational joint and could be configured to either dribble well or to absorb impact energy. The configuration was mainly done with a silicon stop rubber in the structure which was either present or not. As we did not use any sophisticated dribbling moves at previous competitions we always used the configuration for maximum energy absorption. A CAD image of the old dribbling device can be seen in figure 1a.



(a) Old version with a single degree of freedom



(b) New version with two degrees of freedom

Fig. 1: Comparison of dribbling and damping unit. Silicon rubbers are shown in black. Degrees of freedom as arrows.

To improve our ball control capabilities we decided to design a dribbling device that can receive and dribble the ball well. To achieve this a design with two translational degrees of freedom is necessary. Our new dribbler is mounted on two damped linear guides and can be seen in figure 1b. On each side of the dribbler are two screws with a 4mm thick silicon ring around them. The slot allows the whole unit to move up and down while it is limited in the forward/backward direction. With this design, the impact energy of a ball is always absorbed by the silicon rings. When the dribbler is turned on it exerts a backspin on the ball and the counter-force lets the whole unit move slightly upwards. This freedom is necessary as otherwise the whole robot would attempt to drive on the ball.

The new design has been tested and, depending on the carpet, can fully absorb kicks of up to 5m/s. This allows us to receive very fast passes and still maintain good dribbling characteristics. As the two linear guides are independent of each other it may happen that the dribbler is not going upwards perfectly leveled (especially with a ball at one end of the dribbling bar). In the field it turned out that this is not a problem and does not affect functionality of the device. All CAD files for this new dribbling device are already open-sourced with our 2017 yearly release.

1.2 Semtech SX1280 Wireless Transceiver

Since 2013 we are using the nRF24L01+ wireless module from *Nordic Semiconductor* for our communication. These modules are quite common among the SSL and there are various off the shelf breakout board variants available on the market. Most convenient modules use a printed-circuit board antenna. More expensive ones also offer a SMA connector and a power and low-noise amplifier (PA/LNA). We are using a module with PA/LNA on our base station and smaller modules without amplifier but with SMA antenna on the robots.

In recent competitions we experience more and more link quality and connection issues. We suspect the cause of these issues to be the increased usage of this module within the league and the 2.4GHz WiFi band in general. Furthermore, the enlarged field of the RoboCup 2017 Multi-Team Technical Challenge took us to the limit of our wireless range. As this field size will be the new default in 2018 we decided to exchange our primary wireless module.

We selected the SX1280 IC from *Semtech* as our new base to built upon [3]. The SX1280 offers LoRa, FLRC, and FSK modulation modes. We are using the FLRC mode as it offers the best compromise between speed and robustness. The main differences of the SX1280 compared to the nRF24L01+ are shown in table 2. The datasheet of the SX1280 does not note a specific frequency range as a limit but we decided to use 2.3GHz to 2.555GHz and to tune our antenna circuit within this range. Without a front-end module (FEM) the SX1280 delivers 12.5dBm more output power than the nRF24L01+. The bandwidth of the SX1280 is higher and the data rate lower. Overall, this results in a much better link budget (up to 117dB) and link quality. Due to the lower data rate, we needed to increase the time on the base station for communicating with one robot from 1ms to 1.25ms

(see also [4]). This yields a total update rate of 800Hz for one robot or 100Hz for 8 robots.

As there are no off the shelf breakout boards available for this IC we needed to develop the required RF boards ourselves. This also gave us the chance to optimize them in size and shape for our robots. Furthermore, we added the SKY66112 front-end module to our new base station PCB [6]. This FEM incorporates a PA/LNA and also an antenna switch for diversity into a single package. Thereby, we can use two antennas with different orientations on the base station and always use the one with the best receive signal strength. Although the specified frequency range of the SKY66112 only goes from 2.400 to 2.483GHz the module also works outside these limits as out tests have shown. A picture of the new base station can be seen in figure 2.

Fig. 2: Base Station v2018 with exemplary display content showing all robots and an info bar on top.

The new base station also features a 5” touch screen display with a resolution of 800x480 pixels. On the display we can show the status of all connected robots to quickly indicate if any problems are present on the robots. Furthermore, we can manually control a single robot for testing purposes. Future developments may also show a visualization of the field and robot positions as received from the SSL Vision or the current game score from the SSL RefBox. The last use case is appealing as it may also be used as a simple indicator display for the audience to follow the game more easily. The complete base station is managed by a single STM32F767 microcontroller.

At the time of writing all robots have been equipped with new wireless modules and the new base station is operational. We already tested the new equipment at several smaller events and in our lab environment with a heavily crowded

2.4GHz band. Even at WiFi frequencies we do now no longer face any issues with connection quality or lost packets. Further work will be done to verify functionality also on larger fields and during real game situations to make sure this new solutions works well for RoboCup 2018.

	nRF24L01+	SX1280
Frequency Range	2.400 - 2.525GHz	2.300 - 2.555GHz
Output Power (without / with FEM)	0dBm / 21dBm	12.5dBm / 21dBm
Bandwidth	2Mhz	2.4MHz
Data rate	2MBit/s	1.3MBit/s
Diversity	No	Yes
Base Station TDMA slot time	1ms	1.25ms

Table 2: Comparison of wireless modules.

2 Offensive Strategy and Self Learning

Developments of the AI for last years RoboCup were mainly focused on robustness and dynamic plays. The idea was that based on the current situation each robot would decide what it should do and what is the best strategy for this situation. There was no planning in the future and no predefined plays. Plays and passes involving multiple robots were generated naturally and automatically while playing the game. The AI has shown some unique and also efficient plays and strategies against different opponents. Nevertheless, it also has shown to repeatedly make the same mistakes once an opponent has found a weakness. The AI did not learn from its mistakes.

For this year, we developed a strategy that learns from past plays and thus is able to execute better strategies in future plays. The idea is that there are still no fixed plans for future strategies but that potential future plays effect the decision making for the robot in its current situation.

2.1 Offensive Behavior

This section will introduce the basic foundation of the offensive decision making. One key aspect of the offensive strategy are the *OffensiveActionMoves*. An *OffensiveActionMove* represents a specific action a robot can execute. An *OffensiveActionMove* can be a simple pass, a kick on the opponents goal, or a special behavior in close engagements with robots from the opponent team. Currently we have ten *OffensiveActionMoves*. In source code listing 1 the abstract interface of each *OffensiveActionMove* can be seen. Table 3 shows the currently implemented *OffensiveActionMoves*.

OffensiveActionMove	Description
ForcedPass	Forced pass in standard Situation
DirectKick	A direct kick aimed at the opponent goal
ClearingKick	A defensive action to clear a dangerous ball near our penalty area
StandardPass	A normal pass to another robot
LowChanceKick	A direct kick aimed at the opponent goal (low scoring chance)
GoToOtherHalf	We have ball control in our half, but no suitable pass target
KickInsBlaue	A pass to an area that is free of opponent robots
RedirectGoalShot	Redirect the ball into the opponents goal
RedirectPass	Redirect the ball directly to another friendly robot
Receive	Catch and receive an incoming or fast traveling ball

Table 3: Currently implemented *OffensiveActionMoves* ordered by their priority

Algorithm 1 AOffensiveActionMove

```
public abstract class AOffensiveActionMove {
  public abstract EActionViability isActionViable(...);
  public abstract void activateAction(...);
  public abstract double calcViabilityScore(...);
}
```

There are three methods that each *OffensiveActionMove* has to implement. The method *isActionViable* determines the viability of an *ActionMove*. The viability can either be *TRUE*, *PARTIALLY* or *FALSE*. The method *activateAction* controls the actual execution of the move. The method *calcViabilityScore* will determine a score between 0 and 1 for the current Situation. This score should be connected to the likelihood, that this action can be executed successfully.

In our current implementation there are two different sets of *OffensiveActionMoves,* one set for "normal" actions and one set for "redirect" actions. This means that in a first step the robot has to decide whether it can normally move towards the ball or if it has to overtake the ball in order to receive or redirect the ball. The main criteria for this decision are the current position and velocity of the ball.

In source code listing 2 it is shown how the best *OffensiveActionMove*, out of one given *OffensiveActionMoveSet* (normal or redirect), is determined. It is important to notice that the *OffensiveActionsMoves* inside a given set have a specific ordering, which represents the priority. The *OffensiveActionMove* in the first position of the set has the highest priority. An *OffensiveActionMove* will be activated if its viability returns *TRUE* and it has a higher priority than all other *OffensiveActionMoves* that return a *TRUE* viability. Actions that return the viability *FALSE* will be ignored in any further processing. All actions that are *PARTIALLY* viable are sorted by their *viabilityScore* and if there is no action that has a *TRUE* viability, then the action with the highest *viabilityScore* will

be activated. In case all actions have a *FALSE* viability then a default strategy will be executed.

The separation into viable and partially viable actions, combined with priorities leads to a very stable and easy modifiable/extendable algorithm for the offensive strategy. For example, the *OffensiveActionMove* that controls direct kicks on the opponent goal will return a *TRUE* viability if there is a high chance to score a goal. If there is a extremely low chance to score a goal it will return *FALSE*. Otherwise, if the hit chance is reasonable but not really high, it will return *PARTIALLY*. Additionally, this action has a high priority. Thus, the robot will surely shoot on the goal if there is a good opportunity to score a goal. However, if the viability is *PARTIALLY* the action will be compared with the other actions and based on the *viabilityScores* the robot will decide whether it should shoot on the goal or execute another action, e.g. a pass to another robot.

Algorithm 2 Pseudocode - Find the best OffensiveActionMove

```
// Iterate over all offensiveActionMoves in this set
for (AOffensiveActionMove action : actionsSet) {
  EActionViability viability = action.isActionViable(...);
  if (viability == TRUE) {
          // activate first move that got declared as viable
        action.activateAction(...);
        return;
  } else if (viability == PARTIALLY)
    partiallyMoves.add(action);
}

partiallyMoves.sort(...); // sort by viabilityScore
if (!partiallyMoves.isEmpty()) {
  // choose first partially viable move to be activated
  partiallyMoves[0].activateAction(...);
  return;
}
```

2.2 Self Learning Offensive Strategies

For this years RoboCup we developed a self learning system to improve our offensive strategies. The system is based on trees that store executed plays and their outcomes. This information can then be used to improve decision making in the future. Figure 3 shows how such a tree can look like. The actual trees that are generated during a game are much bigger. A node of the tree represents an *OffensiveActionMove* that got executed in the past for this situation in this path of the tree. Additionally, each node has a score that reflects how successful the action has been in the past. The scores are displayed on the arrows and as color of the node. Success is measured in a global score that represents how

good a given situation is for our team. This score is what we try to optimize. A path through the tree represents a dynamically generated play which has been executed in the past, consisting of multiple *OffensiveActionMoves*. We track all executed *OffensiveActionMoves* in our plays during execution. A play starts when we obtain ball control and ends when we lose ball control. When the play ends all containing *OffensiveActionMoves* will be stored in the tree structure and the weights of the tree get updated, based on the success of the executed action.

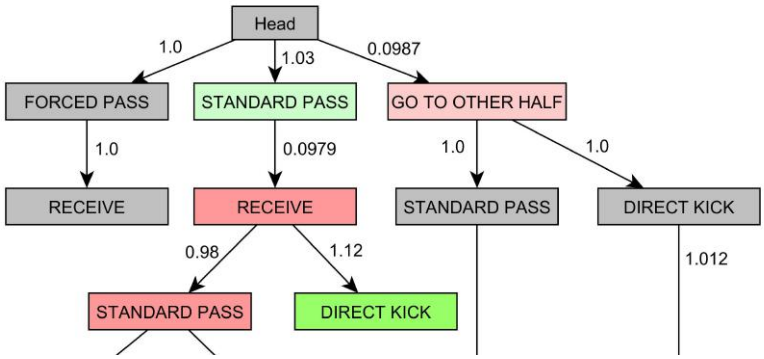


Fig. 3: Offensive Action Tree: This Figure shows an *OffensiveActionTree* for a single *OffensiveSituation*. The numbers on the arrows show the weight for a single *OffensiveActionMove* within its path. The color of the nodes also represents the weight.

Multiple trees will be generated during a game, one tree for each *OffensiveSituation*. An *OffensiveSituation* describes the current game situation in a very abstract way, e.g. "Ball is in the opponents half and we will reach the ball earlier", "Close to opponent penalty area" or "Skirmish around ball". In order to use the trees for our future decision making we have to track the point in time when our team obtains ball control. In the moment we obtain the ball we will use the current *OffensiveSituation* to choose the matching tree. As long as we have ball control this tree will be used and updated, no matter if the *OffensiveSituation* changes during execution. When our team obtains ball control we start at the head of the tree. Then the robot will choose an *OffensiveActionMove*, based on the algorithm described in section 2.1. Once an action has been executed the active position of the tree will move along the executed path. Figure 4 shows the current state in the tree and the potential future paths. Note that there are many more potential future paths than displayed, every path that has not been stored in the tree yet has the weight one.

In order to optimize the offensive decision making, the weights of the tree have a direct effect on the *viabilityScores* of an *OffensiveActionMove*. Starting

from the current position inside a tree we will examine all potential paths. We will look for the path with the maximum weight. Therefore, all the weights along a patch are multiplied. In the situation described in Figure 4 a *FORCED_PASS* has already been executed, now the AI is looking for a *OffensiveActionMove* that could be executed next. Now, we can step along the potential paths and obtain bonus values for the *OffensiveActionMoves.* The values written in the brackets are the products of the maximum weights along a path. In this example, we can see that through our learned tree the algorithm will give the action *REDIRECT_PASS* a slight bonus multiplier of 1.1, while the *REDIRECT_GOAL* action has a standard value of 1. However, this is just a small bonus, depending on the actual viabilities the robot can still choose to execute a *REDIRECT_GOAL.*

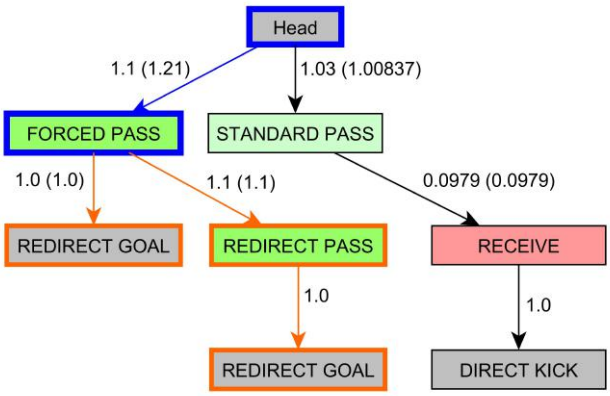


Fig. 4: Currently Active Path: This Figure shows the actual weights of the *OffensiveActionMoves* and the maximized multiplied scores of each path. The multiplied scores are written in brackets. The blue marked *OffensiveActionMoves* have already been executed. The orange marked *OffensiveActionMoves* are the potential moves that can be executed in the future.

The separation of the *OffensiveActionMoves* into 2 different sets and additionally into viable and partially viable actions has shown some good results. It is easy to add more *OffensiveActionsMoves* later and to modify behavior of the offensive. The self learning trees extend this algorithm by dynamically modifying the *viabilityScores based on learned data.* So far, the algorithms for the offensive trees are still in development. Thus, they have not been tested in a real environment in their newest implementation. We hope to collect some data in the upcoming RoboCup 2018, to check if the offensive trees indeed help to improve our offensive strategies.

3 Random Search Optimization

In RoboCup Soccer an AI constantly evaluates the current situation and chooses appropriate actions to win the game. On the one hand, new information is available roughly every 16 milliseconds. We call this information frames. The evaluate-reaction cycle has to be completed within this time constraint to react on the latest information available. On the other hand, the situation is described among others by position and velocity data of all robots and the ball. Calculating the optimal solution from this high dimensional real valued input domain within the given time constraint is difficult. To cope with this input domain, we exploit the fact that the situation does not change significantly from one frame to another. With this in mind, we can calculate several solutions, take the best ones and memorize these solutions for the next frame. In the next frame, those memorized samples get reevaluated and compared against new samples, where the best of both get memorized for the next frame and so forth.

One example where we use this kind of random search optimization is to find the best supportive positions as shown in figure 5 for the yellow team. This could either be pass receiving positions (green circles) or shot on goal positions (red circles). To find these positions we randomly select about five points within a specific shape on the field. In this example it is the red rectangular outlining figure 5. If a position is not valid (e.g. if it is inside the penalty area) a new point is randomly selected until the maximum number of positions or tries is reached. Additionally, the positions are rated (among others) by ball visibility or a goal score chance as described here [5]. In our implementation this is the computationally most expensive part. After that, some robots (depending on the situation) go to these positions.

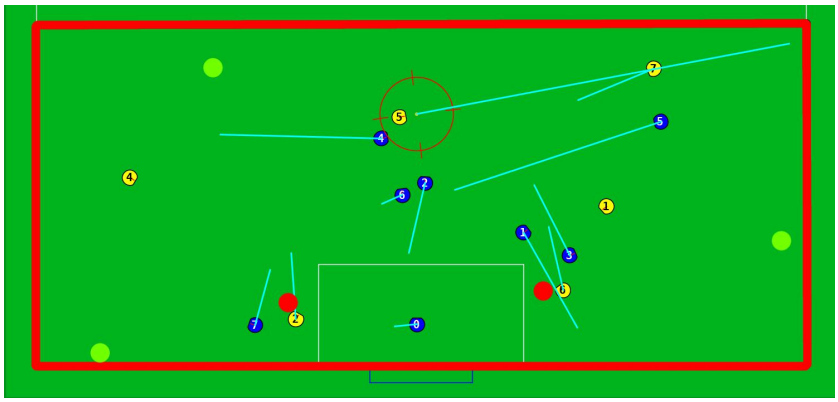


Fig. 5: Random search example: supportive positions. This figure shows calculated receiving positions (green circles) and possible shoot positions (red circles) for the yellow team. The red rectangle is the shape, where all positions are generated. The blue lines represents the velocity of the robots, which is taken in account when calculating the visibility of a position.

In the next frame, the best positions of the last frame will be rated and compared again with newly selected positions. This leads to an optimization, which constantly improves itself and adapts to new situations. As in the presented example, we want more than one position. Additionally, those positions have to have a certain distance from each other to avoid a bulk of robots. Therefore all positions near the best positions are removed during selection. Since it take some time for the robot to drive to the position, the positions have to be stable.

In our implementation, the scores are discretized such that there is a higher probability to be equal. In the equal case, the older position is rated higher than the newer, which provides enough robustness to the positions.

This random search optimization decouples the analysis of the situation from the frame and distribute it over several frames. The number of samples and tries enables the team to dynamically adapt the computational time spend on this specific problem, which can also be situation depended. It enables us to include time consuming methods like trajectory calculations in our rating. In addition, using a random selection, we do not have to divide the field into sectors or a grid, which provides more adaptation to different opponents.

On the contrary, the implementation of the random search optimization is more complex than a simple grid. In addition, it may happen that the algorithm does not find any position, if for example all random positions are inside the penalty area.

RoboCup 2017 has shown that this algorithm reduces the computational effort and returns good solutions within a few frames. A proper visualization of the scoring and a step wise simulation is crucial to tune this algorithm. Nevertheless, it is hard to control the random optimization. Especially when analyzing a running game it is not clear whether it was just bad luck or an error within the scoring of the positions. In addition, the problem of calculating and combining different scores remains unresolved.

This year we will focus on other shapes than a simple rectangle where the positions are generated. Here one can use different shapes depending on the situation to decrease the search space.

4 Publication

Our team publishes all their resources, including software, electronics/schematics and mechanical drawings, after each RoboCup. They can be found on our website[1]. The website also contains several publications[2] with reference to the RoboCup, though some are only available in German.

[1] Open source / hardware: https://tigers-mannheim.de/index.php?id=29
[2] publications: https://tigers-mannheim.de/index.php?id=21

References

1. US Digital. E8P OEM Miniature Optical Kit Encoder, 2012. http://www.usdigital.com/products/e8p.
2. STmicroelectronics. STM32F745xx, STM32F746xx Datasheet, December 2015. http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00166116.pdf.
3. Semtech Corporation. SX1280 Datasheet, May 2017. http://www.semtech.com/images/datasheet/sx1280_81.pdf.
4. A. Ryll, M. Geiger, N. Ommer, A. Sachtler, and L. Magel. TIGERs Mannheim - Extended Team Description for RoboCup 2016, 2016.
5. M. Geiger, C. Carstensen, A. Ryll, N. Ommer, D. Engelhardt, and F. Bayer. TIGERs Mannheim - Extended Team Description for RoboCup 2017, 2017.
6. Skyworks Solutions, Inc. SKY66112 Datasheet, March 2017. http://www.skyworksinc.com/uploads/documents/SKY66112_11_203225L.pdf.