Introduction

In this report I will provide an overview of how Lego Mindstorms operates, along with key programming concepts used in robot design and development. It includes a detailed description of the robot our team created, explaining its functionality and how it works. Ill also add how the report reflects on the collaborative dynamics and synergy within our group and evaluates how effectively we worked together throughout the project which made our robot effective/ineffective.

Robot Description

Our robot was designed to autonomously solve a maze. It uses an ultrasonic sensor to scan the environment, measuring the distance between itself and the surrounding walls or potential paths. This allowed the robot to navigate through tight corners and open spaces. To enhance its decision-making capabilities, the robot is equipped with colour sensors that help identify specific markers within the maze. The colour sensors serve two primary purposes: first, they detect trap doors that are marked with a green colour, signalling the robot to proceed through that specific path. If the robot encounters a red wall, the sensors alert the robot that it has reached the end of the maze, or there is no way forward in that direction. The robot is driven by motors on each side, which allow it to move and rotate efficiently. This motorised movement helps it make precise turns and navigate through the maze without straying from its path. The combination of ultrasonic and colour sensors, along with the motors, enables the robot to autonomously solve the maze, avoiding obstacles and completing the challenge.

Key Concepts

Lego Mindstorms

Lego Mindstorms are programmable bricks equipped with simple computers that run Java Virtual Machines (JVM). These bricks can have Java-based programs uploaded to them and can execute these programs autonomously. The bricks feature buttons for user input and a button and speaker for displaying messages or playing sounds. Additionally, Lego Mindstorms support up to four motors and four sensors, enabling the robot to interact with and respond to its environment through observation and action.

Sensors

Sensors are devices that measure specific data and send the information back to the brick for processing. The Lego Mindstorms sensors include a sound sensor, which measures the loudness of the surrounding environment; an ultrasonic sensor, which emits an ultrasonic beam to detect the distance of objects by measuring the time it takes for the sound to return; and a colour sensor, which detects the colour of objects in its view and can be used for tasks like line

following or distinguishing between different colours. These sensors allow the robot to understand its surroundings and make decisions based on that input. Additionally, Lego Mindstorms robots can establish internet and Bluetooth connections, enabling both input and output through external devices.
Our robot is equipped with an ultrasonic sensor and a colour sensor to navigate and solve the maze. The ultrasonic sensor detects obstacles in front of the robot by emitting a beam and measuring how long it takes for the sound waves to return. If an obstacle is detected, the robot will turn to avoid it and continue moving along its path. The colour sensor helps the robot identify specific markers in the maze, such as trap doors or walls, allowing it to make decisions based on the colours it detects. Additionally, the robot monitors inputs from buttons, one of which is used to manually terminate the program by pressing the "Exit" button. The robot also keeps track of its battery level to ensure it has enough power to continue its task. These sensors and inputs work together to help the robot autonomously navigate the maze and respond to its environment.

MovePilot

MovePilot is a powerful class within the Lejos system that simplifies robot movement. To use MovePilot, you need to provide the diameter of the wheels, their distance from the robot's centre, and the type of chassis. Once these parameters are set, MovePilot enables precise and complex movements. For instance, it can move the robot forward by an exact distance while ensuring it doesn't veer to one side at the start—a common issue that typically requires multiple lines of code to resolve. Additionally, MovePilot can control the robot's maximum speed and acceleration, allowing for smooth straight-line motion and controlled turns.

Navigation

Navigation is a crucial aspect of robotics, and Lejos makes it relatively simple. Thanks to the precision of MovePilot and the support of a separate class called OdometryPoseProvider, the robot's position can be tracked in real-time. Odometry refers to the process of using data from sensors or motors to determine a robot's location. In this case, the OdometryPoseProvider tracks each movement made by the robot, allowing it to calculate its exact position at any given time. Knowing the robot's position is essential for navigating complex environments. With the OdometryPoseProvider, the robot can navigate to specific waypoints by using their coordinates relative to the robot's starting point, and it can even avoid obstacles if their coordinates are provided.

Behaviour

Lejos also supports arbitrators and behaviours, which work together to enable dynamic robot actions based on specific conditions. A behaviour is an action that only executes when a certain condition is met. For example, our robot has a behaviour ("doorcolour") that uses the ultrasonic

sensor to detect a wall. When a wall is detected, the robot rotates 90 degrees each way to check for more obstacles. This rotation continues as the robot scans its surroundings. An arbitrator manages multiple behaviours by checking which ones are ready to run after one finishes its action. If more than one behaviour is applicable, the arbitrator selects the one with the highest priority. For our robot, the highest priority was stopping the program when the battery was low, as this took precedence over all other behaviours.

Reflection

In terms of teamwork, there were definitely some challenges. Two of our group members unfortunately did not contribute to the coding aspect of the project at all. The remaining team member and I maintained consistent communication throughout and collaborated closely to get the robot functioning. We regularly helped each other with understanding different parts of the code, proofreading what the other wrote, and explaining why we implemented certain things the way we did. We made sure to stay flexible and often met outside of scheduled lab times to keep the project moving forward. My teammate focused heavily on structuring the code and organising the logic for the robot's behaviours, while I concentrated more on debugging and troubleshooting hardware issues, as well as working through the strengths and limitations of the LeJOS software. Writing the program itself wasn't too difficult once I understood how behaviours and threads worked. Studying the provided examples and the official documentation helped a lot with that. The more difficult part was actually running and testing the code. At times, unrelated pieces of code would break the robot, needing us to replace the SD card or it causing unexpected behaviour in general, which made debugging a bit of a nightmare but also helped us learn a great deal. We both spent a lot of time exploring the LeJOS EV3 documentation to understand how various classes and methods worked, especially for handling the robot's sensors and movement logic. One of our other teammates worked separately, testing different snippets of code through trial and error. While we appreciated the effort, we ultimately chose to stick with the code that my teammate and I developed, as it was more consistent, easier for us to debug, and better aligned with the direction we were taking for the maze-solving behaviour. Communication with her was limited since she wasn't available often, which made it harder to integrate her work into the main codebase. Because the robot and software only worked properly on my computer, I took it home multiple times to continue testing and debugging. This gave me the chance to solve a lot of issues that arose from hardware limitations or software bugs outside of our lab sessions. Despite the challenges, I'm proud of the work we did. The robot performed well in the end, and I think the collaboration between the two of us who consistently contributed made all the difference.

Technical content

Classes

Driver.java: This is the main class of the program. It begins by initialising the motor ports (A and B) for the left and right wheels, and sets up the sensors on ports S2 (colour sensor) and S3 (ultrasonic sensor). The wheels are configured with the correct diameter and offset to match the

robot's axle length, and combined into a WheeledChassis, which is used to construct a MovePilot for motion control. The pilot is configured with both linear and angular speed settings. When it ends, the program closes all active sensors. The main method creates and registers all the program's behaviours (Trundling, DoorColour, BatteryLevels, and EmergencyStops) which are passed into a LeJOS Arbitrator. The arbitrator handles behaviour switching using subsumption architecture and runs until the program is terminated.

CalibrateCS.java: The CalibrateCS class is responsible for calibrating the EV3 colour sensor by rotating the robot and recording the minimum and maximum red and green values detected during a full 360-degree rotation. These calibrated values are later used by the robot for distinguishing between different surface colours. The class is constructed with references to the MovePilot, the PoseProvider, and the EV3Coloursensor. Upon instantiation, it retrieves the sensor's RGB mode using a SampleProvider, and initialises calibration variables.

MapUpdate.java: A thread responsible for continuously updating a 2D map that tracks the areas the robot has visited. It is initialises with a PoseProvider which is a reference to a shared 2D integer array, and a time interval, which defines how frequently the map should be updated. Once the thread is started, the run() method enters an infinite loop. On each iteration it retrieves the robot's current position from the PoseProvider. It extracts the X and Y coordinates from the pose and scales them down based on the dimensions of the map grid to determine the robot's current cell. If the calculated coordinates is within the map boundaries, it sets the corresponding cell in the map array to 1, indicating that the robot has visited that location. The thread then waits for the specified interval using Delay.msDelay before updating again.

Trundling.java: This behaviour causes the robot to move forward continuously as long as the ESCAPE button is not pressed. The behaviour runs until it is interrupted by another behaviour or command.

DoorColour.java: This class causes the robot to detect a door when it gets close enough, using an ultrasonic sensor to measure the distance. Once the robot is close enough, it rotates 90 degrees to check the colour of the door using the colour sensor. If the door is red, it indicates the maze is solved, and the program exits. If the door is green, the robot continues through the door and resumes its previous behaviour(Trundling). If the door is neither red nor green, the robot rotates 90 degrees in the opposite direction to check again. The behaviour continues until a red or green door is detected.

BatteryLevels.java: This behaviour monitors the robot's battery voltage and displays a warning message when the battery is low. It checks if the battery voltage drops below 7.0V using Battery.getVoltage(). It displays "Battery Low!" on the screen and starts beeping.

EmergencyStops.java: This behaviour is responsible for stopping the robot and exiting the program when the ESCAPE button is pressed using Button.ESCAPE.isDown(). If pressed, it stops the robot's movement and then exits the program.

Code Examples

This part of the code where MovePilot is used to provide precise control of the robot's linear and angular speeds is in the initialisation of the MovePilot object. This allows the robot to move with specific speeds in both straight lines and rotations.

```
BaseRegulatedMotor mL = new EV3LargeRegulatedMotor(MotorPort.A);
BaseRegulatedMotor mR = new EV3LargeRegulatedMotor(MotorPort.B);
Wheel wLeft = WheeledChassis.modelWheel(mL, WHEEL_DIAMETER).offset(-AXLE_LENGTH / 2);
Wheel wRight = WheeledChassis.modelWheel(mR, WHEEL_DIAMETER).offset(AXLE_LENGTH / 2);
Chassis chassis = new WheeledChassis((new Wheel[] {wRight, wLeft}), WheeledChassis.TYPE_DIFFERENTIAL);
MovePilot pilot = new MovePilot(chassis);
pilot.setLinearSpeed(LINEAR_SPEED);
pilot.setAngularSpeed(ANGULAR_SPEED);
```

The PoseProvider helps track the robot's position and direction. It provides the robot with its current location and heading. Odometry is a method used to calculate the robot's position by tracking how far it has moved and how much it has turned using wheel encoders.

```
PoseProvider poseProvider = new OdometryPoseProvider(pilot);
poseProvider.setPose(new Pose());
```

This section initialises the sensors and ensures that they are ready for use. If the sensors cannot be initialised, an error message is displayed, and the program exits, which happened a few times due to faulty wiring.

```
EV3ColorSensor cs = null;
EV3UltrasonicSensor us = null;
try {
    cs = new EV3ColorSensor(SensorPort.S2);
    us = new EV3UltrasonicSensor(SensorPort.S3);
} catch (Exception e) {
    LCD.clear();
    LCD.drawString("Error in intialising sensors", 0, 0);
    System.exit(1);
}
```

This section defines the various behaviours for the robot and feeds them to the Arbitrator. The Arbitrator ensures that only the highest-priority behaviour that can control the robot at a given moment will be active.

```
Behavior[] behaviors = {
    new Trundling(pilot),
    new DoorColour(pilot, poseProvider, us, cs, lightValues),
    new BatteryLevels(),
    new EmergencyStops(pilot)
};

Arbitrator arbitrator = new Arbitrator(behaviors);
arbitrator.go();
```

Calibrating the colour sensor was quite important as the robot can distinguish between different colours, as the sensor was quite sensitive to the light it detects and it can be harder for it to the robot to detect colour in brighter environments. So having minimum and maximum values for

each colour it detects was valid. In future improvements, rather than rotating a full 360 degrees and polling at fixed intervals which can be slow and cause stop-start movement. An alternative method could be implemented. This might involve using a compass or odometer to ensure the robot turns exactly 90 degrees at a time as each door is only 90 degrees away, or scanning for where the green colour ends on either side of a detected door and turning to face the midpoint for more accurate alignment. Another approach could be using a separate thread to detect the first instance of green, enabling quicker, more responsive navigation without continuous polling.

```
private void calibrate() {
    float[] lightSample = new float[light.sampleSize()];
    int pollInterval = 20;
    int pollTimes = 360 / pollInterval;
    float resolution = pollInterval / 2;
    float debt = 0;

    LCD.clear();

    for (int i = 0; i < pollTimes; i++) {
        debt += calibrationRotation(pollInterval, resolution);

        light.fetchSample(lightSample, 0);

        float redLevel = lightSample[0];
        if (redLevel < redMin) { redMin = redLevel; }
        else if (redLevel > redMax) { redMax = redLevel; }

        float greenLevel = lightSample[1];
        if (greenLevel < greenMin) { greenMin = greenLevel; }
        else if (greenLevel > greenMax) { greenMax = greenLevel; }

        LCD.drawString("This is loop " + i + "out of " + pollTimes + " loops", 0, 0);
        Button.ENTER.waitForPressAndRelease();
        LCD.clear();
    }

    pilot.rotate(debt);
}
```

The BatteryLevels behaviour ensures the robot stops if the battery level is low (below 7V), preventing it from running out of power mid-task.

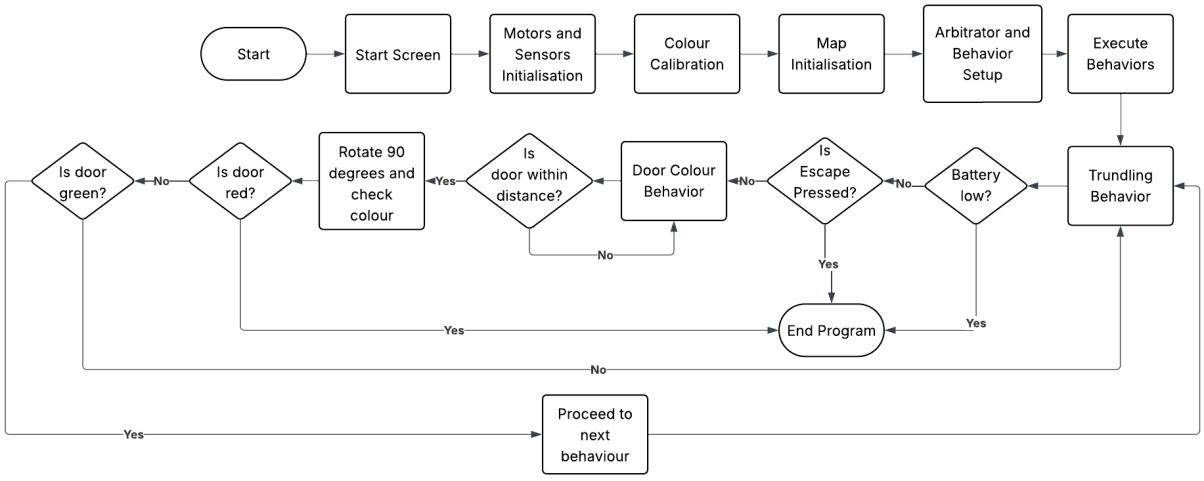
```
class BatteryLevels implements Behavior {
    private boolean suppressed = false;

    public boolean takeControl() {
        return Battery.getVoltage() < 7.0;
    }

    public void action() {
        suppressed = false;
        while (Battery.getVoltage() < 7.0) {
            LCD.clear();
            LCD.drawString("Battery Low!", 0, 0);
            Sound.beep();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }

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

The MapUpdate thread allows the robot to maintain a real-time map of its environment, marking visited positions based on the robot's odometry data. The robot continuously updates a grid (2D array) representing the environment. Each cell is updated based on the robot's current pose. This creates a basic map of the robot's exploration area, which can be useful for path planning and localisation in more complex environments.

```
public class MapUpdate extends Thread{
    private PoseProvider poseProvider;
    private int[][] map;
    private final long interval;

    public MapUpdate(PoseProvider poseProvider, int[][] map, long interval) {
        this.poseProvider = poseProvider;
        this.map = map;
        this.interval = interval;
    }

    public void run() {
        while (true) {
            Pose pose = poseProvider.getPose();
            int x = (int) (pose.getX() / map[0].length);
            int y = (int) (pose.getY() / map.length);

            if (x >= 0 && x < map.length && y >= 0 && y < map[0].length) {
                map[x][y] = 1;
            }

            Delay.msDelay(interval);
        }
    }
}
```

Control Flow

The robot first displays a welcome message on the screen and waits for the user to press the Enter button. It then initialises the motors, sensors, and MovePilot. The robot starts by calibrating the colour sensor. It also initialises the map to track the robot's movement in a grid. Following this, a new thread is started to update the map at regular intervals, constantly checking the robot's position and marking the map with the current location. The arbitrator then starts running the behaviours. When a behaviour runs, it performs its corresponding action. If the Trundling behaviour is active, the robot will continuously move forward unless another behaviour takes control. If the DoorColour behaviour detects an obstacle, it will rotate to check for a door, and if it detects a green door, it carry on through the door and it will start the previous behaviour again (Trundling). If it detects a red door, the program will exit with a "maze solved" message. If no door is detected, the robot will continue rotating to search for one. If the BatteryLevels behaviour detects that the battery level is low (below 7.0V), it will display a warning on the screen and beep and end the program. If the EmergencyStops behaviour detects that the ESCAPE button is pressed, it will stop the robot and terminate the program. If none of these behaviours are triggered, the arbitrator will allow the current behaviour to continue running. The arbitrator ensures that only one behaviour is active at a time, evaluating each behaviour's conditions and selecting the one that takes control based on its criteria.



Conclusion

To conclude, while our robot wasn't the most successful in every aspect, I still believe we accomplished a lot given the circumstances. We were able to program and implement most of the core features we set out to include. The robot can calibrate its sensors, map its surroundings, and react to various conditions like low battery levels and emergency stops. The behaviour-based system allows it to respond to real-time changes, and most functions ran consistently during testing. That said, the door detection behaviour was still a bit buggy and didn’t always work the way we intended, so that’s something we would’ve liked more time to refine. Teamwork wise, the project didn’t go entirely as planned. Two members did not

contribute at all to the code, which meant the remaining teammate and I had to carry most of the workload. However, we collaborated well, regularly stayed in contact, and supported each other by proofreading and explaining different parts of the code. We took initiative to meet up outside of labs and handled most of the development and debugging ourselves. I also worked on the robot from home since the software worked best on my computer, which helped us push through some hardware-related issues. If we were to start the project again, we'd put more effort into getting the whole group involved early on. That could mean asking the TA to check in and apply some pressure, encouraging attendance in labs or we could've arranged library meetups or Microsoft Team calls for those who can't make it in person. Setting specific goals and clear responsibilities with deadlines might've helped with accountability. On our end, we could've also set more dedicated times to physically work on the robot, instead of splitting tasks and delaying testing until later. Communicating with the course staff more often would've also helped us avoid running into the same issues others were facing something we did to an extent, but definitely could've done more. Overall, the robot mostly works and does what we designed it to do with a few bugs here and there. Most functions are reliable, and we're proud of how much we managed to accomplish, especially considering the limitations and how well the two of us worked together to keep things on track.