ECE 3502 Mariobot Spring 2015
Carina Cai, Evaristo Koyama

Documentation of Mariobot Program

Overview

The objective of the project is to design and implement a LabVIEW program on NI MyRio to control iRobot (Mariobot) to autonomously complete a course with obstacles, hills and traffic lights.

The course consists of the road(gray mats), boundaries (white duct tapes), obstacles (bricks with orange circles above them), barcodes (masking tapes), hills (up and down), and traffic lights (alternating blue and red LED circles). The Mariobot needs to stay within bounds, avoid and get around obstacles, navigate up and down the hills, and read the barcode and act according to the instructions. The barcodes are 4 bit long with an initial line. ((mat - tape) = 0; (mat- tape - tape) = 1) Five types of barcodes are valid in our example: speed up, slow down, sharp left turn, sharp right turn, stop line.

Sensors used:

Sensor	**Purpose**
Cliff sensors Reading [0, 4096]	To detect boundaries and barcodes Mat: (~100,~300); Barcode: (~500, ~850); Boundary: (1000+)
Bumpers	To detect obstacles
Accelerometer	To detect orientation of the Mariobot on the hills
Angle	To record the angle turned in one state
Distance	To record the distance traveled in one state
Camera	To detect traffic lights and obstacles

State Variables:

Variable	**Explanation**
Speed	Speed in drive and PID
Turn type	Whether it's a sharp left or a sharp right or a normal turn
Barcode: int array	The time elapsed during each line of tape
x value of the obstacle	The position of the obstacle seen on the screen
Direction in obstacle & boundary handling	Direction of the robot before it detects an edge while it's trying to avoid obstacles

Detailed Explanation of the Program

High Level:

We adopted a hierarchical approach to simplify implementation, allow flexibility and improve readability. The high level state diagram is shown below (Figure 1).

The boundary had priority over all other cases because going out of bound is an immediate failure. So the program was designed to handle boundary case first as soon as it encountered the road edge, as shown in the transition from ***Avoid obstacles*** to ***Avoid obstacles and boundary***, and the transition from ***Adjust to align*** to ***Stay in bound***.

The speed during barcode, edge, and obstacle handling was constant, meaning it was not affected by the *speed up* and *slow down* instructions. The barcode reading speed was slower to obtain better accuracy. The turning speed was also constant since the angle measurements are not very accurate and would change the turning angle.

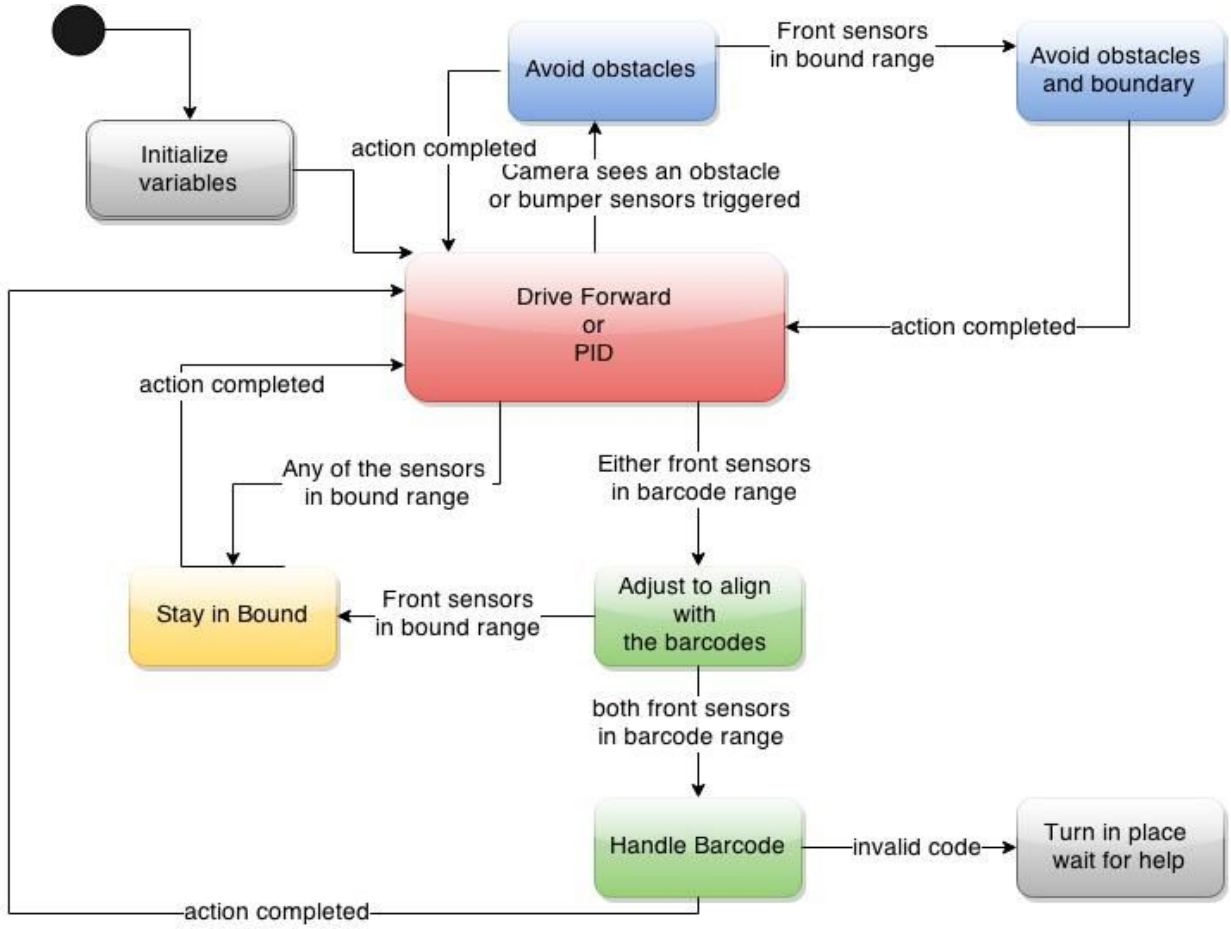

Figure 1

To add substates, first insert a white *region* in the state, and an initial symbol (a black circle). Then multiple substates can be added in that state.

Initialize Variables:

This state is for initializing all state variables such as speed. The initial speed is equal to the max wheel speed given at the beginning.

Drive and PID:

We used the x axis reading of an accelerometer to determine whether the robot is driving uphill or downhill, and we used X and Y axes to determine orientation on the slope. Z axis could not differentiate between uphill and downhill and had a narrow range of values. When it was driving uphill, $x > 0$; when it was going downhills, $x < 0$. We had two thresholds for transitioning between go straight and PID control to allow hysteresis. The state diagram is shown in Figure 2.

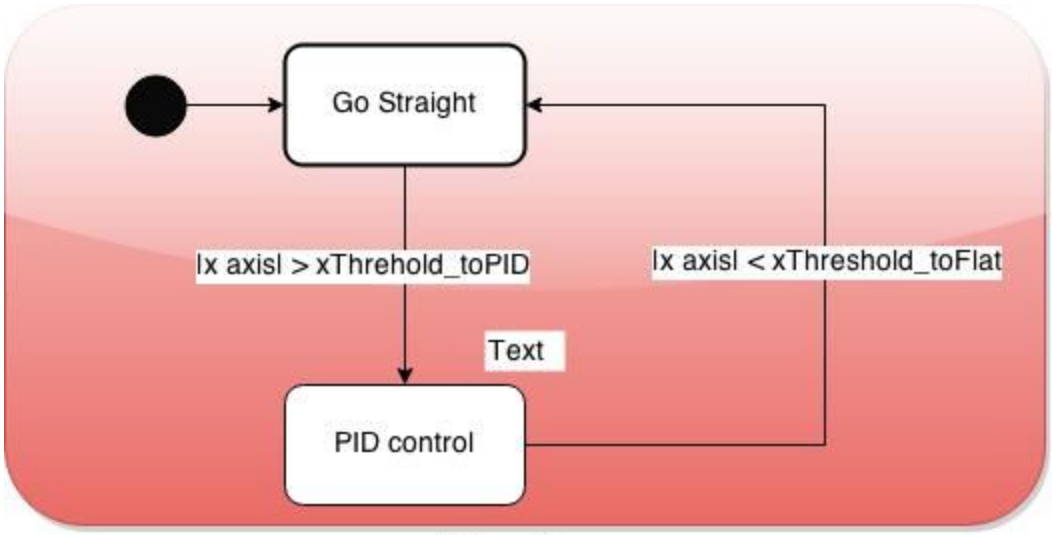

Figure 2

Stay in Bound:

Different types of turns depended on which sensor was triggered and the *turn type* variable. If the side sensors were triggered, then the robot only needed to turn a little bit until it was off the boundary. If the front sensors were triggered, then it turned a set degree. If the *turn type* specified a sharp turn, then the Mariobot would follow the *turn type* regardless of which front sensor was triggered. The decision to only use the front sensor for sharp turns was based on the assumption that the Mariobot would hit the boundary head on after reading the sharp turn barcode. The dummy state was necessary because hierarchical states needed a initial state and the dummy state simplified the transition between high level states. However, we encountered the situation where it got stuck in the dummy state because the sensor value dropped below the threshold in the middle of the transition. So we added a transition from Dummy to Action completed if no sensors are in the bound range. The diagram is shown in Figure 3. *Side right, side left, front right, front left* means that those sensors values are in the bound range. [!] sign indicates otherwise.

We made the assumption that one sensor would always be triggered before the others when the *turn type* is normal and therefore the program would not be confused by multiple possible transitions.

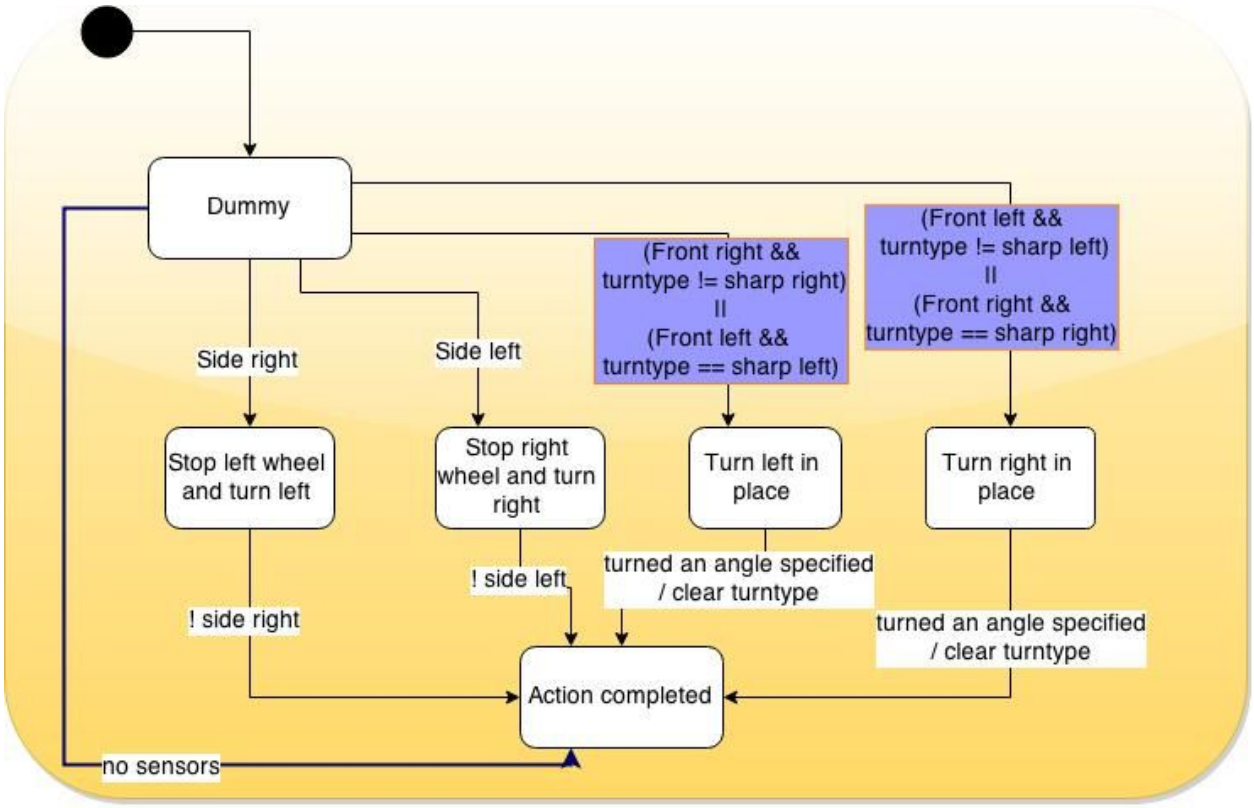

Figure 3

Avoid Obstacles:

We used both camera and the bumper sensors to detect obstacles. There are mainly two benefits of using the camera. First, the Mariobot could react early and actually avoid the obstacles. Second, it could learn the position of the obstacle more accurately. However, the vision was subject to environmental lighting, so it needed to be calibrated every time and may not work if the lighting changed during the course. The bumper sensors were still used in case the vision failed. The state diagram is shown in Figure 4.

In the bumper sensor case, the Mariobot first backs up, then turns a set angle, then goes forward for a certain distance, and finally turns back. It needed to go forward to get around the obstacle. It needed to turn back to face the same direction as before especially when on the hills, otherwise the PID may take the Mariobot to the wrong direction.

In the vision case, the Mariobot performed the same sequence of actions except for backing up. In our design, we took advantage of the x value of the position of the obstacle on the screen to vary the forward distance, so that it was less likely for the Mariobot to hit the edge when it only needed to move a little bit to get around the obstacles. The turn angle in this case was 90 degrees. This helps make detecting the obstacle again easier if the obstacle happens to be wider. The boundary case is also easier to design.

The bottom half of Figure 4 shows the approach of handling boundary with the obstacles. As stated before, the boundary had priority. We only used front sensors to detect edges during obstacle avoiding because we decided that it was unlikely that only the side sensors would sense the boundary. The algorithm was to turn 180 degrees first, and go for a certain distance, turn 90 degrees to check if there was an obstacle. If no obstacle was present, then it would transition to ***Drive forward and PID***. Otherwise it turns back 90 degrees towards the obstacle. If the Mariobot hit the boundary on the other side, then it will repeat the algorithm, searching for an opening the other direction. This happens whether the Mariobot detected the obstacle with a camera or hit the bumper sensors. In the case with the camera, we are only dealing with 90 degree turns. It makes detecting the obstacle again easier. In the case with the bumpers, it deals with the scenario when both left and right bumpers are triggered. If it hit the obstacle and turns away from the boundary, then it got past the obstacle. If it hits the obstacle and turns toward the boundary, then it eventually turns a greater angle away and makes it easier to hit the right bumper.

Case structures are very useful to combine similar states into one single state.

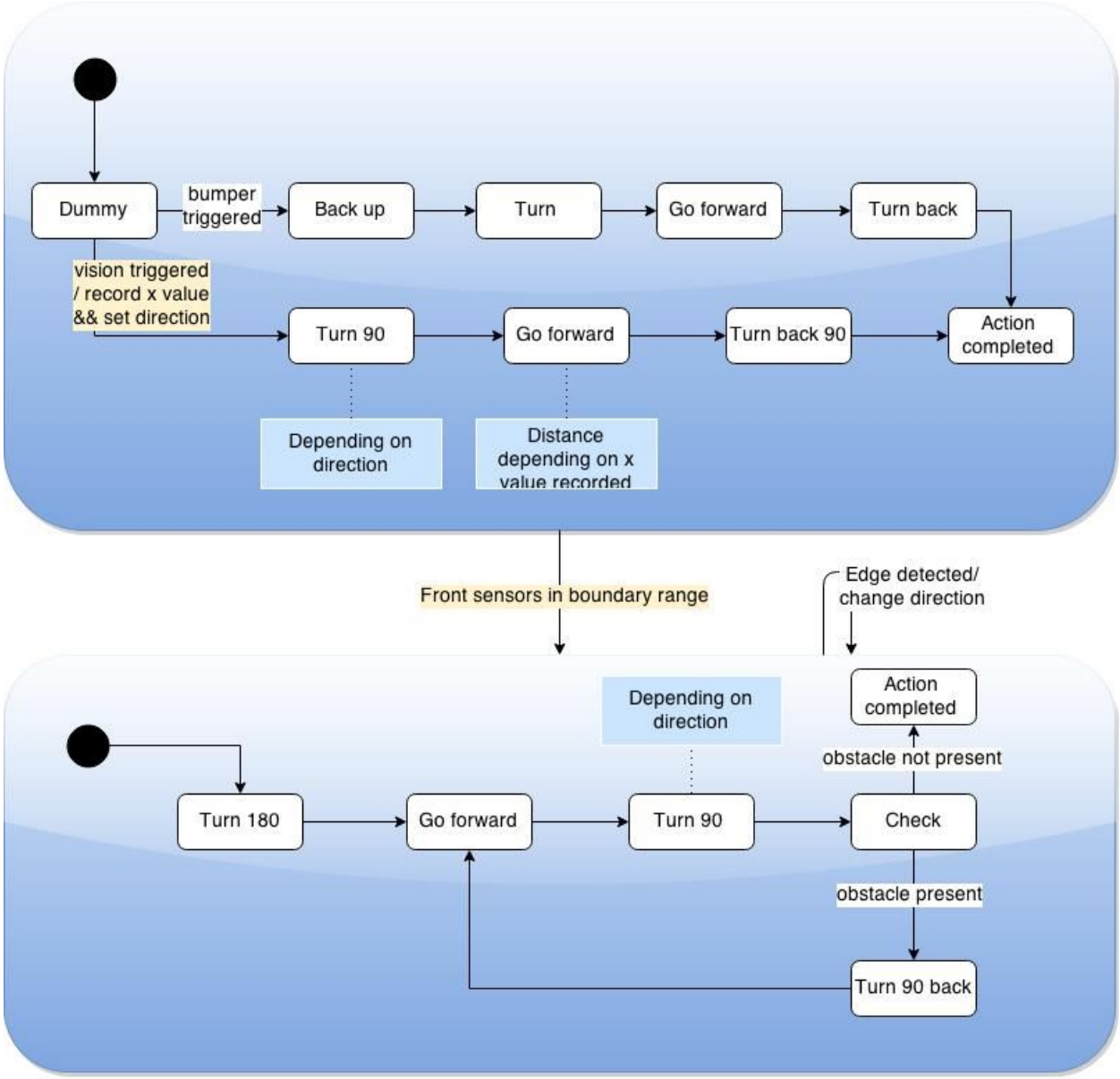
Dummy
bumper triggered
Back up
Turn
Go forward
Turn back
vision triggered / record x value && set direction
Turn 90
Go forward
Turn back 90
Action completed
Depending on direction
Distance depending on x value recorded
Front sensors in boundary range
Edge detected/ change direction
Depending on direction
Action completed
obstacle not present
Turn 180
Go forward
Turn 90
Check
obstacle present
Turn 90 back

Figure 4

Barcodes:

Only two front sensors were used for barcode reading. Because the sensor range for barcodes was between the range for the road and the range for boundary, we made the Mariobot to stop and wait for a bit to allow the sensor values to settle down when it sensed a barcode. The idea is that nothing stops instantaneously. The Mariobot, when told to stop, would have its front sensor on the boundary/bar code. The ***Barcode states*** could also transition to ***Stay in bound*** anytime an edge was detected. A potential problem that was that if the Mariobot was not aligned with the barcode well or it was really close to the edge, and it hit the edge in the middle of barcode reading, it would handle the edge first and ended up ignoring the barcode completely or getting invalid readings. Therefore, it is important to align the Mariobot with the barcode at the beginning. It adjusted by turning in place at a speed that was faster than the drive speed. If it turns at the same speed or slower, then there is a bias towards the first sensor. There was a transition from ***Adjust*** to ***Drive*** because both front sensors may leave the barcode when it was turning in place.

After the Mariobot was aligned, the program transitioned to ***Handle Barcode.*** It first needed to back up to be able to read the full width of the first initial bar, because our algorithm was very much dependent on timing and the width of the first initial line. We used an array of size 5 (4 bits + 1 initial) to record the time elapsed during each bar. In the ***Filter state***, we translated the time recorded to binary bits by dividing all values in the array by the first value and subtracting 1. To simplify comparison, we then translated the barcode bit array to a decimal number in ***Decode*** state. The decimal version of the barcode would then determine the action to be taken. If the barcode was invalid, the program entered an error state where the Mariobot would just turn in place slowly to express its confusion.

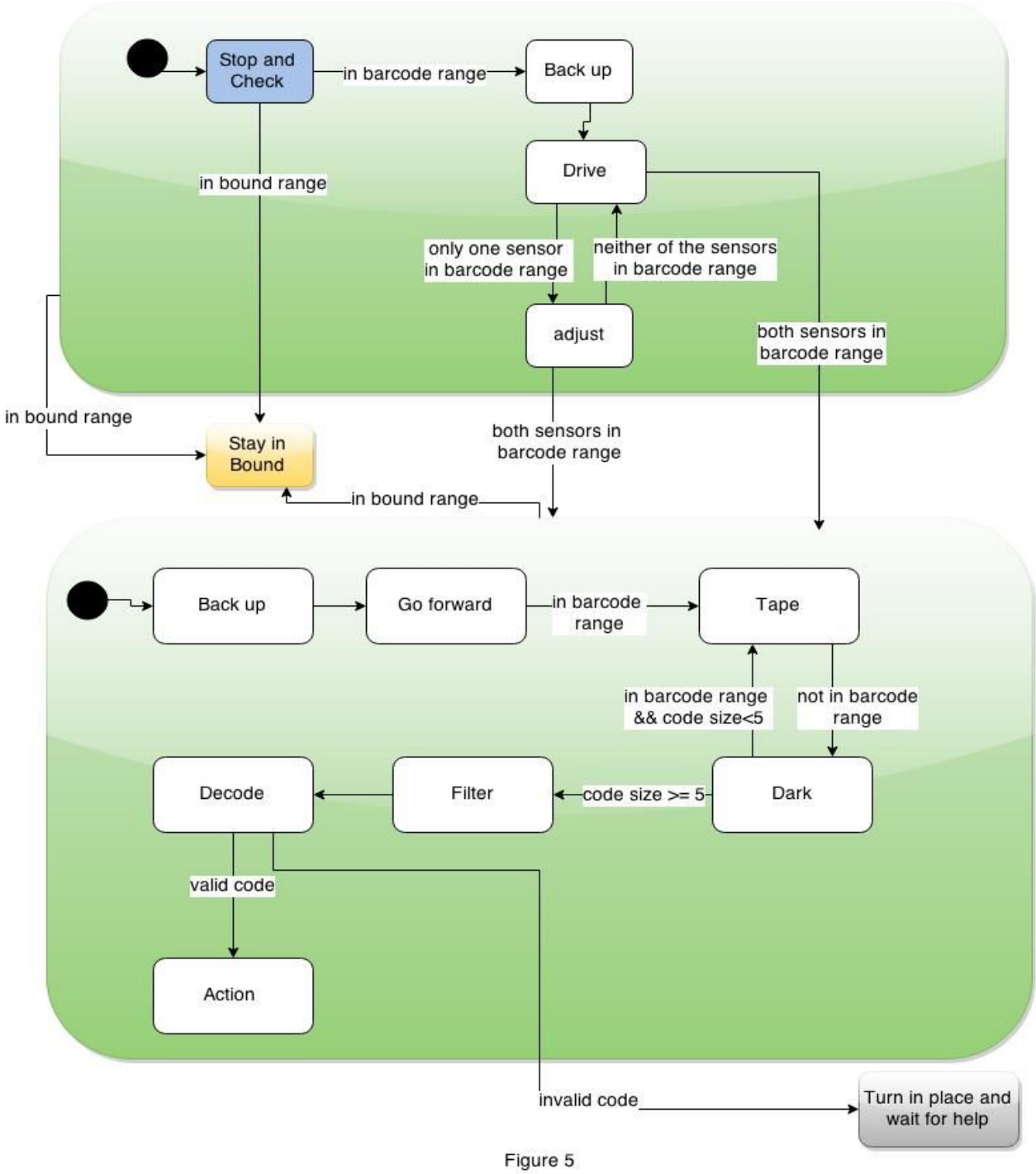
Stop and Check
in barcode range
Back up
Drive
in bound range
only one sensor in barcode range
neither of the sensors in barcode range
adjust
both sensors in barcode range
in bound range
Stay in Bound
both sensors in barcode range
in bound range
Back up
Go forward
in barcode range
Tape
in barcode range && code size<5
not in barcode range
Decode
Filter
code size >= 5
Dark
valid code
Action
invalid code
Turn in place and wait for help

Figure 5

Issues

The main source of error was the calibration. There were a lot of variables to consider. The camera was especially difficult to calibrate because the lighting has a big effect on the variables. There were also some edge cases that we did not have to time address or considered when designing the state machines. What if the Mariobot hits the barcode at a bad angle? This could happen if the barcode was located right after a turn. We assumed this would not happen and that the Mariobot would being going straight. One possible solution is turning in place, then backing up if it hits the side sensors. Another case was the positioning of the circles. If there is an opening between two obstacle, it may detect one obstacle, move, detect the other obstacle, move the other direction and repeat. Proper calibration of the radius and the distance moved helps solves this problem, but this will affect obstacle detection and the total time to get around the obstacle. One possible solution is to make it more like the obstacle-boundary case. We could detect where one obstacle is located and move until we avoid that specific obstacle. A third case was if it got past the obstacle detection range. Our obstacle avoidance algorithm with our bumpers was very simple. We assumed the camera would remove the difficult cases. We could not just back up until we detect an obstacle because we might go out of bounds. A more robust algorithm with the bumper in combination with the camera would solve this issue. Finally, there is the case where we get an invalid code. We have an error state when we receive an invalid code, but it is also possible that we did not read the code correctly. One solution is to allow the Mariobot another chance to read the bar code. We could back up a set distance since we can assume we are facing straight(adjustment was done before reading bar code) and read it a second time. We also stop reading the barcode after reading in 5 values. Instead, we can make it stop reading after 5 values or after it travels a set distance.

Conclusion

Overall, our design succeeded in completing the course. Given more time and actual test cases, we might have been able to add error correction and work out a few bugs.