Simulation for the Management of an Autonomous Warehouse

Daniele Commodaro
daniele.commodaro@studio.unibo.it

April 2024

The project proposes the development of a multi-agent system, based on Jason, to simulate and manage an autonomous warehouse. The research focuses on creating a dynamic and realistic environment, where agents represent robots that load and unload goods at predetermined points. The robot agents will have autonomous behaviors and a realistic navigation system, allowing them to move from one point to another within the warehouse, avoiding obstacles. Goods management will involve the implementation of loading and unloading operations and in the delivery of the goods. During the simulation, the battery level of the robots will be considered, and when it reaches a certain threshold, they will need to recharge. Additionally, will be simulate a scenarios where a robot may break down, and another robot intervenes to repair it. The navigation system will consider obstacles and possible congestion situations, ensuring accurate simulation. Warehouse modeling will include loading/unloading zones and the ability to dynamically move goods. A user interface will provide real-time visualization of the simulation, allowing the user to monitor the robot positions and warehouse status.

Contents

1 Objectives and expected outcomes

Project objectives:

- System development: implement a multi-agent system that includes robot agents, cargo, and loading/unloading points.
- Definition of autonomous behaviors: define autonomous behaviors for the robots, enabling them to make decisions based on the context.
- Achieve a realistic navigation system: ensuring that robots reach their destinations efficiently, considering the use of a basic obstacle avoidance approach, with a focus on simplicity and functionality.
- Managing goods: implementing actions for managing goods such as delivery, loading and unloading operations.
- Warehouse modeling: creating a model of the warehouse with loading/unloading zones, waiting and charging points and the ability to dynamically move goods.
- Implementing a user interface: developing a GUI for real-time simulation visualization, allowing the user to monitor the position of robots and the state of the warehouse.

Expected outcomes include:

- Obtain a working simulation of the autonomous warehouse with agents interacting realistically.
- Verify realistic autonomous behaviors of the robots during goods handling and management activities.
- Demonstrate the effectiveness of the navigation system employed in guiding the robots movements.
- Move the goods by changing their position within the warehouse during the simulation execution.
- Achieve an intuitive user interface that enables effective monitoring of the simulation.

2 Work plan

The work plan is the following:

- Requirements Analysis:
  - Definition of project objectives.
  - Identification of functional and non-functional system requirements.
  - Planning of desired scenarios, including warehouse dynamics, cargo management, and robot agent behaviors.
- System Design:
  - Defining the architecture of the multi-agent system using Jason.
  - Designing the robot agents, warehouse, and goods.
  - Designing the navigation system used by robots to move within the warehouse.
- Implementation:

  - Creation of agents and specific rules for the simulation.
  - Implementation of autonomous behaviors for the robot agents.
  - Implementation of goods characteristics and loading/unloading operations.
  - Integration of a navigation system for the robots.
  - Implementation of the ability to move goods within the warehouse.
- Development of the user interface:
  - Design and implementation of a user interface for real-time visualization of the simulation.
- Testing and optimization:
  - Conducting rigorous tests to ensure the proper functioning of the system.
  - Performance optimization of the system and resolution of any bugs or issues.
- Documentation:
  - Creation of a clear and comprehensive report to describe the system.
- Evaluation and analysis of the results.
  - Assessment of the simulation in relation to the project objectives.
  - Analysis of the obtained results.
- Conclusions and presentation:
  - Drafting project conclusions.
  - Preparing the presentation to illustrate the system, the achieved objectives, and the obtained results.

3 Project requirements

3.1 Functional requirements

- Management of goods delivery, loading and unloading operations.
- Monitoring the quantity of goods available and generating orders when the level drops below a pre-established threshold.
- Management of the robot's charge level and charging when necessary.
- Random simulation of robot breakdowns and intervention of other robots for repairs.
- Communication between agents to coordinate goods and emergency management activities.

3.2 Non-functional requirements

- Efficiency: the system must be able to efficiently manage operations, agents communication and simultaneously monitor the status of the robots.
- Reliability: the system must be robust and handle emergency situations effectively, ensuring that goods are delivered without interruption.
- Usability: the user interface must be intuitive to allow users to monitor system activities efficiently.
- Maintainability: the code must be well structured and documented to allow for easy maintenance and future updates.

3.3 Planning of desired scenarios

3.3.1 Use case planning goals

The use case planning was designed to realize a set of desired scenarios within the simulated warehouse environment. The main objectives include:

- Simulation of warehouse dynamics: use case planning aims to create a realistic simulated environment, where various warehouse elements, such as shelves, loading and unloading areas, and obstacles, influence the movement and behavior of the robotic agents.
- Effective cargo management: an important goal is the efficient management of cargo within the warehouse. This includes monitoring the number of available items, generating orders to replenish the warehouse when necessary, and coordinating loading and unloading cargo activities.
- Defining robotic agent behaviors: use case planning includes defining robotic agent behaviors, such as movement between different locations in the warehouse, loading and unloading operations, and actions to take in the event of failures or need to recharge.

3.3.2 Definition of the desired scenarios

- Load transportation: scenarios include situations where robots need to transport loads from one location to another within the warehouse. This can involve moving around obstacles, which may be other agents or occupied loading and unloading positions, and delivering loads to designated destinations.
- Fault management: the scenarios also include the management of robot faults during transport operations. For example, is simulated the situation in which a robot stops due to a malfunction and requires assistance from another robot to resume activity.
- Charging needs: is considered the scenario where robots need to monitor their battery charge level and plan charging tasks accordingly. This could involve returning to the charging station when the battery reaches a certain critical level and coordinating charging activities to ensure the operational flow of the warehouse is not interrupted.
- Dynamic Navigation: the scenarios involve robots navigating an ever-changing environment, where they must adapt their navigation paths in real time. This includes the robot's ability to avoid obstacles, such as other robots in their path, and to stop when they encounter occupied spots along their path.

By defining these desired scenarios, we aim to test the system's capabilities and performance in a variety of realistic situations, ensuring that the automated warehouse management system is able to successfully address operational and logistical challenges.

3.3.3 Warehouse dynamics

Inside the simulated warehouse, there are different dynamics that influence the movement and behavior of the robots. These include the layout of shelves, which provide destination points for the delivery of goods. Each time a load is completed, the goods are delivered to a central unloading point, where robots come to load the goods to transport them to opposite sides of the simulation area.

In addition to shelves, there are also other elements in the environment that robots need to consider when navigating. There are points where the robots can stand by, in addition, there are charging points, where the robots can stop to regenerate their energy when necessary.

One of the main features of the simulated environment is the presence of a dynamic level that affects the behavior of the robots and the flow of operations in the warehouse. In particular, robots may encounter unexpected situations, such as random failures, which directly affect their operation. When a robot breaks, for example, it starts moving randomly around the warehouse area until it is repaired or recovered by another robot.

Furthermore, load and order management is a crucial aspect of the simulation. At the designated delivery point, known as the "unloading point", a certain number of orders are placed. Once the expected maximum quantity of goods has been successfully delivered, the orders are stopped and the simulation can be concluded. This mechanism defines an end goal for the simulation, allowing us to understand whether the execution can be considered successfully completed.

These warehouse dynamics make the simulated environment more realistic and challenging for robots, requiring them to constantly adapt their behavior based on ever-changing conditions.

3.3.4 Cargo management

Cargo management includes tracking the number of items, generating orders to replenish the warehouse when necessary, and coordinating loading and unloading activities. Below is a detailed explanation of how these aspects should be managed:

- Tracking the number of items: the number of items available in the warehouse is tracked in the project model. This value is updated when items are placed or downloaded. Updating agents' perceptions of the environment can inform agents about the current number of items available in the warehouse.
- Generating orders to replenish the warehouse: when the number of items in the warehouse drops below a certain critical level, agents can generate an action to order new items.
- Coordination of loading and unloading activities:
  - Loading and unloading actions are managed through communication between agents and perception on changes in the warehouse environment.
  - Agents can request access to one of the available racks to place items and to the loading point to retrieve them. The warehouse environment handles these requests and updates the rack status accordingly.
  - Agents can also be coordinated to avoid collisions and overlaps during loading and unloading operations, through the implementation of a motion management algorithm that allows the detection of obstacles and which allows agents to avoid.
  - Management of loading and unloading activities can be implemented through actions managed in the agents' plans and defined in the warehouse environment.

  The warehouse cargo management system includes: tracking the number of items, generating orders to replenish the warehouse, and coordinating loading and unloading activities. The system will allow agents to operate efficiently within the warehouse, maintaining a constant flow of goods and ensuring that goods handling operations are carried out in a continuous and coordinated manner.

3.3.5 Behaviors of robotic agents in the warehouse

Expected behaviors include movement between different points of the warehouse, the robotic agents must be able to move autonomously within the warehouse using a navigation algorithm. The algorithm that manages the movement can be implemented in a simple way, directing the agent in a specific direction allowing it to go around any obstacles that may be in its path or to stop when necessary, for example when trying to access a rack which is already occupied. The navigation strategy employed by the algorithm must recognize whether the type of obstacle obstructing the robot agents' path should be bypassed. When this is possible, the agent will take an alternative route to move around the obstacle. This route may be chosen randomly but the final destination must always be taken into consideration in such a way as to maintain a direction consistent with its destination. Through specific methods, robotic agents are capable of performing unloading operations of items from the delivery point and positioning operations of goods within various types of racks. Interaction with other agents and the warehouse environment enables the coordination of operations across various types of situations that may arise. When items are delivered to the delivery point, a random value can be drawn to specify which rack the item should be placed in by the agent. This type of management allows for a simple representation of different types of items that need to be brought from agents to different positions. Robotic agents must be capable of detecting malfunctions or low battery levels. In case of malfunction or the need for recharging, the agents can take appropriate actions such as returning to their charging station or reporting the malfunction to another agent to request a repair intervention. These actions must be managed through a logic implemented by the robotic agents, which monitors the robot's status and makes appropriate decisions based on the detected conditions.

3.3.6 Test scenarios and performance evaluation

In the context of developing the simulated warehouse management system, the process of testing and performance evaluation can be conducted systematically. The specific test scenarios that are expected to be verified concern the various features that are intended to be implemented, a practical approach is expected to be adopted by testing the simulation behavior throughout the entire development process. This approach allows for the timely identification and addressing of emerging challenges and issues, ensuring that the system responds appropriately and efficiently to the needs of the simulated warehouse. Although this methodology may deviate from traditional test planning, it can still prove effective within the development process context.

4 System architecture and design

The simulation system is based on a centralized infrastructure, managed through a central entity called Multi-Agent System (MAS), which coordinates all activities of the agents in the autonomous warehouse. The system is equipped with a user graphical interface (GUI) for monitoring activities within the warehouse. The system comprises various types of agents: two types of robotic agents responsible for handling goods movement, an agent for activity coordination, and another agent dedicated to warehouse deliveries.

4.1 Components

The architecture of the system consists of the following components:

4.1.1 Warehouse environment

The warehouse environment represents the environment of the simulated warehouse in which agents interact. It manages the actions of agents within the environment, providing methods for executing actions such as accessing racks, releasing racks, unloading merchandise, and others. It updates agents' perceptions based on the current state of the warehouse environment, using information provided by the model. It initializes the warehouse model and the view. The main exposed features are:

- Warehouse environment and graphical interface initialization.
- Updating agents' perceptions based on the current state of the warehouse environment, including agent positions, rack status, merchandise availability, etc..
- Managing agents' actions in the warehouse environment by invoking appropriate methods defined in the model to perform the action requested by the agent.

- Provides auxiliary methods that can be used in support of certain operations, such as obtaining the position of destinations specified by agent actions.

The warehouse environment is responsible for managing interactions between agents primarily through two main functions:

1. Updating agents' perceptions: the environment provides agents with information about the current state of the simulated environment, including data such as agent positions, the presence of goods to be delivered, and other elements relevant to the system's operation. Agents perceive changes within the environment by updating their own perceptions of it.

2. Execution of agents' actions: agents send actions to the environment to influence the state of the simulated environment or to interact with other agents. The environment is responsible for interpreting these actions and applying them appropriately to the environment. Through a dedicated method, actions sent by agents are managed, and the effect of these actions on the environment is determined.

However, in the system, agents can interact directly by exchanging messages without necessarily going through the environment. This is an important aspect to consider because it provides agents with greater flexibility and autonomy in interacting with each other, allowing them to coordinate and collaborate without always having to interact through the environment. Therefore, both modes of interaction, whether through the environment or directly between agents, contribute to the overall functioning of the system.

The environment utilizes the warehouse model to obtain information about the state of the warehouse and update agents' perceptions accordingly. It also handles the execution of agents' actions by forwarding requests to the model for the actual execution of the action and subsequently updating agents' perceptions. Additionally, it provides an interface for environment initialization and management of agents' actions, thus ensuring proper execution of the warehouse simulation.

4.1.2 Warehouse model

The architectural design of the warehouse model involves modeling an environment in the form of a grid, a set of attributes used for the essential data of the application and the definition of methods that incorporate the business logic of the actions that are performed by the agents. The main architectural elements are described below:

- Grid representation: the model provides a representation of the environment in the form of a grid. Through this representation, the topology and specific characteristics of the warehouse environment can be defined.

- State variables and constants: the model contains state variables representing the current state of the simulated environment, such as rack availability, the number of available goods, robot charge levels, and the breakdown status of a robot. Constants are used to identify objects in the environment, such as racks and the delivery point.

- Initial placement of agents and environment objects: in the model, agents and environment objects such as racks and the delivery point are initialized and placed at specified positions on the grid.

- Methods for managing agent actions: the model includes methods for handling agent actions in the environment. These methods include actions such as accessing racks, moving agents to a specific location, placing and delivering goods, motion management system as well as managing robot battery recharge and breakdown conditions.

- Support methods: are operations that support the logic implemented for managing the movement of robot agents, for updating the graphical interface and others

4.1.3 Graphical component

The system view is instantiated by the environment component and is updated through the methods defined in the model. The main functionality that are carried out are the following:

- Represents the view of the environment in the form of a grid.
- Set parameters such as window title and grid size, and set window visibility.
- Draws the environment objects on the grid. It uses information from the model to determine the position of agents, racks, and the delivery point, and then draws them on the grid with appropriate colors and labels.
- Draws the agents on the environment. It uses information from the model to determine the position of agents and draws them on the grid with different colors based on the agent's status, such as the presence of a breakdown or the charge level.

In summary, the view component provides a graphical interface for displaying the warehouse environment and the status of agents in an intuitive and informative manner.

4.2 Introduction to agent design

In the context of the warehouse management system, the core of its operations is represented by its agents. These agents, in three distinct types: Robot, Delivery, and Truck, cooperate to ensure an efficient flow of goods within the environment. In this section of the architectural design, we will explore the dynamics of interaction between these agents, focusing on their modes of communication, the information exchanged, and the overall interactions.

4.2.1 Types of agents and modes of communication

The three main types of agents present in the system are as follows:

- Robot: these agents are responsible for the physical movement of goods within the warehouse. Each robot can perform various actions, such as transporting goods from delivery point to the racks.
- Delivery point: this agent coordinates the arrival and distribution of goods within the warehouse. It is responsible for interacting with the robots to receive and distribute goods, as well as managing incoming orders and communication with the truck for goods delivery.
- Truck: this agent manages the delivery of goods to the warehouse. It receives orders from the delivery point and handles the physical delivery of goods to it.

Communication between these agents occurs via synchronous message exchanges. Messages are used to convey new goals to be achieved or to request new ones. For example, the Delivery agent sends targets to the robots to indicate when and if there are goods that need to be transported and placed inside a shelf. Similarly, the truck receives orders from the delivery point to perform a new delivery task.

4.2.2 Interactions and information exchange

The interactions between agents are characterized by a constant flow of information and goals. For example, robots receive instructions from the Delivery if there are goods that need to be transported, or when they run out, and robots are notified that they can go to their waiting area. Additionally, robots communicate to the delivery point the status of their activities, such as when they complete a series of actions, they send a message to the delivery point to communicate the completion of the task and to ask if there are any other items to unload. The Delivery agent, in turn, manages incoming orders and communicates with the truck to plan deliveries.

The agents communicate with each other through message exchanges that convey the objectives to be achieved. Whenever an agent receives a message with a new objective, it plans and executes the necessary actions to achieve it. This approach allows agents to cooperate and coordinate effectively within the system. To ensure the synchronization of activities and prevent conflicts, agents use semaphores that regulate their availability to execute the received objectives. The semaphores, which are essentially beliefs, can be set as 'free' or 'busy' depending on the agent's state. For example, consider the case of a robot that has depleted its battery. When the robot detects that the battery level is low, it will set its semaphore as 'busy'. This means that if the robot were to receive new objectives for transporting goods, it will not execute them immediately. Instead, it will focus first on recharging the battery. Once the robot has completed the recharge and set the semaphore back to 'free', it will be available again to execute the received objectives. The use of semaphores helps to ensure that agents are instructed to pursue a new objective only when they are ready to do so, thus preventing robots from receiving and consequently attempting to execute conflicting objectives which could lead them to have a incorrect behavior. This architecture allows agents to dynamically manage their activities in response to changing environmental conditions and to collaborate effectively to achieve the system's goals as a whole.

4.2.3 Simulation of additional behaviors

In the design of the autonomous warehouse system, a series of simulated functionalities have been introduced to enrich the simulation experience and provide a more realistic representation of the internal dynamics of the warehouse. These additional functionalities have been deliberately introduced and do not necessarily represent failure situations within the scope of agent goal management. Below are detailed the main simulated functionalities implemented in the system:

- Agent breakdown management: one of the simulated functionalities in the system is the management of agent breakdowns. This behavior has been introduced to simulate a situation where an agent may experience a mechanical or electronic failure that compromises its proper functioning. When an agent detects that it is broken, it activates a rescue mechanism by sending a help request message to other agents. Meanwhile, the broken agent continues to move randomly until it receives assistance.

- Battery charge management: another simulated functionality is the management of the battery charge of robotic agents. This behavior has been introduced to simulate the limited energy autonomy of agents operating in the warehouse. When an agent detects the need for recharging, it temporarily suspends its activities and heads to a specially designated area for recharging. During this period, the agent is not available to perform other tasks.

These simulated functionalities have been designed to enhance the simulation experience and provide a more accurate representation of the challenges that may arise in the autonomous warehouse environment. It is important to emphasize that these functionalities do not necessarily represent failure situations in the context of agent goal management but rather additional elements that contribute to the complexity and dynamism of the system.

4.2.4 Design of the robot navigation system

The robot navigation system is designed to allow robotic agents to move autonomously within the warehouse to reach desired locations, such as delivery areas or storage racks. The system design is based on two main components: action plans defined in the Jason language and movement procedures implemented in the code.

The action plans consists in two main rules that manage the behavior of agents when they reach or dont reach a destination. When an agent reaches its destination, it updates its perceptions of the environment, through which it understands what the next tasks to be performed are. Until the agent reaches its intended destination, it utilizes a rule instructing it to continue advancing towards it using a specific action.

The procedure used for movement calculates the direction towards the desired destination and moves the agent in the correct direction. In case the next position is already occupied by another agent, the agent may attempt to avoid the obstacle. This procedure involves randomly choosing a new direction of movement to try to reach the destination.

However, if two agents attempt to move simultaneously towards the same position, a concurrency issue arises that must be properly handled. In this case, it is necessary to ensure exclusive access to the shared resource, which in this context is represented by the model grid on which agents move. Exclusive access to the model grid prevents two agents from occupying the same position simultaneously and avoids any unwanted collisions or overlaps.

This concurrency management mechanism ensures safe and coordinated movement of agents within the warehouse, minimizing the risk of conflicts and ensuring efficient system operation.

5 Detail design

5.1 Components structure

The following class diagram represents the agent-based system for the automated warehouse. Here's an explanation of the main classes and their relationships:

- WarehouseView: this class represents the view of the warehouse and is responsible for graphically displaying the state of the warehouse. It has a reference to the warehouse model (wModel) to obtain information about the current state of the warehouse. It provides methods for drawing the graphical view of the warehouse, including agents and objects inside the warehouse. It also contains the methods drawAgentInCharge() and drawBrokenAgent() to draw special states for the agents in the view.

- WarehouseModel: this class represents the warehouse model and contains the business logic of the system and the essential data to be stored. It manages the position of objects in the warehouse, agent movement, goods management, and other warehouse-related operations. It also contains methods to update the view and helper methods to manage agent movement. The methods defined in the model are invoked through the environment class following the action plans undertaken by the agents.

- Delivery and Truck: these agents are responsible for managing the goods and sending the goods to the warehouse respectively. The Delivery agent has an associated delivery point which represents the area of the warehouse where the items are actually delivered and has the task of coordinating the other types of agents by assigning objective to be carried out through the exchange of messages. The Truck agent receives new orders from the delivery agent and proceeds to deliver them to the destination.

- Robot: these agents represents the robots in the warehouse system. The robots contact the Delivery agent through messages to inquire if there is any work to be done. If there is, these agents proceed to the delivery point to pick up the items, which are then transported to one of the available racks, provided they are not occupied. If a rack is occupied, the robot waits for it to be freed up. When there are no more items to be placed in the racks, the robot agents can return to their waiting position, waiting for new items to be delivered at the delivery point. After a certain number of trips between the delivery point and the racks, each agent needs to recharge at some point. To do this, it goes to its charging position to recharge its batteries. Once it has completed its charge, the robot can resume

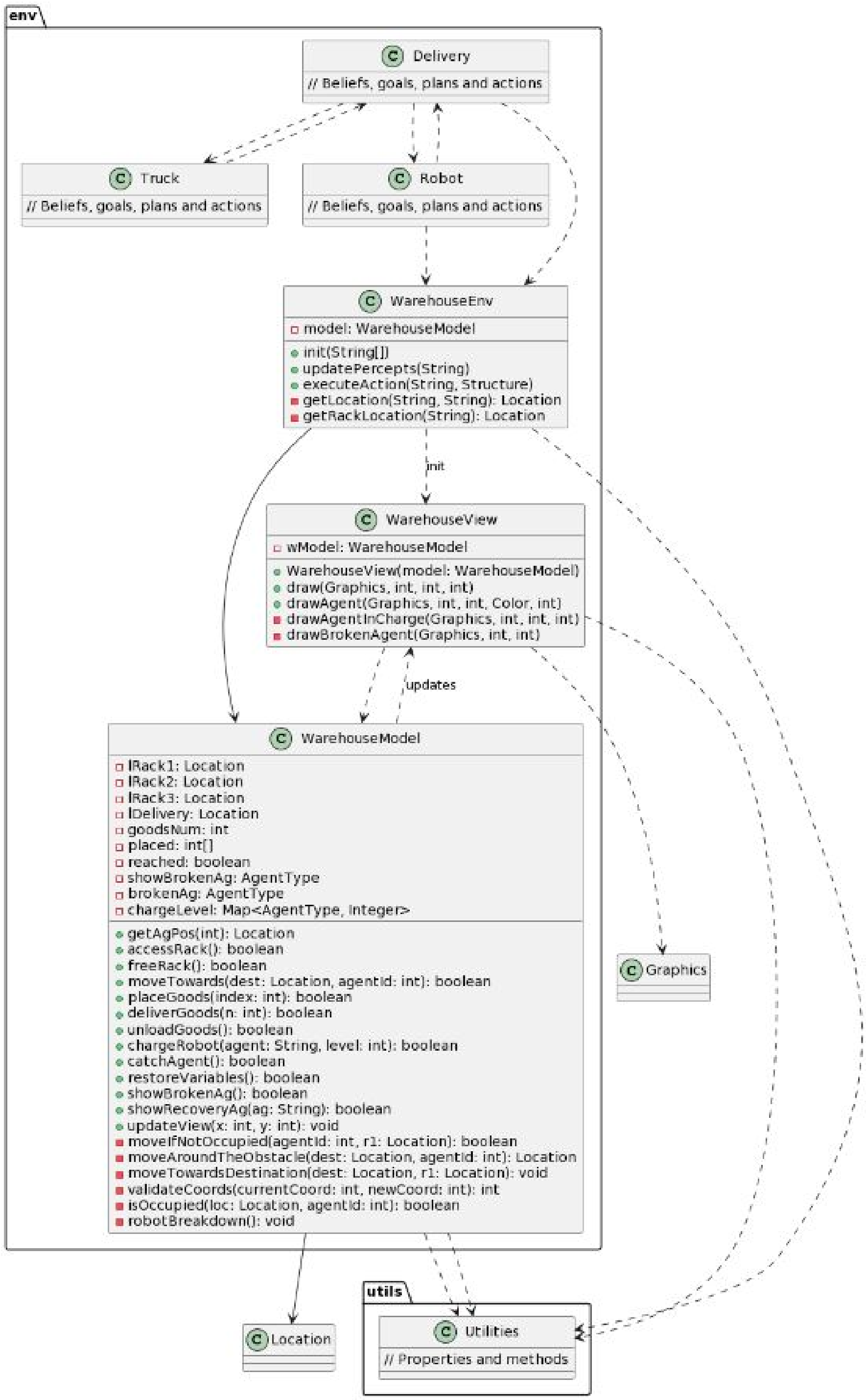

Figure 1: System class diagram

placing items, if any. Within the simulation, one of the two robots present will randomly malfunction at a certain frequency. When this happens, a help message will be sent to the second robot, which will then proceed to assist the malfunctioning robot when available.

- WarehouseEnv: this class represents the warehouse environment, which coordinates interactions between agents, the warehouse model, and the warehouse view. It has a reference to the warehouse model (model) and uses methods such as init, updatePercepts, and executeAction to initialize the environment, update agent perceptions, and execute actions within the environment. It also contains methods getLocation() and getRackLocation() to obtain positions within the warehouse.

5.2 Behavior of the agents

In this section the agents' behavior during the execution of their plans and actions is detailed. An in-depth analysis of the plans and actions carried out by the agents is provided, highlighting the interactions that occur between them. These interactions are based on synchronous message exchange, meaning that agents communicate with each other in a coordinated manner and respond to requests and notifications from other agents in real-time. To facilitate understanding of the execution process and interaction of the agents, specific diagrams are used to visually illustrate how these dynamics occur within the system.

5.3 Robot agents

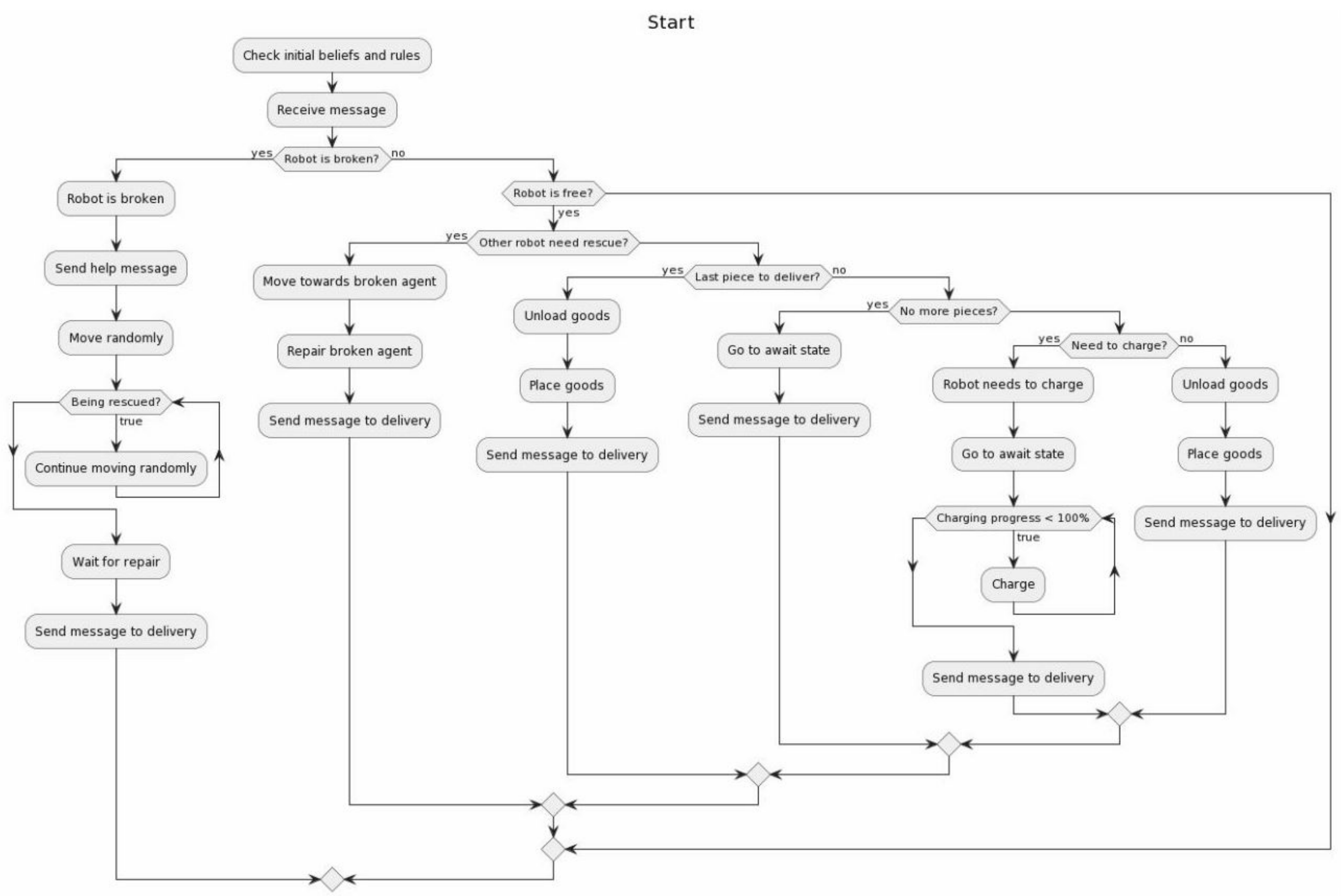

Figure 2: Robot activity diagram

The diagram illustrates the flow of activities of a robot during its operation, considering various situations that may occur. Initially, the robot checks its initial beliefs and established rules. Subsequently, it receives a message from the Delivery agent, representing a request with the next goal to achieve. The robot then checks if it is broken. If it is broken, it sends a help message and starts moving randomly, waiting for the other robot to intervene for assistance. It continues to move randomly until it is rescued and repaired. After the rescue, it waits for the repair and then sends a message to the Delivery agent to communicate the end of the assistance and request a new goal. If the robot is not broken, it checks if it is free. If it is free and another robot needs assistance, it moves towards the faulty robot to provide assistance. After reaching the faulty robot, it proceeds with the repair and then sends a message to the Delivery agent to inform the end of the assistance and request a new goal to pursue.

If the robot needs to be recharged, it moves to the waiting area and starts the battery charging process. Once the charging is complete, it sends a message to communicate the end of the charging and request a new goal. Finally, if the robot is neither broken nor engaged in assisting other robots and does not need to be recharged, it checks the type of task to perform. There is a particular case in which if the Delivery agent communicates that there is the last item to unload, the robot agent will ignore the low charge level and proceed to unload the last piece, allowing the order of new items before recharging. Then if there are items to be arranged, it performs unloading and placing operations and sends a message to report completion and request a new goal from the Delivery agent each time it places an item in the rack. If there are no more items to deliver, it enters the waiting state and sends a message to report the end of operations. The flow ends when the Delivery agent stops communicating new goals.

5.3.1 Occupation management of the robots through semaphores

The robots use synchronous messages to communicate with the Delivery agent. These messages are used to receive new goals to accomplish. During the execution of a goal, the robots use a belief called "semaphore(free)" to determine whether they can accept new goals or not.

The semaphore represents a kind of "occupation" of the robot: if the semaphore is set to "free", the robot is available to perform new tasks and can accept new goals received from the Delivery agent. In this case, the robot will proceed to pursue the received goal.

However, if the semaphore is not free, the robot is not available for new tasks and will not consider any new goals received from the Delivery agent. Instead, it will continue to engage in the current goal until completion.

After successfully achieving a goal, the robot sets its semaphore to "free" and communicates with the Delivery agent to request new goals. This way, the robot prepares to receive and pursue new instructions once it is available again.

This practice helps ensure that the robots do not overload themselves with too many tasks simultaneously and that they are always ready to accept new tasks once they have completed the current ones.

5.3.2 Robot agent's action sequence

In this section, the main sequence of actions that a robot agent performs upon receiving a new delivery objective from the control system, known as Delivery, is depicted. The actions carried out include:

- Prerequisites check: the robot agent begins by checking a series of prerequisites to determine if it can take on the new objective. These prerequisites include:
  - Checking that the robot is not damaged.
  - Checking if the semaphore is free, indicating that the robot is available to take on a new task.
  - Ensuring that the robot does not require a battery recharge.
  - Verifying that the robot is not already engaged in assisting another agent.
- Action execution: if all prerequisites are met, the robot proceeds with executing the actions to fulfill the new objective:
  - The robot sets the semaphore to busy, indicating that it is currently engaged in fulfilling the objective.
  - The robot moves towards the location of the delivery point, and once reached, it picks up an item. The type of item is randomly selected, and based on the item type, it needs to be transported to a specific rack.

– Once reaching the rack, it accesses it to place the goods, and then frees the rack.
– Restores the semaphore to free, indicating that the robot has successfully completed the objective and is available to perform new tasks.
– Sends a message to the Delivery agent to request new objectives to pursue.
– Records the current date and time within a fact "consumed", which is used to count the number of transports carried out by the robot and to understand when it will be necessary to stop for a recharge cycle.

When there is only one last item to unload, the robot receives a specific message from the Delivery indicating this situation. In this case, the sequence of actions will be the same, but the battery level of the agent will not be considered or updated. The agent will proceed to recharge its battery once it receives the next objective.

```

+!sendMsg(ask)
   : true
   <- .send(delivery, achieve, ask(delivery, goods)).

+!has(delivery, goods, Agent)
   : semaphore(free) & not need_charge(Agent) & not broken(_) & not rescuing(_)
    <- .print("Received goal"); -semaphore(free); !at(Agent, delivery);
       unload(goods);
       utils.rand_int(R, 1, 4);
      !at(Agent, R);
      access(rack); place(R); free(rack); +semaphore(free); !sendMsg(ask);
      .date(YY, MM, DD);
       .time(HH, NN, SS);
       +consumed(YY, MM, DD, HH, NN, SS, Agent).

+!go(await, Agent)
    : semaphore(free) & not at(waitingZone) & not broken(_) & not rescuing(_)
    <- .print("I'm going to wait"); -rackType(0); -semaphore(free); !at(Agent,
        waitingZone).

+!at(Agent, P)
   : not at(Agent, P)
   <- move_towards(P); !at(Agent, P).

+!at(Agent, P)
   : at(Agent, P)
   <- .wait(200).
```

In cases where no more items are available to be placed, the robot may receive the objective to return to its waiting position. In this case, it heads towards its waiting position, and once there, it sends a message to the Delivery asking if there are new tasks to be performed. Situations of breakdown, recharging, and repair will be addressed in the following sections.

5.3.3 Management of robot charging

To ensure that the robot maintains an adequate level of battery charge, a calculation based on the number of transports is performed. The robot tracks its energy consumption and compares it with a predefined limit. To facilitate counting, the robot declares a 'consumed' fact for each trip, consisting of the date, time, and agent name. Declaring the fact with the timestamp allows generating a unique fact for each trip, enabling easy counting. When the number of facts exceeds the predefined limit, the robot understands that it needs to recharge. This is checked through the rule need_charge(Ag), which triggers when the energy consumption exceeds the allowed limit.

When the robot receives a new delivery objective and determines the need to recharge, the recharging process is initiated:

1. A number of consumed facts equal to the preset limit are removed from the agent's beliefs, allowing the count to be reset to the starting position.
2. The robot moves to a waiting position.
3. It initiates the recharging process, which occurs in multiple incremental phases, charging the battery from 0% to 100% with charge increments of 20% at each phase. During the charging process, the color of the agent changes according to its battery level.
4. During each charging phase, the robot waits for a specified period of time, indicated by .wait(), to allow the recharging process to complete.
5. Once the recharge is complete, the robot sends a message to the Delivery agent to request new delivery objectives.
6. The semaphore is set to free to indicate that the robot is available again for assignment of new tasks.

```
limit(goods, 5).

need_charge(Ag) :-
   .date(YY, MM, DD) &
   .count(consumed(YY, MM, DD, _, _, _, B), QtdB) & limit(B, Limit) & QtdB > Limit.

+!has(delivery, goods, Agent)
    : semaphore(free) & need_charge(Agent) & not broken(_) & not rescuing(_)
    <- .print("I need to recharge");
       -consumed(_, _, _, _, _, _, Agent);
       -consumed(_, _, _, _, _, _, Agent);
       -consumed(_, _, _, _, _, _, Agent);
       -consumed(_, _, _, _, _, _, Agent);
       -consumed(_, _, _, _, _, _, Agent);
       !go(await, Agent);
       -semaphore(free);
       charge(0);
       .wait(800);
       charge(20);
       .wait(800);
       charge(40);
       .wait(800);
       charge(60);
       .wait(800);
       charge(80);
       .wait(800);
       charge(100);
       .wait(400);
       charge(-1);
       +semaphore(free);
       !sendMsg(ask).
```

In summary, this is the process through which the robot monitors and manages its battery charge, ensuring that a sufficient level of energy is maintained for the execution of its tasks and that recharging is carried out effectively when needed.

5.3.4 Management of breakdown and rescue

Within the simulation model, a system is implemented to manage the random breakdown of robots. This system uses a timer to periodically check the status of the robots and determine if any of them are broken.

The key method for managing robot breakdowns is a method called robotBreakdown(). This method is executed periodically by the timer to check if a robot is broken. If there is no already broken robot, the method randomly selects one of the two available types of robots and marks it as broken. This random selection is made using a random number generator. When an agent is marked as broken in the model, it becomes aware of this through the updating of its perceptions within the environment.

When a robot perceives itself as broken, it sends a help message to request assistance from the other robot. When the other robot receives the help message, it updates his beliefs with a fact to stay aware of what happened, he will complete his current objective if he is executing one and then he will respond to the call for help by giving it priority over everything else that may happen and heading towards the broken robot to be able to help.

To ensure smooth operation of the system, the breakdown situation must be handled in all possible scenarios that may occur at the time of detecting the breakdown. Therefore, agent plans consider a possible breakdown in all possible situations, whether receiving different types of goals from the Delivery or when the battery is low. When a robot detects a breakdown, it immediately communicates its condition by sending a help message. If the robot has been assigned to unload the last item to be delivered, it will also inform the Delivery that it cannot complete this operation due to the breakdown. Once the robot has sent the help message, it begins to move randomly until it is rescued and repaired. This process is managed by a plan that continues to move the robot randomly until it is rescued. The robot perceives from the environment that it has been reached by the assisting robot, and when this happens, it stops and waits for its repair. When it is restored, it sets its semaphore to free and sends a message to request new goals from the Delivery.

```
+!has(delivery, goods, Agent)
    : semaphore(free) & broken(Rescuer)
    <- -semaphore(free); .print("BREAKDOWN"); show_broken(Agent); .send(Rescuer,
        achieve, provide_help(Agent)); !fault_resolved(Rescuer).

+!go(await, Agent)
    : semaphore(free) & broken(Rescuer)
    <- -rackType(0); -semaphore(free); .print("BREAKDOWN") show_broken(Agent);
        .send(Rescuer, achieve, provide_help(Agent)); !fault_resolved(Rescuer).

+!has(delivery, lastPiece, Agent)
    : semaphore(free) & broken(Rescuer)
    <- -semaphore(free); !unmanaged(last_piece); .print("BREAKDOWN");
        show_broken(Agent); .send(Rescuer, achieve, provide_help(Agent));
        !fault_resolved(Rescuer).

+!has(delivery, goods, Agent)
    : semaphore(free) & need_charge(Agent) & broken(Rescuer)
    <- -semaphore(free); .print("BREAKDOWN"); show_broken(Agent); .send(Rescuer,
        achieve, provide_help(Agent)); !fault_resolved(Rescuer).

+!fault_resolved(Rescuer)
   : not fault_resolved(Rescuer)
   <- move_towards(broken); .wait(500); !fault_resolved(Rescuer).

+!fault_resolved(Rescuer)
   : fault_resolved(Rescuer)
   <- .print("I have been reached for rescue");
```

```
      .wait(4500);
      restore(env);
       +semaphore(free);
       !sendMsg(ask).

+!unmanaged(last_piece)
    : true
    <- .send(delivery, achieve, unmanaged(last_piece)).
```

The plan +!fault_resolved(Rescuer) handles the behavior of the robot during the breakdown. The condition of this plan changes and becomes true when the other robot, identified as "Rescuer," reaches the broken robot on the grid. This change is perceived through an update that occurs within the environment.

The plan +!unmanaged(last_piece) is used to communicate to the Delivery that the last item for which pickup was requested will not be unloaded due to the breakdown. In practice, when the robot detects that it cannot complete the delivery of the last item due to the breakdown, it sends a message to the Delivery indicating that the item will not be unloaded. The Delivery can then make the necessary decisions on how to handle this situation.

The rescue management is aimed at ensuring that robots can support each other in case of emergencies. Similar to the previous scenario, this situation has been considered in all possible situations, which can be distinguished into four cases: normal request for unloading goods, request to go on break, request to unload the last item, and the need to recharge their batteries. When a rescue is requested, the robot receiving the request updates the fact +rescuing(Agent) with a plan +!provide_help(Agent), which serves to be aware that it must provide assistance as soon as it finishes its current objective, if it is performing one.

```

+!provide_help(Agent)
    : true
    <- +rescuing(Agent).

+!has(delivery, goods, Ag)
    : semaphore(free) & rescuing(Agent)
    <- -semaphore(free); .print("I go to help"); !reach_robot(Agent).

+!go(await, Ag)
    : semaphore(free) & rescuing(Agent)
    <- -rackType(0); -semaphore(free); .print("I go to help"); !reach_robot(Agent).

+!has(delivery, lastPiece, Ag)
    : semaphore(free) & rescuing(Agent)
    <- -semaphore(free); !unmanaged(last_piece); .print("I go to help");
        !reach_robot(Agent).

+!has(delivery, goods, Ag)
    : semaphore(free) & need_charge(_) & rescuing(Agent)
    <- -semaphore(free); .print("I go to help"); !reach_robot(Agent).

+!reach_robot(Agent)
   : not reach_robot(Agent)
   <- catch(Agent); move_towards(Agent); !reach_robot(Agent).

+!reach_robot(Agent)
   : reach_robot(Agent)
   <- .print("Broken agent reached");
      show_recovery(Agent);
      .wait(4500);
```

```
    -rescuing(Agent);
    restore(env);
    +semaphore(free);
     !sendMsg(ask).
```

During the rescue operation, the assisting agent moves on the model grid until reaching one of the cells adjacent to the position of the broken robot. Once reached, both agents update their perceptions to understand that the repair procedure is in progress. Both agents simulate the repair operation with a wait(). This is made visible through a graphical representation showing the agent in a specific mode indicating that it is performing the repair. Meanwhile, both agents patiently wait for the intervention to be completed. Once the repair is finished, both agents return to their normal availability and communicate with the Delivery agent to receive new objectives. At this point the simulation behavior returns to its standard operation, with all agents ready to receive and execute new activities.

5.4 Delivery agent

5.4.1 Agents interaction

The following sequence diagram is used to represent the interactions that occur among the simulation agents. These agents collaborate for the transfer of goods from one point to another and coordinate their actions through synchronous message exchange. The diagram is used to illustrate clearly and concisely how the interactions between agents occur during the goods delivery process.

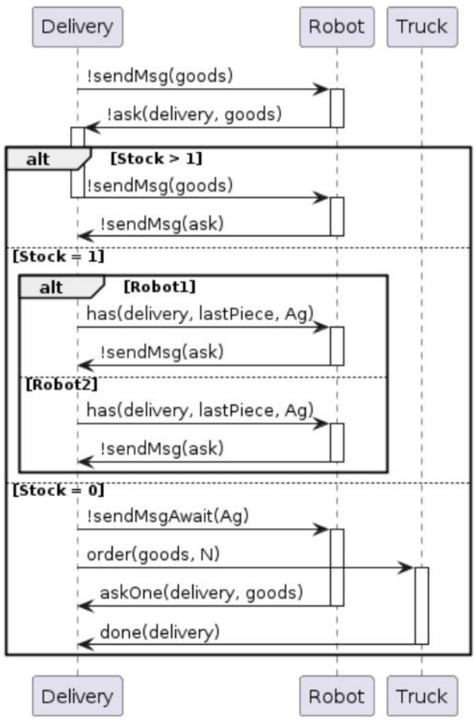

Figure 3: Agent interaction sequence diagram

Each line in the diagram represents a message sent from one agent to another. The arrows indicate the flow of communication between the agents. The diagram depicts three distinct alternative cases that may occur during the message exchange:

1. If the goods stock has a cardinality greater than 1, the Delivery sends objectives to both robots. After receiving the objectives, each robot executes the planned action sequence to

complete the objective. Once the objective is completed, the robot sends a new message to the Delivery to request a new objective to pursue via "!sendMsg(ask)".

2. If there is only one remaining item to be delivered, the Delivery sends an objective to only one of the two robots. The choice of the robot falls upon the one that requested a new objective at that specific moment when the stock reached a single unit. After receiving the objective, like in the previous case, the robot executes the necessary actions to reach it, then contacts the Delivery again to inquire about what to do next.

3. If the stock is zero, the Delivery responds with a message to the robot that requested the objective, informing it to enter the waiting state. When the robot performs the actions necessary to enter the waiting state, it sends a new message to inquire if there are any new objectives. At the moment when the Delivery perceives from the environment that there are no more items to unload, it sends a message to the Truck to order new goods. The truck will carry out the necessary actions to complete the delivery, then send a message to the Delivery to confirm the successful delivery of the goods.

The Delivery interacts with the robots through various types of messages to assign them specific objectives during the goods delivery process. The message sendMsg(goods) is used by the Delivery to assign the general objective of going to pick up the goods to the robot agents. This message is employed when the goods stock is greater than one, and both robots can be involved in transporting the goods. The message has(delivery, lastPiece, Ag) is instead used by the Delivery to send the objective of picking up the last available item to only one robot. This is done when there is only one item left to be picked up, requiring a single robot to complete the retrieval. Lastly, the message sendMsgAwait(Ag) is used to tell the robot that has just requested a new objective to go into the waiting state. This prompts the robot to reach its waiting position and await further instructions from the Delivery before proceeding with new objectives.

In addition to the messages sent from the Delivery to the robots to assign them specific objectives, it is also important to consider the requests sent from the robots to the Delivery to obtain new objectives or to confirm their arrival in a waiting area.

The robots can send a request message ask(delivery, goods) to the Delivery to ask for new objectives to pursue. Whenever the Delivery receives this request, it evaluates the current stock situation and responds with one of the previously described messages, based on the current conditions. The message askOne(delivery, goods) is used by the robots to request a new objective after reaching the waiting area. This message was introduced to properly handle cases where the robots received new objectives while they were heading to the waiting area, which could lead to errors in some specific scenarios. If no response is received to this message from the Delivery, the robots understand that there are currently no new objectives available and wait for further instructions accordingly.

The message order(goods, N) is sent from the Delivery to the Truck agent to request the delivery of a new stock of items to the warehouse. The requested quantity is randomly generated and represents the number of items that need to be added to the warehouse.

Once the Truck has completed the delivery and brought the items to the warehouse, it sends a response message to the Delivery with the message done(delivery). This message informs the Delivery that the merchandise delivery has been successfully completed. Through this exchange of messages, the Delivery can keep track of the delivery status and efficiently manage the flow of goods in the distribution system.

5.4.2 Agent behavior

The Delivery agent constantly monitors the stock of items and sends messages to the robots to assign them new objectives based on the current situation. These messages are sent in response to the robots' requests for new objectives. When the number of items to be stocked reaches zero, the Delivery agent places a new order to receive new items in the warehouse. This is handled through the orderGoods plan. As long as the overall stock is below 100 units, the Delivery agent generates random orders for a certain number of items using the orderGoods plan. Once the order is generated, the Delivery agent initiates the delivery process by communicating with the Truck agent through a synchronous message. When the total number of items in the warehouse reaches or exceeds the threshold of 100 units, the Delivery agent realizes that the maximum limit has been reached, and further orders are not necessary. This situation serves to determine the end of the simulation as it halts the generation of new objectives. The Delivery agent uses facts to recognize and manage situations where the merchandise is no longer available or when it is waiting to receive a new order. In summary, the Delivery agent manages the distribution of goods by coordinating order and delivery operations through the exchange of synchronous messages with other agents.

5.5 Truck agent

The Truck agent receives orders from the Delivery agent and the plan used to handle orders called "order" is activated to manage the requests.

When the Truck agent receives an order for a certain quantity, it commits to executing the delivery of the products in the requested quantity. This is accomplished through the execution of a specific action called deliver, which symbolizes the physical process of transporting and delivering the cargo. This action is defined within the project's model and will be illustrated in subsequent points.

Once the delivery is completed, the Truck agent sends a synchronous message to the Delivery agent to inform it that the task has been successfully executed. This message, represented by done(delivery), indicates to the Delivery agent that the delivery has been completed and that it can proceed with further stock management operations or assign new objectives to the robots, if necessary.

In summary, the Truck agent is responsible for the actual delivery of the requested goods, ensuring that orders are fulfilled promptly and efficiently, and then communicating the completion of the delivery to the Delivery agent for further actions.

5.6 Environment

The WarehouseEnv environment is responsible for managing the interaction of agents within the simulated environment. This environment provides an interface for agents to interact with it, both to receive perceptions and to execute actions. Each agent interfaces with the actions of the environment and updates its perceptions of it in response based on the actions performed.

When an agent performs an action within one of its plans, it interacts with the warehouse environment. This process occurs in several steps:

- Execution of the action: the agent decides to perform a specific action within its plan. This action can be a movement, an interaction with an object in the environment, or any other action defined in the model.

- Invocation of the corresponding method in the model: once the agent has decided on the action to perform, the environment invokes the corresponding method in the environment model. This method implements the specific logic of the action and modifies the environment's state accordingly. For example, if the action is "movement," the method in the model calculates the new position of the agent and update the new location in the model accordingly.

- Update of agent perceptions: after performing the action, the environment updates the agents' perceptions of the current state of the environment. Perceptions include information such as the agent's position, the state of objects in the environment, and other relevant information. This update allows agents to make informed decisions and adapt their behaviors based on the environmental conditions.

- Update of the view: after executing the action and updating perceptions, the graphical view of the environment is also updated. This allows visually monitoring the agents' interactions with the environment and better understanding the current state of the simulation.

The sequence diagram describes the flow of interaction between the agents, the environment, the model, and the view within the system architecture.

1. The agent performs an action by invoking it through the environment.
2. The environment calls the method defined in the model to perform operations in response to the agent's action.
3. Following the execution of the action the state of the model can be changed.
4. If updating the view is necessary:
   - The model method requests the update of the GUI.
   - The view updates itself providing an updated view.
5. The environment reads the state from the model.
6. The environment updates the agents' perceptions based on the current state of the model.

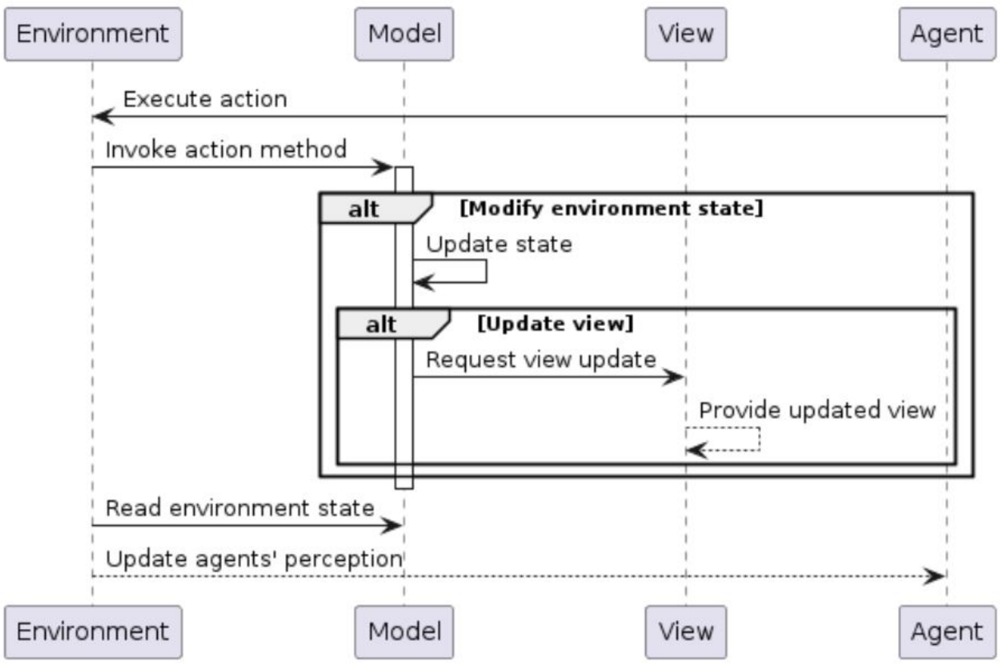

Figure 4: Interaction Flow between Agents, Environment, Model, and View

The actions managed by the environment include:

1. access(rack): simulate the access to the rack to position the items.
2. free(rack): simulate the frees the rack after placing the item.
3. move_towards: moves the robots to the various destinations.
4. place: places the items in a specific rack.
5. unload_goods: used to pick up goods from the delivery point.

6. deliver: used by the Truck agent to deliver the goods.
7. charge: used by robots to recharge their batteries.
8. catchAgent: report that a robot has reached another robot when it is broken.
9. restoreVariables: restores the variables used by the view to recognize particular situations such as breakage and repair.
10. showBrokenAg: tells the view to represent the broken state of a robot.
11. show_recovery: tells the view to represent the recovery status of a robot.

These are the actions that are handled by the environment to manage the interactions of the agents with the model and to maintain the consistent state of the simulation.

In the environment, agents' perceptions are updated, meaning agents receive information about the surrounding environment that influences their decisions and actions. This is the list of perceptions that are updated

1. end(goods): this perception is used by the Delivery agent to detect the absence of items at the delivery point. When the number of items to be placed by the robots reaches zero, this perception is generated. At this point, the Delivery agent understands that it needs to wait before assigning new objectives to the robots and that it needs to forward a new order of items to the Truck agent. This perception allows the Delivery agent to manage merchandise logistics efficiently, ensuring there is always an adequate supply of items available in the warehouse.
2. broken(Agent): this perception serves to indicate to the robot agent that it has broken. The value assigned to the Agent variable corresponds to the reference to the name of the other robot identified in the code as rescuer, to which the help message will be sent.
3. reach_robot(Agent): this perception is used until it is negated to instruct the rescuer robot to move towards the malfunctioning robot to provide assistance. When the two robots are in an appropriate position, this perception becomes true, and the rescuer robot begins simulating the repair intervention. In this case, the Agent variable takes the name of the robot to be assisted, which is used to retrieve its position from the move_towards action.
4. fault_resolved(Rescuer): until this perception is negated, it instructs the malfunctioning agent to move randomly within the map. When the robot perceives that this condition is met, it stops moving randomly and waits for the repair intervention.
5. stock(goods, N): where N represents the quantity of items that need to be retrieved by the robots at the Delivery point. This perception is used by the Delivery agent to be aware of the number of items to be unloaded and to manage the agents accordingly.
6. at(Agent, Destination): this perception is used to make agents aware that they have reached their destination. The variable Agent in this case is populated with the name of the agent in question, while the Destination can be one of the racks, the delivery point, one of the waiting/recharge zones, or another robot when it is malfunctioning.

Perceptions in the environment are updated every time an action is successfully completed. This means that when an agent performs an action and the environment handles it correctly, perceptions regarding the state of the environment are updated accordingly. For example, if a robot successfully completes the action of moving towards a rack, the environment updates agents' perceptions to reflect the robot's new position.

Moreover, before each perception update, they are reset to ensure that only the most recent information is reflected in agents' perceptions. This process ensures that agents always have the most up-to-date information about the state of the environment, enabling them to make informed decisions and act accordingly.

Finally, it is important to emphasize that perceptions are essential for enabling agents to interact effectively with the environment. Thanks to updated perceptions, agents can gain an accurate understanding of the surrounding world and react accordingly to achieve their goals and adapt to changes in the environment.

The environment also provides two functions that play an important role in helping agents determine their next actions based on their position in the simulated environment.

The getLocation() function is used to obtain the desired position for an agent's movement action. The function receives as input the object that represents the destination of the agent in question, and returns its position within the environment. For example, if the agent wants to reach the rack or the delivery point, this function will return the corresponding position of one of these entities. If the agent needs to move to the waiting area, this function will retrieve the position of this point based on the agent type.

The getRackLocation() function is specifically designed to obtain the position of the desired rack. It takes the rack number as input which can be 1, 2 or 3 and returns the corresponding position in the environment model.

These functions are crucial for the proper functioning of agents within the simulated environment. They provide agents with the necessary information about the position of environmental elements, enabling them to plan and execute their actions to achieve their goals.

The getLocation() method, in the case of a broken robot, generates random movement on the map, limiting it to half of the grid, which can be either the left or right part, depending on where the waiting zone positions of the robots are located, so that the broken robot moves only in a specific part of the environment. This approach of limiting random movement aims to keep the simulation more organized, avoiding uncontrollable movement of the robot across the entire map area, which could make it difficult to understand and monitor the environment activities. Additionally, reducing the randomness in the random movement of the robot to a portion of the grid also decreases the randomness of events in the simulation. This can make the simulation more realistic and predictable, allowing better control over the behavior of agents within the environment. Overall, this limitation of random movement of the broken robot helps maintain an organized, focused, and controllable simulation.

```java
int robot1X = this.model.rnd.nextInt(WarehouseModel.GSize - 3);
int robot0X = this.model.rnd.nextInt(WarehouseModel.GSize - 3) + 3;
dest = ag.equals(Utilities.AgentType.ROBOT1.getValue())
        ? new Location(robot1X, this.model.rnd.nextInt(WarehouseModel.GSize - 2) + 1)
        : new Location(robot0X, this.model.rnd.nextInt(WarehouseModel.GSize - 2) + 1);
```

5.7 Model

5.7.1 Attributes

The attributes that are part of the model are the following:

- RACK_1, RACK_2, RACK_3, DELIVERY_POINT: these constants represent the unique identifiers of these objects in the simulated environment.
- GSize: this constant defines the size of the grid representing the environment. The grid is 7x7 in size.
- goodsNum: the total number of goods available in the environment. This attribute keeps track of the availability of goods and is used to determine when a new order of goods needs to be placed.

- placed: it's an array that keeps track of the number of items placed on each rack. It's used to monitor the number of items present on each rack.
- chargeLevel: it's a map that keeps track of the charge level of the robots. This attribute is used to ensure a proper representation of the battery level of the robots by the view while they are recharging.
- reached: a boolean flag that indicates whether a robot has reached its destination, it's used only in the case where the destination is a broken robot that requires assistance. It's used to coordinate the movement of the robots.
- catching: another boolean flag that indicates whether a robot is trying to reach and rescue another broken robot. It's used to coordinate the rescue operations between the robots.
- brokenAg: it's an enumerated value representing the type of agent that is currently broken. It's used to identify the broken robot.
- showBrokenAg: an enumeration representing the type of agent that should be displayed as broken in the user interface. It's used to instruct the view to accurately represent the broken status of a robot.
- rnd: an instance of the Random class used to generate random numbers during the simulation. It's used to simulate random behaviors in the environment.
- lRack1, lRack2, lRack3, lDelivery, lWaitingZone, lWaitingZone1: The positions of objects in the environment, such as racks, the delivery point, and waiting zones. These attributes define the initial positions of objects in the simulated environment.
- DELAY: represents the delay in milliseconds before the TimerTask is first executed. This is used to set an initial delay before starting the periodic check for robot failures.
- PERIOD: indicates the time interval in milliseconds between two consecutive executions of the TimerTask. Here, it's used to specify the frequency at which the periodic check for robot failures is performed.

5.7.2 Methods

The explanation of the methods present in the model and their definitions follows:

- constructor():
  - Calls the constructor of the superclass Environment, passing as arguments the dimensions of the grid of the warehouse and the number of agents.
  - Place robots in the start zone of the warehouse using the setAgPos() method, which assigns a position to the agents within the grid.
  - Place the racks and the delivery point in the predefined positions of the warehouse using the add() method, which adds environmental objects to the grid.
  - Initialize the robot charge levels to -1. This is a default value used when the robot charge level is not being considered.
  - Create a Timer object and schedule a periodic task that calls the robotBreakdown() method to periodically check if a robot has broken down. The constant DELAY specifies the initial delay before starting the task, while the constant PERIOD specifies the time interval between successive executions of the task.

- accessRack() and freeRack() methods: the two methods respectively simulate accessing and freeing a rack in the warehouse. However, both methods simply print a message to the console and return true because the navigation system prevents robots from accessing a rack whose position is already occupied, therefore it is not necessary to carry out further operations.
- moveTowards() method: is responsible for moving one robot towards a specific destination position in the warehouse. It performs the following operations:
    1. Gets the agent's current location.
    2. Calculates the direction towards which the agent should move using the moveTowardsDestination() method.
    3. Checks if the position it wants to move to is occupied, and if it corresponds to one of the critical points such as a rack or the delivery point, the agent does not attempt to move and returns true. This is to prevent the robot from trying to go around an obstacle in these situations.
    4. If the current location is unoccupied, the agent attempts to move towards the destination. If it succeeds, it returns true.
    5. If the current position is occupied and does not correspond to a critical point such as the rack or the delivery point, the agent tries to avoid the obstacle. This is done by pseudo-randomly choosing the next position to move to using the moveAroundTheObstacle() method. If the newly calculated position is not occupied, the agent moves to it. However, if it is occupied, the agent will not change its position, and the process will need to be repeated.
- moveIfNotOccupied(): this method checks if the new position of the agent is not occupied by another agent. If the position is free, it updates the agent's position with the new one and updates the visualization of view to reflect the new state. It returns true if the agent successfully moved, otherwise, it returns false if the desired position is occupied. The method's synchronization ensures exclusive access to grid cells during the movement operation, preventing two robots from moving simultaneously to a new cell that turns out to be the same, thus avoiding overlap.
- moveTowardsDestination(): it calculates the direction in which an agent must move to reach a given destination within the warehouse. It takes as input the current position of the agent and the destination position. It examines the x and y coordinates of both the current and destination positions and determines whether the agent should move left, right, up, or down. It increments or decrements the agent's coordinates at every invocation until it reaches the destination.
- isOccupied(): it checks if a specific position in the grid is occupied by another agent different from the one specified by agentId. If the position is occupied by another agent, it returns true; otherwise, it returns false. Additionally, if the agent specified by agentId is attempting to reach a broken agent (indicated by the catching variable), it sets the reached flag to true to indicate that the agent has reached the broken agent. This last operation is used to signal to the two agents with a perception that they can stop their movement in order to perform the repair operation.
- moveAroundTheObstacle(): This method simulates an agent's attempt to circumvent an obstacle, which can be another robot. Based on the agent's current position (r1) and the destination position (dest), the method calculates a new position that allows the agent to move towards the destination without crossing the obstacle. The method handles three main cases based on the destination's position relative to the agent's current position: if the destination is above, below, or at the same height as the agent. In each case, new

values for x and y are determined so that the agent moves in the appropriate direction. The randomDirection variable is used to randomly determine whether the agent should move left, right, up or down when encountering an obstacle. Finally, the method validates the new coordinates to ensure they fall within the grid space defined and returns the new position of the agent after circumventing the obstacle. When the robot randomly chooses the direction to circumvent the obstacle, it always considers its final destination. This means that, while randomly selecting the direction, the robot will choose a movement that brings it closer to the desired destination, ensuring that the movement remains consistent with the ultimate goal. In this way, the robot tries to maintain a path as efficient as possible towards its destination, avoiding movements in directions opposite to the final destination.

- validateCoords(): it ensures that the coordinates do not go beyond the predefined area, represented by the grid of the warehouse. If the newly calculated coordinates turn out to be outside the grid's limits, then the current coordinate is kept; otherwise, the newly calculated coordinate is returned. This way, the method ensures that the robot always remains within the defined space of the warehouse.
- placeGoods(): this method is called when a robot places a good on a rack. It increments the counter for the type of rack specified by the index. It also updates the display of the rack corresponding to the type of good placed. Finally, it returns true to indicate that the operation has been successfully completed.
- deliverGoods(): the method is called when the Truck agent delivers some quantity of goods to the delivery point. It increments the total number of goods available at the delivery point. It updates the view in the delivery point position, and finally, it returns true to indicate that the operation has been successfully completed.
- unloadGoods(): this is called when the robot unloads an item at the delivery point. If the total number of goods at the delivery point is greater than zero, the method decreases the number of available goods by one. It also triggers the update of the view for the delivery point to reflect the decrease in goods.
- chargeRobot(): this method handles the robot recharging process. It takes as parameters the type of robot and its charge level. It obtains the name of the relevant agent and then retrieves its current position. Subsequently, it updates the charge level of the corresponding robot in the model using the provided level. It also updates the view at the robot's position to reflect the new charge level. This method is called repeatedly during the robot charging process. It ends by returning true to indicate that the operation has been completed.
- catchAgent(), restoreVariables() and showBrokenAg(): these methods are primarily used to update variables. The variable "catching" is used to indicate the end of the catching operation, while "restoreVariables()" restores a set of variables once the repair operation is completed. Most of these variables are used by the view to distinguish between different phases and provide appropriate representation. The "showBrokenAgent" variable is also used by the view for representational purposes.
- showRecoveryAg(): this method is used to update the view when an agent is performing the repair operation on another robot. It uses the agent's ID to retrieve its current position and then updates the view to reflect this state.
- updateView(): actually updates the view.
- robotBreakdown(): is called periodically by a timer to simulate the random breakdown of a robot. It checks if there are no robots already broken. If there are none, it randomly chooses one of the two robots and marks it as broken by setting the variable brokenAg

with the type of the selected robot. This simulates the random and unexpected breakdown of a robot during the simulation.

5.8 View

The View represents the visual part of the application, responsible for the graphical representation of the model. Its attributes include a series of colors used to distinguish different conditions and states in the graphical interface. Additionally, it contains an instance of the warehouse model that provides the data necessary for accurate visualization.

The methods defined in the graphical interface follows:

- constructor(): it takes an instance of the warehouse model as an argument, sets the window title to "Warehouse," and the width to 700 pixels. It stores the reference to the model (wModel). It defines a default font type for the text displayed in the view. It also makes the view visible.
- draw(): the method is used to draw objects on the grid of the view. It takes arguments including a graphics object used for drawing, the coordinates of a grid cell, and the identifier of the object to draw. Based on the object identifier, the method draws an agent, a rack, or a delivery point. If a robot is positioned on a rack or the delivery point, the object is colored yellow to indicate that it is occupied by a robot. Additionally, the number of goods on the rack and the number of goods at the delivery point are displayed next to the drawn object. The method also draws descriptive text above each object, indicating the type of object and, if applicable, the number of goods present.
- drawAgent(): the method is used to draw an agent on the view. It takes arguments including a graphics object for drawing, coordinates of a grid cell, the color of the agent, and the ID of the agent. The method checks if the agent's position is different from the position of the delivery point and the racks, and if so, it draws the agent with a specific color. Additionally, text is drawn above the agent indicating whether the robot agent is in recovery phase or if it is broken. If the agent is a robot and needs recharging, a graphical representation is drawn to indicate the agent's charge level, and text indicating the agent's charging status is displayed. Within this method, it checks whether the current agent is faulty or needs recharging. If the agent is broken, identified by comparing it with the stored broken agent in the model, the drawBrokenAgent() method is called to draw the agent with a visually distinct appearance. Similarly, if the agent needs recharging (chargeLevel variable is not -1), the drawAgentInCharge() method is called to draw the agent with coloring representing the charge level. Finally, corresponding labels are drawn above the agents to indicate their status.
- drawAgentInCharge(): this method is used to draw a graphical representation of the agent with different coloring based on the specified charge level. The method selects the color based on the agent's charge level. Colors are predefined and assigned based on specific charge level ranges. Once the appropriate color is selected, the agent is drawn with that color on the grid.
- drawBrokenAgent(): it's used to draw a graphical representation of an agent that is broken. Within the method, the color RED_ALERT is defined to represent this type of state. Then, the agent is drawn with this color on the grid using the drawAgent() method.

5.9 Utility class

The Utilities class was created to provide a set of utilities and constants used throughout the project. Inside, three types of enumerations are defined:

- AgentsIds: this enum defines the IDs of the agents, such as ROBOT and ROBOT1. It provides methods to obtain the ID of an agent type, check if two agents have the same type, and verify if a given ID corresponds to ROBOT or ROBOT1.
- AgentType: this enum defines the types of agents, such as ROBOT, ROBOT1, DELIVERY, and TRUCK. Each agent type has an associated value which consists in the name of the agent.
- LocationType: this enum defines the types of positions on the grid, such as ROBOT, DELIVERY, and the various RACKs. Each position type has an associated value.

These enumerations provide a clear and organized structure for handling the different types of agents, their identifiers, and positions on the grid within the project. Additionally, they simplify the code through the use of constants and utility methods.

6 Testing and optimization process

During the development of the simulation, a series of comprehensive application tests were conducted to ensure the correct functioning of each implemented feature. Each functionality was individually tested, observing its behavior in isolation within the simulation whenever possible. Subsequently, once integrated with the rest of the system, further testing was carried out to evaluate the overall behavior of the simulation and verify the absence of regression situations.

During system testing, it was crucial to use the debugging features provided by the MAS platform. This allowed monitoring the current state of beliefs, plans, and actions of the agents at specific moments in the simulation. This support was crucial for identifying any anomalies in the behavior of the agents and for optimizing the application.

Some issues encountered during testing were difficult to identify and understand, as they occurred sporadically and it was not immediately apparent what caused them. In these cases, extensive observations and various simulation tests were necessary to identify, understand, and correct the issues. This process required patience and determination, but ultimately led to problem resolution and overall improvement of the simulation.

During the overall system testing and regression checks, there was often a need to increase the simulation speed. This was done by reducing the wait times caused by the sleep operation performed by agents after executing an action to very low values. The goal of this approach was to create a kind of "stress test" for the system.

Increasing the simulation speed put pressure on the system, exposing it to higher load conditions and a higher frequency of actions. This made it more likely for errors or issues to manifest that might not have surfaced during execution at normal speed.

Moreover, running the simulation at a higher speed accelerated the testing process and yielded results more quickly. This was particularly helpful during the development and optimization phases of the system, allowing for the prompt identification and resolution of any issues.

Ultimately, increasing the simulation speed during testing contributed to enhancing the robustness and reliability of the system, enabling the more efficient detection and correction of any defects.

7 Evaluation and analysis of the results

7.1 Assessment of the simulation in relation to the project objectives

An evaluation follows regarding the achievement of the project objectives:

- System development: the project achieved this goal by implementing a multi-agent system that includes robotic agents, cargos, and loading/unloading points. Interactions between agents and the warehouse's operational logic were defined.

- Definition of autonomous behaviors: in the context of the project, autonomous behaviors referred to the ability of all agents to make decisions and actions based on the context without requiring direct control from the user or an external authority. This means being able to operate autonomously within the warehouse environment, dynamically adapting to the conditions and situations that arise. This entails defining strategies and behavior rules that allow the agents to achieve assigned goals and interact with the surrounding environment effectively and efficiently. This also included communication among agents within the system. All types of agents in the system, such as those responsible for load management or the delivery process, needed to be able to operate autonomously and interact with each other to achieve predetermined goals. This involved defining strategies and behavior rules that allowed agents to coordinate their actions, exchange information, and collaborate to achieve common objectives. The concept of autonomous behaviors in the project encompasses the ability to act independently and includes communication and collaboration among the various agents to achieve the system's overall goals. This objective can also be considered achieved.

- Achieve a realistic navigation system: the goal of achieving a realistic navigation system has been fully accomplished and has proven effective during numerous tests. The system enables robots to reach their destinations efficiently, using a basic obstacle avoidance approach. The tests conducted did not highlight the presence of significant bugs, confirming the robustness and functionality of the implemented system. However, it is important to note that the current system has limitations. In particular, the simulation is not able to scale effectively when increasing the number of robot agents beyond the current limit of two. This is due to the risk of congestion that can occur with more than two agents, a situation that the current navigation system is unable to handle. Therefore, although the goal of creating a realistic navigation system has been achieved, it would have been desirable to develop a simulation capable of supporting a larger number of robot agents. This would have allowed for greater scalability and a simulation more faithful to real warehouse scenarios with a higher number of robots operating simultaneously.

- Managing goods: the goal of managing goods has been successfully achieved through the implementation of fundamental actions such as delivery, loading, and unloading of goods. These actions have been implemented relatively easily, contributing to the achievement of the set goal. However, it must be said that some components, such as goods diversification, have been realized in a rather unsophisticated manner. In particular, the random generation of numbers at the time of goods loading by robot agents to determine the type of goods and their destination rack represents a rather simplistic implementation. This approach could be considered rudimentary, and certainly, it could have been done better. Nevertheless, considering the scope and objectives of the project, the current implementation meets the basic requirements and allows the system to function effectively for the purposes of the simulation.

- Warehouse modeling: the goal of modeling the warehouse has been achieved by creating a model that includes loading and unloading zones, waiting areas, and charging points, in addition to the ability to dynamically move goods within the simulated environment. The warehouse model is effectively represented through the graphical interface, which provides a clear and intuitive visualization of the various components. In the displayed image, it is possible to identify the delivery point where goods are delivered and picked up by robots, which move towards one of the available racks to place the items. The robots are positioned in their respective waiting zones at the beginning of the simulation and move towards them to recharge their batteries when necessary.

Figure 5: Starting point of the simulation

The warehouse model offers a realistic representation of the working environment and its dynamics, allowing for an accurate simulation of the activities performed by the robot agents within the system.

- The goal of implementing a user interface was achieved through the development of a GUI for real-time visualization of the simulation, allowing the monitoring of robots and warehouse status. To create the GUI, the classes provided by the MAS platform were exploited, which significantly simplified the development process. The APIs made available offered a vast range of features and tools, allowing for quick and effective creation of the interface.

7.2 Analysis of the obtained results

For the analysis of the obtained results, the list of the various functionalities tested and verified during the simulation follows. This list includes:

1. Goods delivery by the Truck agent to the delivery point: the correctness of this functionality was verified by checking the increase in the counter of the items present in the delivery point and the subsequent message sent to the Delivery agent once the delivery was completed.
2. Goods retrieval by the robots:

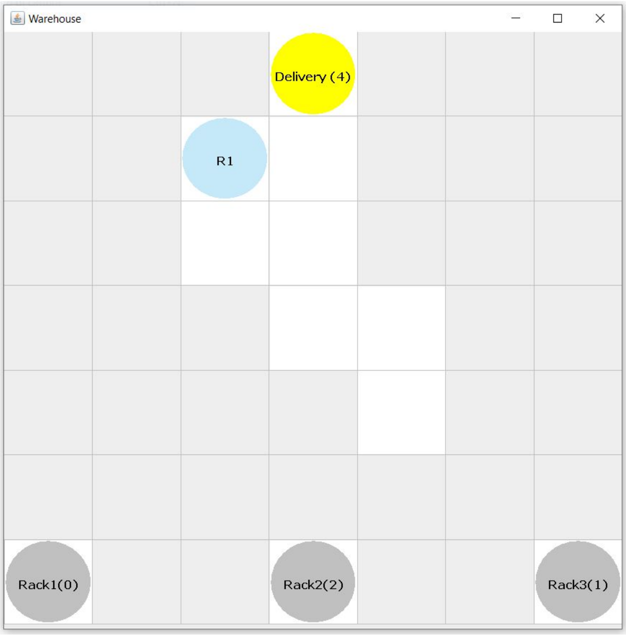

Figure 6: Retrieve of goods at the delivery point

This point highlights the process of goods retrieval by the robots at the delivery point. In the displayed image, we can observe a robot approaching the delivery point to retrieve an item. This operation is visually highlighted in the interface as the delivery point is illuminated in yellow, indicating that the robot is successfully carrying out its retrieval action.

3. Goods placement on racks:

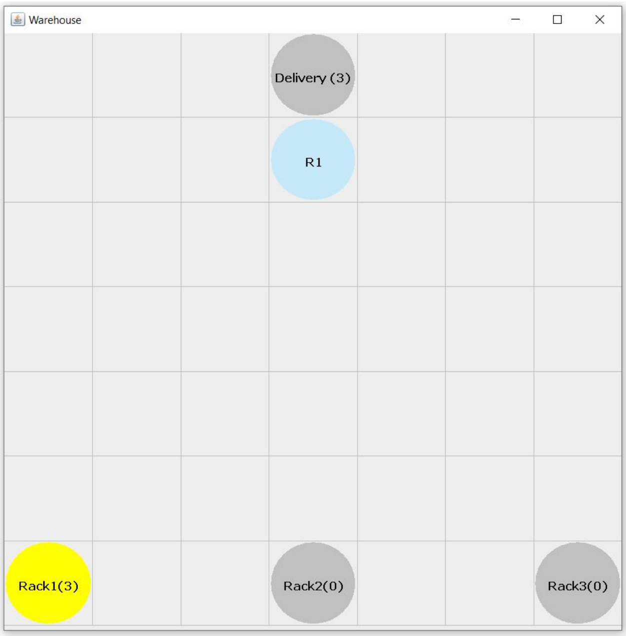

Figure 7: Robot accesses the rack to place an item

This point describes the process of placing goods on the racks by the robots. In the displayed image, a robot is approaching a rack to place an item. This action is visually highlighted in the GUI by the fact that the respective rack is illuminated in yellow, indicating that the robot is performing the process of placing the goods.

4. Utilization of all available racks. Through the GUI we can observe how all the racks are used for positioning the goods, thanks also to the counter of each of them which is increased with each item added.

5. Movement through the navigation system and obstacle avoidance. The proper functioning of the navigation system can be highlighted in various scenarios, including obstacle presence along the path in all possible directions and the presence of an obstacle within a critical point.

6. Battery discharge process and subsequent recharge process. The image shows robots while they are charging. It is clear that the robots are stationary in their charging positions, as they do not move and are focused on the battery charging operation. Thanks to the dedicated color, which goes from red to green, and the printing of the charge percentage for each of them, we have clear visual feedback on the status of the charging process. Different colors indicate the battery charge level, allowing you to quickly identify which robots are fully charged (green) and which may need additional charging (red or intermediate colors):

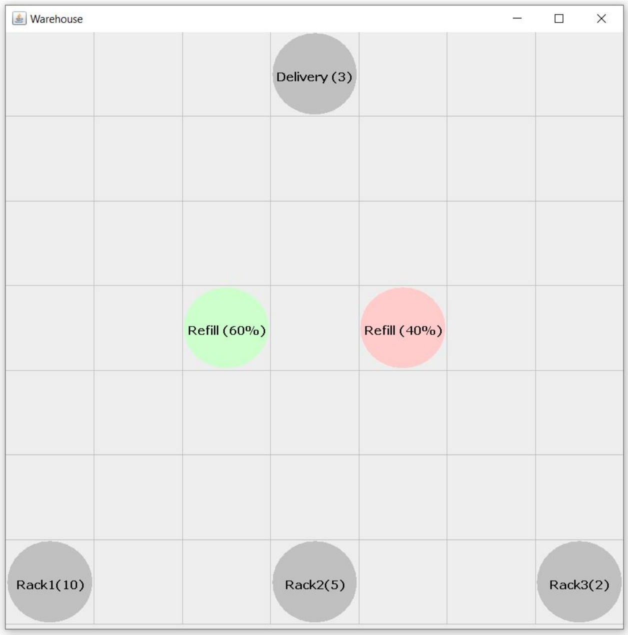

Figure 8: Robots during the charging process

7. Reaching a broken agent:

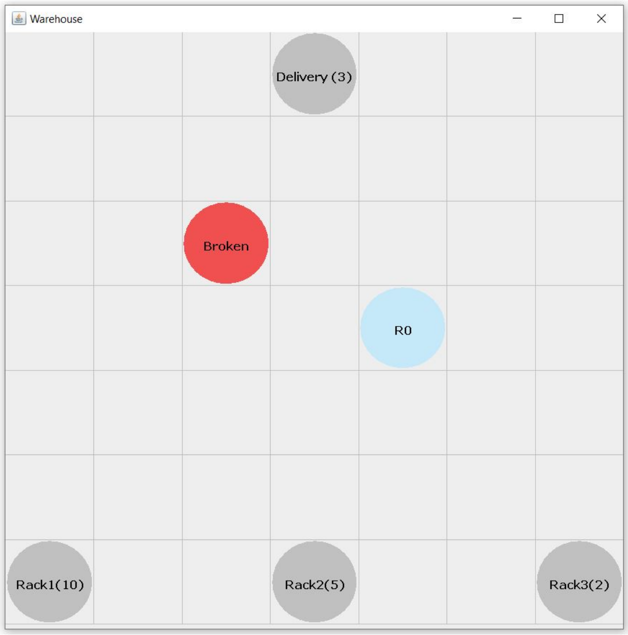

Figure 9: A robot heads towards the broken one

The image shows one robot that has broken while the other interrupts the flow of loading and unloading operations giving priority to the objective of repairing the broken robot. At the moment represented by this image, the non-faulty robot is heading towards the broken one to be able to repair it, both are moving, when the R0 robot reaches a position adjacent to that of the faulty robot, both will stop and the repair process can begin.

8. Breakage of an agent and typical behavior of the breakage. Refers to the movement performed to simulate breaking, which is a random movement within a predetermined grid area.

9. Repair process:

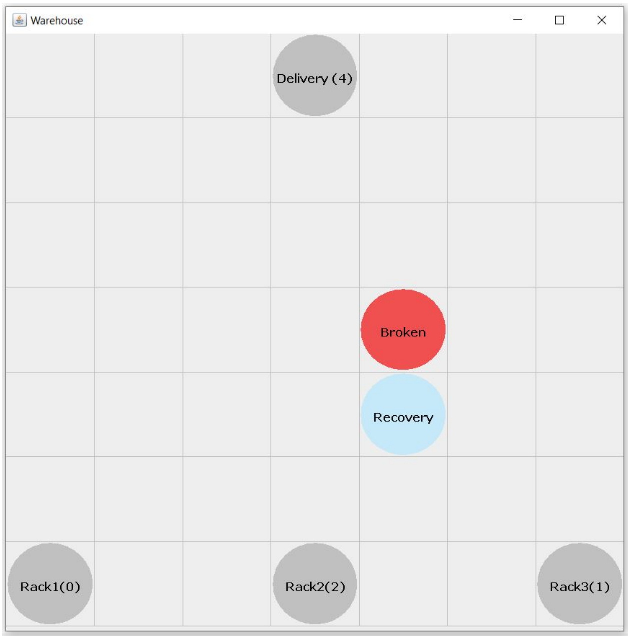

Figure 10: Process of repairing a broken agent

The image shows the process of repairing the broken robot. So one of the two robots has reached the faulty one and both have stopped, a string is printed on the robot carrying out the repair to have evidence during the duration of this procedure. When the operation ends both robots will resume their normal flow of activity.

10. Correct representations by the interface in various scenarios. In all test cases, correct behavior of the representation with the GUI could be observed.

11. Absence of conflicts between different situations that may occur. In this regard, it was verified that there are no conflicts between the different possible situations during the simulation. For example, if a robot is in a low battery situation but needs to rescue another broken robot, then rescuing the broken robot will take priority. Only after successful recovery will the drained robot begin the battery charging process. All such situations have been recognized and tested to ensure they are handled correctly in all possible scenarios.

12. Continuity in the flow of operations that is not interrupted during the simulation. For this point it was necessary to ensure that the flow of operations continues without interruptions throughout the entire simulation. The main objective is to ensure that the operational flow ends only as expected and scheduled. In other words, the agents must continue to collaborate, exchanging information and objectives to achieve, until the planned end of the simulation. This aspect has been carefully tested to ensure that the simulation proceeds smoothly until the intended operations are completed.

13. Operations end correctly when a certain number of correctly positioned goods are reached.

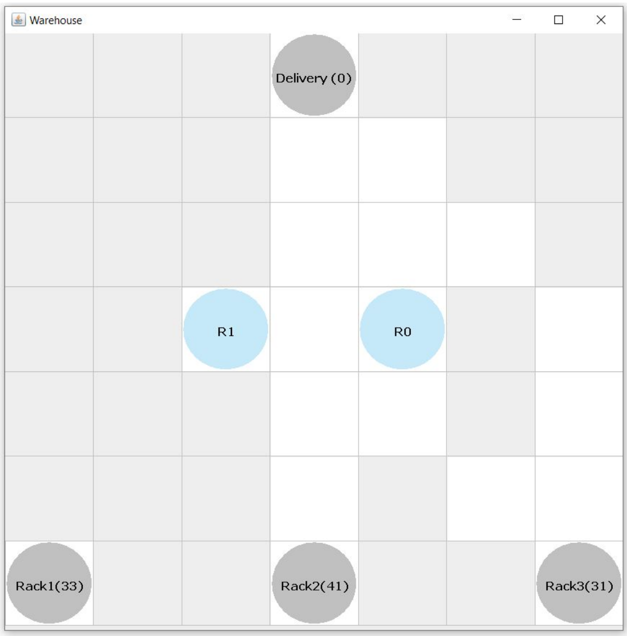

Figure 11: End of the simulation

When the expected limit of items to deliver and place was reached, the simulation ended correctly. In the displayed image, it can be observed that the maximum number of items to place has been reached, and consequently, the simulation was terminated. All items have been correctly placed in the racks, and the agents have returned to their waiting positions. Additionally, the delivery point no longer has any items to pick up, and the sum of the number of items in all racks has passed the expected threshold of 100 units.

When examining the simulation results, several significant observations emerge.

First, it is found that the simulated system was effective in achieving the set objectives. It verifies that key functionalities, such as goods delivery and retrieval, rack management and autonomous robot movement, have been successfully implemented. The numerous application tests conducted confirmed the correct functioning of these features, without the appearance of bugs or critical errors.

However, it is important to note that the system has some limitations, especially when it comes to managing a large number of robot agents within the simulation. In particular, it has been found that congestion situations can occur with more than two robot agents, leading to inefficiencies in the current navigation system. This suggests the need for further improvements to make the system more scalable and suitable for simulations with greater complexity.

Another important observation is that despite the satisfactory implementation of basic functionality, there are still areas where the system could be improved. For example, the approach to managing merchandise diversification could be made more sophisticated, in order to improve the realistic representation of product management within the warehouse.

Looking to the future, further developments and improvements could be considered to make the system even more robust and adaptable to real-world contexts. This could include the implementation of more complex algorithms for robot navigation, more sophisticated handling of goods, and greater flexibility in the interaction between agents within the system.

In conclusion, despite the identified challenges and limitations, it is believed that the simulated system has proven to be a valid tool for exploring and understanding the basic mechanisms of the MAS platform and the Jason language.

8 Conclusions

The development of this project provided the opportunity to learn and work with new technology for developing multi-agent systems. The learning process was stimulating and allowed us to acquire valuable skills in simulation and artificial intelligence. Although some of the implemented solutions were kept simple, however, we are satisfied that they demonstrated effective operation during testing. Regarding the achievement of the set objectives, we are satisfied with the results obtained. The considerations made previously in the point relating to considerations regarding the achievement of the objectives reflect one's judgment on the effectiveness of the developed system and its potential. Learning to use this new technology took time and dedication, but we are happy to have understood the fundamentals and to have developed practical skills in its use. However, it would have been exciting to be able to add additional features to the simulation, such as a navigation system capable of handling a larger number of robot agents and cargo drop-off points. This additional potential would have made the system even more complex and interesting.