Radboud University

Introduction Robotics

BSc Artificial Intelligence

Faculty of Social Sciences

Report: the Expedition Robinson Challenge – From Theory to Practice

Group 12: ScorpI/O

Authors:
Max Driessen
s4789628

Jill Muris
s4787579

Vera Tukke
s4766970

Marc-William Verwoert
s4718801

Teachers:
Louis Vuurpijl
Frits Vaandrager
Pim Haselager
Luc Seelen

July 2nd, 2017

Contents

1 Introduction

A lot of computer applications used in our daily lives are systems that perform a set of actions when the user directs them to do so. These systems are made for a specific set of actions; they listen to the user and do not bring new ideas to the table. Nowadays, people want to see systems that come up with creative ideas and create solutions by themselves to be able to reach a goal. These self-deciding systems are called 'agents' [2], and they act upon the environment with the help of sensory inputs. Agents are, for example, used in space projects and in search programs.

For the course 'Introduction Robotics', we had to create our own agent. This agent had to be capable of performing actions in the environment in which it is situated, without being explicitly programmed for that environment. The main assignment for this course was to make our own autonomous agent that had survived a so-called 'dramatic boat incident that occurred near a remote volcanic archipelago in the Pacific'. Some robots survived and washed upon the shores of several isolated islands. The robots had to struggle to stay alive. The robot had to perform several tasks to be able to stay alive. These tasks were, among others: finding paths, following paths, crossing bridges, crossing a seesaw, orienting itself by means of detecting and recognizing salient landmarks in the environment, escaping from a difficult maze, picking up small blocks and moving them from one island to another, distinguishing enemies from friends (by fighting the enemies and showing friendly behavior towards friends).

The main assignment was divided into 4 challenges. These challenges have to be performed on the demo day, the 7th of July. A brief description of the 4 challenges is provided below.

Explore your island and find your way out
Every place on the island can be useful for survival. That is why this first challenge was designed. The first challenge is a combination of the vacuum cleaner task and the maze problem. The vacuum cleaner task is used to explore the island. The maze problem is used to find an escape route. For this challenge, the robot first has to find the grid, after which it has to find all nine grid points. After the grid points have been visited, the blue pillar that marks the start of the maze has to be found by the robot. When the maze has been found, an appropriate sound has to be played. Then, the maze has to be solved to reach a red pillar; when the red pillar is found, happy behavior has to be performed and the robot has to stop.

Run for your life!
There exists evidence for the existence of a dangerous animal on the island. This challenge is designed to run away from the creature to stay alive. The robot starts at a point on a 'racetrack' and has to complete the track twice. Completing the track includes crossing a bridge, crossing a seesaw and following the line. The goal is to complete the track as fast as possible.

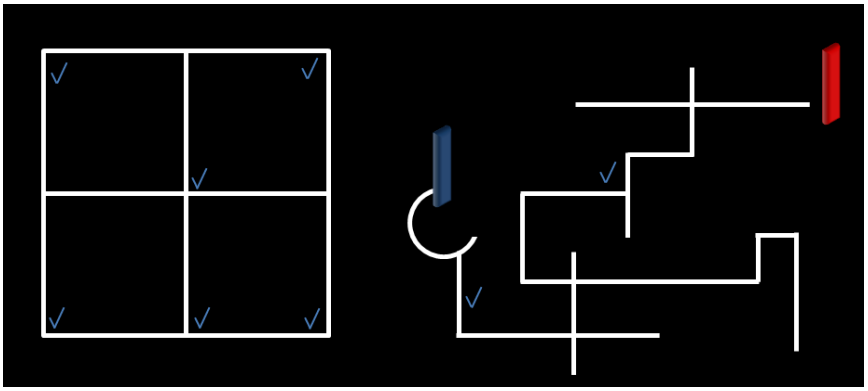
Friend or foe?
The robot should be able to detect whether objects are friends or enemies on a beach. This has to be done: otherwise, the robot would be less likely to survive. The robot has to detect the beach and start at some random location. There are 3 blue and 3 red pillars on the beach which are also positioned at random locations.The red pillars have to be 'killed' and when detecting the blue pillars, romantic behavior has to be performed.

Search and rescue
An island with poisonous plants and food is next to the island the robot is located. The food can be found by following salient landmarks. When the food is detected by the robot, it has to pick it up. With this food, the robot can survive. The robot has to follow a path, across a very small passage, pick up found food, head back 'home' and drop food in a circle when it detects that 'home' has been reached by using salient landmarks.

2 Ecological Niche

An important concept in behavior-based robotics is the ecological niche. The ecological niche describes the current status of an animal (in this case the robot) in its surroundings. In the ecological niche, the relations to enemies and food are very important. In the animal kingdom, if animals find a stable niche in nature, they are more likely to survive. Evolution has helped animals find their ecological niche. Ecological niches are also very important in robotics. If the robot can successfully exist and face the environment, the ecological niche of the robot is stable; if the ecological niche of the robot is not stable, the goal from the robot will probably not be reached. It is therefore important that the person that creates the robot is fully aware of the environment where the robot will be located in [3]. The ecological niches of the robot and thus the robot for the four challenges are described below.

2.1 Challenge 1: Explore your island and find your way out

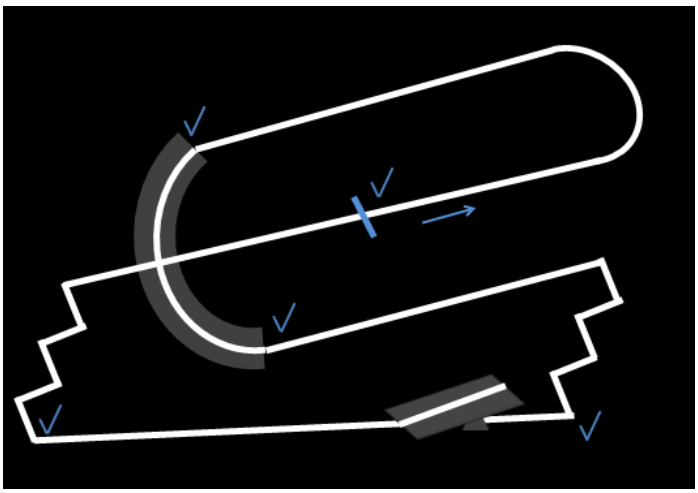


Rendering of the environment of the first challenge

{task = path_following; robot = rgb_sensor; environment = grid}
The rgb sensor in this challenge measures red-values to detect the difference between the white line and the black background. The white lines on the background are positioned in such a way in this challenge that they resemble a 3 by 3 grid. The robot has to use the differences between the black and the white values to stay on track of the white line and to succeed in visiting all nine grid points.The rgb sensor is located at its original position (see Appendix [1]), which is approximately vertically below the ultrasonic sensor and below the grasping hooks of our robot. The rgb sensor was placed in this position, because then it was directly placed above the ground, which resulted in more or less the same red values without too much variation of those values like it would in other placements of the rgb sensor (for example, higher above the ground).

{task = solve_maze; robot = ultrasonic_sensor + rgb_sensor; environment = maze}
The ultrasonic sensor is placed on top of the front end of the robot. The ultrasonic sensor was placed this way, because then it had a decent visual field, without any parts of the robot giving the robot some ‘white noise’, causing the robot to detect parts of itself as objects. This enabled our robot to properly find the blue pillar, to indicate the entrance of the maze. The robot should move towards the maze entry, a not-completely-closed oval surrounding the blue pillar, and then solve the maze. The maze consists of a bunch of white lines positioned and connected in a random way, without any ‘islands’/loops. The robot should follow the lines and has to stop when the red pillar is detected, at the end of the maze, where it displays happy behavior.

2.2 Challenge 2: Run for your life!



Rendering of the environment of the second challenge

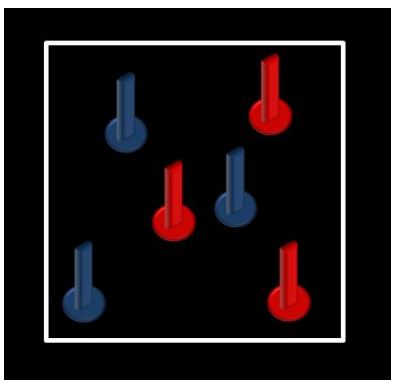
{task = path_following; robot = rgb_sensor; environment = bridge}
We placed the rgb sensor that detects the line the robot has to follow a little higher for this challenge (see Appendix [10]), to prevent the rgb sensor from touching the bridge and be stuck and/or damage the rgb sensor. The bridge consists of an uphill slope and a downhill one, with a line on top that the agent has to follow; to stop ScorpI/O from being top-heavy, his head is removed during this challenge (see Appendix [11]). This means that he does not have access to the ultrasonic sensor and the grabbing hooks, both of which are not necessary during this challenge.

{task = path_following; robot = rgb_sensor; environment = seesaw}
In this part of the challenge, the placement of the rgb sensor mentioned above again avoids the rgb sensor from touching the seesaw and get stuck and/or damage the sensor. The seesaw consists of a platform which can go up and down, pivoting on a point in the middle. The seesaw has a line which lies on top of it.

{task = path_following; robot = rgb_sensor; environment = path}
During this part of the challenge, the rgb sensor does not necessarily have to be higher than normal; the line just has to be followed as fast as possible. However, because this part of the challenge was between the bridge and seesaw for which the rgb sensor does have to be higher than normal, the rgb sensor is also located higher for this part. It is not necessary (and rather impossible with our robot's build) to change the height of the rgb sensor for only the parts were the seesaw and bridge did not have to be crossed. However, the higher placement does lead to different rgb values being registered for the line and the background.

2.3 Challenge 3: Friend or foe?

{task = detecting_pillars; robot = ultrasonic_sensor; environment = black island, bordered by white beach, with three red and three blue pillars}
For this challenge, the ultrasonic sensor is placed at the same position as in the first challenge, again because then it had a decent visual field, without causing the robot to detect parts of itself as the objects, instead of the pillars. The environment is black background of 2 by 2 metres (resembling an island) surrounded by a white line (resembling a beach), and after the white line there is still some some black background. On this island there are six pillars in total. Three of those pillars are colored blue and three of those pillars are colored red. The blue pillars need to be mated by the robot, while the red pillars need to be killed by the robot.

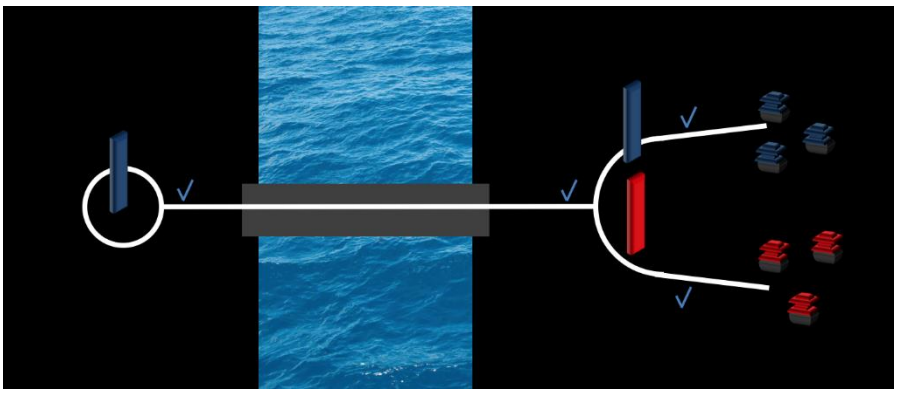


Rendering of the environment of the third challenge

{task =determining_pillar_colour; robot = rgb_sensor; environment =red or blue pillar}
The rgb sensor for this challenge was placed as far to the front as possible (see Appendix [2]). This enables the robot to stop at a certain distance before the pillar and still be able to detect the color of the pillar. When the rgb sensor was in its original position (more backwards), we ran into some issues with the robot not being able to detect the color of the pillar. This happened because the robot couldn't stop any closer to the pillar, otherwise the pillar was unintentionally tackled, but our robot still wasn't able to shine its rgb sensor light on the pillar background to detect its color. In this part of the challenge, the robot discriminates the color of a (red or blue) circle (which has the same color as the pillar) surrounded by a black background. Based on the color detected the robot will perform a certain behavior. When the pillar is detected to be red, the robot will perform its killing behavior which means that it will play an appropriate tune and that the robot will tackle the red pillar. When the pillar is detected to be blue, the robot will perform its mating behavior which means closing its claws and playing an appropriate tune followed by reopening its claws.

{task=kill_or_mate_behavior; robot = tail; environment = red or blue pillar}
For this challenge, the tail of our robot, which resembles the tail of a scorpion, was removed. The reason for this removal, was that the robot would unintentionally sometimes throw down pillars with its huge tail, when it drove in a circle to detect pillars.

2.4 Challenge 4: Search & Rescue



Rendering of the environment of the fourth challenge

{task = path_following; robot = rgb_sensor; environment = bridge}
We placed the rgb sensor that detects the line the robot has to drive on a little higher for this part of the challenge, to prevent the rgb sensor from causing the robot to get stuck and/or damage the rgb sensor when trying to cross the bridge. The bridge consists of an upward slope, a vertical piece, and a downward slope, and is about 30 cm wide.

{task = path_following; robot = rgb_sensor + ultrasonic_sensor, environment = path with pillars next to it}
The rgb sensor is used to follow a path on the floor which leads to the food, with a pillar sitting adjacent to the path somewhere around the middle of its length. The ultrasonic sensor is placed on the 'head' at the front end of the robot. It points straight ahead so it 'sees' the pillar if it is straight in front of the robot.

{task = driving_to_object, robot = ultrasonic_sensor, environment = black underground with red or blue pillar}
The robot drives up to the object until it is 5 centimeters away. The rgb sensor is placed at the left side of the robot; to prevent the robot from knocking down the pillar with its arms, we let the robot drive with a slight turn right.

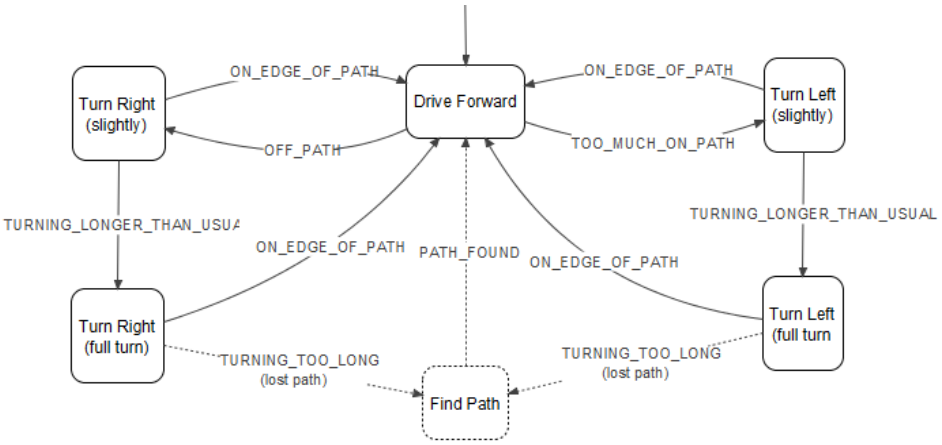
{task = distinguishing_color, robot = rgb_sensor, environment = blue or red pillar}
The robot uses its rgb sensor to check for the color of the pillar. If the pillar is blue the robot opens its arms and continues following the path until it approaches an object (i.e. food). If the pillar is red the robot makes a turn and follows the path in the other direction, again until it approaches an object.

{task = pick_up_food, robot = grabbing_hooks, environment = food pieces on black underground}
When the robot approaches an object and this object is on a black underground, the robot slowly approaches the object and closes its arms, 'picking up the food'. Because of the height of our arms our food pieces are placed on a five centimeters high wooden block so that the food pieces are the correct height for the grabbing hooks to grasp.

{task = drop_food, robot = grabbing_hooks + ultrasonic_sensor, environment = blue pillar 'at home'}
Once the robot has driven back over the bridge, it detects the blue pillar that indicates 'home' using the ultrasonic sensor. It then drives towards the pillar, and when it has reached it, it drops the food that it is holding.

3 Sub-assignments

3.1 Assignment 1: modeling your path following algorithm with FSM



This FSM is implemented (in part or completely) in some behaviors of ScorpI/O. The part where he loses the path and tries to find it again was not implemented due to time limitations and since it rarely ever happens that the robot completely loses the path.

The implemented algorithm works essentially like a P-controller, taking the difference between measured values and the average value (the average of black and white) and using this difference to change the speed of the wheels proportionally so that the robot moves forward rather steadily.

The implemented algorithm has two timers. One measures how long the robot has consecutively been measuring 'black color values', the other measures the same for 'white color values' When the timer threshold for black values has been reached, the robot makes a sharp turn right, and when the threshold for white values has been reached, it makes a sharp left turn.

3.2 Assignment 2: Model Friend or foe

Below, Challenge 3: friend or foe will be explained with the formalisms from MAS chapter 2 [1].

3.2.1. Abstract architectures

E= {e0, e1, e2, e3, e4}

Where:
Initial state e0= A black, two by two meter 'island' which is surrounded by a white line (representing the island's beach). On the island there are three red pillars and three blue pillars. The pillars are positioned at random. The starting position of the robot is also randomly determined.
e1= pillar is standing.
e2= pillar is mated.
e3= pillar is thrown down.
e4= beach is hit or not.

Ac= {a1, a2, a2', a3, a4, a5, a6, a6'}

Where:
a1= drive in a circle, while looking for pillars.
a2= If (pillar==found), drive towards pillar.
a2'= If (pillar==not found), drive forwards and continue with a1.
a3= if (hitBeach== True) then turn around.
a4= Stop at a certain distance before the pillar.
a5= Use the color sensor to detect if the pillar is blue or red.
a6= if (pillar == "blue") then perform_courtship_dance.
a6'= else if (pillar == "red") then perform_killing_behavior.

A run R:

Where en means, that the entire process is repeated n times, where n is the amount of pillars in the challenge. So in this challenge the robot should perform the entire cycle from e0 to en, six times because there are six pillars.

3.2.1 Flow charts/Agents with state

For the flowchart, see appendix [5].

3.2.2 Environment/perception(s) / actions / behaviors

We have a reactive agent for the friend or foe exercise. The robot bases its decision-making processes and thus its actions on the present state. The robot does not have a reference to its history/past.This was done, because we could remove the pillar after our robot has interacted with it and thus there was no point in keeping track of the robot's previous behaviors. The robot can thus purely react to what it perceives at the current moment, without a reference to its history.

We have for action: E →Ac. In which the actions are described in the abstract architecture above. The action function represents the agent's process of decision making.

We also have a perception function, 'see'. The see function is the agent's ability to observe its environment: E → Per.

E = {{pillar found}, {blue pillar, ¬ red pillar}, {red pillar, ¬ blue pillar }}
e1 e2 e3

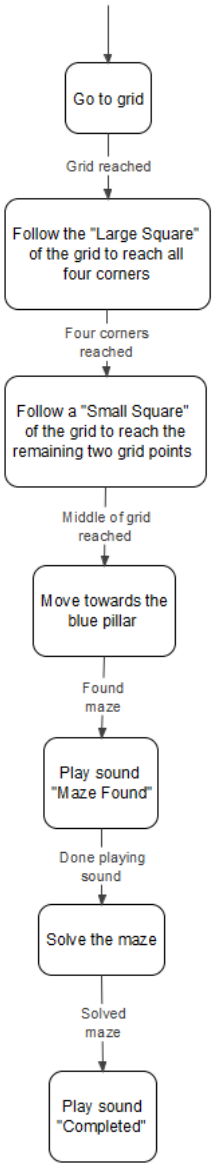
see (e) {p1 if e=e1
{p2 if e=e2 or e=e3

The output of the see function is now a percept p: see : E ↛ Per.
We now however map the environment states to percepts and thus action is now a function: action : Per* ↛ A. Which maps sequences of percepts to actions.

4 Behaviors

4.1 Challenge 1: Explore your island and find your way out

For this challenge, the robot has to visit the nine points of a grid, then find a maze and solve it. Below we will describe the behaviors used for this challenge.



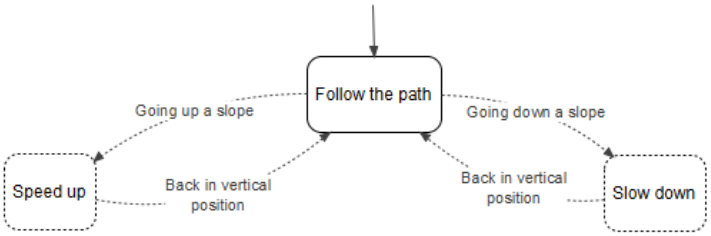
First of all, the robot has to move towards the grid and then turn left so that it is aligned with the path. This is done by the DriveForward behavior, which is suppressed after the turn has been made. Then, FollowPathLeft is activated, which follows the left side of the path by using our path-following algorithm. This algorithm works essentially like a P-controller, taking the difference between measured values and the average value (the average of black and white) and using this difference to change the speed of the wheels proportionally so that the robot moves forward rather steadily. This behavior also contains a timer that counts how long the sensor has been measuring 'black values', which is reset each time it measures 'white values'. When this timer reaches a certain threshold, the robot makes a sharp right turn and counts this as one 'real corner'. When four of these corners have been made, FollowPathLeft is suppressed.

After these four corners have been reached, the robot has visited at least 7 grid points. To reach the remaining two, it first moves over to the other side of the path using the CrossLine behavior (which does nothing else than move the robot over to the other side of the line). Then, FollowPathRight is activated, which in essence does the same as FollowPathLeft, only on the other side of the path. The timer and the 'real turn', in this case, depend on the sensor measuring 'white' values instead of black ones. When the remaining two grid points have been visited, this behavior is suppressed.

Now that the grid has been completed, the robot has to search for the maze, the entrance of which is indicated by a blue pillar. The behavior MoveToPillar makes the robot spin around in place until it sees the pillar, after which it moves towards the pillar. When the robot is close enough to the pillar and the maze is detected, MoveToPillar is suppressed and MazeFound is activated, which plays an appropriate sound. After the sound has been played, SolveMaze takes over, which is the full implementation of our path following algorithm described in section 3.1.
When the robot is close enough to the end of the maze, which is indicated by a red pillar, SolveMaze is suppressed and the Finished behavior becomes active. This behavior makes the robot spin around in a circle, after which it plays an appropriate victory song.

4.2 Challenge 2: Run for your life!

For this behavior, we tried implementing a gyro sensor to make the robot go faster when going uphill and slowing down when going downhill, but we could not get this to work in the end. Instead, we complete this challenge with a single behavior.

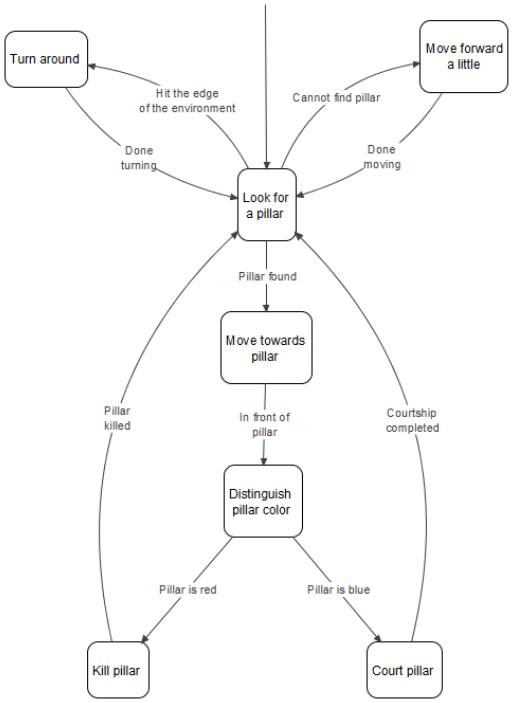


The robot starts somewhere on the 'racetrack', after which it has to complete the track twice. Since following the track is nothing more than following a line, this challenge's single behavior, Race, is nothing more than the full implementation of our line following algorithm, just like SolveMaze in challenge 1. The only place where it differs from SolveMaze is that it never gets suppressed, and the black and white values had to be changed somewhat to accommodate for the higher placement of the rgb sensor described in section 2.0.

4.3 Challenge 3: Friend or foe?

The behaviors we used for the friend or foe challenge are listed below.

The first behavior the robot performs in this challenge is PillarDetector. This behavior has the lowest priority of the two behaviors in the arbitrator. The main course of action in PillarDetector is driving around in a circle trying to detect pillars. When a pillar is detected, the robot will drive towards the pillar. If after a certain amount of time the robot still has not found a pillar, it will drive a bit forwards, after which it will continue with its main course of action. This is done until a pillar is detected. The PillarDetector behavior also checks if the robot has hit the white border (the 'beach'). A beach is detected when a red value is measured resembling that of white and the distance it sees in front of it is bigger than 10 cm, such that it can never be confused with the red values of a pillar color, which can only be measured at 5 cm. When the white line is detected, the robot will turn around, after which it starts searching for pillars again. When the robot is close to a pillar, the PillarDetector behavior is suppressed.



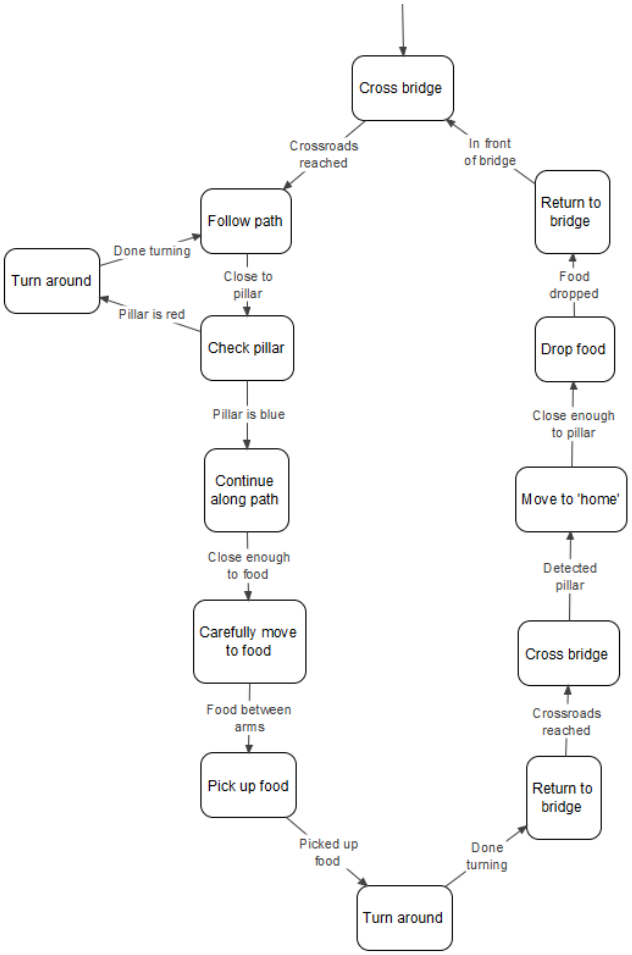
After PillarDetector is suppressed, the next behavior in the arbitrator to be activated: KillOrMate. KillOrMate is the behavior with the highest priority in the arbitrator. When KillOrMate is activated, the robot will measure (via the color sensor) whether the detected pillar is red or blue. When the measured values indicate that the pillar is red, the robot will perform the killing action: it will drive backwards for a certain amount of time, then the appropriate song will be played, after which the robot will drive forwards at a fast speed. This will cause the pillar to be tackled by the robot, causing it to fall down. When a blue pillar is detected instead, the robot will close its grabbing hooks. In this way, the robot 'hugs' the pillar. Then, an appropriate romantic tune is played. Lastly, the claws of the robot are opened again via rotating the connected motor positively.

After any interaction with a pillar, there is a delay of a few seconds, which allows us some time to remove the pillar from the island. When KillOrMate has interacted with a pillar, it will be suppressed and PillarDetector will activate again, causing the process described above to recur.

4.4 Challenge 4: Search & Rescue

For this challenge, the robot has to find food, bring it back to its island and then drop it in a circle when 'home' has been detected. The behaviors we used for this challenge are listed below.

The robot starts at a line that is in front of the bridge. The behavior that we use for this part is the CrossBridge behavior. This CrossBridge behavior is the same as the path following behavior that we also used in challenge 1, but with adjusted speeds. With this behavior, the bridge is to be crossed. More changes in behavior compared to the path following behavior in the first challenge are that we changed the 'red value' of white from 0.45 to 0.35 (due to a higher placement of the sensor) and we lowered the threshold for making a full corner.



When the bridge is crossed, the robot can go left or right. The food that is 'good' can be at the left or at the right side. We do not know the 'good' food is in advance. We decided to first make a right turn. When the rgb sensor detects white for a long time, the robot makes a right turn. Then the CrossBridge behavior suppresses. After this we use FollowPathToObject behavior. In this behavior, we did not use a cornercounter like in the first challenge and the CrossBridge behavior. We used the distance sensor in this behavior. When the distance from the robot to the object, which is either a pillar or foodpiece, is bigger than 30 cm, the robot keeps looking. This behavior suppresses when the object is within a reach of 30 cm.

After that, the MoveToObject behavior is executed. In this MoveToObject behavior, the distance to the object is checked again. If the distance to the object is smaller than 5 cm, the suppress function is used. If the distance is between 5 and 30 cm, the robot drives to the pillar. When the distance is bigger than 30 cm, the robot lost the object and looks for it again.

Then the CloseToObject behavior is used. Blue and black are so similar in rgb-values which makes it hard to see if the object is a food piece or a pillar so we had to implement an extra check. In this behavior we keep track of the amount of times the robot is close to an object and sees the colour black/blue in a blueCheckCounter in Filter. In case the object is a pillar this behavior checks if the pillar is blue or red. When the pillar is red the robot makes a turn backwards and continues following the path to the otherside. When the pillar is blue the robot continues following the path until it detects another object (foodpiece). When the object is a food piece and the robot has already checked and found blue (so the blueCheckCounter is 1), the robot closes its arms and picks up the food. After picking up the food, the robot makes a turn and zigzags back to the white line. The robot then crosses the bridge again with CrossBridge. When the robot sees the final landmark and sees it has a blue/black rgb-value and the blueCheckCounter is 2, it drops the food piece, makes a turn and the process starts all over again to get a second food piece.

5 Design Considerations

The considerations we made for designing our own Lejos robot were based on two principles. First and foremost, the robot was first designed to be practical, effective and efficient with respect to speed, size etc. Secondly, the robot should have a fashionable/distinguishable look. Both of these design considerations will be thoroughly explained below.

5.1 Fashionable design considerations

The robot has a pretty distinguishable look if you compare it to the other robots created within this course (see Appendix [3]). We came up with this look, because in the lectures of robotics, especially in the lecture from Pim Haselager a lot of examples from the animal world were discussed as an inspiration source for the design and the behaviors of a robot. With Pim's talk in mind, we decided to design our robot based on something from the animal world. The robot has two grabbing hooks which are needed to pick up food in challenge 4. These two hooks were in our eyes a bit similar to the pincers of an scorpion. This gave us the idea to design the robot in general with resemblance to a scorpion. The robot therefore not only has the pincers of a scorpion, but also something resembling a scorpion's tail. In the end we thought that we succeeded pretty well in creating a robot which has its own distinguishable, fashionable and awesome look.

5.2 Practical design considerations

The box of the robot which we were provided with predetermined a lot of the design considerations normally done within the creation of a robot. This way, we did not have to make choices in material, battery and other such design considerations. What did remain for us to consider with respect to the practical design of the robot, was to design the robot in such a way that the challenges could be performed smoothly, efficiently and of course successfully. This required some practical design considerations with respect to the size, weight and balance of the robot, but also some with respect to the placement of the sensors, among others. All of the design considerations we made, are discussed below.

5.2.1 Rgb sensor

One of the major design considerations which caused some problems was the placement of the rgb sensor. Like we already explained above, we preferred to place the rgb sensor as close to the ground as possible and a bit below our grabbing hooks. Sadly, this caused some issues with respect to challenges two, three and four. So we decided, based on the fact that you could change your robot between each challenge, to create an easy way of connecting the rgb sensor to multiple places on the robot (see Appendix [4]). This way it would work for each challenge, by simply changing its position when switching between the challenges.

5.2.2 Extra wheel covered in rubber bands

The second challenge also caused some issues with respect to the wheels of the robot. The wheels sometimes did not have enough friction to cross the seesaw or the bridge. To solve this problem, we came to conclusion that we had to add something to our robot to create the necessary friction. This resulted in the usage of an extra (third) wheel covered with rubber bands at the back of our robot (see Appendix [8]). This created the necessary friction to drive up the seesaw and the bridge and thus allowed us to succeed in this challenge.

5.2.4 Grabbing hooks

The design of the grabbing hooks (see Appendix [9]), was mostly based on our own perception of how grabbing hooks should function: two grabbing hooks which could close and open, based on the rotation of a motor. To create a proper motor mechanism behind the grabbings hooks we used some previously created grabbing hooks by other people [4] as inspiration. The logical place of the grabbing hooks on the robot was at the front end of the robot. Some rubber bands between the hooks were also added; these resulted in a higher success rate of correctly picking up an item, because they allowed for a smaller and tighter space around the object grasped within the hooks.

5.2.5 Brick at top and diagonal

Due to some of the previous choices we made, we had to place the brick in such a way that it was possible for us to use the buttons on the brick properly and for the sensors to still be able to be connected to the brick. This resulted in the placement of the brick in a diagonal way on the back of our robot, because this was the most practical solution (see Appendix [6]). This also allowed us to easily read the screen on the brick at all times.

5.2.6 Tail

As mentioned above, we tried to make our robot look like a scorpion. To do this, we created a tail (see Appendix [7]) of lego and added this to our robot. However, since the robot needs to drive under the bridge in challenge two, the tail needed to be made and implemented in such a way, that it would not get stuck under the bridge. The tail was in the placed a little to the side at the back end of the robot, and the tail was made a bit smaller than the original, so that it would not get stuck.

Another problem we ran into with challenge two, was that the robot was a bit top heavy which caused it to sometimes topple over forwards when descending from the bridge. The scorpion tail of the robot to balances out the weight at the front and at the back of the robot a little, so that it does not completely tumble over anymore when descending from the bridge.

5.2.7 Head flexibility

Challenge 2 caused some troubles with the 'head' of our robot, that is the ultrasonic sensor and the grabbing hooks. The head of the robot sometimes got smashed on the ground when descending from the bridge in challenge 2, since it made the robot more front-heavy. To solve this, we had to do a similar thing as with the rgb sensor: we had to design the head in such a way that it could be easily removed between challenges. We also added some rubber bands to the head of the robot (see Appendix [9]), making it more stable and causing it to more accurately see pillars during challenges one, three and four.

6 Solutions, inventions, construction & implementation details

6.1 Solutions/inventions/construction

We came across several problems while programming the behaviors for the robot and designing the robot. One of the first problems which we encountered was the problem with the slipperiness of the wheels and the stability of the robot, when performing certain actions. To solve this problem, a third wheel, covered in rubber bands, was added to the back of the robot. This solution (mostly) stopped the robot from sliding or being out of balance, when making turns and going up slopes. There was also a tail made of lego at the end of the robot, which serves as a counterweight for the robot, to make the robot less front-heavy, causing it to be more stable and balanced. The tail also made the robot go down slopes smoother, and als made it look like a scorpion, which was the design look of our robot we were aiming for.

The hardest design problem was that the placement of the rgb sensor interfered with completing most of the challenges. With each challenge, the sensor needed to be placed in a different position. For some challenges the sensor was preferred to be placed more backwards, while for other challenges it was desired to be placed more in the front and preferably more closer to the ground, to accurately measure the color values of the path and the pillars. However, when the sensor was almost touching the ground, crossing the bridges in challenges two and four with this positioning was not possible, because the sensor would get stuck when trying to go up the slope, forcing the robot to a halt.

Another problem we encountered, was that the distance between the color sensor and the ground was sometimes too big, depending on the placement of the sensor. The values measured by the sensor differed significantly depending on the sensor's vertical positioning. We tried to solve this problem by taping some paper around the sensor to block out surrounding light, but this sadly did not quite work; instead, the sensor values of certain colors are coded differently in the robot's code depending on the placement of the sensor for any particular challenge.

Since the positioning of the color sensor had to be changed with each challenge, we build the part of the robot to which it was connected in such a way that its positioning could be easily changed with each challenge. Like we already explained in section 5.0. (design considerations), we also constructed the head of the robot, or better said the ultrasonic sensor and grabbing hooks, in a flexible way to easily take it off for challenge two.

The two main wheels are located around the middle of the robot, because then the weight would be more evenly dispersed throughout the robot. In the end this also resulted in more balance.

6.2 Implementation details

Below, we will explain two classes we used during each challenge (with some alterations made for particular challenges, but the overall idea stayed the same). We used these two classes for each challenge (see Appendix [9.1]): a SampleRetrieval class, which retrieves sensor values, and a Filter class, which averages sensor values.

The SampleRetrieval class is responsible within each challenge for the retrieving sensor values in the environment. This class can retrieve the color ID, the red mode sample or the RGB sample from the sensor and return it to the caller. We used this class to simplify the way we retrieve values; the functions for retrieving sensor values in Lejos are rather complicated. SampleRetrieval calls these functions, but uses function names that are easier to use in the rest of our program.

The filter class that we used, is a class containing three functions for the different sensor values the robot could retrieve from its environment. These values are the 'red values', 'rgb values' and 'distance values' from the color/rgb and ultrasonic sensors, respectively. All of the three functions in these class take the average of 100 sensor values retrieved from the SampleRetrieval class, and return them simply as a float. With the use of the filter, the sensor values were more accurately measured (less affected by sudden outliers) and retrieving samples became easier (the values being separate floats instead of float arrays).

7 Conclusion

In the end we not only created a robot which looks pretty decent, but we also managed to program the behaviors of the robot in such a way, that the robot was in general quite successful in performing each challenge. However, besides the strong points of our robot, there are still some weaker aspects that remain, with respect to its construction and behaviors. The strong and weak aspects of the robot will be discussed in this part of the report.

7.1 Strong aspects of the robot

The strong aspects of our robot are mostly connected to parts of the construction, the overall look and some of the programmed behaviors of the robot.

One of the strong aspects of the robot's appearance is, in our eyes, that it looks like a robot scorpion made of lego. The scorpion look of the robot resulted in a robot which didn't look so dull like some of the other robots we saw during this course, which is in our opinion a good thing to have achieved; other students have on multiple occasions pointed out that the robot indeed does look like a scorpion.

Furthermore, we programmed certain behaviors very successfully. The robot has, for starters, pretty smooth path-follower behavior. The robot manages successfully to follow the white line (the path) within each challenge. Our robot usually does not shake around very much and, in general, it correctly and robustly follows the line. The robot also almost never gets stuck in dead-ends in the paths of the challenges, which is also a strong aspect of our robot.

We also succeeded in creating an overall stable robot. The robot does not tumble over that easily or fall to its side when performing certain actions, because of some of the solutions and constructions we came up with, which were already discussed before. We mainly succeeded in creating a stable robot via the usage of an extra (third) wheel at the back end of the robot, via the usage of rubber bands and by using a 'tail' made out of lego.

Another one of the strong aspects of the robot, are the distinguishable sounds the robot plays when it has performed certain actions. We received some compliments and weird looks of other people for the sounds our robot made during the trial runs of the challenges. From those reactions we concluded that the sounds our robot made were another strong aspect. We believed this also to be true, because of the fact that we uploaded wav files to the robot to play as sounds which are more clear and distinguishable than the beeping of the robot in a certain rhythm.

Lastly, one of the other strong aspects of the robot is the fact that the percentage of the challenges that we hard-coded is very close to zero. Overall, this resulted in a more robust robot (when the rgb values properly fit the current state of lighting, which is still a weak aspect; see below). The reason for this is that the robot can recover from certain mistakes when, for example, following a path. When the robot hits a dead end, he can turn around a couple of times and easily find its way back to the right track.

7.2 Weak aspects of the robot

Our robot also has some flaws, and sadly not all of our bugs are 'hidden features'. We have (at least) two weak aspects with respect to the coding of the robot's behaviors and its hardware.

The weakest aspect of our robot is the one with respect to its construction/hardware, which was already discussed many times before: the placement of the rgb sensor. Sadly, the placement of the rgb sensor was a hardware issue for us within each individual challenge. Due to its positioning certain parts of the challenges could not be performed successfully, meaning we have to change its positioning with each individual challenge.

Another weak aspect of our robot was that when the light in the environment changed even a little bit, the color values had to be altered, such that the robot still would be able to perform the challenge successfully. This caused issues sometimes, because our behaviors suddenly do not work anymore correctly when the background lighting in the room has changed. We were unable to properly implement a calibrator and thus have to hardcode the rgb values for specific conditions every time.

Those are also the two points of improvement for future designs of robots of our group. For example, to solve the problem of the changing light in the room, we could have created a calibrator class which changed the general values for the environment colors by simply probing values or doing something similar. To solve the problem of the position of the rgb sensor, perhaps it would have been possible for us to change our design in such a way that none of the troubles with the rgb sensor occurred, by placing the sensor on a part that can moves somewhat independently from the rest of the robot.

7.3 General conclusion

We succeeded pretty well in the creation of our own robot which has strong aspects, but sadly also some weaker aspects. Hopefully the robot will perform well during the demo day. In the end we can say that building your own robot to solve certain predefined challenges can be fun and extremely rewarding, but can also be very difficult and frustrating. We stumbled across a lot of hardware issues, programming issues and more; however, we solved almost all of them and in doing so learned a lot about the process of creating your own robot with a team.

8 References

[1] Michael Wooldridge. Chapter 2 Intelligent Agents. In *An Introduction to MultiAgent Systems.* Second Edition, pages 15-42. John Wiley & Sons, May 2009.

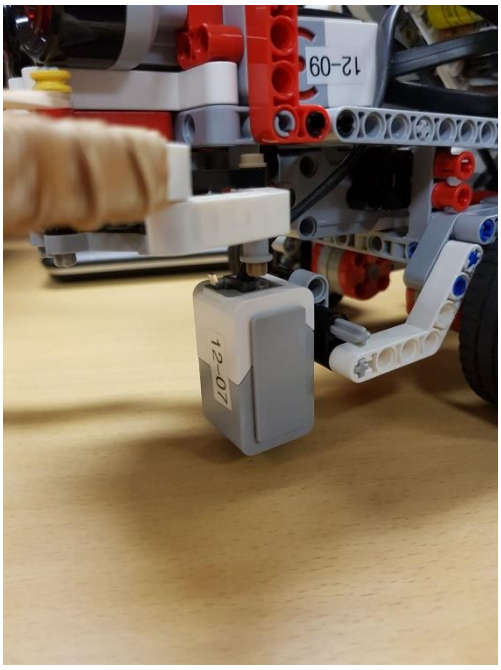
[2] Michael Wooldridge. Intelligent Agents. In *An Introduction to MultiAgent Systems.* Second Edition, pages 3-5. John Wiley & Sons, May 2009.

[3] Ronald C. Arkin. Behavior-based Robotics. First Edition, pages 51 - 52. The MIT Press, 1998.

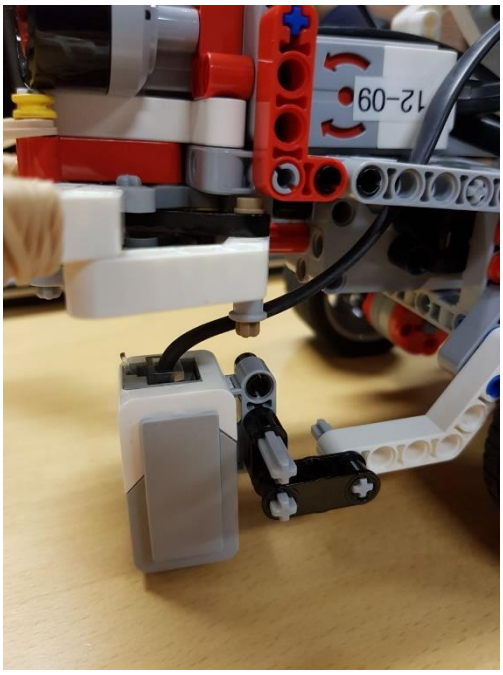
[4] Username W1ll14m. Simplest EV3 Robot Claw/Gripper. http://www.instructables.com/id/Simplest-EV3-Robot-ClawGripper/. 15-04-2015.

9 Appendices

[1] Position of the rgb-sensor in the first challenge

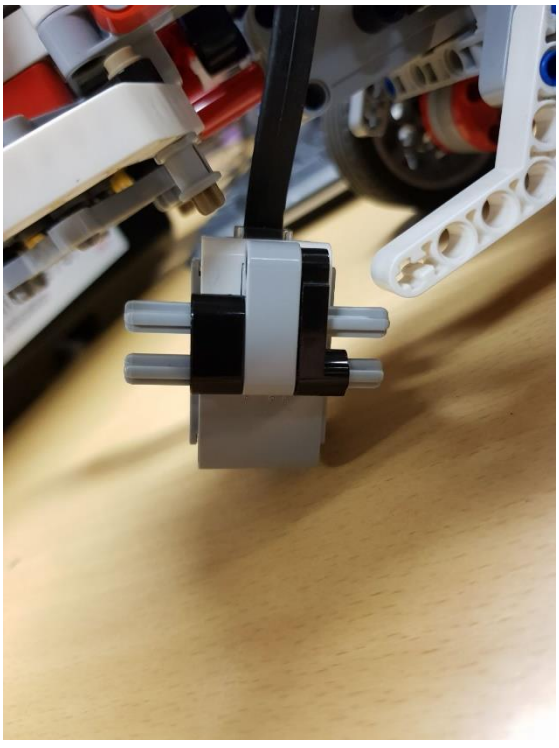


[2] Position of the rgb-sensor in third challenge.

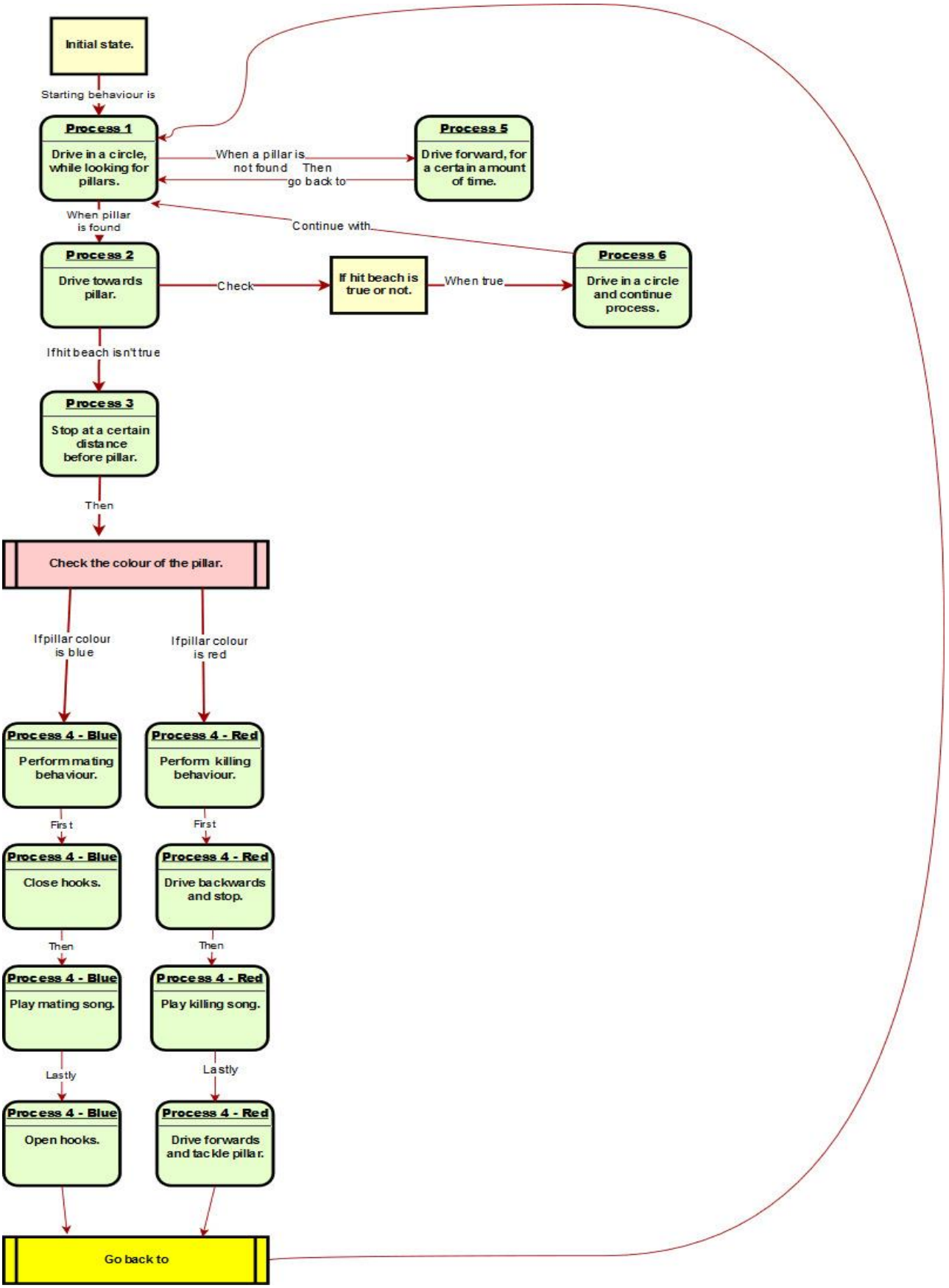


[3] General look of the robot.

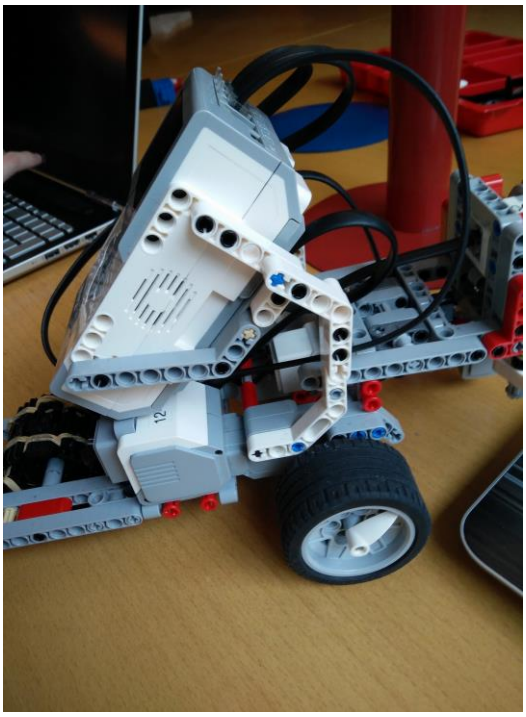
[4] Design consideration of rgb-sensor. The different parts make it easy to change its location on the robot.



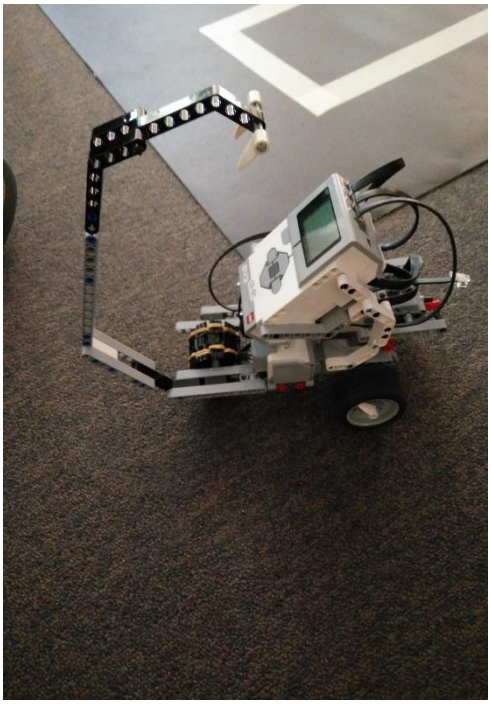
[5] Flowchart by sub-assignment 2 from friend-or-foe.



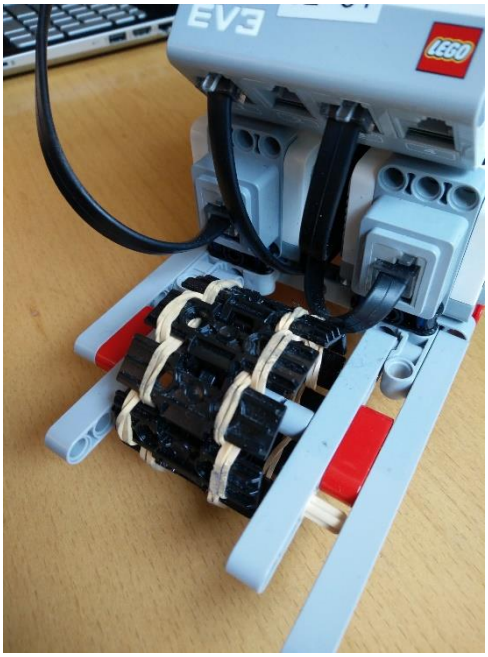
[6] Placement of the brick.



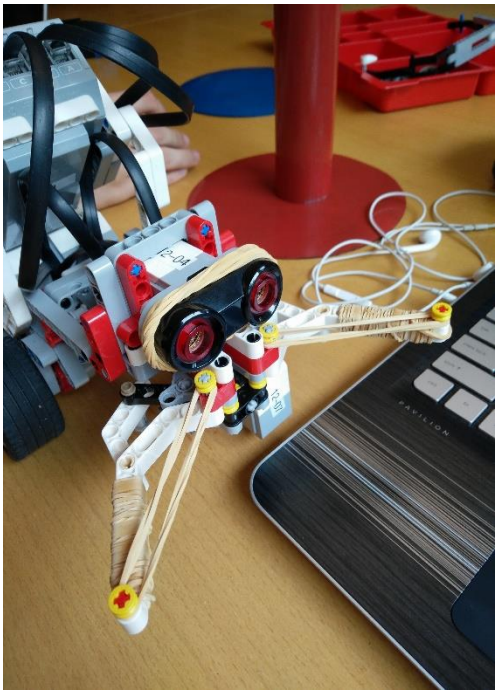
[7] Tail of the scorpion.



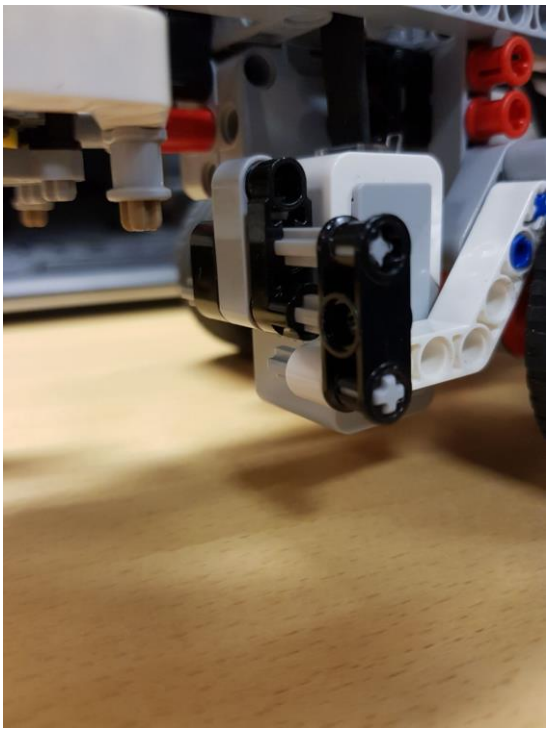
[8] Wheel at the backside of the robot covered with elastics.



[9] Grabbing hooks of the robot.



[10] Rgb-sensor placement in challenge 2.



[11] Robot without head.

9.1 Code Listings

9.1.1 Challenge 1 - Explore your island and find your way out

Main1

```
package nl.ru.ai.MMJV;

import lejos.robotics.subsumption.Arbitrator;
import lejos.robotics.subsumption.Behavior;

public class Main1 {

	public static void main(String[] args) {
		SampleRetrieval sr = new SampleRetrieval();
		Filter sample = new Filter(sr);
		Behavior b1 = new DriveForward(sample);
		Behavior b2 = new FollowPathLeft(sample);
		Behavior b3 = new CrossLine(sample);
		Behavior b4 = new FollowPathRight(sample);
		Behavior b5 = new MoveToPillar(sample);
		Behavior b6 = new MazeFound();
		Behavior b7 = new SolveMaze(sample);
		Behavior b8 = new Finished();
		Behavior [] bArray = {b8, b7, b6, b5 , b4, b3, b2, b1};
	    Arbitrator arby = new Arbitrator(bArray);
	    arby.start();
	}
}
```

DriveForward

```
package nl.ru.ai.MMJV;

import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;
import lejos.utility.Delay;

public class DriveForward implements Behavior {
	static final int SPEED = 200;
	static final double WHITE = 0.35;
	private boolean suppressed = false;
	public Filter sample;
	boolean done = false;

	public DriveForward(Filter s) {
		this.sample = s;
	}

	public boolean takeControl() {
		return !done;
	}

	public void suppress() {
		suppressed = true;
	}
```

```
	public void action() {
		suppressed = false;
		// Move forward in a straight line
		Motor.B.setSpeed(SPEED);
		Motor.B.forward();
		Motor.C.setSpeed(SPEED);
		Motor.C.forward();
		while (!suppressed) {
			// As soon as you find white, turn left and suppress
			if (sample.redValue() > WHITE) {
				done = true;
				Motor.B.setSpeed((int) (SPEED * 0.8));
				Motor.C.setSpeed((int) (SPEED * 0.8));
				Motor.B.backward();
				Motor.C.forward();
				Delay.msDelay(300);
				suppress();
			}
			Thread.yield();
		}
		// Clean up
		Motor.B.stop();
		Motor.C.stop();
	}
}
```

FollowPathLeft

```
package nl.ru.ai.MMJV;

import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;
import lejos.utility.Delay;

public class FollowPathLeft implements Behavior {
	static final double WHITE = 0.45;
	static final double BLACK = 0.1;
	static final int NUMBER_OF_CORNERS = 4;
	static final int SPEED = 300;
	static final int CORNER_TIME = 30;
	private boolean suppressed = false;
	private Filter sample;
	private int cornerCount = 0;

	public FollowPathLeft(Filter s) {
		this.sample = s;
	}

	public boolean takeControl() {
		// Return true if not all 4 corners have been visited yet
		return cornerCount < NUMBER_OF_CORNERS;
	}

	public void suppress() {
		suppressed = true;
	}
```

```
	public void action() {
		suppressed = false;
		double colorValue;
		double diff = WHITE - BLACK;
		double invDiff = 1 / diff;
		double BSpeed;
		double CSpeed;
		int counter = 0;
		while (!suppressed) {
			// If 4 corners have been reached, suppress
			if (cornerCount >= NUMBER_OF_CORNERS)
				suppress();
			colorValue = sample.redValue();
			// If the colorValue is black, increment the counter
			if (colorValue < BLACK + 0.05)
				counter++;
			// If the coloValue is not black, reset the counter
			else
				counter = 0;
			// If the counter reaches a certain threshold, the robot
has to turn
			// longer than usual, meaning that it has reached a
corner:
			// increment cornerCount and make a full turn
			if (counter >= CORNER_TIME) {
				if (counter == CORNER_TIME)
					cornerCount++;
				Motor.B.setSpeed((int) (SPEED * 0.8));
				Motor.C.setSpeed((int) (SPEED * 0.8));
				Motor.B.forward();
				Motor.C.backward();
			}
			// If no corner is reached, continue following the left
side of
			// the path
			else {
				BSpeed = -(invDiff) * colorValue + (1 + 0.1 *
invDiff);
				CSpeed = (invDiff) * colorValue - (0.1 * invDiff);
				Motor.B.setSpeed((int) (SPEED * BSpeed));
				Motor.B.forward();
				Motor.C.setSpeed((int) (SPEED * CSpeed));
				Motor.C.forward();
			}
			Delay.msDelay(10);
			Thread.yield();
		}
		// Clean up
		Motor.B.stop();
		Motor.C.stop();
	}
}
```

CrossLine

```
package nl.ru.ai.MMJV;

import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;
import lejos.utility.Delay;

public class CrossLine implements Behavior {
	private boolean suppressed = false;
	static final int SPEED = 300;
	static final int DELAY = 500;
	public Filter sample;
	boolean done = false;

	public CrossLine(Filter s) {
		this.sample = s;
	}

	public boolean takeControl() {
		return !done;
	}

	public void suppress() {
		suppressed = true;
	}

	public void action() {
		suppressed = false;
		// Turn around while moving slightly forward
		Motor.B.setSpeed((int) (SPEED * 1.5));
		Motor.B.forward();
		Motor.C.setSpeed((int) (SPEED * 0.5));
		Motor.C.backward();
		Delay.msDelay(DELAY);
		while (!suppressed) {
			// If you find black background again after having
crossed the line,
			// suppress
			if (sample.redValue() < 0.15) {
				done = true;
				suppress();
			}
			Thread.yield();
		}
		// Clean up
		Motor.B.stop();
		Motor.C.stop();
	}
}
```

FollowPathRight

```
package nl.ru.ai.MMJV;

import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;
import lejos.utility.Delay;

public class FollowPathRight implements Behavior {
	static final double WHITE = 0.45;
	static final double BLACK = 0.1;
	static final int SPEED = 250;
	static final int CORNER_TIME = 15;
	static final int NUMBER_OF_CORNERS = 2;
	private boolean suppressed = false;
	private Filter sample;
	private int cornerCount = 0;

	public FollowPathRight(Filter s) {
		this.sample = s;
	}

	public boolean takeControl() {
		// Return true if not all 2 inner corners (required for the
last two
		// gridpoints) have been reached
		return cornerCount < NUMBER_OF_CORNERS;
	}

	public void suppress() {
		suppressed = true;
	}

	public void action() {
		suppressed = false;
		double colorValue;
		double diff = WHITE - BLACK;
		double invDiff = 1 / diff;
		double BSpeed;
		double CSpeed;
		int counter = 0;
		while (!suppressed) {
			// Suppress if both inner corners have been reached
			if (cornerCount >= NUMBER_OF_CORNERS)
				suppress();
			colorValue = sample.redValue();
			// If the colorValue is white, increment the counter
			if (colorValue > WHITE)
				counter++;
			// Otherwise, reset the counter
			else
				counter = 0;
			// If the counter passes a certain threshold, the robot
is turning
			// longer than usual, meaning a corner has been reached:
increment
			// the cornerCount and make a full turn
			if (counter >= CORNER_TIME) {
				if (counter == CORNER_TIME)
					cornerCount++;
```

```
                        Motor.B.setSpeed((int) (SPEED * 0.8));
                        Motor.C.setSpeed((int) (SPEED * 0.8));
                        Motor.B.forward();
                        Motor.C.backward();
                        Delay.msDelay(300);
                  }
                  // Otherwise, keep following the right side of the path
                  else {
                        BSpeed = (invDiff) * colorValue - (0.1 * invDiff);
                        CSpeed = -(invDiff) * colorValue + (1 + 0.1 *
invDiff);
                        Motor.B.setSpeed((int) (SPEED * BSpeed));
                        Motor.B.forward();
                        Motor.C.setSpeed((int) (SPEED * CSpeed));
                        Motor.C.forward();
                  }
                  Delay.msDelay(10);
                  Thread.yield();
            }
            Motor.B.stop();
            Motor.C.stop();
      }
}
```

MoveToPillar

```
package nl.ru.ai.MMJV;

import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;

public class MoveToPillar implements Behavior {
      static final double WHITE = 0.35;
      static final int SPEED = 200;
      static final double DIST_TO_PILLAR = 1.0;
      private boolean suppressed = false;
      private Filter sample;
      private boolean done = false;

      public MoveToPillar(Filter s) {
            this.sample = s;
      }

      public boolean takeControl() {
            return !done;
      }

      public void suppress() {
            suppressed = true;
      }
```

```
	public void action() {
		suppressed = false;
		double dist = 10;
		double red = 0;
		while (!suppressed) {
			dist = sample.distanceValue();
			red = sample.redValue();
			// If no pillar is found, turn around in place to
find it
			if (dist >= DIST_TO_PILLAR) {
				Motor.B.setSpeed(SPEED);
				Motor.B.backward();
				Motor.C.setSpeed(SPEED);
				Motor.C.forward();
			// If the pillar is close and white has been found,
suppress
			} else if (dist <= 0.2 && red >= WHITE) {
				done = true;
				suppress();
			// Otherwise, move forward
			} else {
				Motor.B.setSpeed(SPEED);
				Motor.B.forward();
				Motor.C.setSpeed(SPEED);
				Motor.C.forward();
			}
			Thread.yield();
		}
		// Clean up
		Motor.B.stop();
		Motor.C.stop();
	}
}
```

MazeFound

```
package nl.ru.ai.MMJV;

import java.io.File;

import lejos.hardware.Sound;
import lejos.robotics.subsumption.Behavior;

public class MazeFound implements Behavior {
	static final int VOLUME = 150;
	static final int SPEED = 100;
	private File song = new File("Weeeeee Sound.wav");
	private boolean suppressed = false;
	private boolean done = false;

	public MazeFound() {
	}

	/**
	 * Play the song "Weeeeee Sound.wav"
	 */
```

```
	public void play() {
		Sound.playSample(song, VOLUME);
		done = true;
	}

	@Override
	public boolean takeControl() {
		return !done;
	}

	@Override
	public void suppress() {
		suppressed = true;
	}

	@Override
	public void action() {
		suppressed = false;
		// Play a song and suppress
		play();
		while (!suppressed){
			if (done)
				suppress();
			Thread.yield();
		}
	}
}
```

SolveMaze

```
package nl.ru.ai.MMJV;

import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;
import lejos.utility.Delay;

public class SolveMaze implements Behavior {
	static final double WHITE = 0.25;
	static final double BLACK = 0.1;
	static final int SPEED = 300;
	static final int BLACK_CORNER_TIME = 30;
	static final int WHITE_CORNER_TIME = 15;
	static final double GOAL_DIST = 0.2;
	private boolean suppressed = false;
	private Filter sample;
	boolean done = false;

	public SolveMaze(Filter s) {
		this.sample = s;
	}

	public boolean takeControl() {
		return !done;
	}

	public void suppress() {
		suppressed = true;
	}
```

```
	public void action() {
		suppressed = false;
		double colorValue;
		double diff = WHITE - BLACK;
		double invDiff = 1 / diff;
		double BSpeed;
		double CSpeed;
		int blackCounter = 0;
		int whiteCounter = 0;
		while (!suppressed) {
			colorValue = sample.redValue();
			// If the colorValue is black, increment the blackCounter
			if (colorValue < BLACK + 0.05)
				blackCounter++;
			// otherwise, reset the counter
			else
				blackCounter = 0;
			// If the colorValue is white, increment the whiteCounter
			if (colorValue > WHITE)
				whiteCounter++;
			// Otherwise, reset the counter
			else
				whiteCounter = 0;
			// If the blackCounter passes a certain threshold, make a
full turn
			// right
			if (blackCounter >= BLACK_CORNER_TIME) {
				Motor.B.setSpeed((int) (SPEED * 0.8));
				Motor.C.setSpeed((int) (SPEED * 0.8));
				Motor.B.forward();
				Motor.C.backward();
			}
			// If the blackCounter passes a certain threshold, make a
full turn
			// left
			else if (whiteCounter >= WHITE_CORNER_TIME) {
				Motor.B.setSpeed((int) (SPEED * 0.8));
				Motor.C.setSpeed((int) (SPEED * 0.8));
				Motor.B.backward();
				Motor.C.forward();
			}
			// Otherwise, keep following the left side of the path
			else {
				BSpeed = -(invDiff) * colorValue + (1 + 0.1 *
invDiff);
				CSpeed = (invDiff) * colorValue - (0.1 * invDiff);
				Motor.B.setSpeed((int) (SPEED * BSpeed));
				Motor.B.forward();
				Motor.C.setSpeed((int) (SPEED * CSpeed));
				Motor.C.forward();
			}
			Delay.msDelay(10);
			// If you are at the end of the maze, suppress
			if (sample.distanceValue() < GOAL_DIST) {
				done = true;
				suppress();
			}
			Thread.yield();
		}
```

```
            // Clean up
            Motor.B.stop();
            Motor.C.stop();
      }
}
```

Finished

```
package nl.ru.ai.MMJV;

import java.io.File;

import lejos.hardware.Sound;
import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;

public class Finished implements Behavior {
      static final int VOLUME = 150;
      private boolean suppressed = false;
      private File song = new File("Victory Sound.wav");
      private boolean done = false;

      public Finished() {
      }

      /**
       * Play the song "Victory Sound.wav"
       */
      public void play() {
            Sound.playSample(song, VOLUME);
            done = true;
      }

      @Override
      public boolean takeControl() {
            return !done;
      }

      @Override
      public void suppress() {
            suppressed = true;
      }

      @Override
      public void action() {
            suppressed = false;
            // Spin around
            Motor.C.rotate(1440, true);
            Motor.B.rotate(-1440, true);
            // Play a song
            play();
            while (!suppressed){
                  if (done)
                        suppress();
                  Thread.yield();
            }
            // Clean up
            Motor.B.stop();
            Motor.C.stop();
      }
}
```

SampleRetrieval

```
package nl.ru.ai.MMJV;

import lejos.hardware.port.Port;
import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3ColorSensor;
import lejos.hardware.sensor.EV3UltrasonicSensor;
import lejos.robotics.Color;
import lejos.robotics.SampleProvider;

public class SampleRetrieval {
	final static Port COLOR = SensorPort.S3;
	final static Port SONIC = SensorPort.S1;
	private EV3ColorSensor colorSensor;
	private EV3UltrasonicSensor sonicSensor;
	private SampleProvider redProvider;
	private SampleProvider rgbProvider;
	private float[] redSample;
	private float[] rgbSample;
	private float[] distanceSample;

	public SampleRetrieval() {
		colorSensor = new EV3ColorSensor(COLOR);
		sonicSensor = new EV3UltrasonicSensor(SONIC);
		sonicSensor.getDistanceMode();
		redProvider = colorSensor.getRedMode();
		rgbProvider = colorSensor.getRGBMode();
		redSample = new float[redProvider.sampleSize()];
		rgbSample = new float[rgbProvider.sampleSize()];
		distanceSample = new float[sonicSensor.sampleSize()];
	}

	public int colorID() {
		// Get the color ID from the sensor and return it to the
caller.
		return colorSensor.getColorID();
	}

	public float[] redSample() {
		// Get the red mode sample and return it to the caller.
		colorSensor.setFloodlight(true);
		redProvider.fetchSample(redSample, 0);
		return redSample;
	}

	public float[] rgbSample() {
		// Get the RGB sample and return it to the caller.
		colorSensor.setFloodlight(Color.WHITE);
		rgbProvider.fetchSample(rgbSample, 0);
		return rgbSample;
	}

	public float[] distanceSample() {
		// Get the distance to an object and return it to the caller.
		sonicSensor.fetchSample(distanceSample, 0);
		return distanceSample;
	}
}
```

Filter

```
package nl.ru.ai.MMJV;

public class Filter {
	private static final int NUMBER_OF_SAMPLES = 100;
	public SampleRetrieval sample;

	public Filter(SampleRetrieval s) {
		this.sample = s;
	}

	public float redValue() {
		// Get the average of 100 redSample values and return it to the
caller
		float sum = 0;
		for (int i = 0; i < NUMBER_OF_SAMPLES; i++) {
			sum += sample.redSample()[0];
		}
		return sum / NUMBER_OF_SAMPLES;
	}

	public float rgbValue() {
		// Get the average of 100 rgbSample values and return it to the
caller
		float sum = 0;
		for (int i = 0; i < NUMBER_OF_SAMPLES; i++) {
			sum += sample.rgbSample()[0];
		}
		return sum / NUMBER_OF_SAMPLES;
	}

	public float distanceValue() {
		// Get the average of 100 distanceSample values and return it
to the caller
		float sum = 0;
		for (int i = 0; i < NUMBER_OF_SAMPLES; i++) {
			sum += sample.distanceSample()[0];
		}
		return sum / NUMBER_OF_SAMPLES;
	}
}
```

9.1.2 Challenge 2 - Run for your life!

Main2

```
package nl.ru.ai.MMJV;

import lejos.robotics.subsumption.Arbitrator;
import lejos.robotics.subsumption.Behavior;

public class Main2 {

	public static void main(String[] args) {
		SampleRetrieval sr = new SampleRetrieval();
		Filter sample = new Filter(sr);
		Behavior b1 = new Race(sample);
		Behavior [] bArray = {b1};
	    Arbitrator arby = new Arbitrator(bArray);
	    arby.start();
	}
}
```

Race

```
package nl.ru.ai.MMJV;

import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;
import lejos.utility.Delay;

public class Race implements Behavior {
	static final double WHITE = 0.25;
	static final double BLACK = 0.1;
	static final int SPEED = 300;
	static final int BLACK_CORNER_TIME = 30;
	static final int WHITE_CORNER_TIME = 15;
	static final double GOAL_DIST = 0.2;
	private boolean suppressed = false;
	private Filter sample;
	boolean done = false;

	public Race(Filter s) {
		this.sample = s;
	}

	public boolean takeControl() {
		return !done;
	}

	public void suppress() {
		suppressed = true;
	}
```

```
	public void action() {
		suppressed = false;
		double colorValue;
		double diff = WHITE - BLACK;
		double invDiff = 1 / diff;
		double BSpeed;
		double CSpeed;
		int blackCounter = 0;
		int whiteCounter = 0;
		while (!suppressed) {
			if (done)
				suppress();
			colorValue = sample.redValue();
			// If the colorValue is black, increment the blackCounter
			if (colorValue < BLACK + 0.05)
				blackCounter++;
			// otherwise, reset the counter
			else
				blackCounter = 0;
			// If the colorValue is white, increment the whiteCounter
			if (colorValue > WHITE)
				whiteCounter++;
			// Otherwise, reset the counter
			else
				whiteCounter = 0;
			// If the blackCounter passes a certain threshold, make a
full turn
			// right
			if (blackCounter >= BLACK_CORNER_TIME) {
				Motor.B.setSpeed((int) (SPEED * 0.8));
				Motor.C.setSpeed((int) (SPEED * 0.8));
				Motor.B.forward();
				Motor.C.backward();
			}
			// If the blackCounter passes a certain threshold, make a
full turn
			// left
			else if (whiteCounter >= WHITE_CORNER_TIME) {
				Motor.B.setSpeed((int) (SPEED * 0.8));
				Motor.C.setSpeed((int) (SPEED * 0.8));
				Motor.B.backward();
				Motor.C.forward();
			}
			// Otherwise, keep following the left side of the path
			else {
				BSpeed = -(invDiff) * colorValue + (1 + 0.1 *
invDiff);
				CSpeed = (invDiff) * colorValue - (0.1 * invDiff);
				Motor.B.setSpeed((int) (SPEED * BSpeed));
				Motor.B.forward();
				Motor.C.setSpeed((int) (SPEED * CSpeed));
				Motor.C.forward();
			}
			Delay.msDelay(10);
			Thread.yield();
		}
		// Clean up
		Motor.B.stop();
		Motor.C.stop();
	}
}
```

SampleRetrieval

```
package nl.ru.ai.MMJV;

import lejos.hardware.port.Port;
import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3ColorSensor;
import lejos.hardware.sensor.EV3UltrasonicSensor;
import lejos.robotics.Color;
import lejos.robotics.SampleProvider;

public class SampleRetrieval {
	final static Port COLOR = SensorPort.S3;
	private EV3ColorSensor colorSensor;
	private SampleProvider redProvider;
	private SampleProvider rgbProvider;
	private float[] redSample;
	private float[] rgbSample;

	public SampleRetrieval() {
		colorSensor = new EV3ColorSensor(COLOR);
		redProvider = colorSensor.getRedMode();
		rgbProvider = colorSensor.getRGBMode();
		redSample = new float[redProvider.sampleSize()];
		rgbSample = new float[rgbProvider.sampleSize()];
	}

	public int colorID() {
		// Get the color ID from the sensor and return it to the caller.
		return colorSensor.getColorID();
	}

	public float[] redSample() {
		// Get the red mode sample and return it to the caller.
		colorSensor.setFloodlight(true);
		redProvider.fetchSample(redSample, 0);
		return redSample;
	}

	public float[] rgbSample() {
		// Get the RGB sample and return it to the caller.
		colorSensor.setFloodlight(Color.WHITE);
		rgbProvider.fetchSample(rgbSample, 0);
		return rgbSample;
	}

}
```

Filter

```
package nl.ru.ai.MMJV;

public class Filter {
	private static final int NUMBER_OF_SAMPLES = 100;
	public SampleRetrieval sample;

	public Filter(SampleRetrieval s) {
		this.sample = s;
	}

	public float redValue() {
		// Get the average of 100 redSample values and return it to the
caller
		float sum = 0;
		for (int i = 0; i < NUMBER_OF_SAMPLES; i++) {
			sum += sample.redSample()[0];
		}
		return sum / NUMBER_OF_SAMPLES;
	}

	public float rgbValue() {
		// Get the average of 100 rgbSample values and return it to the
caller
		float sum = 0;
		for (int i = 0; i < NUMBER_OF_SAMPLES; i++) {
			sum += sample.rgbSample()[0];
		}
		return sum / NUMBER_OF_SAMPLES;
	}
}
```

9.1.3 Challenge 3 - Friend or foe?

Main3

```
package nl.ru.ai.MMJV;

import lejos.robotics.subsumption.Arbitrator;
import lejos.robotics.subsumption.Behavior;

public class Main3 {

	public static void main(String[] args) throws Exception {
		SampleRetrieval sr = new SampleRetrieval();
		Filter sample = new Filter(sr);
		Behavior b1 = new KillOrMate(sample);
		Behavior b2 = new PillarDetector(sample);
		Behavior[] bArray = {b2, b1};
		Arbitrator arby = new Arbitrator(bArray);
		arby.start();
	}
}
```

KillOrMate

```
package nl.ru.ai.MMJV;

import java.io.File;

import lejos.hardware.Sound;
import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;
import lejos.utility.Delay;

public class KillOrMate implements Behavior {

	private boolean suppressed = false;
	static final int SPEED = 180;
	static final int KILLSPEED = 300;
	static final int VOLUME = 150;
	private File songRed = new File("Western theme.wav");
	private File songBlue = new File("Sexy Music.wav");
	public Filter sample;

	public KillOrMate(Filter s) {
		this.sample = s;
	}

	public boolean takeControl() {
		// Return true if you are close to a pillar
		return (sample.distanceValue() <= 0.05);
	}

	public void suppress() {
		suppressed = true;
	}

	/**
	 * Play the song "Western theme.wav"
	 */
	public void playRed() {
		Sound.playSample(songRed, VOLUME);
	}

	/**
	 * Play the song "Sexy Music.wav"
	 */
	public void playBlue() {
		Sound.playSample(songBlue, VOLUME);
	}
```

```
	public void action() {
		suppressed = false;
		while (!suppressed) {
			double redValue = sample.redValue();
			// If the found redValue corresponds to blue, "hug" the
pillar and
			// play some music, then suppress
			if (redValue < 0.15) {
				Motor.A.rotate(-1000);
				playBlue();
				Motor.A.rotate(1000);
				Delay.msDelay(3000);
				suppress();
			}
			// If the found redValue corresponds to red, drive
backwards, play
			// music and drive forwards to tackle the pillar, then
suppress
			else if (redValue > 0.20) {
				Motor.B.setSpeed(SPEED);
				Motor.C.setSpeed(SPEED);
				Motor.B.backward();
				Motor.C.backward();
				Delay.msDelay(1000);
				Motor.C.stop();
				Motor.B.stop();
				playRed();
				Motor.B.setSpeed(KILLSPEED);
				Motor.C.setSpeed(KILLSPEED);
				Motor.B.forward();
				Motor.C.forward();
				Delay.msDelay(2000);
				Motor.B.stop();
				Motor.C.stop();
				Delay.msDelay(3000);
				suppress();
			}
			Thread.yield();
		}
		// Clean up
		Motor.B.stop();
		Motor.C.stop();
	}
}
```

PillarDetector

```
package nl.ru.ai.MMJV;

import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;
import lejos.utility.Delay;

public class PillarDetector implements Behavior {
	static final int SPEED = 75;
	static final double WHITE = 0.45;
	static final double DIST_TO_PILLAR = 0.6;
	private boolean suppressed = false;
	private Filter sample;

	public PillarDetector(Filter s) {
		this.sample = s;
	}

	public boolean takeControl() {
		return true;
	}

	public void suppress() {
		suppressed = true;
	}

	public void action() {
		suppressed = false;
		double dist;
		double redValue;
		double timer = 0;
		while (!suppressed) {
			dist = sample.distanceValue();
			redValue = sample.redValue();
			// If you are at the edge of the environment, turn around
			if (redValue > WHITE && dist > 0.1) {
				Motor.B.setSpeed(SPEED);
				Motor.B.backward();
				Motor.C.setSpeed(SPEED);
				Motor.C.forward();
				timer = 0;
				Delay.msDelay(3500);
				// If no pillar is in sight, turn around and
increment a timer
			} else if (dist >= DIST_TO_PILLAR) {
				timer += 1;
				// If the timer passes a threshold, move forward
slightly and
				// try searching again
				if (timer > 1000) {
					Motor.B.setSpeed(SPEED);
					Motor.B.forward();
					Motor.C.setSpeed(SPEED);
					Motor.C.forward();
					Delay.msDelay(1000);
					timer = 0;
				}
```

```
                        Motor.B.setSpeed(SPEED);
                        Motor.B.backward();
                        Motor.C.setSpeed(SPEED);
                        Motor.C.forward();
                  }
                  // If you are close to a pillar, suppress
                  else if (dist <= 0.05) {
                        suppress();
                  }
                  // Otherwise, move towards the pillar in sight
                  else {
                        Motor.B.setSpeed(SPEED);
                        Motor.B.forward();
                        Motor.C.setSpeed(SPEED);
                        Motor.C.forward();
                        timer = 0;
                  }
                  Thread.yield();
            }

            // Clean up
            Motor.B.stop();
            Motor.C.stop();
      }
}
```

SampleRetrieval

```
package nl.ru.ai.MMJV;

import lejos.hardware.port.Port;
import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3ColorSensor;
import lejos.hardware.sensor.EV3UltrasonicSensor;
import lejos.robotics.Color;
import lejos.robotics.SampleProvider;

public class SampleRetrieval {
      final static Port COLOR = SensorPort.S3;
      final static Port SONIC = SensorPort.S1;
      private EV3ColorSensor colorSensor;
      private EV3UltrasonicSensor sonicSensor;
      private SampleProvider redProvider;
      private SampleProvider rgbProvider;
      private float[] redSample;
      private float[] rgbSample;
      private float[] distanceSample;

      public SampleRetrieval() {
            colorSensor = new EV3ColorSensor(COLOR);
            sonicSensor = new EV3UltrasonicSensor(SONIC);
            sonicSensor.getDistanceMode();
            redProvider = colorSensor.getRedMode();
            rgbProvider = colorSensor.getRGBMode();
            redSample = new float[redProvider.sampleSize()];
            rgbSample = new float[rgbProvider.sampleSize()];
            distanceSample = new float[sonicSensor.sampleSize()];
      }
```

```
	public int colorID() {
		// Get the color ID from the sensor and return it to the
caller.
		return colorSensor.getColorID();
	}

	public float[] redSample() {
		// Get the red mode sample and return it to the caller.
		colorSensor.setFloodlight(true);
		redProvider.fetchSample(redSample, 0);
		return redSample;
	}

	public float[] rgbSample() {
		// Get the RGB sample and return it to the caller.
		colorSensor.setFloodlight(Color.WHITE);
		rgbProvider.fetchSample(rgbSample, 0);
		return rgbSample;
	}

	public float[] distanceSample() {
		// Get the distance to an object and return it to the caller.
		sonicSensor.fetchSample(distanceSample, 0);
		return distanceSample;
	}
}
```

Filter

```
package nl.ru.ai.MMJV;

public class Filter {
	private static final int NUMBER_OF_SAMPLES = 100;
	public SampleRetrieval sample;

	public Filter(SampleRetrieval s) {
		this.sample = s;
	}

	public float redValue() {
		// Get the average of 100 redSample values and return it to the
caller
		float sum = 0;
		for (int i = 0; i < NUMBER_OF_SAMPLES; i++) {
			sum += sample.redSample()[0];
		}
		return sum / NUMBER_OF_SAMPLES;
	}

	public float rgbValue() {
		// Get the average of 100 rgbSample values and return it to the
caller
		float sum = 0;
		for (int i = 0; i < NUMBER_OF_SAMPLES; i++) {
			sum += sample.rgbSample()[0];
		}
		return sum / NUMBER_OF_SAMPLES;
	}
```

```
	public float distanceValue() {
		// Get the average of 100 distanceSample values and return it
to the caller
		float sum = 0;
		for (int i = 0; i < NUMBER_OF_SAMPLES; i++) {
			sum += sample.distanceSample()[0];
		}
		return sum / NUMBER_OF_SAMPLES;
	}
}
```

9.1.4 Challenge 4 - Search & Rescue

Main4

```
package nl.ru.ai.MMJV;

import lejos.robotics.subsumption.Arbitrator;
import lejos.robotics.subsumption.Behavior;

public class Main4 {

	public static void main(String[] args) {
		SampleRetrieval sr = new SampleRetrieval();
		Filter sample = new Filter(sr);
		Behavior b1 = new CrossBridge(sample);
		Behavior b2 = new FollowPathToObject(sample);
		Behavior b3 = new MoveToObject (sample);
		Behavior b4 = new CloseToObject(sample);
		Behavior [] bArray = {b3, b4, b2, b1};
	    Arbitrator arby = new Arbitrator(bArray);
	    arby.start();
	}
}
```

CrossBridge

```
package nl.ru.ai.MMJV;

import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;
import lejos.utility.Delay;

public class CrossBridge implements Behavior {
	static final double WHITE = 0.35;
	static final double BLACK = 0.1;
	static final int SPEED = 250;
	static final int CORNER_TIME = 20;
	private boolean suppressed = false;
	private Filter sample;
	private boolean done = false;

	public CrossBridge(Filter s) {
		this.sample = s;
	}
```

```
	public boolean takeControl() {
		return !done;
	}

	public void suppress() {
		suppressed = true;
	}

	public void action() {
		suppressed = false;
		double colorValue;
		double diff = WHITE - BLACK;
		double invDiff = 1 / diff;
		double BSpeed;
		double CSpeed;
		int counter = 0;
		while (!suppressed) {
			colorValue = sample.redValue();
			// If the colorValue is white, increment the counter
			if (colorValue > WHITE)
				counter++;
			// Otherwise, reset the counter
			else
				counter = 0;
			// Whenever the robot is on the line longer than the
threshold,
			// a sharp turn is made.
			if (counter >= CORNER_TIME) {
				Motor.B.setSpeed((int) (SPEED * 0.8));
				Motor.C.setSpeed((int) (SPEED * 0.5));
				Motor.B.forward();
				Motor.C.forward();
				Delay.msDelay(1000);
				done = true;
				suppress();
			}
			// Otherwise, keep following the right side of the path
			else {
				BSpeed = -(invDiff) * colorValue + (1 + 0.1 *
invDiff);
				CSpeed = (invDiff) * colorValue - (0.1 * invDiff);
				Motor.B.setSpeed((int) (SPEED * BSpeed));
				Motor.B.forward();
				Motor.C.setSpeed((int) (SPEED * CSpeed));
				Motor.C.forward();
			}
			Delay.msDelay(10);
			Thread.yield();
		}
		Motor.B.stop();
		Motor.C.stop();
	}
}
```

FollowPathToObject

```
package nl.ru.ai.MMJV;

import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;
import lejos.utility.Delay;

public class FollowPathToObject implements Behavior {
	static final double WHITE = 0.45;
	static final double BLACK = 0.1;
	static final int SPEED = 200;
	static final int CORNER_TIME = 15;
	private boolean suppressed = false;
	private Filter sample;

	public FollowPathToObject(Filter s) {
		this.sample = s;
	}

	public boolean takeControl() {
		return sample.redValue()>WHITE;
	}

	public void suppress() {
		suppressed = true;
	}

	public void action() {
		suppressed = false;
		double colorValue;
		double diff = WHITE - BLACK;
		double invDiff = 1 / diff;
		double BSpeed;
		double CSpeed;
		while (!suppressed) {
			// Stop following path when an object is in a range of 30
cm
			if (sample.distanceValue() < 0.30)
				suppress();
			// Otherwise, keep following the path
			else {
				colorValue = sample.redValue();
				BSpeed = -(invDiff) * colorValue + (1 + 0.1 *
invDiff);
				CSpeed = (invDiff) * colorValue - (0.1 * invDiff);
				Motor.B.setSpeed((int) (SPEED * BSpeed));
				Motor.B.forward();
				Motor.C.setSpeed((int) (SPEED * CSpeed));
				Motor.C.forward();
			}
			Delay.msDelay(10);
			Thread.yield();
		}
		Motor.B.stop();
		Motor.C.stop();
	}
}
```

MoveToObject

```
package nl.ru.ai.MMJV;

import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;

public class MoveToObject implements Behavior {
	static final int SPEED = 100;
	static final double DIST_TO_PILLAR = 0.4;
	private boolean suppressed = false;
	private Filter sample;

	public MoveToObject(Filter s) {
		this.sample = s;
	}

	public boolean takeControl() {
		// Return true if the robot is close to an object
		return sample.distanceValue()<0.30;
	}

	public void suppress() {
		suppressed = true;
	}

	public void action() {
		suppressed = false;
		double dist;
		while (!suppressed) {
			dist = sample.distanceValue();
			// Stop with approaching when the object is within a
range of 5 cm
			if (dist < 0.05)
				suppress();
			// Start driving in circles whenever the object is out of
sight.
			else if (dist > DIST_TO_PILLAR) {
				Motor.B.setSpeed(SPEED);
				Motor.B.backward();
				Motor.C.setSpeed(SPEED);
				Motor.C.forward();
			}
			// Slowly approach the object
			else {
				Motor.B.setSpeed(SPEED+100);
				Motor.B.forward();
				Motor.C.setSpeed(SPEED);
				Motor.C.forward();
			}
			Thread.yield();
		}
		// Clean up
		Motor.B.stop();
		Motor.C.stop();
	}
}
```

CloseToObject

```
package nl.ru.ai.MMJV;

import lejos.hardware.motor.Motor;
import lejos.robotics.subsumption.Behavior;
import lejos.utility.Delay;

public class CloseToObject implements Behavior {
	static final double WHITE = 0.45;
	static final double BLACK = 0.1;
	static final double BLUE = 0.15;
	static final int SPEED = 200;
	private boolean suppressed = false;
	private Filter sample;

	public CloseToObject(Filter s) {
		this.sample = s;
	}

	public boolean takeControl() {
		return sample.distanceValue() <= 0.05;
	}

	public void suppress() {
		suppressed = true;
	}

	public void action() {
		suppressed = false;
		double colorValue = sample.redValue();
		// Blue check counter is 1 (or 4 in second run) when the robot
has
		// already passed the first blue pillar. So when the redsensor
is
		// again close to an object and sees the blue/black colorvalue
it
		// knows that this object is food so it can pick up the object.
		if ((sample.getBlueCheckCounter() == 1 ||
sample.getBlueCheckCounter() == 4) && colorValue < BLUE) {
			sample.setBlueCheckCounter(1);
			// Close arms
			Motor.A.rotate(-2000);
			while (!suppressed) {
				Thread.yield();
			}
			// Make a turn
			Motor.B.rotate(-400);
			Motor.C.rotate(400);
			while (!suppressed) {
				Thread.yield();
			}
			int switcher = 0;
			Motor.B.setSpeed(SPEED);
			Motor.C.setSpeed(SPEED);
			// Robot searches for path
			while (colorValue < WHITE) {
				Motor.B.forward();
				Motor.C.forward();
				Delay.msDelay(1000);
```

```
                        if (switcher == 0) {
                              Motor.B.rotate(-100);
                              Motor.C.rotate(100);
                              switcher += 1;
                        } else {
                              Motor.B.rotate(100);
                              Motor.C.rotate(-100);
                              switcher -= 1;
                        }
                  }
                  suppress();
            }
            // Blue check counter is 2 (or 5 in second run) so the robot
has already
            // seen the blue pillar and a food piece. So whenever it again
detects
            // blue/black it is 'home', so the food can be dropped.
            else if ((sample.getBlueCheckCounter() == 2 ||
sample.getBlueCheckCounter() == 5) && colorValue < BLUE) {
                  Motor.A.rotate(2000);
                  sample.setBlueCheckCounter(1);
            } else if (colorValue < BLUE) {
                  // The first pillar is blue so the robot can continue
this path
                  // looking for food.
                  sample.setBlueCheckCounter(1);
                  Motor.B.setSpeed(SPEED);
                  Motor.C.setSpeed(SPEED + 20);
                  while (colorValue < WHITE)
                        Motor.B.backward();
                  Motor.C.backward();
                  Motor.A.rotate(2160);
                  suppress();
            }
            // Otherwise, the pillar is red, so the robot has to make a
turn
            else {
                  Motor.B.setSpeed(SPEED + 20);
                  Motor.C.setSpeed(SPEED);
                  while (colorValue > WHITE)
                        Motor.B.backward();
                  Motor.C.backward();
                  suppress();
            }
            Motor.B.stop();
            Motor.C.stop();
      }
}
```

SampleRetrieval

```
package nl.ru.ai.MMJV;

import lejos.hardware.port.Port;
import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3ColorSensor;
import lejos.hardware.sensor.EV3UltrasonicSensor;
import lejos.robotics.Color;
import lejos.robotics.SampleProvider;

public class SampleRetrieval {
	final static Port COLOR = SensorPort.S3;
	final static Port SONIC = SensorPort.S1;
	private EV3ColorSensor colorSensor;
	private EV3UltrasonicSensor sonicSensor;
	private SampleProvider redProvider;
	private SampleProvider rgbProvider;
	private float[] redSample;
	private float[] rgbSample;
	private float[] distanceSample;

	public SampleRetrieval() {
		colorSensor = new EV3ColorSensor(COLOR);
		sonicSensor = new EV3UltrasonicSensor(SONIC);
		sonicSensor.getDistanceMode();
		redProvider = colorSensor.getRedMode();
		rgbProvider = colorSensor.getRGBMode();
		redSample = new float[redProvider.sampleSize()];
		rgbSample = new float[rgbProvider.sampleSize()];
		distanceSample = new float[sonicSensor.sampleSize()];
	}

	public int colorID() {
		// Get the color ID from the sensor and return it to the
caller.
		return colorSensor.getColorID();
	}

	public float[] redSample() {
		// Get the red mode sample and return it to the caller.
		colorSensor.setFloodlight(true);
		redProvider.fetchSample(redSample, 0);
		return redSample;
	}

	public float[] rgbSample() {
		// Get the RGB sample and return it to the caller.
		colorSensor.setFloodlight(Color.WHITE);
		rgbProvider.fetchSample(rgbSample, 0);
		return rgbSample;
	}

	public float[] distanceSample() {
		// Get the distance to an object and return it to the caller.
		sonicSensor.fetchSample(distanceSample, 0);
		return distanceSample;
	}

}
```

Filter

```
package nl.ru.ai.MMJV;

public class Filter {
	private static final int NUMBER_OF_SAMPLES = 100;
	public SampleRetrieval sample;
	public int blueCounter;

	public Filter(SampleRetrieval s) {
		this.sample = s;
	}

	public float redValue() {
		// Get the average of 100 redSample values and return it to the
caller
		float sum = 0;
		for (int i = 0; i < NUMBER_OF_SAMPLES; i++) {
			sum += sample.redSample()[0];
		}
		return sum / NUMBER_OF_SAMPLES;
	}

	public float rgbValue() {
		// Get the average of 100 rgbSample values and return it to the
caller
		float sum = 0;
		for (int i = 0; i < NUMBER_OF_SAMPLES; i++) {
			sum += sample.rgbSample()[0];
		}
		return sum / NUMBER_OF_SAMPLES;
	}

	public float distanceValue() {
		// Get the average of 100 distanceSample values and return it
to the caller
		float sum = 0;
		for (int i = 0; i < NUMBER_OF_SAMPLES; i++) {
			sum += sample.distanceSample()[0];
		}
		return sum / NUMBER_OF_SAMPLES;
	}

	public int setBlueCheckCounter(int count) {
		// Increment the blueCounter by an int and then return it to
the caller
		blueCounter += count;
		return blueCounter;
	}
public int getBlueCheckCounter() {
		// Return the blueCounter
		return blueCounter;
	}
}
```

10 Peer Review

10.1 Peer review form - Marc

Grade (between 0-10) each of your group members on the following factors.

- *(c1) collaboration: communication and working with others*
- *(c2) contribution: amount of work (both software and writing)*
- *(c3) contribution: showing determination and skill*
- *(c4) contribution: quality of the work*
- *(i) inspiring: quality of ideas*
- *(o) overall: your own weighted average of these aspects*

Student	***C1***	***C2***	***C3***	***C4***	***I***	***O***
Max	*10*	*10*	*10*	*10*	*10*	*10*
Jill	*10*	*9*	*9*	*9*	*9*	*9.5*
Vera	*10*	*9*	*9*	*9*	*9*	*9.5*

Well overall I thought that our teamwork within this robotics course went pretty well. We ran into the occasional problems with respect to bugs in our program and we had some clear hardware problems, but in the end we managed to solve all of them via the usage of the teaching assistants or brainstorming together for a solution. We didn't run into any other problems overall with respect to planning, communicating or teamwork.

I thought that working together with Max, Jill and Vera was great and easy-going. There weren't any issues between each other and all three of them always did their work on time and also always managed to deliver very high quality end results. In my opinion there also weren't any other communication or teamwork issues that I know of. In general Max did the biggest portion of the programming, he especially did the biggest portion of the hardest challenges, which helped the entire project to move steadily forward during this course. So, props to Max for doing a lot of the harder programming work. He was an immense factor to the success of this project. Jill and Vera also did their work with respect to this entire project very well. They programmed most of the parts of the second challenge and in the end delivered a very successful result. Jill also did a lot of the coding for the last challenge and delivered a very good result, so also props to her for doing that. Vera also managed to do a big portion of the report in a high quality way. Especially her work on the poster/flyer of our group was very well done and resulted in a very awesome looking flyer. So in general I had nothing to complain about my fellow group members, it was a pleasure working with them.

10.2 Peer review form - Jill

Student	*C1*	*C2*	*C3*	*C4*	*I*	*O*
Max	*10*	*9.5*	*10*	*10*	*10*	*10*
Marc	*10*	*9.5*	*9.5*	*9.5*	*9.5*	*9.5*
Vera	*10*	*9*	*9*	*9.5*	*9*	*9.5*

At the beginning of the course we worked a bit slow because we were all figuring out how to program in LeJos. We started with working on challenge 1. Max made the sample retrieval, filter and the path following algorithm, I wrote the sounds that are being played when the pillars are found.

Challenge 2: Vera and I used the path-following algorithm of challenge 1 and we updated this. We tried to give the robot extra speed whenever it was riding uphill and we tried to slow it down whenever it was driving downhill. Eventually we found that this did not work so we simply use the path following algorithm of challenge 1 for this challenge.

Challenge 3: Marc programmed most of this challenge, at the end he got some help from Max. I equipped him with the correct wav-files and corresponding code to run whenever a pillar is found.

Challenge 4: Max made a beginning for the challenge. I used his code and finished it.

Report: Vera started with making the layout. She wrote a lot of text, same did Marc. Max and I wrote less on the report but Max did check everything that has been written for grammar and contextual errors whilst I continued working on challenge 4.

Overall I would say that it was pleasant to work with this team. We communicated well and it was clear what everyone was working on. We all worked hard and did our part.

10.3 Peer review form - Max

Student	*C1*	*C2*	*C3*	*C4*	*I*	*O*
Jill	*10*	*9.5*	*9*	*9*	*9*	*9.5*
Marc	*10*	*9.5*	*9*	*9*	*9*	*9.5*
Vera	*10*	*9*	*9*	*9.5*	*9*	*9.5*

Overall, I am very positive about our collaboration for this course. Nobody did significantly less than others, and communication went very well (hence the high grades in the table above).

In the first few weeks, we figured out how Lejos worked and made some basic functions that were required to solve the challenges. I made the path following algorithm, Jill figured out how to play sounds, Marc made the distance sensor work and vera looked at the color sensor.

For the first challenge, I did most of the programming to make the behaviours based on the functions mentioned above. Vera and Jill did most of the second challenge; they tried implementing a gyro sensor to be able to adjust speed based on how the robot is oriented, but this did not work so eventually we opted for the same algorithm as the one that solves the maze in challenge 1. Marc did most of challenge 3 on his own. We ended up having little time for challenge 4, which I started just 2 weeks before the demo day, and Jill is working on finishing it.

During the final two weeks, I mainly worked on programming the robot, making the final touches to the design and the code of all challenges. This meant that I did not have much time to work on this report during the scheduled hours; Marc and Vera did most of the work there, which was very much appreciated. I did work on the report at home, so in the end, everybody ended up doing some of everything.

10.4 Peer review form - Vera

Student	*C1*	*C2*	*C3*	*C4*	*I*	*O*
Max	*10*	*9.5*	*10*	*10*	*10*	*10*
Marc	*10*	*9.5*	*9.5*	*9.5*	*9.5*	*9.5*
Jill	*10*	*9.5*	*9.5*	*9.5*	*9.5*	*9.5*

Overall, I am very pleased with the collaboration between the group members. We had some trouble getting started with the challenges the first few weeks. But after we got the hang of it, it went a lot faster. We first tried to get the hang of how the sensors worked. The first challenge we started on was challenge 1. After a lot of time brainstorming, we started programming. The first challenge was primarily programmed by Max but little parts were programmed by other group members.

Jill and I started working on the second challenge. We used a lot of code from the first challenge. This was the case because in both challenges a line had to be followed. After spending a lot of time on this challenge, we decided it did not work good enough. We decided to just use the path following part of the first challenge. This was a better solution. The third challenge was programmed by Marc. He has spent a lot of time on it which paid off. We did not have a lot of time left for the fourth challenge. This caused some stress. Jill programmed a big part of the fourth challenge and used a part of the code which Max had written.

We all programmed a part of the project to let the robot execute the challenges as good as possible. But eventually, the main part of the programming was done by Max. I made the setup of the report, started with writing the report and made the poster when Max and Jill were busy with writing code for the fourth challenge. Marc and I contributed a lot to the report. I think the workload was equally distributed between the four of us. The communication between the group members went smoothly and we all tried our best.