System Design Project Technical Specification

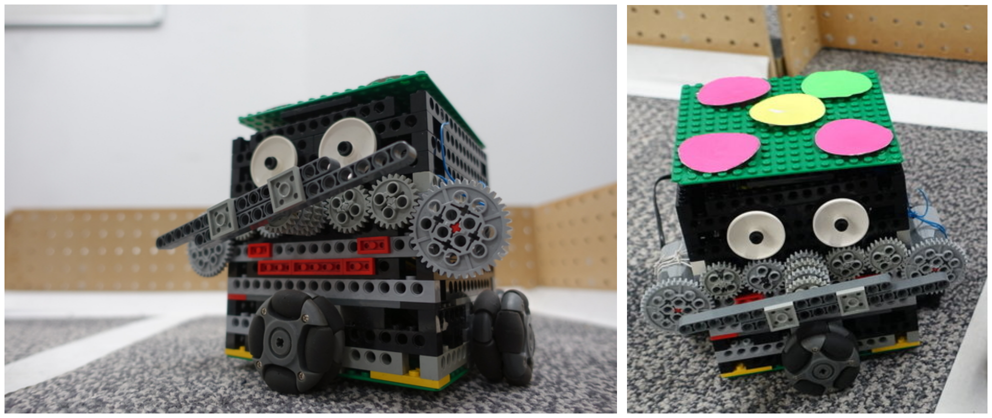
Group 2𝔸

1 Introduction

The robot 𝔽.ℝ.𝔼.𝔻 was designed as to meet requirements given in the *System Design Project* website by *The University of Edinburgh*, which is assigned to third year Informatics students. The final version of our robot has to be able to play a two-a-side football game, similar to the Robocup competition small sized league.
The requirements are as follows:

- The robot must be able to move in any direction.
- The robot must be able to rotate.
- The robot must be able to kick the ball to various distances ie. pass or shoot.
- Moving with the ball is not allowed.
- The robot must be able to play an attacking or defending role.
- The robot must be able to be fit in a box with the dimensions 20cm x 18cm x 18cm.

2 Overall Architecture



2.1 Movement

A simple four-wheeled holonomic design was chosen for 𝔽.ℝ.𝔼.𝔻. A wheel is attached to each of the four lightweight motors, located on each side of the cubic model. The firmware allows an external system to control each wheel independently, meaning it allows for fully free holonomic motion (It is possible to use different motor powers to achieve motion where axial rotation is independent from planar motion).

2.2 Kicking

During a game of two-a-side football, each robot has to be able to act in a defensive role by somehow taking ownership of the ball from the defence area and passing it towards its team-mate. 𝔽.ℝ.𝔼.𝔻 is unique in the way that he does not have a grabber, but instead a propeller-like kicker. This is located at the front-facing side of 𝔽.ℝ.𝔼.𝔻. The kicker uses two motors, located on the left and right sides of 𝔽.ℝ.𝔼.𝔻, which link to the kicker by several gears situated across the front of 𝔽.ℝ.𝔼.𝔻. Using gearing and two motors increases the power of 𝔽.ℝ.𝔼.𝔻's kick. This design was chosen in order to be an effective defensive player, as well as make powerful, accurate shots when attacking.

3 Hardware

3.1 Basic Hardware

𝔽.ℝ.𝔼.𝔻 was designed in a way that allows one to build the "basic" version of 𝔽.ℝ.𝔼.𝔻 and then expand upon it. An extensive guide on how to build the basic Box design that moves around can be found at:

`http://fred.rovder.com/building.html`

3.2 Extended Hardware

Our specific version of 𝔽.ℝ.𝔼.𝔻 is a defensive robot with a propeller designed to deflect the ball from its path to our goal. These are the additional components used:

Part	**Location on Robot**	**Amount**
Power Functions Medium Motor	Left and Right sides	2
24 Teeth Gear	Across front-facing side, attached to motor	8
40 Teeth Gear	Across front-facing side, attached to motor	2
Arduino Motor board	Inside box	1

3.3 Movement

- An Electric Technic Mini-Motor 9v was connected at the base of each of the four sides of the robot, attached to a bi-directional wheel.

- The Electric Technic Mini-Motor 9v motors are a slightly older model of motor, however we have found them to be reliable and simple to use. These motors provide high RPM at the cost of torque, which, combined with the motor multiplexor board, made the fastest robot. The Electric Technic Mini-Motor 9v motors are also very small, therefore allowing more room for further modifications to the robot and simple replacement in case of failure.

- The small grey bi-directional wheels were one of two choices which would allow for holonomicity. We considered the larger yellow holonomic wheels, but those proved too large and the motors were not able to produce enough torque to move 𝔽.ℝ.𝔼.𝔻.

3.4 Kicking

- The kicker is similar to that of a propeller, which is large and stable enough to enable for powerful kicks. 𝔽.ℝ.𝔼.𝔻 is fundamentally a defence robot, so accuracy was not of high priority.

- The kicker works as a spinning mechanism. The direction of spinning is determined by the position of 𝔽.ℝ.𝔼.𝔻 to the ball. The direction of spinning will change such that it will always kick towards the opponents half. This increases the chance of a goal against the opponent and decreases the chance of an own goal. This spinning mechanism is controlled by the Strategy System.

- Using two motors to control the kicker maximises the power of the kicker.

- Gearing the kickers to higher rotational speed adds some accuracy to $\mathbb{F}.\mathbb{R}.\mathbb{E}.\mathbb{D}$'s kicks. This is because $\mathbb{F}.\mathbb{R}.\mathbb{E}.\mathbb{D}$ will most often face the ball as it approaches it. The way 'sphere impacts' work in physics require the kickers to hit the ball as soon as it is within reach. The further "through" the propeller kicks the ball on impact, the less accurate the kick will be. (It's similar to Billiards, when the Cue ball strikes a regular ball ever so slightly on the side, the regular ball will move almost perpendicularly to the direction of the Cue ball)
- The 24 and 40 teeth gears were used for gearing up at a 5:3 ratio.
- We used the Power Functions Medium Motors for the kicker as they are small and lightweight. This aids quick movement. These motors were also chosen because they are powerful enough for the kicker to shoot to the goal as well as pass to a team-mate.

4 Firmware

Firmware functions and descriptions are shown in Appendix A.

5 Communication Between PC and Robot

5.1 Introduction

The communication system for the robot implements methods that are called by the Strategy Module. After being called by the Strategy Module, the methods send the appropriate commands to the robot's Arduino via the provided *Ciseco SRF Stick*.

5.2 Connecting

Once the Strategy module is turned on, it automatically creates a communications class. This class will attempt to reach Fred by sending "ping" to all the available ports and listening to "pang". Once "pang" is received, the class keeps that port open and uses it for all further communication. This whole process is done on a different thread, so the GUI will remain responsive while the ports are being scanned.

5.3 Command Set

Each of the methods in the Communications class calls the CommandSender method with the "args" being whatever the strategy module is inputting. Implementation of these methods can be found in `brobortdriver.src.robot.Fred`. The command each method sends to the arduino can be seen below.

Method	**Command**	**Input Parameters**
holonomicMotion	r	The powers of the motors (in order: front, back, left, right).
halt	h	N/A
ping	ping	N/A
kick	kick	1, 0 or -1 to spin kicker clockwise, not spin or counter-clockwise respectively.

6 Vision System

6.1 Vision System Pipeline

The Vision System is composed of 5 pipeline stages. Each pipeline stage is described below.

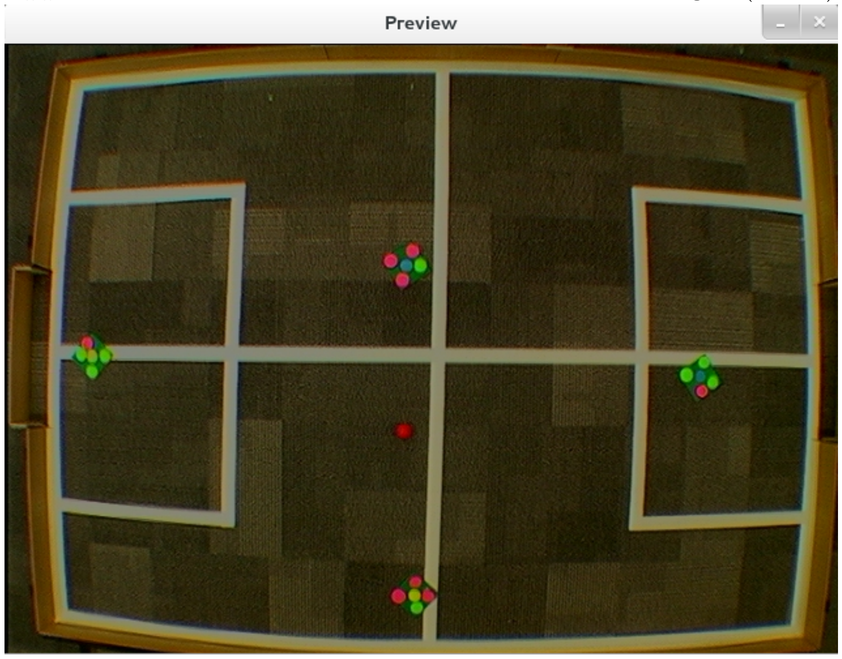
6.1.1 Raw Input

The vision system has a `RawInputInterface` and an abstract `AbstractRawInput` class. Extending the `AbstractRawInput` class enables you to add new raw input methods to the system. The camera feed is one, a static image is another. Video files will be streamed into the vision system in the future too. This is for debug purposes. Each raw input extends a JPanel, which can be added to the GUI and should contain all the controls for that input method. These can be seen in the Input Selection tab.

When implementing a raw input feed, one must carefully make sure that all errors are handled within the scope of the class and that the class will either successfully pass a `BufferedImage` to the next pipeline stage, or terminate the feed informing the user of the error. This can be done via a message dialog.

6.1.2 Raw input multiplexer

At this stage of the pipeline, the various raw input methods are handled and the streamed image is passed forward. This stage unites the GUIs of the individual Raw Inputs, adding them to the vision control window, and it also make sure only one input stream is open at any time. If everything up to this pipeline stage is implemented correctly, the expected output should appear in the Preview window as a stream of images (video):



6.1.3 Spot analysis

At this stage, the image is scanned for coloured spots using the following procedure:

1. The Raster of the `BufferedImage` is fetched (Using `BufferedImage.getData()`). The Raster gives us access to the actual pixel RGB colour values.

2. Using `Raster.getPixels`, we then fill an array with time image's RGBA values. This array is preallocated on start-up, hence no memory allocation is needed.

3. The array of RGBA values is passed onto the *ImageTools.rgbToHsv* function, which converts the RGBA values into a more analysis-friendly HSV format.

4. The next steps are repeated for every defined colour the vision system is detecting:

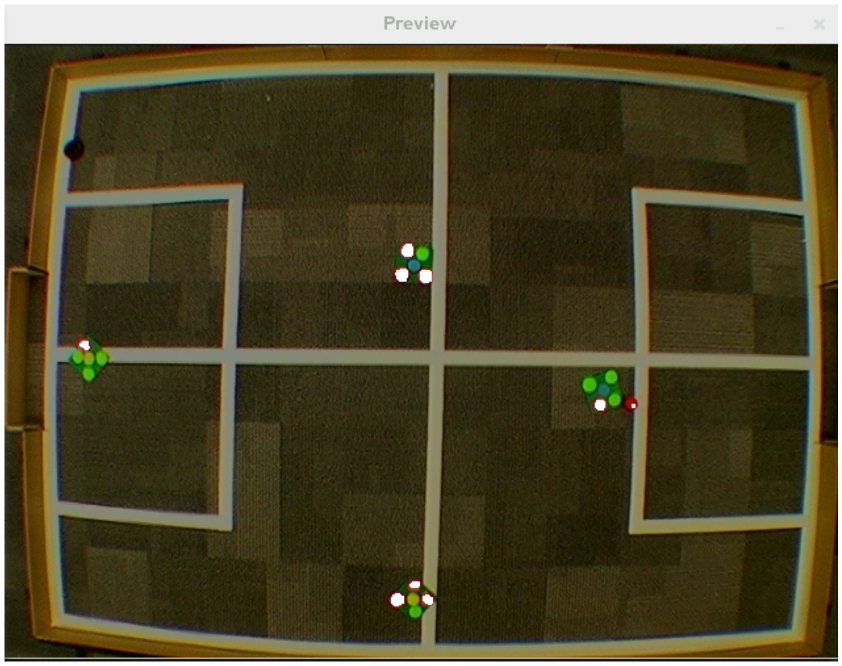
    (a) Reset the `private SDPColor[] found` array to null values.

(b) Iterate over all the pixels, passing each one to the *RecursiveSpotAnalysis.processPixel* method. This method recursively fills all potential spots, finding all the adjacent pixels of that colour. Every time a pixel is determined to be of the correct colour, set its corresponding location in the "found" array to the colour. This flag is used to make sure no pixel gets processed twice.

(c) Every cluster of pixels is recorded into an ArrayList.

5. In the end, all the ArrayLists of clusters of different colours (clusters also referred to as Spots) are then put into a HashMap, the colour of the spots in the list being the Key.

The final resulting HashMap is then passed onto the next stage of the pipeline. The pixels detected as the colour "Pink" are highlighted here:



6.1.4 Distortion

The spots passed into this stage from the Spot Analysis stage are undistorted and turned from pixel coordinates to actual centimetre coordinates. This is the sequence of operations every Spot is forced to undergo:

1. **Barrel Undistortion** - Gets rid of the camera distortion.

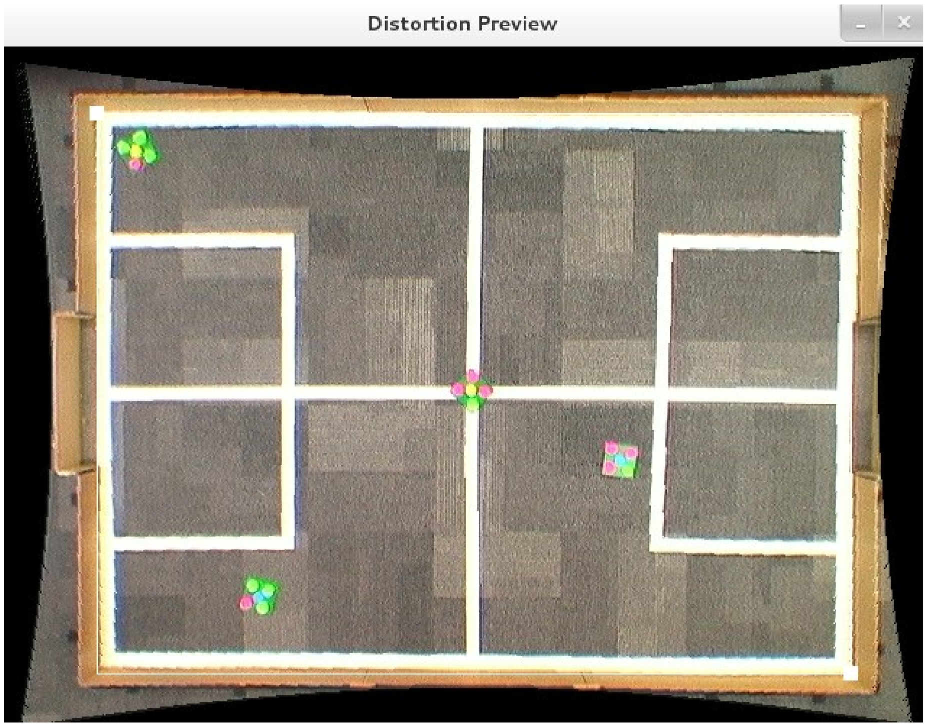
2. **Zoom** - Zoom the picture in or out.

3. **3D Tilt** - Removes distortion caused by camera being angled off-centre.

4. **Rotation** - Removes the rotation of the pitch.

5. **XY Shift** - Allows the user to centre the pitch.

There is no such thing as an "Undistorted image preview", since the image itself never gets undistorted. What the system does provide though is a preview of how the distortion affects a single frame:



6.1.5 Robot Recognition

The undistorted colour spots reach the robot recognition stage, where the robot plate patterns are detected.

This was by far the trickiest part to implement well and was incrementally developed over four iterations:

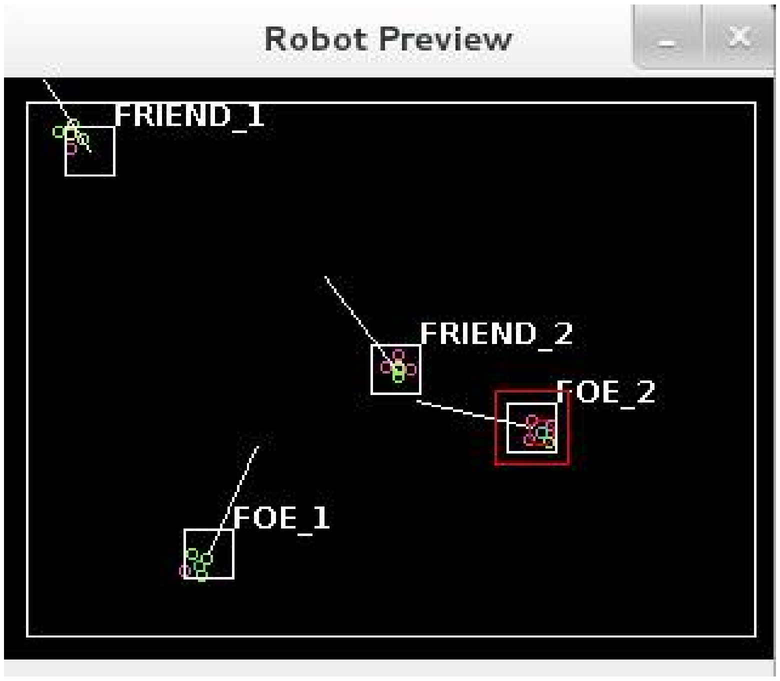
1. **Deterministic detection with team spots as plate bases** - System found all the team spots and tried to find surrounding spots to form plate. (Failed because team spots were not always visible.)

2. **Deterministic detection using plate outlines for clustering** - System found all plates and assumed all spots contained within that plate belonged to a single robot (failed because plates were not always visible.)

3. **Probabilistic detection** - System detected spots and tried to solve the constraint satisfaction problem describing the robot plates. (Failed because camera quality sometimes detected several spots in place of a single spot, creating anomalous probabilistic ghost robots.)

4. **Deterministic brute force detection using non-team spots as plate base** - System used brute force of complexity $O(n^3)$ (in the amount of non-team spots) to detect top plates by their primary colour spots. Then, using their locations, checked for corresponding secondary spot and team spot in locations where it expected them to be. (Succeeded)

In the end, the brute force detection performed much better than the others. This is because non-team spots had a much easier detectable colour. So we set the thresholds for the non-team spots very strictly and loosened the thresholds on team-spots. Since plates are detected using non-team spots, only real plates are detected (no false positives). Detection is so accurate, no smoothing of any kind is required. Once the robots are detected, they are drawn onto the `RobotPreview` window:



Note that the further the robots are from the centre of the pitch, the less the square lines up with the spots. This is because of the robots' heights, which distort their actual locations when viewed at an angle. The offset is caused by removing this distortion.

6.2 Design Decisions - Justification

6.2.1 Post Detection Distortion

In order to minimize computation, undistorting takes place after colour spots have been detected. On average, about 20 to 30 spots will be detected with any frame. Only these 30 spots will be undistorted. This is more efficient than undistorting the image and then looking for spots, because the image is approximately 300 000 pixels. Undistorting each pixel is a waste of computation power.

6.2.2 Arrays

The code takes `BufferedImage` inputs, gets the Raster from it and turns the image into an array of `rgb` values. These values are then used in the computations, bypassing having to call the `BufferedImage` methods to access the colours, which are very slow and allocate unnecessary objects. The arrays themselves are recycled and only need to be allocated on start-up, so no per-frame memory allocation is needed. Being single dimensional arrays of integers, the integers are all located next to each other in physical memory, which greatly improves memory access times by avoiding pointer chasing and utilizing spacial locality optimizations of the hardware.

6.2.3 Pipeline

While bashing everything together into one massive system may have been simple at the start, splitting the system into pipeline stages provides several advantages:

1. Code can be tested independently of the other parts of the program.
2. If one stage of the pipeline is not performing well, it can be replaced easily.
3. If necessary, this approach allows dynamic swapping of pipeline stages in and out of the pipeline. One may implement (for example) three different spot analysis systems, where each one is good for a different situation. A smart overviewing system could track the efficiency of each stage and swap them in and out based on the current state of things.
4. One may also take a leaf out of the book of airplane on-board computers and run several systems in parallel, unifying their results, filtering away outliers.

6.3 Connecting the Vision System to your code

To run the Vision System with your code, follow these instructions:

1. One or more of your classes should implement the `VisionListener` interface. These classes are then able to receive output from the Vision System.

2. Instantiate the `Vision` class. Located in vision.Vision. Instantiate it with
   `Vision vision = new Vision();`
3. Add your `VisionListener` classes as listeners: `vision.addVisionListener(`*yourClassHere*`)`.

7 Strategy

The Strategy module is the part that makes 𝔽.ℝ.𝔼.𝔻 play football. It consists of a timer, which launches a method every 100 milliseconds (Referred to from now on as ”Strategy Iterations”). This method takes a look at the current world state (provided by the Vision System) and determines the best course of action. Once it decides, it sends the appropriate commands to 𝔽.ℝ.𝔼.𝔻.

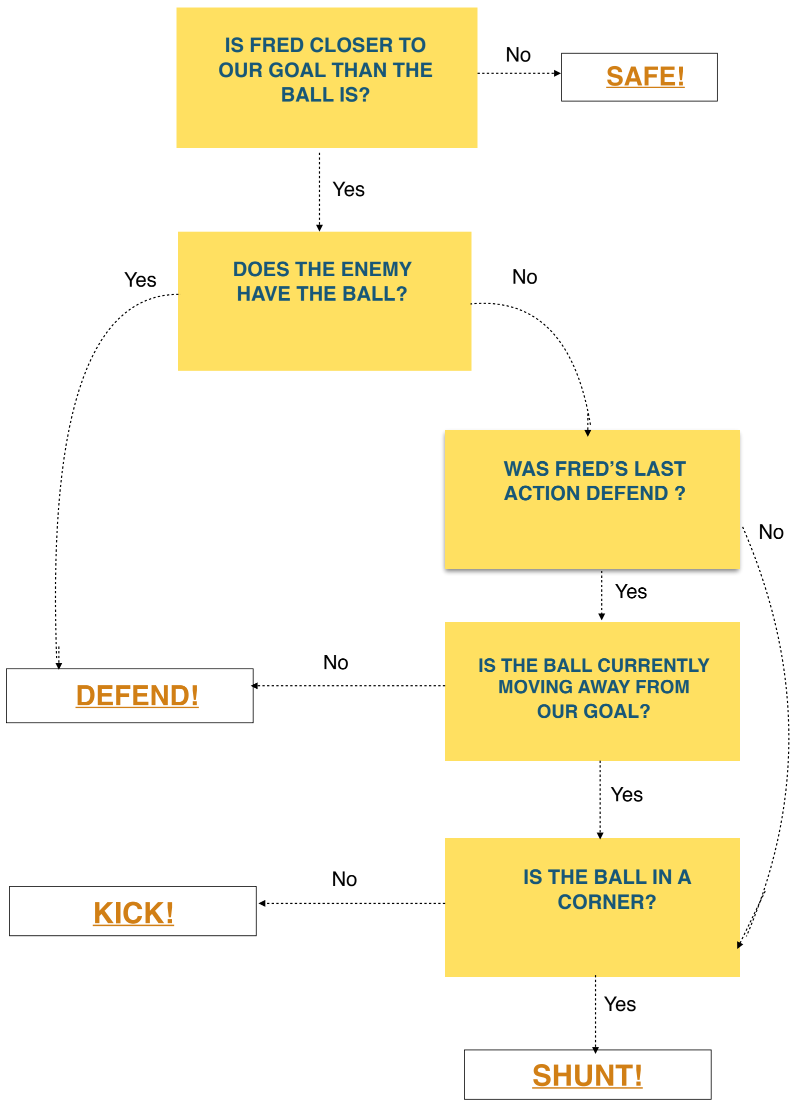
7.1 Behaviour Modes

Since the game is fairly simple, there was no need for extra clever strategy. Due to the simplicity of the game, there is only a very small amount of different situations that can occur in the game. This makes it possible to identify the best behaviour for each situation and hard code the most desired behaviour for each of them. There are a total of 4 types of behaviour (Referred to from now on as ”Behaviour Modes”), which cover all of the game states:

1. **Attack** - Kick the ball.
2. **Defend** - Hold position between ball and goal.
3. **Safe** - Go to our goal without touching the ball. (A precautionary measure to make sure no own goals are scored.)
4. **Shunt** - When the ball gets stuck in a corner, defend it. (𝔽.ℝ.𝔼.𝔻 is unable to fetch a cornered ball, so it will make sure nobody else can either, forcing a ball reset.)

7.2 Decision Making

The main thing the Strategy module needs to be able to do, is decide which of the 4 Behaviour Modes it should follow. For every Strategy Iteration, the Strategy Module follows the following flowchart in order to decide the most appropriate Behaviour Model.


IS FRED CLOSER TO OUR GOAL THAN THE BALL IS?
No
SAFE!
Yes
DOES THE ENEMY HAVE THE BALL?
Yes
No
WAS FRED'S LAST ACTION DEFEND ?
No
Yes
IS THE BALL CURRENTLY MOVING AWAY FROM OUR GOAL?
No
DEFEND!
Yes
IS THE BALL IN A CORNER?
No
KICK!
Yes
SHUNT!

7.3 Strategy Tools

In order to make everything clear and modular, there are several tool packages implemented, which are very low-level oriented and are used by a higher behaviour modelling system. These are the holonomicDrive package and the navigation package.

7.3.1 Holonomic Drive

Holonomic motion was implemented using a holonomic motion matrix. To make sending commands to 𝔽.ℝ.𝔼.𝔻 easy, the `holonomicDrive.HolonomicMove` class was implemented. This class provides the methods *setDesiredHeading()*, *setRotation()* and *perform()*, which should be used to make 𝔽.ℝ.𝔼.𝔻 move.

Keep in mind that the `VectorGeometry` object passed to *setDesiredHeading()* must be a relative direction "from 𝔽.ℝ.𝔼.𝔻's point of view" and NOT the desired location on the pitch. The `HolonomicMove` class also does not ensure avoiding obstacles. It simply makes 𝔽.ℝ.𝔼.𝔻 move.

7.3.2 Navigation

The Navigation System should be thought of as a black box. The desired destination is put into it and it will spit out the direction in which 𝔽.ℝ.𝔼.𝔻 should move *right now.* It calculates the shortest path from where 𝔽.ℝ.𝔼.𝔻 is, to where we want him to be, avoiding obstacles and obeying rules.

The Navigation System uses a combination of Potential Fields and A* search to find the path:

- **Potential Field Navigation** - The idea is that the whole pitch gets transformed into a potential field. Obstacles (such as robots, defence areas, and walls) are made repulsive and the desired destination is made attractive. The system then simulates 𝔽.ℝ.𝔼.𝔻 "fall" through this potential field. This way 𝔽.ℝ.𝔼.𝔻 is forced to follow the potentials to a potential minimum.
  - **Pros:** Continuous and fast.
  - **Cons:** Unreliable for long distances (getting trapped in local minima).
- **A* Navigation** - Dividing the pitch into small squares of area roughly 10cm x 10cm and running standard A* search from the robot's location to the destination.
  - **Pros:** Reliable and fast.
  - **Cons:** Inaccurate for short distances (maximum accuracy of destination is determined by the square size, since A* search is Discrete).

In order to avoid the cons and make use of the pros of each method, the Navigation System implements and uses both. With every call to `Navigator.move()` (which happens every Strategy Iteration) the `Navigator` class decides which of the two navigation systems to use. Since they both implement the same interface, they can be swapped in and out of the pipeline seamlessly. The interfaces also make potentially implementing and integrating more systems in the future very simple.

7.4 Modelling the Behaviour

To model all of the behaviour, we use a system of dynamic points (points that automatically adjust themselves) and Actions (recursive finite state machines implemented using a model called the *Tik Tok Model*). First, it will be explained what these are and then how they are tied together.

7.4.1 Dynamic Points

The idea of Dynamic Points is that you can define an arbitrary point by specifying how to calculate its X and Y coordinates from a `DynamicWorld` object. One can create a class, which implements the `ActionPoint` interface. This means it will provide the methods *getX()*, *getY()* and *recalculate()*, which you get to implement yourself. The recalculate method should contain the "instructions" on how to calculate that point. So one can implement points such as:

- `BallPoint` - Follows the ball around.
- `DangerPoint` - The most dangerous thing on the pitch. (If the ball is visible, it becomes the ball. If not, becomes the most likely ball holder. Otherwise it becomes the closest enemy to the goal.)
- `DefencePoint` - The point located exactly halfway between the goal and a `DangerPoint`.

The reason this is useful is that you can then pass these `ActionPoint`s to the Robot's `Navigation` class like so:

```
Fred.MOTION.setDestination(new DefencePoint());
Fred.MOTION.setHeading(new DangerPoint());
```

Making these two calls to `Fred.MOTION` will make 𝔽.ℝ.𝔼.𝔻 follow the `DefencePoint`, wherever it may be and make 𝔽.ℝ.𝔼.𝔻 face the `DangerPoint`, avoiding obstacles and taking the shortest optimal path. 𝔽.ℝ.𝔼.𝔻 will continue to do this until told otherwise.

Coincidentally, this is exactly what the 'Defend' Behaviour Mode is. These two lines are all it takes to describe the defending strategy.

7.4.2 Actions

We have seen how `ActionPoints` work, the remaining question is, what code tells 𝔽.ℝ.𝔼.𝔻 to follow which point? That's where Actions come in.

Actions are objects that extend the `TikTokBase` class, meaning they implement the `TikTokInterface` and require implementing the *tok()* method. Instantiating an Action and setting it as $\mathbb{F}.\mathbb{R}.\mathbb{E}.\mathbb{D}$'s current action via `Fred.ACTION.setAction(yourActionHere)` will ensure that the *tok()* method of your class will be called with every Strategy iteration. Within this method one should check what is going on in the game and take appropriate action. It is these *tok()* methods that make calls to `Fred.MOTION` and set $\mathbb{F}.\mathbb{R}.\mathbb{E}.\mathbb{D}$'s destination and headings to make him move.

Actions work like a finite state machine. Finite state machines may have sub-machines within them and Actions can do something similar. An Action can create a new action and set it as it's sub-action. This means that until the sub-action is finished, the *tok()* method of the parent action will be bypassed and the system will use your action's *tik()* method to call the sub-action. This is all done automatically. If you want to call a sub-action, you can do it within your action by calling `this.enterAction(`*yourSubaction*`)`.

The only problem of chaining actions is that parent actions have no way of "switching" to a different sub-action without the current sub-action terminating. This may be necessary in some higher level behaviour definition action (for example the `Behave` action, which is basically the topmost action of $\mathbb{F}.\mathbb{R}.\mathbb{E}.\mathbb{D}$'s behaviour - it plays football). For this you may extend the `ActionDispatcher` class, which allows you to check different world conditions before allowing the sub-actions to continue. A great example of this is the `Behave` class.

8 Appendix A: Functions in Firmware - firmware.ino File

Function	**Description**
void setup()	Sets up the motors and board.
void loop()	Reads input from serial and launches appropriate function.
void dontMove()	Stops 𝔽.ℝ.𝔼.𝔻. (only wheels. If there are other motors attached, this will not affect them.)
void motorControl(int motor, int power)	Sets the speed of the specified motor to the specified power. Power must be in range from -255 to 255 (0 means stop).
void pingMethod()	Sends ”pang” over the serial port.
void completeHalt()	Stops absolutely everything (all motors).