Connection Timed Out (CTO)

Ege Engin
CS, University of Antwerpen
Antwerpen, Belgium

Eren Copcu
CS, University of Antwerpen
Antwerpen, Belgium

Ioannis Tzortzakis
CSD, University of Crete, Greece
Erasmus Semester at the University of Antwerp, Belgium
Heraklion, Crete, Greece

Luigi Ferrara
DISCo, University of Milano Bicocca, Italy
Erasmus Semester at the University of Antwerp, Belgium
Milan, Italy

***Abstract*—This paper presents the development and simulation of an autonomous Lego robot designed for warehouse logistics, specifically targeting improvements in service-level agreements within a fruit-handling warehouse. Utilizing the Lego Mindstorms Education EV3 platform and Java-based agent-oriented programming, the study assesses the robot's capabilities through simulations in Carla, an open-source autonomous driving simulator. The robot successfully navigates various scenarios, including obstacle avoidance, collision detection, crate transfer, and battery recharge, showcasing the efficacy of PID controllers, collision avoidance algorithms, and battery management logic in handling complex warehouse environments. A collaborative experiment with another robotics team underscores the robot's adaptability and collision avoidance capabilities in shared spaces. The study emphasizes the need for ongoing technological evolution and suggests potential hardware and software upgrades for real-world applications. Hardware enhancements include sturdier motors, advanced sensors, and improved power management, while software improvements involve integrating machine learning for better adaptability and real-time analytics for more informed decision-making. The research provides practical insights into robotics and cyber-physical systems, offering a valuable learning experience with Lego Mindstorms and the SPADE multi-agent platform. The findings lay the groundwork for developing advanced and adaptive autonomous solutions to address the challenges in contemporary warehouse logistics.**

***Index Terms*—Autonomous robots, Lego robots, Warehouse logistics, Cyber-Physical Systems (CPS), Multi-agent systems, Collision avoidance, Simulation, PID controller, Machine learning, Robotics Education**

I. Introduction

Our paper is about a study on Cyber-Physical Systems (CPSs) with a particular focus on agents, aiming to understand the performances of such a system in real-life scenarios.

We will build a Lego robot aiming to simulate the functionality of a real robot capable of transporting fruit.

Regarding our statement to simulate real-life scenarios, let's see what it is for a more concrete perspective.

A. Problem

BelgaFruitPorts, a successful terminal in the Antwerp harbor specializing in dealing with fruits and other perishables, aims to enhance its service-level agreement to boost customer satisfaction and expand its market share. The primary objective for improvement is:

Guarantee that the transportation time between the ships and the temperature-controlled storage facilities is less than 7 minutes.

To achieve this objective, BelgaFruitPorts is exploring the feasibility of employing a fleet of smart robots. The focus is on coordinating the robots' behavior to ensure that goods are transported within the specified time frame and that collisions between vehicles are avoided. The secondary objective is to increase the detection rate of fruits that start to rot by 5%.

B. Cyber Physical Systems

A cyber–physical system (CPS) is a computer system that has physical and software components that are deeply intertwined. Such components can operate on different spatial and temporal scales, exhibit multiple and distinct behavioral modalities, and interact with each other in ways that change with context

Examples of CPS include smart grids, autonomous automobile systems, medical monitoring, industrial control systems, robotics systems, recycling, and more.

Multi-agent CPSs are Cyber-Physical systems in which the multi-agent paradigm implies autonomous software capable of acting dynamically, reactively, and intelligently that alters its environment to create a state change based on defined behavioral features.

II. State of arts

For the structure of our project, we drew inspiration from the research conducted by Burak Karaduman, Geylani Kardas and Moharram Challenger [1]

It is also worth mentioning (Rational software agents with the BDI reasoning model for Cyber–Physical Systems [2]) from which we draw inspiration since the paper is a study on agents integrated into CPS. Even though the focus of the paper is a bit different.

III. Scenarios

In building our Lego robot to simulate a real-world robot, we have to make some **assumptions**. These assumptions are simplifications that help us create a practical simulation.

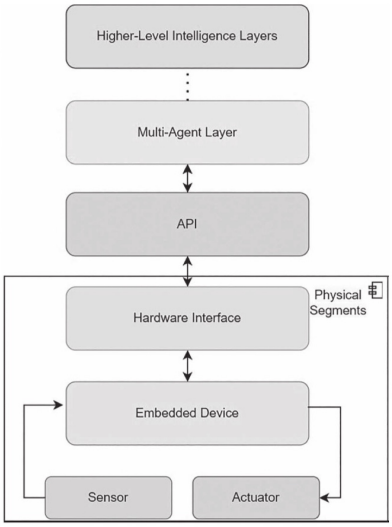

FIGURE 11.3 Proposed architecture for implementing CPS.

Fig. 1. Reference structure

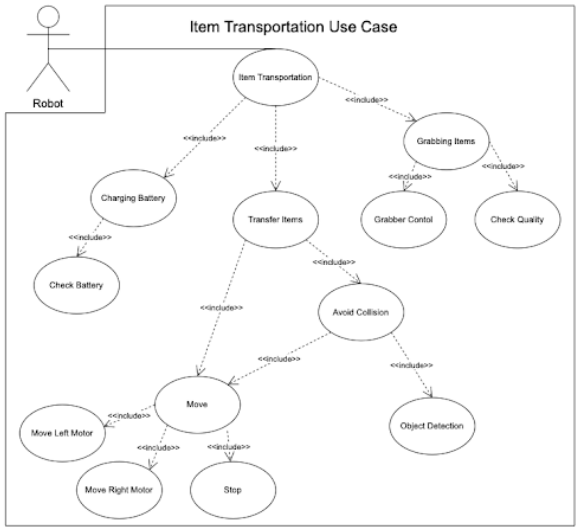

Fig. 2. Usecase Diagram.

They allow us to focus on what's important without getting bogged down by complex details. While these assumptions might not match the reality exactly, they make our Lego robot simulation more manageable and practical.

1) Artificial battery simulation: The system assumes the presence of a battery, while in reality, no physical battery exists. Instead, the artificial battery time is employed to replicate its behavior.

2) Obstacle avoidance: Our robot can avoid obstacles only if they are over 4.5cm in height and over 1.5cm in width, ensuring visibility by the ultrasonic sensors. Therefore we assume the system to face just these kinds of objects.

3) Representing crate positioning: We assume our robot can detect where the objects are, but in reality, we tell the robot a representative position for each item.

4) Manual object handling: We assume the object can precisely acquire and release an item thanks to its grabbing claw, but in reality, we manually have to place out the object from the grabbing claw.

5) Item Positioning: We assume our robot can detect the exact position of an item.

6) Size of Object: The object must have 4-6cm height and 3-4cm width.

Our system has been developed to satisfy one particular use case, that is, the Item Transportation use case.

From the next figure, it is also possible to notice the various scenarios. Let's go deeper into the most relevant of them to gain a more comprehensive understanding of our system.

A. Avoid Collision

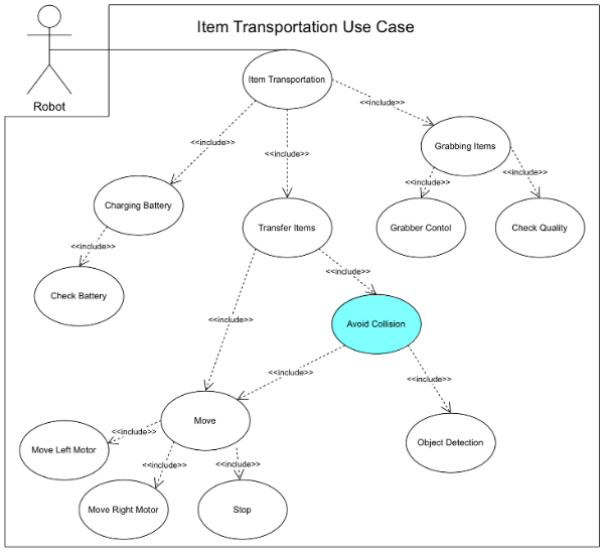

Fig. 3. Avoid Collision.

Our robot employs the PID technique in conjunction with three ultrasonic sensors to navigate around obstacles of sufficient size and avoid walls.

It's important to note that the sensors may not detect obstacles that are too short and fail to reach the required height, as previously mentioned.

B. Charging Battery

We assume our system has a battery that can be recharged simply thanks to the pressing of a button

We assume there exists a place in which our robot can recharge autonomously

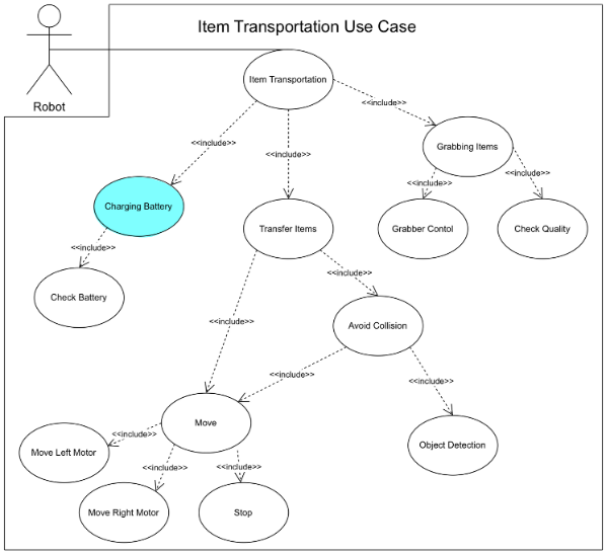

Fig. 4. Avoid Collision.

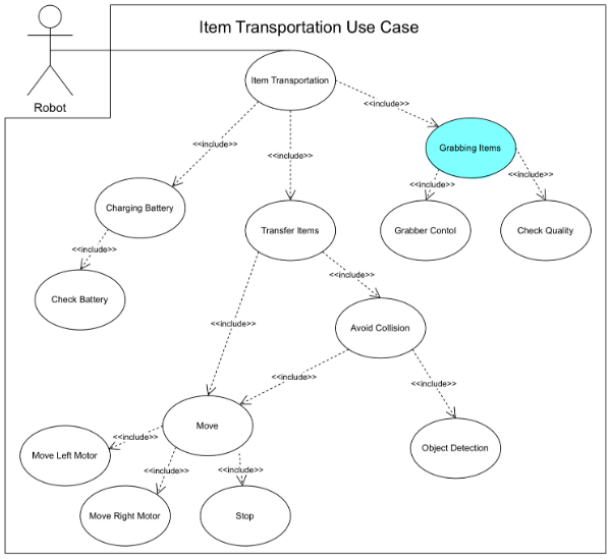

Fig. 5. Avoid Collision.

C. Grabbing Items

Our robot is equipped with a grabbing claw, and thanks to a moving mechanism, it can grab objects of a reasonable size.

At first, the robot will open its claw followed by closing it to securely grasp the item. To release the object, the robot performs the reverse action by first opening the claw

IV. Implementation

Requirement Specification

Our implemented scenarios offer practical insights into the Lego robot's functional needs. By breaking down these scenarios, we can identify essential features necessary for effective operation in a warehouse setting. These scenarios closely align with the system's requirements, and the accompanying diagram visually outlines these connections.

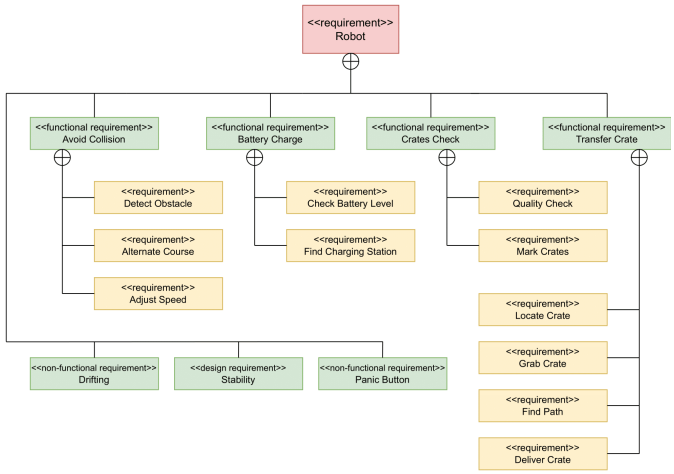

Fig. 6. Requirments Diagram.

A. Hardware

In this study, we used the Lego Mindstorms Education EV3, a user-friendly and affordable robotics platform designed for educational purposes.

The robot consists mainly of regular Lego parts. Some other parts are from the Lego Mindstorm Education EV3 set. These are:

- EV3Ultrasonic Sensor
- EV3TouchSensor
- EV3LargerRegulatedMotor
- EV3MediumRegulatedMotor

Other Hardware pieces are:

- Batteries
- Raspberry Pi3

The Raspberry Pi 3 is a small and affordable computer that serves as the main hub in our project. With enough power to handle tasks smoothly, it can connect wirelessly to the internet, has USB ports for other devices, and even lets us attach things like sensors. Its compact size and versatility make it a handy addition to our hardware lineup.

Initially, our robot had a very wide base, making maneuvers challenging to the battery. So we decided to move to a new approach, making the base smaller and sticking the hardware on top of the other. This made the robot more stable and overall better. The key adjustment involved distributing the robot's weight evenly across the entire base, significantly improving its mobility. As we progressed, we introduced a "grabber" attachment to the robot, introducing a challenge due to the additional weight at the front, as illustrated in pic 3. To address this, we came up with the idea of introducing incorporated suspensions at the wheels, effectively stabilizing the robot and enhancing its overall performance.

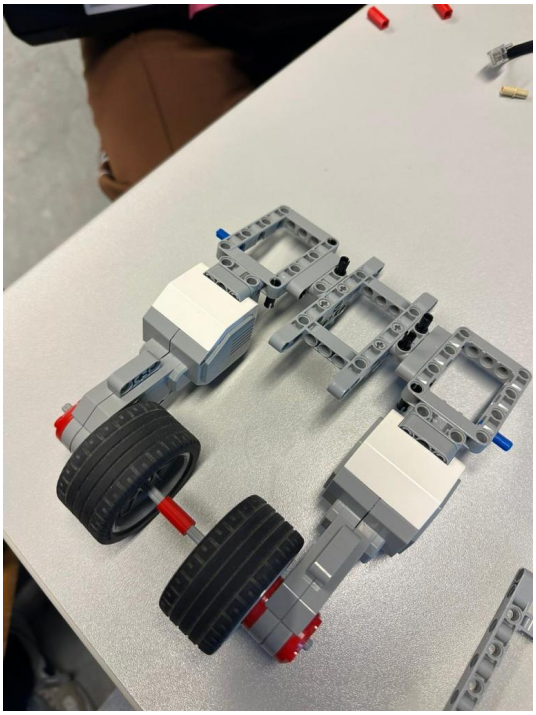

Fig. 7. The base of the first robot.

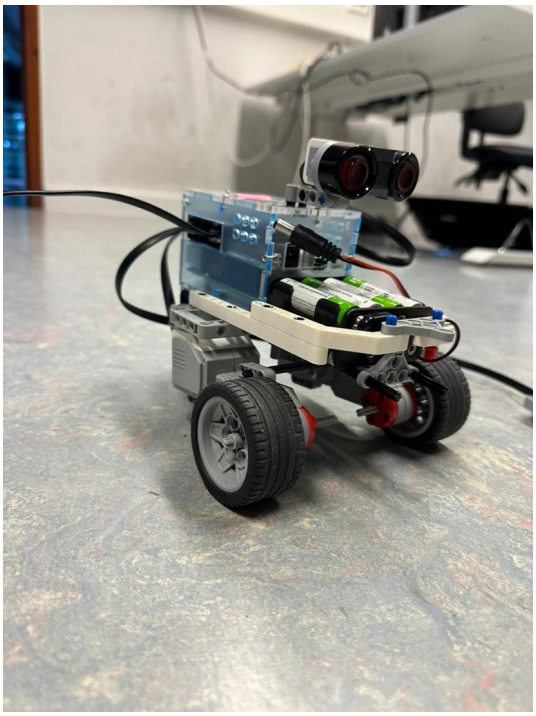

Fig. 8. The second version.

The main aim of upgrading our robot was to strike a balance—ensuring it moved smoothly and efficiently while keeping it light and avoiding unnecessary complexity. A significant improvement was achieved by evenly distributing the weight across the base. This adjustment not only enhanced the robot's agility and responsiveness but also contributed to a more efficient use of power. Throughout the design process, we maintained a careful balance, using only essential components to avoid unnecessary complexity. This approach allowed us to keep the robot light yet decently complex, optimizing its performance without unnecessary complications.

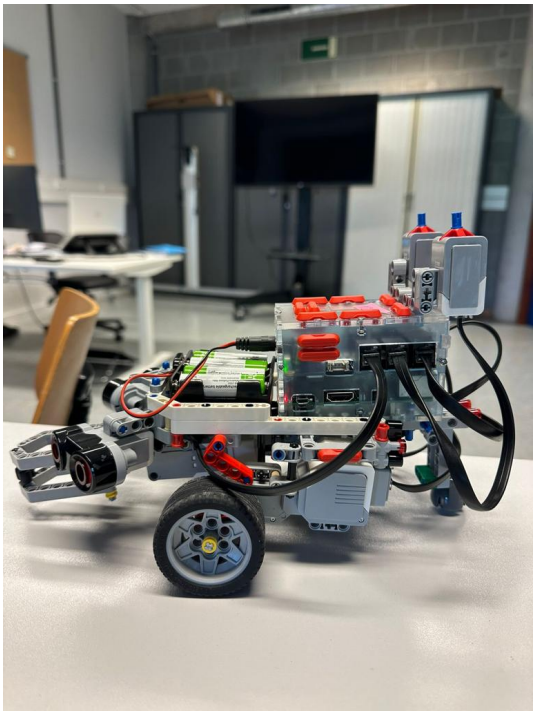

Fig. 9. The finale version.

B. *Software*

For the software part of our warehouse robot project, we went with Java because it's compatible with the BrickPI system and has a solid set of tools for handling complex robot tasks. Java's object-oriented approach made it easy to organize our code into different sections, making it easy to maintain and troubleshoot.

We used an agent-based approach to mimic how robots act in a warehouse. This means we treated each robot like its own agent, each with specific roles and behaviors. This allows the robots to make decisions based on what's happening around them, much like how they'd adapt to a changing warehouse environment.

This agent-based approach worked well for the various tasks our robots were built for, like moving to different locations, avoiding obstacles, and managing their battery levels. It also lets us use smart algorithms for things like planning the best path and avoiding collisions, which are crucial for keeping things safe and efficient in a busy warehouse.

Before putting our robots into action, we tested extensively every part of our software in a simulated setting, which we'll

explain more about later. This simulation helped us make sure our algorithms worked well in different situations, ensuring reliability before putting the robots in a real warehouse.

First, we'll talk about the overall design of our software, explaining the main parts and how they work together. Then, we'll get into the details of how we made our software handle specific situations in a warehouse, like managing material transport, finding the best route based on the layout and inventory, and making sure each robot uses energy efficiently to stay operational as long as possible.

Transfer Crates Logic:

1.1 - Navigation and Positioning

The robot utilizes an internal map of the warehouse, represented in the code by a stack of Point2D objects (coordinates targets), which denote target locations for crate pick-up and delivery. The MovementBehaviour class governs the robot's movement towards these targets. It calculates the most efficient turn angles and paths to reach these points using methods like calculateTurnAngle and isArrived. If the robot is not close to any obstacles, it tries to go to the next target location based on these functions.

```
private double calculateTurnAngle(Point2D target){
            try{
                yaw = Math.toDegrees(tag.getYaw());
                double angleToTarget =
Math.toDegrees(Math.atan2(target.y - position.y,
                        target.x - position.x));
                angleToTarget = (angleToTarget + 360) % 360;
                double angleDifference = angleToTarget -
yaw;
                angleDifference = (angleDifference + 180) %
360 - 180;
                return angleDifference;
            }
            catch (Exception e){
                System.out.println(e);
            }
            return 0;
        }
```

Fig. 10. calculateTurnAngle() function.

The calculateTurnAngle function is designed to determine the angle the robot needs to turn to align itself toward a target location. It calculates the angle between the robot's current orientation and the direction of the target. This involves computing the arctangent of the difference in y and x coordinates between the robot's current position and the target's location and then adjusting this angle relative to the robot's current heading. This function is crucial for enabling the robot to navigate efficiently toward its destination by determining the correct steering angle.

The isArrived function is used to determine if the robot has reached its target location or waypoint in the warehouse. This function involves calculating the distance between the robot's current position and the target position. If this distance is within a specified threshold, which indicates the robot is close enough to the target, the function returns true, signaling that the robot has "arrived" at its destination. This is crucial

```
private boolean isArrived(Point2D target){
            try{
                double xdiff = Math.abs(target.x -
position.x);
                double ydiff = Math.abs(target.y -
position.y);
                double result = Math.sqrt((ydiff*ydiff) -
(xdiff * xdiff));

                if (result < 75){
                    return true;
                }
                return false;
            }
            catch (Exception e) {
                System.out.println(e);
                return false;
            }

        }
```

Fig. 11. isArrived() function.

for tasks like stopping the robot, performing actions at the destination, or moving on to the next target.

1.2 - Crate Handling Mechanisms

The robot is equipped with a motorized grabber to handle the carriage of the crates within the warehouse. The operateGrabber method within the MovementBehaviour class controls this grabber to pick up and release crates at the appropriate locations

1.3 - Adaptive Path Planning

The robot continuously adapts its path based on real-time environmental data. This adaptation is evident in the code where the robot checks for obstacles using ultrasonic sensors (ultrasonicSensorLeft, ultrasonicSensorRight, ultrasonicSensorMiddle) and alters its path accordingly using the PID controller algorithm (more explained in 4. Collision Avoidance). The checkDistance method plays a crucial role here, feeding sensor data into the navigation logic to avoid collisions and ensure smooth movement toward the targets.

Collision Avoidance:

2.1.0 - Local Collision Avoidance

This is primarily managed by a PID controller, but its activation is contingent on the proximity of nearby objects. The PID controller is not constantly influencing the robot's movements; instead, it intervenes when an obstacle is detected within a critical range. This selective activation ensures that the robot's general motion remains smooth and uninterrupted under normal circumstances, while still providing responsive maneuvering in the presence of immediate obstacles.

2.1.1 Ultrasonic Sensors for Obstacle Detection

In our robot, we employed three ultrasonic sensors (ultrasonicSensorRight, ultrasonicSensorLeft, ultrasonicSensorMiddle) to detect obstacles. These sensors

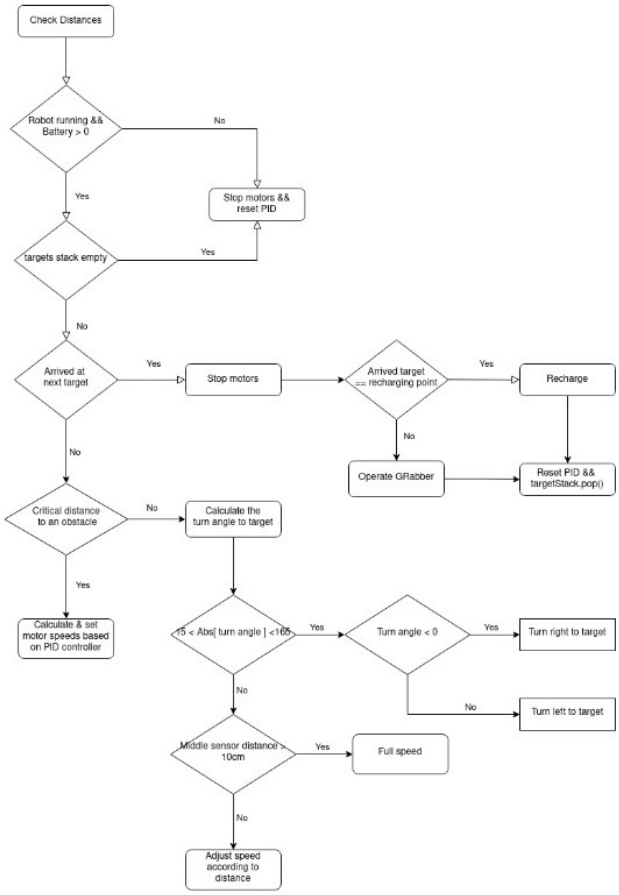

Fig. 12. Move logic diagram.

provide real-time distance measurements from various angles around the robot.

2.1.2 - PID Controller for Dynamic Adjustment

The PID controller is utilized to calculate precise motor adjustments. The controller takes the error (the difference between the desired state and the current state) as input and produces an output to correct the robot's course. This is evident in the calculateAndSetMotorSpeeds method. The PID parameters (kp, ki, kd) are tuned to ensure smooth and responsive control.

2.1.3 - Distance-Based Motor Control

The robot's movement is governed by distance values obtained from the sensors. If an object is detected within a certain range, the PID controller outputs are used to adjust the speeds of the left and right motors (motorLeft, motorRight), helping the robot navigate around obstacles.

2.1.4 - Handling Close Proximity Scenarios

In scenarios where an obstacle is detected at a very close range, the robot employs more drastic steering adjustments to avoid collisions. This is managed by the calculateAndSetMotorSpeeds method, which dynamically alters motor speeds based on the proximity to detected obstacles.

2.1.5 - Continuous Monitoring and Real-Time Response

The robot continuously checks for obstacles (checkDistance method) and adjusts its path accordingly, ensuring ongoing collision avoidance and safe navigation in a dynamic environment.

2.2.0 - Global Collision Avoidance

At a higher level, a central control mechanism oversees the overall safety of the warehouse environment. Implemented within the agent's container for this prototype, this system continuously monitors the positions and trajectories of all robots. It calculates collision probabilities based on their paths, angles, and distance differences. If this system determines a high probability of collision, it preemptively intervenes by halting the involved robots. This intervention is a safety measure to prevent potential collisions. However, it necessitates human intervention to reset and restart the robots, ensuring that any false positives or system errors are properly addressed.

2.2.1 - Agent Communication and Monitoring

The system establishes communication with individual agents (robots) using MQTT protocols, as seen in the instantiation of TagIdMqtt objects for both our_tag and target_tag. Continuous monitoring of each agent's location is performed, utilizing the getLocation method of TagIdMqtt.

2.2.2 - Collision Probability Calculation

The check_preventCollision function continuously evaluates the risk of collision. It retrieves the positions of the agent and a target and calculates the angle to the target using the calculateTurnAngle method. The calculateCollisionProbability method is then used to estimate the likelihood of a collision based on the distance between agents.

2.2.3 - Decision Making and Intervention

If the calculated collision probability exceeds a certain threshold, the system considers this a potential collision scenario. The code suggests a mechanism for intervening in such cases, although, in the current implementation, the intervention (like stopping the agent) is commented out for safety reasons.

2.2.4 - Human Intervention and System Reset

In the case of a high collision probability, the system is designed to require human intervention. This ensures that any false positives or unforeseen scenarios are handled safely.

Battery Recharging:

The robot monitors its battery level using the BatteryCheck-Behaviour class. When the battery level drops to a predefined

threshold, the robot navigates to the charging station, as indicated by pushing the recharging point to its target stack. The recharging process is simulated by resetting the artificial battery time when the robot reaches the charging point.

```
private class BatteryCheckBehaviour extends CyclicBehaviour {
    @Override
    public void action() {
        if (batteryRechargeSensor.isPressed()) {
            System.out.println("Recharging battery, please wait.");
            artificialBatteryTime = CONSTANT_BATTERY_TIME;
            Delay.msDelay(2000);
            resetPID();
        }
        else if (artificialBatteryTime == 480 && !targets.empty() && targets.peek() != rechargingPoint) {
            System.out.println("Low battery going to charge.");
            targets.push(rechargingPoint);
        }
        else if (targets.empty()){
            System.out.println("Finished job going to charging dock");
            targets.push(rechargingPoint);
        }
        else if (artificialBatteryTime > 0) {
            if (isRunning) {
                artificialBatteryTime--;
            } else {
                System.out.println("Battery on hold");
            }
            position = tag.getLocation();
            Delay.msDelay(250);
        }
        System.out.println(artificialBatteryTime);
        Delay.msDelay(1);
    }
}
```

Fig. 13. BatteryCheckBehaviour class.

The BatteryCheckBehaviour class assumes the responsibility of monitoring the robot's battery level and orchestrating its behavior according to predetermined conditions. Nestled within a threaded behavior, this design choice ensures the concurrent execution of battery monitoring alongside other tasks in the system. This parallel approach is especially advantageous in scenarios where various operations can unfold simultaneously without compromising the overall efficiency of the system.

During the battery decrement process, the class intricately incorporates the retrieval of the robot's current position through the tag.getLocation() method. This particular action, categorized as a secondary priority, occurs exclusively when the robot is operational (i.e. when isRunning is true). The acquisition of the robot's location serves a dual purpose: it not only facilitates battery management but also acts as a supplementary mechanism for spatial awareness. By regulating the frequency of location updates with a controlled delay of 250 milliseconds, the system strikes a harmonious balance between responsive behavior and periodic spatial updates.

3.1 - Recharging Process

If the robot detects that it is positioned at the charging point, the class initiates the recharging process. The artificialBatteryTime, symbolizing the simulated battery life, undergoes a reset to a predefined constant value (CONSTANT_BATTERY_TIME). To mimic the time required for recharging, a deliberate delay of 2 seconds is introduced. Simultaneously, the Proportional-Integral-Derivative (PID) controller undergoes a reset to its initial state.

3.2 - Low Battery Condition

When the artificialBatteryTime reaches a predefined threshold, simulating critical battery levels, and there are still pending targets for the robot to visit, the class appends the recharging point to the target stack. This conditional action prompts the robot to autonomously navigate towards the charging station, addressing the urgency of low battery levels.

3.3 - Completion of Tasks

In the absence of remaining targets in the stack, the class adds the recharging point, signaling that the robot has successfully completed its designated tasks and should proceed toward the charging dock.

3.4 - Battery Depletion During Operation

If the artificialBatteryTime remains above zero, the class decrements the battery time exclusively during the robot's operational phases (when isRunning is true). The robot's current position is updated based on the information retrieved from tag.getLocation(). This comprehensive approach to battery management, coupled with the concurrent execution in a threaded behavior, underscores a well-balanced integration of tasks, seamlessly blending battery monitoring with spatial awareness in the operational context of the robot.

Triaging Rotting Crates:

Each crate picked up by the robot is assigned a random value, representing the probability of it being rotten. This introduces an element of unpredictability, mimicking real-life scenarios where crates might contain perishable goods with varying shelf lives.

We simulate advanced sensory capabilities in the robot, allowing it to detect the state of the crate. This could involve hypothetical sensors that detect ethylene gas, commonly released by ripening or rotting produce.

Upon detection of a rotting crate, the robot's logic dictates a change in its task. Instead of proceeding to the regular drop-off point, it reroutes to a designated area for rotting crates, effectively segregating compromised goods from the rest.

The robot then proceeds to handle the rotting crate with priority, ensuring it is safely and quickly transported to the designated drop-off point to prevent contamination or further degradation

After dealing with the rotting crate, the robot's system is designed to automatically update its task list and resume standard operations, picking up the next crate in line and continuing its workflow without manual intervention.

C. Simulation

In our simulation, we utilized Carla [3], an open-source simulator for autonomous driving research. Carla is designed primarily for urban self-driving car scenarios, offering a realistic environment with various traffic conditions and layouts. Its advanced simulation capabilities, including detailed urban environments and a wide range of weather conditions, make it

a valuable tool for testing autonomous navigation algorithms. However, adapting Carla for our warehouse simulation posed certain challenges due to its focus on urban landscapes. Despite this, we successfully simulated all use cases by customizing Carla's environment to replicate warehouse conditions.

For the agent-based architecture, we employed SPADE (Smart Python multi-agent Development Environment). SPADE is chosen for its robustness in creating complex, multi-agent systems and its compatibility with the Python programming language, which complements our use of Carla. SPADE facilitates the development of autonomous agents with capabilities such as communication, planning, and decision-making, mirroring the operations of our physical robots.

To support SPADE's functionalities, we required an XMPP (Extensible Messaging and Presence Protocol) server, for which we selected OPENFIRE. OPENFIRE is known for its stability, scalability, and ease of setup and management. It supports a wide range of XMPP features, making it an excellent choice for handling real-time communication and coordination between agents in our simulated environment.

In the Carla simulation, we integrated multiple sensors to replicate the sensory capabilities of our real-world robots. Specifically, we used a Collision Detector and a Radar Sensor. The Collision Detector helps in simulating the robot's ability to detect and respond to immediate physical obstacles, while the Radar Sensor provides data crucial for understanding and navigating the environment, crucial for testing our robot's navigation and obstacle avoidance algorithms.

*1) **Simulating Collision Avoidance**:* In the simulation of Collision Avoidance, we meticulously configured the Radar sensor to closely mimic the behavior of the ultrasonic sensors in our actual robot. We set both the horizontal and vertical fields of view (FOV) of the Radar sensor to 10 degrees, optimizing it to detect objects in a narrow range, akin to the real-world scenario. Further fine-tuning involved adjusting the sensor's range and sensitivity to align with the physical constraints and detection capabilities of the ultrasonic sensors on the actual robot. This careful calibration ensured a realistic simulation of how the robot detects and reacts to nearby obstacles.

Regarding the PID Controller used in the simulation, we adapted the controller's parameter values (proportional, integral, and derivative gains) to suit the different dynamics of the simulated environment. This entailed recalibrating the controller to respond effectively to the simulated physics and response characteristics of the vehicle in Carla, which differ from the real robot's environment.

In line with our actual robot's logic, the PID controller in the simulation is activated only when an object is detected within a close range, ensuring that the robot's standard movement towards the target is not unnecessarily altered. This selective use of the PID controller is crucial for mimicking realistic navigation scenarios, where the robot smoothly transitions from regular path tracking to obstacle avoidance maneuvers only when required.

*2) **Simulating Collision Detection**:* In our simulation for collision detection, we employed a Collision Detector sensor. This sensor operates continuously in the background, monitoring for any physical contact with objects in the environment. Upon detecting a collision, the sensor triggers a signal, which is programmed to immediately stop the vehicle's engine. This approach effectively simulates a real-world response to collisions, ensuring immediate action to prevent further damage or complications. This simulation aspect is crucial for testing the robot's ability to safely navigate and interact within a dynamic environment, where avoiding and appropriately responding to collisions is paramount.

*3) **Simulating Crate Transfer**:* In simulating the crate transfer process, we meticulously replicated the logic used in the actual robot implementation, enhancing it to suit the simulated environment of Carla:

Coordinate-Based Crate Location:

As in the actual robot, crate locations in the simulation are defined by precise coordinates. This ensures that the simulated vehicle navigates to the exact points for crate pick-up and drop-off, mirroring the real-world scenario.

Turn Angle Calculation: The calculateTurnAngle function plays a pivotal role in determining the vehicle's navigation. While the function's core logic remains consistent with the actual implementation, its application differs. In the simulation, instead of controlling individual motor speeds, we adjust the vehicle's steering angle based on the output from calculateTurnAngle. This modification aligns with Carla's vehicle control parameters, ensuring realistic navigation.

Arrival Detection: The arrivedAtTarget function is crucial for the crate transfer simulation. It calculates the positional difference between the vehicle and the target crate location. When this difference falls within a predefined range, it signifies that the vehicle has arrived at the target, triggering the simulated pick-up or drop-off actions.

*4) **Simulating Battery Recharge**:* We set up designated recharge points on the map, strategically located to mimic the layout of a warehouse. These points represent the charging stations where the vehicle can replenish its battery.

A separate thread in the simulation continuously monitors the battery level of the vehicle. This mimics the real-world scenario where robots need to keep track of their battery status to avoid running out of power during operations.

Once the battery level falls below a predetermined threshold, our simulation logic alters the vehicle's current target to the nearest recharge point. This decision-making process is critical for ensuring uninterrupted operation and reflects the autonomous nature of the robot. Upon reaching a recharge point, the vehicle simulation includes a 'waiting' state, rep-

resenting the recharging process. This pause is crucial for simulating the time taken for a real robot to recharge. After the simulated recharging is complete, the vehicle resumes its course toward its last target.

V. Evaluation

Our evaluation demonstrates the effective performance of our autonomous Lego robot in warehouse logistics. The robot excels in obstacle avoidance, crate handling, and dynamic battery management within simulated real-world scenarios. Collaborative experiments with other robotics teams confirm its adaptability in shared environments, showcasing practical viability. For a visual representation of the robot's capabilities, you can watch the following YouTube links:

- Transfer Crates (link)
- Collision Avoidance (link)
- Battery Recharging (link)

These videos provide clear evidence of the robot's successful navigation and task execution, highlighting its potential for enhancing warehouse operations.

Collaboration with another team

One key aspect of evaluating the robustness and practicality of our autonomous Lego robot in warehouse logistics involved collaborative experiments with another robotics team. The goal was to simulate a shared warehouse environment and assess the effectiveness of our collision avoidance mechanisms in the presence of other autonomous agents. The results from this collaborative experiment significantly contribute to affirming the operational success and adaptability of our cyber-physical system.

In the collaborative setting, our Lego robot effectively demonstrated its ability to avoid collisions with other robots. The implementation of advanced obstacle detection and avoidance algorithms allowed for real-time coordination, showcasing the system's responsiveness and adaptability. The positive outcomes from this experiment not only validate the collision avoidance capabilities but also highlight the potential for seamless collaboration among multiple autonomous systems within a shared logistical space.

This collaborative evaluation serves as a valuable metric in affirming the real-world applicability of our autonomous robot, reinforcing its reliability in dynamic, multi-agent environments. As we delve into the implications of collaborative robotics, this successful collision avoidance experiment exemplifies the practical viability of our cyber-physical system in addressing the challenges of contemporary warehouse logistics.

VI. Conclusion

Our study focused on developing and simulating a Lego robot for autonomous operations within a warehouse setting. Using a PID controller for obstacle avoidance and collision detection, our Lego robot successfully navigated through the warehouse, showcasing its ability to handle tasks such as avoiding obstacles, detecting collisions, and managing crates within a simulated environment.

The study emphasized the significance of coordinating multiple robots to ensure safe and efficient operations in shared environments, highlighting their potential applications in logistics. The findings underscore the feasibility of employing autonomous robots for tasks like item transportation and decision-making in logistics scenarios.

Our investigation examines the adaptability of our cyber-physical system by simulating real-world scenarios, including obstacle avoidance, crate handling, and dynamic battery management. These successful simulations demonstrate the potential of multi-agent autonomous systems, as exemplified by the Lego robot. However, it is essential to acknowledge the continuous evolution of technology. Ongoing research and development are crucial to refining system performance, addressing limitations, and seamlessly integrating autonomous technology into various warehouse environments.

Collaboration between academia and industry remains essential for translating theoretical models into practical solutions. Future efforts should concentrate on scaling the technology, exploring diverse use cases, and fostering collaborations to make autonomous systems integral to mainstream warehouse operations. Reflecting on our journey from conceptualization to simulation, this study not only contributes insights into robotics and cyber-physical systems but also serves as an educational tool. The engagement with Lego Mindstorms Education EV3 and the SPADE multi-agent platform provides a hands-on learning experience, developing skills essential for navigating the complexities of autonomous systems. In this spirit of continuous improvement, our study lays the foundation for creating more advanced and adaptive autonomous solutions that redefine modern warehouse logistics.

VII. Further implementations

A. Hardware

Upsizing the Lego robot to real-world proportions provides a chance for hardware upgrades to better suit the demands of a full-scale warehouse. This involves incorporating sturdier motors, advanced sensors, and larger battery capacities. The shift to real-life dimensions not only improves the robot's physical resilience but also enables the integration of more advanced sensing technologies, enhancing autonomy. This transition creates an opportunity to optimize hardware components, aligning them with the challenges presented in larger logistical environments.

In addition to hardware upgrades for upscaling, further advancements can be explored in the area of power management. Implementing more sophisticated energy-efficient systems and exploring alternative power sources, such as renewable energy options, could contribute to prolonged operational durations. Moreover, enhancing the communication modules, such as incorporating 5G connectivity, can facilitate faster data exchange between robots and the central control system, improving real-time decision-making and coordination. Additionally, research

into innovative materials for the robot's physical structure could lead to lighter yet durable constructions, optimizing energy consumption and overall performance. These considerations collectively contribute to a comprehensive approach for refining the hardware aspects of the autonomous robot for more effective deployment in real-world logistics scenarios.

B. Software

To boost the Lego robot's abilities, the navigation algorithms can be improved for better adaptability in dynamic warehouse scenarios. This involves integrating machine learning to improve decision-making and path planning. Through deep reinforcement learning, the robot becomes more adept at handling complex situations, enhancing overall efficiency. Regular software updates with learning capabilities allow the robot to effectively tackle unexpected challenges, increasing its autonomy.

Simultaneously, the integration of real-time data analytics and decision support systems into the software could enable the robot to analyze sensor data and adjust its behavior on the fly. Additionally, we could incorporate machine learning for predictive maintenance to anticipate hardware issues, potentially reducing downtime and improving reliability. The inclusion of a robust decision support system could empower the robot to make informed choices in dynamic environments, potentially ensuring safe and efficient navigation through the warehouse. These potential software enhancements might collectively contribute to creating a more intelligent, adaptive, and collaborative autonomous system tailored to the challenges of warehouse logistics.

C. Simulation

To further enhance the use of autonomous robotics in warehouse environments, a specialized framework can be developed on top of Carla's existing simulation capabilities. While Carla provides a solid foundation with its focus on urban and vehicular simulations, it requires additional features for warehouse-specific scenarios. This new framework would integrate extra sensors and algorithms to tackle unique warehouse challenges like complex navigation and object handling. The use of Python in Carla's architecture facilitates this integration, allowing for rapid development and customization of the simulation environment. This enhancement can be critical for advancing autonomous robotics in demanding and dynamic warehouse settings, leading to more adaptive and efficient systems in real-world applications.

REFERENCES

[1] Burak Karaduman, Geylani Kardas, and Moharram Challenger, *Development of Autonomous Cyber-Physical Systems Using Intelligent Agents and LEGO Technology*, January 2023, https://www.researchgate.net/publication/367548933_Development_of_Autonomous_Cyber-Physical_Systems_Using_Intelligent_Agents_and_LEGO_Technology.

[2] Burak Karaduman, Baris Tekin Tezel, Moharram Challenger, Rational software agents with the BDI reasoning model for Cyber–Physical Systems, Engineering Applications of Artificial Intelligence, Volume 123, Part C, 2023, 106478, ISSN 0952-1976, https://doi.org/10.1016/j.engappai.2023.106478. https://www.sciencedirect.com/science/article/pii/S0952197623006620

[3] Carla Simulator. (n.d.). Open-source simulator for autonomous driving research. Retrieved from https://carla.org//