Design and Implementation of an Autonomous Arduino-Based Vehicle with Line Following and Obstacle Avoidance Capabilities

Team A2: Moeez Mufti, Muhammad Ali
Systems Engineering and Prototyping, BSc Electronic Engineering
Hochschule Hamm-Lippstadt, Lippstadt, Germany
{moeez.mufti, muhammad.ali}@stud.hshl.de
23rd June, 2025

Abstract

This project presents the design, modelling, and implementation of an autonomous Arduino-based vehicle capable of line following, obstacle detection, and context-aware color-based responses. Developed within the framework of a systems engineering and physical prototyping course, the vehicle integrates multiple hardware modules including infrared sensors, an ultrasonic sensor, a colour sensor, dual DC motors, and a LiPo battery. The system architecture was modeled using SysML, covering structural (BDD, IBD), behavioural (state machine, activity, sequence), requirements and parametric aspects to ensure consistency and traceability from concept to realization. Colour-specific behaviour: parking on red, stopping on green, and rerouting on blue was implemented based on environmental detection logic. The mechanical components were custom-designed and 3D printed within strict time and design constraints. The resulting prototype demonstrates modular 3D design of each component offering a compact example of embedded autonomous systems engineering.

Table of Contents

List of Figures and Tables

List of Figures:

List of Tables:

I. Introduction

The concept of an autonomous Arduino car merges the fields of robotics and embedded systems to develop an autonomous vehicle capable of navigating its environment without human intervention. Utilizing the Arduino Uno as the central control unit, the system integrates various sensors, actuators, and control logic to create a compact, self-driving car. Through the combination of hardware and software, the vehicle can perceive its surroundings, make real-time decisions, and perform actions such as line following, obstacle avoidance, and colour-based responses [1].

Creating a self-driving car with Arduino represents a compelling blend of technology, innovation, and hands-on engineering. This project brings together multiple hardware and software components to achieve autonomous navigation in a compact, embedded system. At the heart of the system is an Arduino microcontroller, which acts as the central processing unit, coordinating sensor input, motor control, and decision-making logic. The development process involves careful hardware integration, structured software programming, and iterative testing to ensure reliable and safe operation [1].

Beyond the technical challenges, the project also involved the use of 3D modelling software to design custom holders for various components, ensuring precise mechanical integration. Additionally, SysML diagrams were created using Visual Paradigm to streamline our development approach, allowing us to align system requirements, structure our planning, and maintain traceability throughout the design and implementation process [2].

II. Systems Design and Engineering

A. Systems Architecture

The architecture of the autonomous Arduino-based vehicle is modular and structured around four primary subsystems: the **control unit**, **locomotion system**, **sensor system**, and **power system**.

As shown in the Block Definition Diagram, the control unit (Arduino Uno) serves as the central processing element, interfacing with input sensors and actuating the motors via the motor driver.

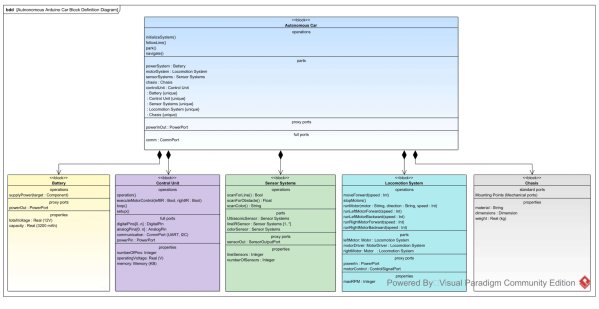

Fig. 1. Block Definition Diagram

The locomotion system consists of two DC motors, each independently controlled and powered through an H-bridge motor driver. Each motor is connected via two direction control pins (IN1/IN2 for the right motor, IN3/IN4 for the left motor) and one speed control pin (ENA for the right motor, ENB for the left motor). By setting the logic high or low on the direction pins, the motor can be made to rotate forward, backward, or stop.

The sensor system includes two infrared (IR) sensors for line detection which are placed closed to the ground and mounted on the frontside of the chassis to ensure maximum detection of the line; an ultrasonic sensor for obstacle avoidance which is mounted on a servo motor for rotation, and a colour sensor for contextual environmental analysis. These sensors send digital or analogue signals to the control unit.

The power system, composed of a 12V LiPo battery, supplies electrical energy to all components through regulated connections achieved through a common power hub in the breadboard.

B. Behavioural Logic

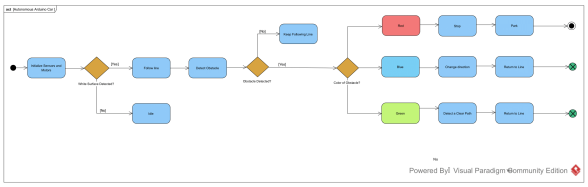

Fig. 2. Activity Diagram

The behavioral model, based on the **Activity Diagram**, defines the core operational flow of the vehicle. It starts with an initialization of its sensors and motors, then transitions into line following function: the IR sensors function by maintaining a set distance apart, with both sensors initially detecting the white surface to maintain straight-line movement. When a single IR sensor encounters a black line, the robot immediately turns in the direction of that sensor to follow the line's path. When both sensors simultaneously detect the black line, the robot stops completely, creating a simple yet effective line-following system that uses differential sensor feedback for navigation control.

While following the line, the ultrasonic sensor constantly monitors for obstacles and changes its path and navigates it paths back to the line through predefined angle movements.

For colour sensor, we had planned to follow two paths for two types of colours: red and blue/green; for red, the car is to stop and park while for green/blue the car is supposed to navigate around the obstacle and return back to the line following the logic of obstacle detection.

1) Improvements

- The current activity diagram can be enhanced to better reflect the actual control flow and behavior of the robot. Specifically, the first decision node currently routes a false condition (i.e., when the line is not detected) directly to an "Idle" state, which inaccurately suggests that the robot stops operating. In reality, as shown in our code, the robot

actively attempts to reacquire the line using corrective motor actions.

- To more accurately represent this logic, the diagram should be updated so that the false branch leads to a ”Reacquire Line” activity instead of terminating. This adjustment will align the diagram with the intended autonomous behavior of the vehicle and provide a more realistic depiction of its continuous navigation process.

C. SysML Modeling Use

SysML was employed throughout the design process to ensure a clear, traceable, and modular system structure. It allowed us to visualize, plan, and validate both functional and structural aspects of the vehicle.

- The **Use Case Diagram** outlines all interactions between the system and the user (e.g., switch on, follow line, park) to visualise the user's perspective on how the car should operate.

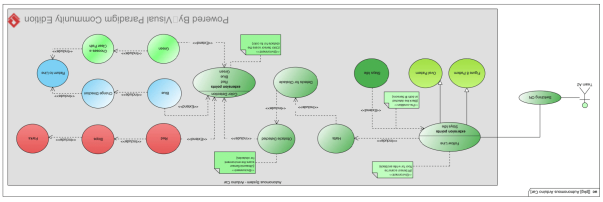

Fig. 3. Use Case Diagram

- The **State Machine Diagram** models the internal behavioural states and transitions based on input conditions, providing a systematic framework that helps map out the various operational states and their corresponding transitions, which is essential for coding each state's specific functions and behaviours.

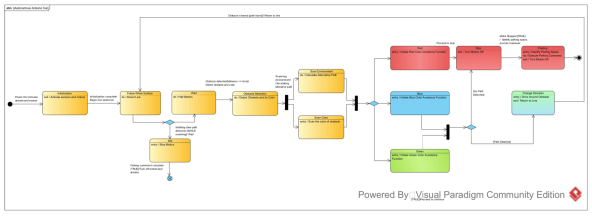

Fig. 4. State Machine Diagram

- The **Sequence Diagram** illustrates the communication of messages between the control unit and the components.
- The **Internal Block Diagram (IBD)** illustrates the interactions between the major components of the Arduino-based autonomous car, emphasizing both the flow of data signals (e.g., sensor readings, control commands) and the power connections based on the system's voltage specifications. It maps how each hardware module is interconnected within the overall architecture.

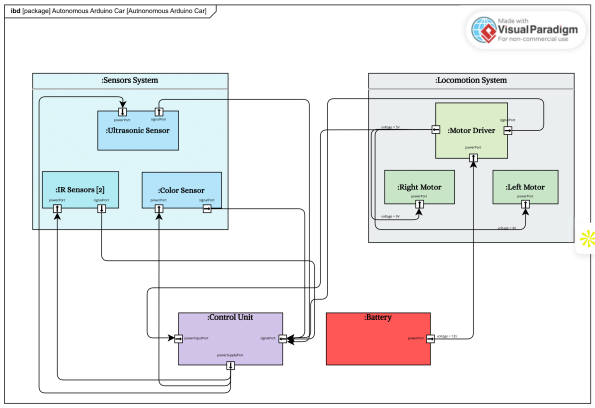

Fig. 5. Internal Block Diagram

D. Parameterization

The vehicle's design incorporates several constraints, as captured in the Parametric Diagram V3, to ensure safe and efficient operation. Key operational parameters include:

- Battery Voltage: 12V
- Minimum Obstacle Detection Range: 15 cm
- Maximum Line Deviation: 2 cm (for effective line-following)
- Each motor should get around the same voltage to allow for equal speed and operation.
- Light conditions can impact the accuracy of the color sensor. These variables were accounted for through parameter tuning in both the hardware (e.g., IR sensor placement and shielding) and software (e.g., threshold calibration).

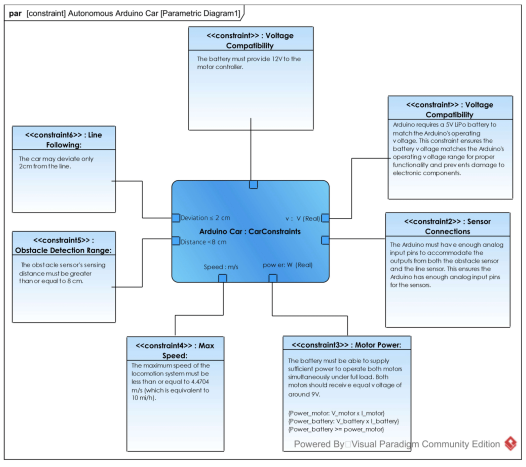

Fig. 6. Parametric Diagram

III. Prototyping and Design

This section will look into the electronic components used in the car, the visual prototypes and physical assembly of each and every component to allow functioning as a single unit.

A. Electronic Components

1) Microcontroller

- **Arduino Uno R4 Wifi**:
  - Acts as the control unit.
  - 14 digital I/O pins (used for sensors/motors), 6 analog inputs.
  - Handles PWM for motor speed control and digital logic for direction.

2) Sensors

- **IR Sensors (2x)**: Line detection by sensing black/white contrast.
- **Ultrasonic Sensor**: Detects obstacles in front of the car (15 cm).
- **Color Sensor**: colors and decides the behavior triggered based on result.

3) Motor Driver

- L298N:
  - Directly receives the entire power supply from the battery through direct connection.
  - Receives signals from Arduino.
  - Controls 2 DC motors (direction + speed).
  - 2 inputs per motor (IN1/IN2 for right, IN3/IN4 for left) + ENA/ENB for PWM speed control.

4) Motors

- **DC Gear Motors (2x)**: Drive wheels independently.
  - Controlled via motor driver.

5) Power Supply

- **12V LiPo Battery**: Powers both Arduino and motors.
  - Ensure correct voltage regulation for Arduino (5V via Vin).
  - Must supply enough current to drive both motors simultaneously.

B. Virtual and Physical Properties

We relied on 3D printing to create three-dimensional objects from digital models designed in AutoDesk Fusion [3]. This technology enables precise part production that is crucial for high-level prototyping and manufacturing, but since the process is time-consuming and material-intensive, we had to carefully consider the size and importance of each component before printing. We primarily used this method to create holders for sensors and actuators, as these components required greater precision and structural integrity compared to the car's sidewalls, roof, and base.

1) Breadboard and Battery Holder

Since we had already finalized the placement of components on the chassis, we designed the breadboard and battery holder to fit securely on top of each other. The breadboard is enclosed within a cavity in the holder, while the battery is placed underneath this cavity and secured using screws on the protruding sides of the holder.

For the second version, we kept the fundamentals intact and just increased the thickness of the protruding sides to ensure the sides do not crack under the tension and pressure of screwing the screws.

Improvements:

There are three protruding sides on our holder; however, we utilized only the ones on the left and right. The front protrusion was inward-facing instead of outward, making it impossible to screw into the chassis. To improve this, we plan to redesign the front extension by elongating it to allow proper securing to the chassis.

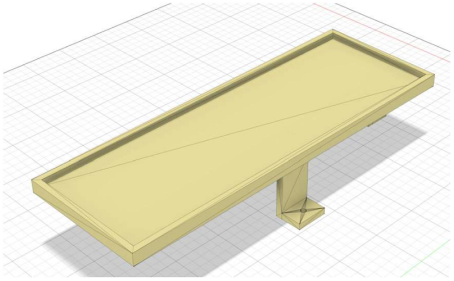

Fig. 7. 3D Modelled Breadboard and Battery Holder V1

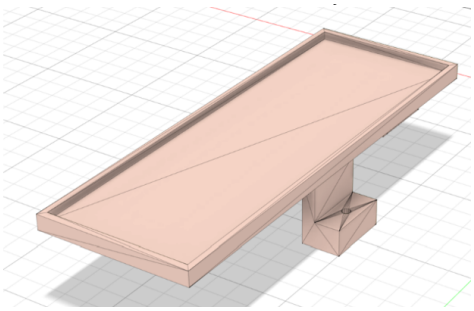

Fig. 8. 3D Modelled Breadboard and Battery Holder V2

2) IR Sensors Holder

The IR sensor holder was designed by imagining the IR Sensors mounted on a railing. The two rectangular bricks had the IR Sensors mounted on it through screw holes aligning directly with the holes in the IR Sensors. Those two bricks would then have a long rod inserted collectively through both of them using the square hole on the front of the bricks. This allowed us to vary the position and distance between the IR Sensors based on our requirements. Then, that rod with the bricks is connected through the holes in the screw holders which are then screwed into the front side of the chassis.

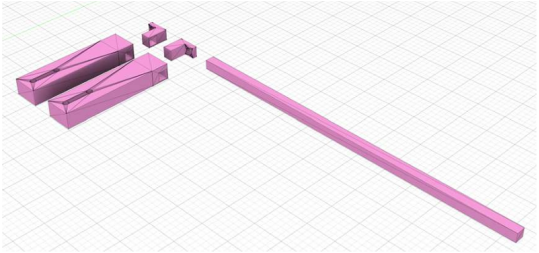

Fig. 9. 3D Modelled IR Sensors Holder V1

For the second version, we centred the square holes on the bricks because the holes in the first version were not properly printed which rendered our design as unusable as the railing would not be secured inside them. Lastly, we made the screw holders thicker for stability.

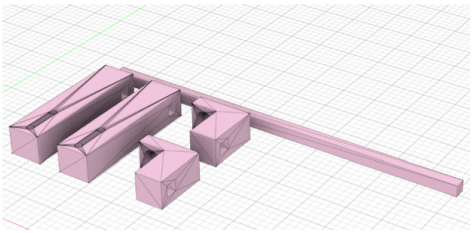

Fig. 10. 3D Modelled IR Sensors Holder V2

3) Motor Holders

We designed 3D printed motor holders to ensure a snug fit for the motors. The holders included covers on both the front and back to prevent the motors from slipping during sharp turns.

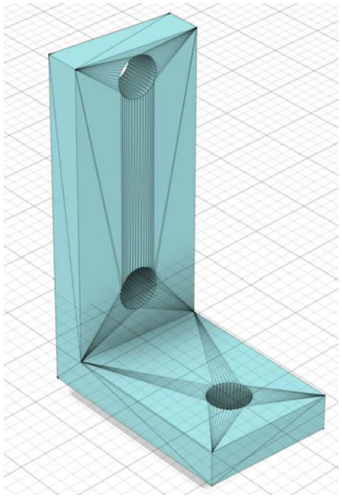

Fig. 11. 3D Modelled Motor Holders V1

For our second version, we added champers to our screw holes to better accommodate the screws to avoid them protruding outwards. We did not, however, use these in our final assembly because our car tilted heavily towards the front which hindered its ability to drive properly.

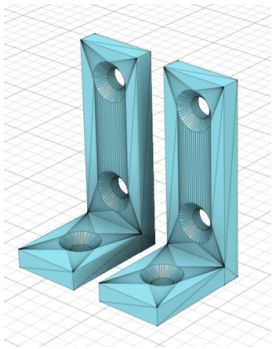

Fig. 12. 3D Modelled Motor Holders V2

4) Ultrasonic Holders

We designed a straightforward ultrasonic sensor holder to be mounted on the front of the chassis by referencing the provided 3D ultrasonic sensor file and creating a thicker base for enhanced rigidity and stability. The holes for the ultrasonic sensor's speakers were designed to match the exact diameter specified in the 3D file, with minor tolerances incorporated to ensure a secure, fixed attachment.

We did not, however, use these in our final assembly because we had assumed the usage of two ultrasonic sensors on the left and right, and not accommodating the servo motor. We, instead, used the standard ones available in the lab.

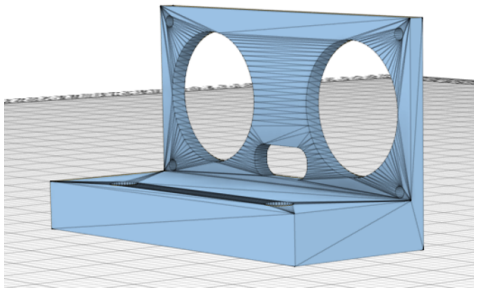

Fig. 13. 3D Modelled Ultrasonic Holder

C. Hardware Assembly

The hardware assembly of the autonomous Arduino car involved mounting all electronic components onto a laser-cut wooden chassis. A central breadboard was positioned across the bridge between the two extended sides to optimize space and wiring management. The Arduino Uno and motor driver module were situated at the front-left and front-right corners, respectively, allowing efficient signal routing. For obstacle detection, an ultrasonic sensor mounted on a servo motor was fixed at the front edge of the chassis to provide a wide scanning range. The two DC motors were secured to the sides of the bridge using motor holders, enabling stable locomotion and balanced weight distribution.

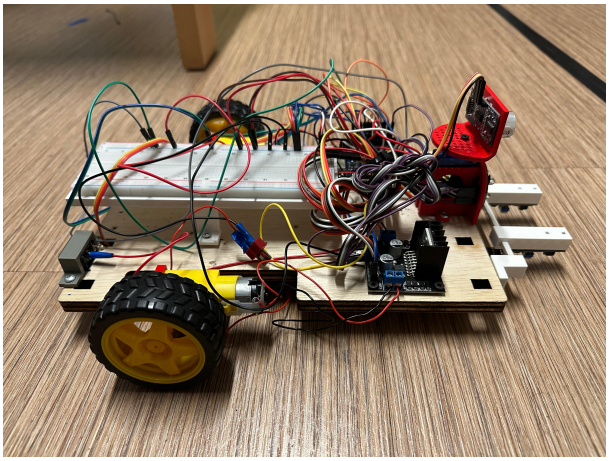

Fig. 14. Assembled Arduino Car (side view)

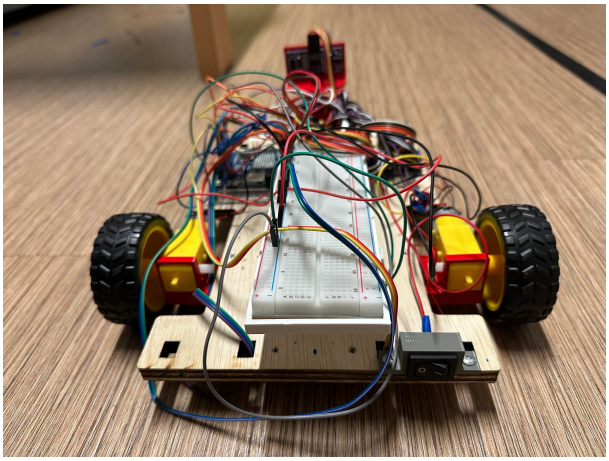

Fig. 15. Assembled Arduino Car (back view)

IV. Circuit Design

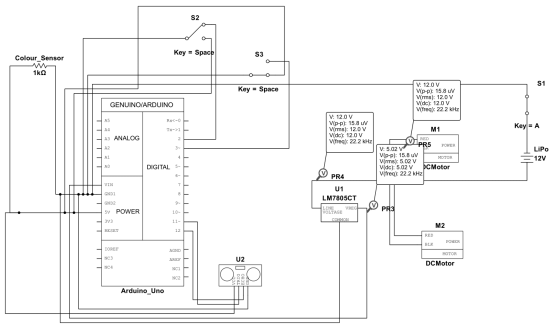

Fig. 16. Circuit Design

A. Motors (Connected via Motor Driver):

- Right Motor:
  - Speed Control (ENA) = Pin 10
  - Direction Control = IN1: Pin 4, IN2: Pin 5
- Left Motor:
  - Speed Control (ENB) = Pin 11
  - Direction Control = IN3: Pin 6, IN4: Pin 7

B. Infrared (IR) Line Sensors:

- Left IR Sensor = Pin 12
- Right IR Sensor = Pin 13

C. Ultrasonic Sensor (HC-SR04):

- Trigger Pin (TRIG) = Pin 9
- Echo Pin (ECHO) = Pin 8

D. Color Sensor (TCS3200):

- Frequency Scaling Pins = S0: Pin 2, S1: Pin 3
- Color Filter Selection Pins = S2: A0, S3: A1
- Output Signal (OUT) = A2
- Output Enable (OE) = A3 (Set LOW to enable sensor)
- Onboard LED Control = A4 (Set HIGH to turn on)

V. Software Implementation

A. Initialization of Sensors and Actuators

The setup() function in the Arduino sketch is responsible for initializing all sensors, actuators, and peripheral components. This includes setting pin modes for input/output functionality, starting serial communication for debugging, and configuring the color sensor for operation. Below is a code snippet demonstrating the initialization logic:

```
void setup() {
  Serial.begin(9600);

  // Initialize Motor Driver pins
  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(ENA, OUTPUT);
  pinMode(IN3, OUTPUT);
  pinMode(IN4, OUTPUT);
  pinMode(ENB, OUTPUT);

  // Initialize IR sensors
  pinMode(IR_LEFT, INPUT);
  pinMode(IR_RIGHT, INPUT);

  // Initialize Ultrasonic Sensor
  pinMode(TRIG_PIN, OUTPUT);
  pinMode(ECHO_PIN, INPUT);

  // Initialize Color Sensor (TCS3200)
  pinMode(S0, OUTPUT);
  pinMode(S1, OUTPUT);
  pinMode(S2, OUTPUT);
  pinMode(S3, OUTPUT);
  pinMode(OE, OUTPUT);
  pinMode(LED, OUTPUT);
  pinMode(OUT, INPUT);

  // Configure Color Sensor frequency scaling and enable it
  digitalWrite(S0, HIGH);
  digitalWrite(S1, HIGH);
  digitalWrite(OE, LOW);    // Enable sensor output
  digitalWrite(LED, HIGH);  // Turn on sensor LEDs

  stopMotors(); // Ensure motors are stopped at startup
}
```

Fig. 17. Motor and Sensors Initialization Code

B. Movement Control Algorithms

The vehicle's movement is controlled via a set of modular motor functions. These functions allow the car to move forward, stop, or perform turns by independently controlling each DC motor through an H-bridge motor driver. Pulse-width modulation (PWM) is used on the speed control pins (ENA and ENB) to regulate motor speed.

Key movement patterns implemented:

- **moveForward():** Both motors run in the forward direction.
- **runLeftMotor() / runRightMotor():** Enables movement of one motor for turning behavior.
- **stopMotors():** Immediately halts all motor activity.
- **runLeftMotorReverse() / runRightMotorReverse():** Enables backward motion used during obstacle avoidance routines.

C. Behavioural Decision Logic

The vehicle's high-level logic is implemented inside the loop() function. The logic combines line-following, obstacle detection using ultrasonic distance measurement.

- **Line Following Logic:**

It starts with an initialization of its sensors and motors, then transitions into line following function: the IR sensors function by maintaining a set distance apart, with both sensors initially detecting the white surface to maintain straight-line movement. When a single IR sensor encounters a black line, the robot immediately turns in the direction of that sensor to follow the line's path. When both sensors simultaneously detect the black line, the robot stops completely, creating a simple yet effective line-following system that uses differential sensor feedback for navigation control.

- **Obstacle Detection:**

If the ultrasonic sensor detects an obstacle within 15 cm, the car stops and enters the detection and avoidance routine.

- **Avoidance Logic:**

If an obstacle is found, the robot scans both left and right directions to find a clear path using a sequence of turning and distance-checking motions. If neither side is clear, the robot halts.

- **Absence of Colour Logic:**

The TCS3200 colour sensor wasn't integrated into our final build during development and limited time for proper calibration. If we were to revisit the project, we would take a more gradual approach: focusing on properly understanding the sensor, testing it early on, and using available guides or tutorials to help with calibration and integration. This would allow us to successfully implement colour-based actions in future versions of the car.

VI. Git Usage and Collaboration

The project development process was meticulously managed using GitHub, enabling version control, collaboration, and progress tracking. A branching strategy was adopted to separate different tasks such as wire testing, SysML diagram development, and component testing.

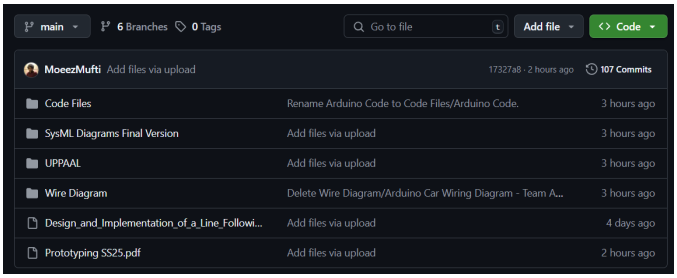

Fig. 18. The Main Branch

The GitHub repository includes the main directory and branches like Component-Testing, SysML-Diagrams, Wire-Testing, Version-3, and Line-Following-Code-Versions, each reflecting different stages and focus areas of the development process [4].

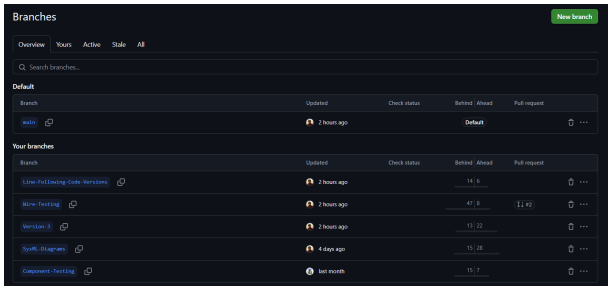

Fig. 19. Branches

A. Improvements

1) Lack of Pull Requests & Reviews

- Most branches have **no pull requests (PRs)**, and there's no evidence of code review or merge documentation.

2) Branch Overlap and Inconsistency

- Some branches like Wire-Testing and Line-Following-Code-Versions appear to **duplicate development effort**.
- Ahead/behind statistics suggest merges have not occurred properly or frequently, risking merge conflicts.

3) No Issue Tracking or Milestones

- No use of **GitHub Issues**, **Milestones**, or **Projects** to plan tasks or track bugs/features.

4) Commit Messages

- Our commit messages such as *"Add files via upload"* or *"Rename Arduino Code..."* are vague. This weakens long-term traceability.

B. Possible Future Implementation

We faced the most complications regarding our line following capabilities and its documentation in our git.

For improvements we can use the following structure to help us document better and do version control for our github which includes: introducing branches like **develop** (merge all feature branches for final testing before promoting to main), **feature/obstacle-logic**, and **hotfix/ir-sensor-failure**.

This approach ensures stable code in **main** (only store final, tested versions of our code and other tasks), organized feature development, and fast bug fixes without disrupting ongoing work [5].

In the future, if we were to replicate this functionality, we can implement the following steps specifically on our GitHub:

- *Line-Following-Code-Versions*
  We can use this branch to experiment with new algorithms (e.g., PID control or state-based turning). Commit frequently, test different approaches, and document performance.
- *develop*
  This will act as the integration branch. After successfully testing a version from Line-Following-Code-Versions, merge it here to combine with other modules (e.g., obstacle logic).
- *main*
  This holds only the **fully tested and stable version** of the car's code.
  Once develop has been verified on hardware, merge into main.
- *hotfix/ir-sensor-failure*
  If an IR sensor behaves unexpectedly during a test or demo, we can use a hotfix branch to patch just that bug.

1) Example Flow

1) Work on updates inside Line-Following-Code-Versions (e.g., tweak IR sensor thresholds, add smoothing logic)
2) Once tested, **merge into develop** and integrate with obstacle avoidance or wire-testing features.
3) After final validation on hardware, **merge into main** for release.

C. Contribution Breakdown:

TABLE I
GIT COMMITS PER TEAMMATE

Team Member	Number of Commits
Moeez Mufti	85
Muhammad Ali	22
Total	107

- Moeez Mufti (85 commits): Took lead in GitHub repository management, SysML modeling, version control, and testing. He was also responsible for detailed diagram creation and software-hardware integration.
- Muhammad Ali (22 commits): Led the design of the physical structure using AutoCAD, was involved in early-stage assembly, component mounting, and initial Arduino coding. Also contributed to testing and wiring validation.

VII. KEY ACHIEVEMENTS

- Successfully 3D modelled a modular IR sensor mount
- Completed all tasks assigned on a weekly basis which helped put us on the right track.

ACKNOWLEDGMENT

We would like to express our sincere gratitude to our instructors at Hochschule Hamm-Lippstadt for their invaluable guidance and support throughout the development of this project. Their feedback and encouragement played a crucial role in helping us navigate the challenges of system integration, design, and prototyping. This project would not have been possible without the combined knowledge, creativity, and teamwork of everyone involved.

REFERENCES

[1] M. N. R. Bhukya, R. M. Sanapathi, V. S. V. Koti, and D. S. V. Sravya, "Self Driving AI Robot Car Using Arduino Uno," *International Journal of Engineering Research & Technology (IJERT)*, vol. 12, no. 5, pp. 1210–1214, May 2023.

[2] Visual Paradigm International, "Visual Paradigm," 2024. [Online]. Available: https://www.visual-paradigm.com/

[3] Autodesk Inc., "Autodesk Fusion 360," Autodesk, 2024. [Online]. Available: https://www.autodesk.com/products/fusion-360/overview

[4] MAliSohail, "GitHub - MAliSohail/Prototyping— Team$_A$2$_2$025, "$GitHub, 2025.https$: $//github.com/MAliSohail/Prototyping - - - Team_A2_2025$.

[5] V. Driessen, "A successful Git branching model," nvie.com, 2010. https://nvie.com/posts/a-successful-git branching-model/

DECLARATION OF ORIGINALITY

We hereby declare that this report is our own work and that we have acknowledged all sources used.