

G53ARS Assignment

**A Subsumption Architecture Implementation on LEGO Mindstorms EV3 Robots**

**Report**

By

# **Natalia Sharon**

Date of Submission: 21/11/15

Student ID: 4204656

Student Username: psyns3

School of Computer Science, at the University of Nottingham

# Introduction

The purpose of this project is to implement a program between myself and a partner which, when installed into the provided robot, causes it to complete a set of tasks and abide to a set of behaviours. Sensors are used by the robot to provide feedback regarding its surroundings, including a gyro, light and sonar sensors. Calculations are made by the robot based on readings from these inputs which ultimately affect the robots motors and hence its movement. The behaviours include:

* Forage – where the robot must search for a line.
* Follow – where the robot must continue to follow the (not necessary straight) line around the room.
* Avoid – where the robot must successfully avoid any obstacles that lay in its path, and then continue to move along the line.

This report will highlight the development and implementation process of this project, and I shall explain in detail the thought process and algorithms behind each implemented behaviour as well as elaborate on any problems/errors encountered along the way.

## Key

Throughout this report there are finite state machine diagrams representing the states and transitions of behaviours. A circle indicates a state, and arrow shows a transition based on a condition being true and a circle with a glow illustrates the starting state.

Above is a diagram which illustrates the hierarchy of our Subsumption Architecture. Level 0 indicates Forage, Level 1 indicates Follow and Level 2 indicates Avoid. The S symbol shows where the behaviour is supressed and at which conditions this is done so.

# Implementation of Behaviours

## Forage

Upon turning the robot on and running our program, we manually place the robots light sensor over the black track which it will eventually attempt to follow. For 4 seconds the robot reads in a stream of RGB light values, for which a weighted average is calculated, and then are placed into an array. An average is calculated across the black sensor input values - this is then repeated over the white floor and a white average is also calculated. A static threshold is then calculated by adding an offset value of 400 to the black average to account for any imbalance regarding light projection around the track (as the track directly underneath the light will be lighter). A dynamic threshold is calculated and constantly updated with each movement, this will be explained in the Follow behaviour. Once the robot has ‘lost’ the black line, (which it will have initially as it will be placed in the middle of the track), a timer is set (this resets every time the robot cannot detect the track). Once this timer reaches 3 seconds, it can be assured that the robot is not on track and must search for the line, and hence the Forage task executes in attempt to locate or re-locate the track.

The behaviour of this task causes the robot to move in a circular motion by setting the right motor to 0 and the left motor to 20. The speed of the right motor increases by 1 every half a second in order to increase the diameter of the circular motion and therefore allowing the robot to cover as much space as possible in a short amount of time whilst it searches for the black line. Whilst this occurs, the Follow method is running simultaneously – however the arbiter task is not ‘paying attention’ to its outputs – and is constantly reading in the light values which the robot traverses across. The moment the input of the light sensor is below the threshold (which classified as part of the track), the variable ‘blackFound’ is set to true and the arbiter task begins to assign the outputs of the Follow task to the robots motors, now paying no attention to the Forage tasks’ outputs.

**Figure 1 - FSD of Forage. SR represents the right motor**

SR += 1

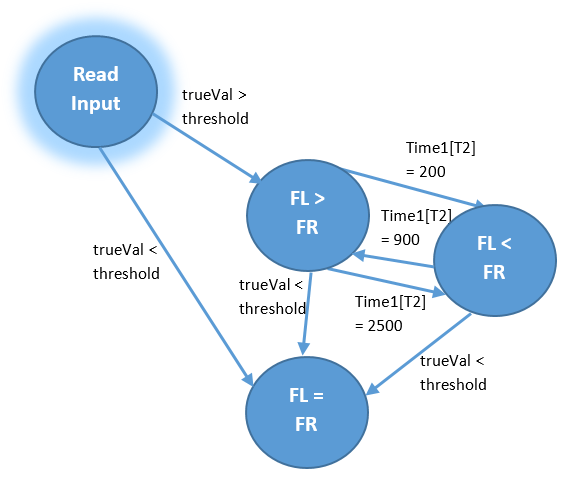
Wait 500 ms SR += 1

Wait 500 ms SR += 1

SR = 0

As the timer is constantly resetting and we keep track of the amount of time the robot has not found the track for, the robot is able to perform the search behaviour at any time and not at just the beginning.

## Follow

As soon as the black track has been found, the robots primary behaviour switches to Follow. Within this task, the light values are constantly being read in and compared to the threshold. A decision is then made based on this comparison – if the light value is below the threshold (and therefore black) then the motors continue at the same speed as it is on track, meaning the robot appears to move in a straight line. If a value read in is above the threshold and therefore white, the robot halts and attempts to relocate the black track. The method used to check the immediate surroundings for the track is as such:

A timer is set and values are still being continuously read in. The robot searches to its right for 200 milliseconds, it then checks to its left for a further 900 milliseconds (meaning it has turned past where it initially halted and is checking the other side for black values), and then revolves back to searching the right until the timer reaches 2500. If the robot still has not located the track be checking left and right, once the timer is up it turns all the way around to the left (if the robot has not found it now, the timer checking whether the robot has lost the line would have reached its limit and Forage behaviour will be induced). Once the robot picks up where the track is again, it will continue moving forward once more.

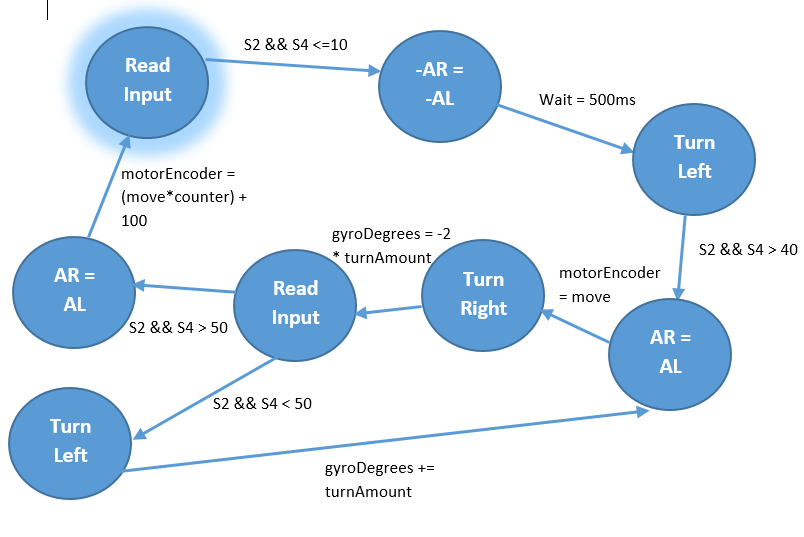
**Figure 2 - FSD of Follow. FL and FR represent motors**

Using this algorithm, the robot can move clockwise or anticlockwise around the track, and additionally follow turns within the track, be it a curved line or a sharp corner.

In order to make a more robust solution, we update the black average and threshold value constantly to account for any changes in the lighting around the track. We do this by keeping an array of 10 values and constantly reading in a value below the threshold – the array always keeps the 10 most recent values and each time it is updated, a new threshold is calculated.

For example, if a robot is moving towards a lighter area of the track, the input values gradually get higher and higher, and so the threshold value increases accordingly, meaning by the time the robot has found the lightest part of the track, it is able to detect it as part of the track.

## Avoid

The robots’ two sonar sensors, positioned at its front, are constantly retrieving an input whilst the robot is moving. However, the Avoid behaviours outputs do not have any affect until the variable obstaclePresent is equivalent to true, which occurs when both sonar sensors retrieve a value of less than 10, meaning that an obstacle is directly ahead and close. Upon this condition becoming true, the robot reverses slightly, giving it enough room to turn to the left and away from the object, until it cannot detect it anymore. The robot then moves forward by x amount, before stops again to turn to its right and check if the obstacle is still present. If the obstacle is still present, the robot turns back so it can move x amount again, and this is repeated until there is no object present to its right. Once the passage is clear, the robot moves by a set amount back towards the line (where the set amount takes into consideration how many times the robot had moved distance x). Whilst it is moving back, the obstaclePresent variable is set to false, so that when the robot stops on the track the Arbiter Task switches back to using the outputs from the Follow behaviour.

**Figure 3 - FSD of Avoid. AR and AL represent Motors**

We had many ideas which were more dynamic (meaning the robot would have been more adaptable to different types of obstacles) and would have provided a smoother motion, however due to lack of time and knowledge of how to implement the better options, we went with the currently implemented idea.

# Interaction of Behaviours

The structure of our code follows the Subsumption Architecture – which is greatly used with behaviour-based robotics. The robots behaviour is decomposed into a subset of behaviours (in this case, the 3 described above) which are ordered within a hierarchy. Essentially, the program runs all tasks simultaneously, however the Arbiter task determines which of the tasks’ outputs to follow.

# 

**Figure 4 - Subsumption Architecture**

# Personal Reflections

I believe that the entire workload – including ideas, constructing algorithms and implementing/testing code – was appropriately divided between my partner, Joshua Twynham, and myself. We attended all workshops and discussed ideas outside of our limited time within the workshop before taking it in turns to produce algorithms and implement our ideas into the code.

My main contributions were designing the Avoid Tasks’ algorithm and implementing it. In addition to this, our implementation of the dynamically updating threshold and search algorithm were primarily of my doing. However, we both worked together to produce all behaviours and algorithms. For example, Josh would be testing out the robot on the track and reading back output values whilst I watched its behaviour and updated our variables and conditions accordingly, or Josh would be writing a few lines of code whilst I proof read to ensure it implemented the idea which we had just discussed, and vice versa.

# Discussions and Conclusions

Our biggest achievement regarding this project would be the Forage behaviour, as this worked every time without fail.

We made improvements regarding implementation of the Follow behaviour, as previously the robot traversed along the edge of the track, constantly making left and right movements to weave to and from white and black – this worked only in one direction however. After changing this to our current implementation of steering in a straight line whilst on the track, the motion was smoother and considerably more aesthetic, as well as working under all conditions. The only weakness concerning our Follow behaviour would be that when traversing anti-clockwise, the movement is less smooth as it is when traversing the track clockwise. The reason being that as it always checks first to its right for the track; and so in this scenario, the track would always be on the robots left. The robot still completes these turns in a fitting amount of time, however when traversing clockwise the time taken for the turn and the motions smoothness is more optimum.

The Avoid behaviour has very little, perhaps close to no, flexibility regarding the range of obstacles that it must avoid. Our method is particularly static, as although we attempt to allow the robot to continuously check if the object is still there, and move accordingly either further away or towards the black line, the method to turn to the black line involves the robot traversing a fixed length towards that direction. The length which it travels back to the track allows the robot to return to the line and continue following it, but only for that specific object and track.

Unfortunately during the last week, instead of hastily attempting to implement the observe method, we worked on perfecting and optimizing the code which we already had to ensure the first 3 behaviours worked as best as we could get them too, and so we did not have enough time to implement Observe.

Evaluating how we went about this project, if I were to re-do this task I would initially take some time to study the architecture to be implemented and analyse how the behaviours would interact with each other through this way, rather than diving straight into implementing the behaviours which was done in this case.