Universitat Autònoma de Barcelona

Execution-driven dynamic Multi-Robot Task Allocation model easily applicable to real cases

Daniel Rivas Alonso

Universitat Autònoma de Barcelona

Microelectronics and Electronic Systems Department

Execution-driven dynamic Multi-Robot Task Allocation model easily applicable to real cases

Ph.D. Dissertation in the Doctoral Program in

Electronic and Telecommunication Engineering by

Daniel Rivas Alonso

Thesis Advisor:
Prof. Lluís Ribas Xirgo
Universitat Autònoma de Barcelona (UAB)
Escola d'Enginyeria

December, 2022

To my family, this achievement is also yours.

“La vida es como un pasaje de una canción que marea,

Y aquel que no se la sabe, la tararea.”

- Frank Delgado

Abstract

Many modern warehouses and factories use autonomous mobile robots for their internal logistics operations. Due to the workload of these facilities, it is impractical to manually assign transport tasks to fleet robots. Frequent changes in planned orders and unexpected events during their execution, such as robot malfunctions or traffic jams, increase the complexity of the task assignment problem. Taking all this into account, exhaustive searches for solutions must be limited in computing time and their results become suboptimal. Practical solutions use less computationally intensive allocation mechanisms that may not provide optimal results but are well suited to dynamic environments such as those described. One of these mechanisms consists in auctioning the tasks between the robots so that those who make the best offer end up doing the assigned transport.

This work proposes a model of a multi-agent system to make allocations through auctions that improves the quality of the solutions of a system with simple auctions while incorporating repetition mechanisms.

Unlike other task assignment applications, which use estimates to determine robot availability, the one we developed uses a physical simulator, allowing the effect of traffic or other factors to be considered. To make this possible, in addition to the models of the deliberative part of the agents of the system, the models of the reactive part of the robots have also been created. These models are formally described with a specific type of state machine that, in addition to guaranteeing predictability and facilitating the verification and implementation of the system, makes it possible to express the deliberative behavior of the agents.

The proposed system includes a synchronization mechanism with the physical part so that it makes the assignments and controls the robots while evolving the set of machines until it requires time to pass, such as to let the robots move.

The experiments that have been done show reductions in the time and distance traveled by the robots when the auction parameters are adjusted, and repetitions are used. The results obtained suggest that the allocations are close to the optimal ones.

Since the task assignment system that has been developed controls the simulated robots during the execution of tasks, the integration with a plant with real robots would be straightforward.

Resumen

Muchos almacenes y fábricas modernas utilizan robots móviles autónomos para sus operaciones logísticas internas. Debido al volumen de trabajo que tienen estas instalaciones, no es práctico asignar manualmente las tareas de transporte a los robots de la flota. Los cambios frecuentes en las órdenes planificadas y los eventos inesperados durante su ejecución como, por ejemplo, el mal funcionamiento del robot o los atascos de tráfico, aumentan la complejidad del problema de asignación de tareas. Teniendo todo esto en cuenta, las búsquedas exhaustivas de soluciones deben limitarse en tiempo de cálculo y sus resultados acaban siendo subóptimos. Las soluciones prácticas utilizan mecanismos de asignación menos intensivos en cálculo que quizás no proporcionan resultados óptimos pero que se adaptan muy bien a los entornos dinámicos como los que se han descrito. Uno de estos mecanismos consiste en subastar las tareas entre los robots de forma que los que hagan la mejor oferta acaben realizando el transporte asignado.

En este trabajo se propone un modelo de un sistema multi-agente para realizar las asignaciones mediante subastas que mejora la calidad de las soluciones de un sistema con subastas simples incorporando mecanismos de repetición.

A diferencia de otras aplicaciones de asignación de tareas, que utilizan estimaciones para determinar la disponibilidad de los robots, la que se ha desarrollado utiliza un simulador físico, lo que permite tener en cuenta el efecto del tráfico o de otros fenómenos. Para ello, además de los modelos de la parte deliberativa de los agentes del sistema, también se han creado los modelos de la parte reactiva de los robots. Estos modelos se describen formalmente con un tipo propio de máquina de estado que, además de garantizar la predictibilidad y facilitar la verificación e implementación del sistema, permite llegar a expresar el comportamiento deliberativo de los agentes.

El sistema que se propone incluye un mecanismo de sincronización con la parte física por lo que hace las asignaciones y controla los robots haciendo evolucionar el conjunto de máquinas hasta que requiere que pase tiempo, como, por ejemplo, para dejar que los robots se muevan.

Los experimentos que se han realizado muestran reducciones en el tiempo y la distancia recorrida por los robots cuando se ajustan los parámetros de la subasta y se utilizan repeticiones. Los resultados obtenidos sugieren que las asignaciones son próximas a las óptimas.

Dado que el sistema de asignación de tareas que se ha desarrollado controla a los robots simulados durante la ejecución de las tareas, la integración con una planta con robots reales sería directa.

Resum

Molts magatzems i fàbriques modernes utilitzen robots mòbils autònoms per a les seves operacions logístiques internes. A causa del volum de treball que tenen aquestes instal·lacions, no és pràctic assignar manualment les tasques de transport als robots de la flota. Els canvis freqüents en les ordres planificades i els esdeveniments inesperats durant la seva execució com, per exemple, el mal funcionament del robot o els embussos de trànsit, augmenten la complexitat del problema d'assignació de tasques. Tenint tot això en compte, les cerques exhaustives de solucions s'han de limitar en temps de càlcul i els seus resultats esdevenen subòptims. Les solucions pràctiques utilitzen mecanismes d'assignació menys intensius en càlcul que potser no proporcionen resultats òptims però que s'adapten molt bé als entorns dinàmics com els que s'han descrit. Un d'aquests mecanismes consisteix en subhastar les tasques entre els robots de manera que els que facin la millor oferta acabin fent el transport assignat.

En aquest treball es proposa un model d'un sistema multi-agent per fer les assignacions mitjançant subhastes que millora la qualitat de les solucions d'un sistema amb subhastes simples tot incorporant-hi mecanismes de repetició.

A diferència d'altres aplicacions d'assignació de tasques, que fan servir estimacions per determinar la disponibilitat dels robots, la que s'ha desenvolupat fa servir un simulador físic, cosa que permet tenir en compte l'efecte del trànsit o d'altres fenòmens. Per fer-ho possible, a més dels models de la part deliberativa dels agents del sistema, també s'han creat els models de la part reactiva dels robots. Aquests models es descriuen formalment amb un tipus propi de màquina d'estat que, a més de garantir la predictibilitat i de facilitar la verificació i la implementació del sistema, permet arribar a expressar el comportament deliberatiu dels agents.

El sistema que es proposa inclou un mecanisme de sincronització amb la part física de manera que fa les assignacions i controla els robots tot fent evolucionar el conjunt de màquines fins que requereix que passi temps, com, per exemple, per deixar que els robots es moguin.

Els experiments que s'han fet mostren reduccions en el temps i la distància recorreguda pels robots quan s'ajusten els paràmetres de la subhasta i s'hi fan servir repeticions. Els resultats obtinguts suggereixen que les assignacions són properes a les òptimes.

Atès que el sistema d'assignació de tasques que s'ha desenvolupat controla els robots simulats durant l'execució de les tasques, la integració amb una planta amb robots reals seria directa.

Acknowledgements

I want to thank many people who helped and supported me during the realization of this research. First and most important to my thesis advisor Lluís Ribas, who guided me with patience and dedication over all these years. Thank you to Quim Saiz and the personnel from the MiSE department, who advised and helped me from the start to the end of this project.

I am also grateful to my parents, siblings, and family, who inspired and encouraged me to overcome the difficulties during this period. I am especially thankful to Rosa Lliberia, who has patiently listened to me every day and helped me like no other. Without her support and understanding, the last months of this work would have been much harder for me.

Last but not least, I would like to thank my friends at the UAB, those with whom I share many thoughts, songs, and laughs. In particular to Eloi Guerrero, Lluis Acosta, Carlos Caballero, Ivan Pisa, Noemi Contreras, and Marti Raya. Also, a special thank you to Patricia Silveira, who told me of the opportunity of doing my Ph.D. research at the UAB and helped me even before my arrival in Barcelona.

A huge thank you to all of you and those I do not explicitly mention.

Daniel Rivas Alonso

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, many factories, warehouses, and other industrial facilities use automated guided vehicles (AGV) and autonomous mobile robots (AMR) to automate their internal transportation [Pol15, RX13b, RX13a, RX12]. These robots are not fixed to a line or rail and can modify their routes easily. This characteristic makes them highly desirable for internal transportation tasks due to the high variability of materials' input and demand for products in the current market scenario.

Minimizing operating costs while complying with high quality-of-service constraints makes multi-robot systems (MRS) more efficient [Lee20, Che19, Liu19]. These costs are related to the number of resources spent to complete the transportation tasks arriving at the system. The fewer resources spent, on average, to accomplish a task, the more efficient the transportation system. Calculating the minimal cost is complicated since it depends on the accuracy of the operations cost estimation [Das18], finding a global minimum on top of a non-optimal structure [TL16], and how to assign the incoming tasks to each agent.

Operations cost estimation is a base element for the improvement of the efficiency of these systems; for that reason, many studies focus on this topic. Most of these researches estimate the time consumption of a given transportation task using the distance to travel [Woo18]. However, these approaches do not consider any variations that may occur in the environment, such as new obstacles appearing or liquids spilling on the roads. In this regard, the work developed in [Das18] by members of the Software/Hardware Agent-based Distributed Embedded Systems (SHADES) research group registers and updates the time consumed when traveling the environment pathways and uses these data to calculate future travel costs.

There are many different ways to calculate the route to take [Ste19]. One of the most common is using an optimization algorithm, like Dijkstra [Cor90], to calculate the shortest path from one place to another. This type of solution could lead to new problems since many of the algorithms they use are greedy. In other words, they will find the shortest path to the destination, which would benefit the agent in question the most but would not consider the whole system situation, which could lead to jamming some sectors of the environment. In [TL16], members of the SHADES research group calculate the paths cooperatively, which may not be optimal for each robot individually but is optimal for the whole system.

A critical part of managing these systems is finding the best combination when assigning tasks to the agents [Whi19, Buc19]. When allocating tasks, users pursue a specific goal, like minimizing resource consumption, such as time [Lin23, Nun17]. A simple allocation strategy is to assign an incoming task to one of the agents with enough energy to fulfill it and from that group to the one that will spend the smallest amount of resources in the system. This last element is usually translated to the shortest time to job completion and the most efficient use of energy [Dai19, Kaw12]. This approach is fit for finding local optimums but may not offer the best solution for improving the overall system performance. Finding optimal solutions for the multi-robot task assignment problem requires some planning and foresight of the incoming orders and the availability and capacity of the robots. The higher the fleet size and the number of concurrent tasks, the higher the number of possible assignment combinations. These growths exponentially increase the complexity of finding the optimal solution, making it almost impossible for a single person to solve it in medium to large systems. So, many companies look to automate the TA or provide appropriate decision-support tools to their specialists. Another issue is that not all users require the same optimization criteria: some may want to reduce the average distance traveled per order, and others may prefer to minimize the maximum distance traveled by any robot, to name a few. Also, the system characteristics vary from one implementation to another, adding even more variability to the problem and eliminating the existence of a universal optimal solution.

Since previous research in our research group improved the cost estimation of the operations and path planning methods and algorithms, we will focus our studies on the tasks' management mechanism. The primary motivation for this research is that in recent years, many businesses have been increasing their fleet size due to the growth of their operations, making it quite difficult for a single person to handle the assignment and management of transportation tasks. This situation leads to needing to automate the Task Allocation (TA) process or having a decision support tool that allows the planner to evaluate the impact of making changes in the allocation criteria or paths used by the robots. Since many of these plants cannot stop their operations to test the performance of different configurations, it is helpful to have ways to simulate the work environment and assess the magnitude that such changes imply and the benefits they bring to the system.

Finding optimal assignment distributions is one of the main concerns when using multi-robot systems. Many studies focus on this problem, which they call the multi-robot task allocation (MRTA) problem. These studies use different approaches, like methods from diverse areas of knowledge for the assignment [Zha16, Kim12] or different control architectures [Sar18b, Tur18a, Tur18b, Mic08]. These problems have many characteristics and variations, leading to several studies in their taxonomy, like [Nun17, Kor13].

Even though the MRTA problem has been heavily studied, many problems emerge from the current solutions found in the literature. Some approaches focus on solving MRTA problems in static environments, where all the transportation orders are known beforehand, and only the participating robots use the operation area. In these situations, centralized MRTA solutions [Sar18b, Kim12], based on exploration algorithms or integer linear programming, can provide near-optimal allocations. However, they are hard to scale and have problems adapting to changes in the system. Furthermore, the quality of the results highly depends on the *a priori* information on the tasks, and their performance degrades when dealing with certain randomization levels in the system [Sch17]. Such limitations make them inappropriate for dynamic environments since the TA system would have to reevaluate the solution every time there is a change in the scenario, like the appearance of tasks not included in the original plan.

When dealing with dynamic environments, distributed solutions show better results than centralized ones. However, these solutions have also room for improvement, like using more accurate data from the physical system's conditions and more elaborated temporal models [Nun17], optimizing the agent's time and energy consumption, or including robust and intelligent controllers that support hybrid control architectures [Ism19]. The latter solutions would use *deliberative control* for inter-agent coordination and *reactive control* for handling the physical aspects of the environment.

Solving MRTA problems usually requires large amounts of computation time due to the complexity of the scenarios. Such complexity is related to the sheer number of possible assignment combinations, even for small numbers of robots and orders. This situation makes the increment in solution quality directly proportional to the computation time required to find it [Tur18b].

Other studies present limitations regarding the environmental conditions used. In [Tur18b], they manifest that the scenarios they used for their tests were not realistic enough. Another example appears in [Tur18a], where the author expresses his interest in developing task assignment algorithms based on the network topology in the future.

Finally, a general survey on the field [Ism19] proposes some guidelines for obtaining better results based on state-of-the-art studies. Regarding communications, it is interesting using a combination of implicit and explicit methods rather than relying solely upon one of them. The first one performs better in local environments, so it is more suited for navigation and crash avoidance between the robots. The latter allows the transmission of specific data and gives better results in global events, so it is more advantageous to use it for task planning and assignment.

Our research's objective is to develop a task management model capable of finding competitive allocation solutions for multiple MRTA problems. The solutions consist of distributing the transportation orders that arrive to a logistics system between the available mobile agents. To do so, our allocation model must take information regarding the environment, the tasks, and the robots and provide a solution based on their characteristics.

The allocation algorithm must be able to continuously adapt its solution to any changes that may appear in the scenario during its operations. This capability is necessary for it to operate in static and dynamic environments. The best way to achieve these characteristics is by designing our TA mechanism in a distributed control structure. This method eliminates single points of failure in the system, improving its robustness to errors. To that end, our assignment algorithm must be distributed among all agents participating in the allocation solution. Our model needs to quickly adapt its solution to any changes in the planned tasks or the mobile agents executing them, either due to additions or subtractions. So, the agents in our system must continuously deliberate to determine the best allocation combination in the changing environment. This continuous coordination should also increase the robustness of our solving mechanism when facing errors and obstructions of any kind. It is also vital that our model behaves both predictably and verifiably to eliminate the occurrence of erratic or inappropriate behavior. We plan to ensure both traits by formalizing the description of the controllers and system behavior.

We also want to provide the users of our model with ways to adjust its operations to fit the requirements and limitations of the different logistics scenarios they look to represent. Also, the system should include simulation capabilities to represent as many details as possible from the environment where the transportation tasks take place and their execution. This feature should allow the user to introduce changes or disturbances at various control levels and evaluate their impact on the overall performance of the logistics system. To increase the versatility of our tool, we look to design it modularly, with each module attending a different part of the control. This structural division should allow the generation of the allocation solution at various control levels enabling the use of our model in diverse modalities, depending on how the modules get used. The users might employ the whole model as a standalone decision support tool for obtaining data on the behavior and performance of the system under different conditions. They may use only some of the modules to complement other execution systems or use them as an automated manager for physical agents. Another feature we would like to include is some mechanism that improves the early detection and reaction to errors or changes in the allocation scene. Finally, the model must provide enough statistical data on the systems' performance to allow the users to compare different configurations and modalities. All these features should capacitate our model to provide solutions to a large variety of allocation problems and even improve the efficiency of transportation and delivery systems in internal logistics environments. That, in turn, should allow its users to test new route distributions, the effect of adding or eliminating mobile agents, and many other changes.

Our lead approach builds upon a physical simulator containing the details of the physical plant and the mobile agents, which includes a control layer for the MRTA at the top to achieve the previously mentioned goals. This control module receives tasks from, e.g., a manufacturing execution system, distributes them to the robots, and redistributes them as the plan (or the general situation) changes. As the scenario in the physical simulator includes the vehicles and the plant itself, the model accurately represents the time consumption of different types of operations along with the physical interactions in the system. Thanks to this representation, the TA mechanism uses the information gathered during the tasks' execution to elaborate and modify the planning. That way, the allocation results include the impact of unexpected events, making them more exact than most approaches in the literature which only use estimated data for the allocation. The physical simulation also allows the users to introduce changes or disturbances in the system at any level and study how they affect its behavior and performance. Even when experimenting on actual plants could provide reliable data, a simulator allows testing plant modifications and hazardous situations without disturbing the operations of real-world systems.

Our MRTA algorithm is built upon a Multi-Agent System (MAS) divided into layers. Each layer deals with a different level of complexity in the allocation solution: one for task allocation, one for route planning, one for route execution, one for traffic management, and one for executing simple movement commands. This segmentation makes it easier to obtain the system answers at different levels, allowing the communication of our framework with many robots or models, regardless of their complexity. Also, each layer can be used as an individual module as long as its communication protocols are respected. So, the users may use our framework as a standalone decision support tool or any of its independent modules as complements for their models or multi-robot management systems.

The control architecture of our model uses a decentralized structure where each task management is individual without knowledge about the rest of the other tasks. This architecture provides our system with adaptability and robustness when facing dynamic environments [Sch17]. The model contains two types of agents: task managers and executors. The managers represent the transportation tasks to perform, and the executors represent the mobile robots that perform the tasks. The task assignment occurs through a series of closed-envelope auctions, where the managers offer their transport orders for bidding, and the executors are the bidders [Riv19a]. Many approaches use similar task allocation mechanisms because they provide a distributed method to partition tasks, allowing various robots to have different costs or values over them [Ott19]. However, since the TA mechanism is independent of other components in the framework, it is easy to change it for any other.

However, the best-suited candidate for a TO may appear after the latter carried out its auction, adding complexity to finding the optimal solution. To deal with this common situation, especially in dynamic allocation problems, and increase the fault tolerance of our model, our MRTA algorithm uses a reassignment mechanism that dynamically updates the tasks' assignments. This method eliminates the requirement for knowing when and where each mobile agent will finish their current assignments and plan accordingly [See20]. For the reassignment, our system repeats the auctions for already assigned tasks, which allows it to account for recently available vehicles that might reach the client at a lesser cost.

To synchronize TA operations with lower-level controllers, its behavior, and those of the reactive controllers, is described by a specific type of state machine we call Extended Finite-State Stack Machines or EFS^{2}M (Section 2.6). This representation ensures a formal description of the model, prevents it from making decisions based on undefined or indeterministic cases, and provides the means to systematize its development and implementation.

The quality of the allocation solution from our system depends on several parameters, like time to place bids, winner selection criteria, auction repetition, etc. Tuning the values of these parameters changes the model's behavior without changing the environment. So, it allows the users to study how different work conditions in their locations impact the logistics system's performance.

For the implementation, we used a multi-robot scene in the CoppeliaSim [Roh13] simulator, with robots modeling real ones, which we have been using for educational purposes since 2014. In this model, each robot is an agent with several controllers distributed in layers: the lower levels have a version inside the real robot and another in the simulated counterpart; meanwhile, the upper ones are strictly virtual. The data from upper levels to lower ones go to the virtual agents and the real ones (if they are connected).

1.1 Thesis Outline

In the next chapter of this document, we define the MRTA problem (Section 2.1) and present a Mixed Integer Linear Programming (MILP) model (Section 2.3) that searches for the allocation solution with the minimum distance traveled by all mobile agents. This model provides optimal solutions regarding this metric for assessing the quality of the results from our multi-robot system. Section 2.4 illustrates how the agents coordinate in our distributed model for allocating the incoming tasks and how the system adapts its solution before changes. The rest of Chapter 2 contains a detailed description of the controllers that constitute our task allocation system, their representation, and the integration of the algorithm with a physics simulator.

Chapter 3 starts with a description of the scenarios used for testing the resolution capabilities of our models (section 3.1). Then, we proceed to describe the metrics used in many approaches, including our own, to measure the performance of a logistic system in section 3.2. This chapter concludes with the testing procedures to assess the performance of the MILP model, which will be used later to evaluate the quality of our primary approach's results.

We compared the results from our two models in a warehouse-like setup with variations regarding the number of agents and tasks (section 4.1.1). In Chapter 4, we also assess how different configurations in the TA mechanism and the insertion of disturbances at the physical level (communication failures and mobile obstacles) impacted the system's performance in our distributed system. The experiments show that the MRTA model can provide reliable allocation solutions in complex logistics environments, maintaining its solving capabilities in both static and dynamic environments (concluding chapter).

1.2 Research Contributions

The main contribution of our research is the formalization of an agent-based distributed MRTA algorithm with reallocation capabilities integrated with a physics simulator. The reallocation mechanism adapts its solution to any sudden changes in the planned tasks, errors in the agents' operations, or the appearance of disruptive elements in the environment. So, it improves the performance of auction-based allocation methods while maintaining their adaptability benefits. The model is designed as a Multi-Agent System with the behavior of all its controllers formally represented by a particular type of state machine. The allocation algorithm is integrated into a physics simulator to obtain accurate data from the execution of the tasks and adapt the plan to more realistic conditions. Since not all the controllers' operations consume the same amount of time, the use of the traditional execution engines inside physical simulators could cause some delay in operations that should happen instantaneously, deteriorating the performance of the system. For that reason, a new time model is introduced to separate the execution of instantaneous operations from those that consume a significant amount of time.

The resulting model includes a distributed TA, which takes into account data in real time and adapts its solution to any change in the planning or physical system, including robot failures or traffic jams. Thus, it overcomes the limitations of other methods from the literature and also from our previous approaches [Riv19a, Riv19b, Riv18]. The system uses time as a parameter, discerning how various operations incur different time consumptions in the simulations. With these characteristics, it is possible to adapt the system to use it as a digital twin for controlling and managing real-life transport systems [Cha16].

Chapter 2

Formalization and modeling of the task allocation problem

2.1 Problem formalization

Our research focuses on a logistics environment containing a set of mobile agents, an area for them to remain while not operating (e.g., charging their batteries), a storage area with many products, and a loading area where the products are processed. In this facility, a transport task entails carrying a product from a port in the loading area to another in the storage area and vice versa.

The problem addressed in our research involves a set of transport tasks carried out by a fleet of homogeneous mobile agents $E = \{e_1, e_2, \dots, e_m\}$ that travels the environment following a predefined road network. Each robot can only carry one product at a time. To do so, it must go to the starting port of the corresponding product, pick it up, take it to its destination port, and drop it there. The maximum number of tasks an agent can perform is defined as Q. We denote the set of n individual products $P = \{p_1, p_2, \dots, p_n\}$, their established starting ports $S_i(i \in \{1, 2, \dots, n\})$, and their corresponding destination ports $D_i(i \in \{1, 2, \dots, n\})$.

To transport a product, the mobile agent cannot carry any other product at the time and must not have reached the maximum number of tasks to perform ($q_k \leq Q$). So, any loaded robot must drop its items at its destination before picking any other. With these requirements, we must find an assignment of products to agents that minimizes the total traveled distance by all participating vehicles. The solution must establish which agents transport which products and the order in which they do it.

2.2 Problem modeling

We represent the previously described problem with a weighted graph $W = (V, A, C)$, where:

- V is a set of nodes denoting the tasks to perform, one for each product $P = \{p_1, p_2, \dots, p_n\}$. The

v_0 node represents the recharging operations, which is the start and finish of all mobile agents' operations. Each node $v_i(i \in \{1, 2, \ldots, n\})$ corresponds to the transportation task for product $p_i(i \in 1, 2, \ldots, n)$.

- A is the set of arcs $(i, j) \in V^2 : i \neq j$. An arc $(i, j) \in A$ indicates that a mobile agent can go from the destination of product p_i to the location of product p_j.

- C is the set of costs for executing one task after another. The value of c_{ij} represents the cost of using arc $(i, j) \in A$, which involves executing task j after i. Its value is the sum of two distances, one for traveling from the destination of task i to the start of task j and the other for going from there to the goal of task j.

The solution to the problem consists of finding a route for each robot. The routes are ordered sequences of different nodes that must follow some rules:

- The routes must start and finish at the v_0 node.

- The number of nodes of a route $U_k(k \leq m)$ must be lesser or equal to the maximum number of tasks an agent can carry out Q.

- The solution's objective is to minimize the sum of the costs of the routes of all agents. The cost of a route is the sum of the costs of all the arcs it contains.

During our research, we developed two models for solving the MRTA problem. The main one is a distributed approach based on a Multi-Agent-System, which makes our solution more robust to errors and adaptable to changes in the system. Our second system is a centralized approach whose main objective is to provide results to assess the solutions from the distributed one.

2.3 Centralized approach

We formulated this problem as a Mixed Integer Linear Programming (MILP) model to design a centralized solution. During this process, we used as starting points the examples included in the IBM ILOG CPLEX Optimization Studio and the one for solving the Capacitated Vehicle Routing Problem using the Python programming language [Cac20]. To start, our model needs information about the tasks to perform and the environment where they take place. We introduce how many ports the layout contains, the minimal distances between them, and indicate from which one of them the robots start their operations (charging station). The input data includes a list of all the tasks to carry out, along with their corresponding starting and destination ports. Based on that information, our MILP model generates the set of tasks' nodes (V), the set of arcs linking them (A), and the set of costs corresponding to those arcs (C).

The model has three decision variables for tuning up to achieve the optimal objective value. These variables are the maximum number of tasks a mobile agent can execute (Q), the tasks that belong to a route (u_i, with $i \in N$), and variable x_{ij} indicating if the arc $(i, j) \in A$ is part of the solution, which means that a robot performed task j after i.

The system objective is to minimize the solution cost (*solCost*), which corresponds to equation 2.1, and Q. With this design, the optimization algorithm finds combinations of the decision variables' values that reduce solutions costs and guarantee an even task distribution among the mobile agents. In equation 2.2, we combine both metrics in a single expression where their values share similar orders of magnitude. In there, the *fullCost* value is the sum of all the costs in *C*, and *n* is the number of tasks to perform. The *fullCost* would correspond to executing each task after each of the other tasks, meaning that its value is comparable to the cost incurred when performing n^2 tasks. So, dividing *fullCost* by n^2 returns a value similar to the average task cost, and if we multiply it by Q, the result would be of a similar order of magnitude to *solCost*.

$$solCost = \sum_{i,j \in A} c_{ij} x_{ij} \tag{2.1}$$

$$solCost + Q \times \frac{fullCost}{n^2} \tag{2.2}$$

The search, however, is subject to a set of constraints also defined in our program. The first constraint establishes that whenever a robot leaves a task node, it must go to another node (equation 2.3).

$$\sum_{j \in V, j \neq i} x_{ij} = 1 \quad \forall i \in N \tag{2.3}$$

The second constraint follows equation 2.4 and has a formulation similar to the first. It defines that a mobile agent can only reach a task node if it comes from another node.

$$\sum_{i \in V, i \neq j} x_{ij} = 1 \quad \forall j \in N \tag{2.4}$$

With the constraint in equation 2.5, the model sets that including any arc in an agent route implies that the number of tasks for that route increases by one.

$$\text{if } x_{ij} = 1 \Rightarrow u_i + 1 = u_j \quad \forall i, j \in A : j \neq 0, i \neq 0 \tag{2.5}$$

The number of tasks carried out by any agent must always be lesser or equal to the maximum number of tasks per robot. So, the fourth constraint takes the form of equation 2.6.

$$1 \leq u_i \leq Q \quad \forall i \in N \tag{2.6}$$

The constraint in equation 2.7 informs the search algorithm that the variable x_{ij} only takes binary values.

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in A \tag{2.7}$$

Finally, we included a constraint to prevent stalled traffic and frequent conflicts between the robots. To that end, we limit the number of mobile agents to use in a solution to avoid high levels of traffic saturation, which is the percentage of road area occupied by vehicles from the total road area [RX21]. In our case, we established the maximum traffic saturation percent (*TS*) at 30%. The model uses this value to calculate the maximum number of robots (M) using equation 2.8, where $area_{RN}$ is the total road network area in the facility and $area_{MA}$ is the area occupied by a mobile agent. The constraint in equation 2.9 limits the number of agents in the solution.

$$M = \frac{area_{RN}}{area_{MA}} \times TS \tag{2.8}$$

$$\sum_{j \in N} x_{0j} \leq M \tag{2.9}$$

The IBM ILOG CPLEX Optimization Studio provides, as the execution result, the objective variable value. Our model contains a section to extract and show the values of other variables and deliver a more detailed description of the allocation solution. Those variables are the number of robots used in the solution, the maximum number of tasks per agent, and the set of active arcs, which contains the sequence in which the tasks were attended.

2.4 Distributed approach

Our distributed solution is a multi-robot task allocation mechanism organized as a Multi-Agent System (MAS) divided into layers. Each layer attends to a different control level in the allocation solution: task allocation, route planning, route execution, traffic management, and executing simple movement commands. This segmentation allows the resolution of each aspect of the allocation problem independently from the rest. Each layer contains a set of mechanisms to solve any emerging problem, change, or disruption in their area of influence.

A decentralized control architecture provides adaptability and robustness to the system, which is crucial when facing dynamic environments [Dra20,Sch17] and allows the distribution of the computation system components. Many approaches have similar architectures to implement TA based on auctions [Ott19].

There are two types of agents in our model: *managers* to control the execution of the tasks and offer them for bidding, and *executors* to participate in those auctions as bidders [Riv19a, Riv19b]. The task allocation happens through a series of closed-envelope auctions, where the bidders do not know the bidding values of other participants. To increase the adaptability and robustness of the system, it includes a reallocation mechanism that allows an already assigned task to launch new auctions while waiting for its vehicle to look for better transportation options. With these procedures, our allocation system reacts faster when a robot encounters a problem (in the environment or internal) that causes it to stop being the fittest for its current task. It also allows the model to reallocate the tasks to closer mobile agents that were busy during the first auction. Thanks to this feature, our model does not need to keep a register of when and where any occupied agent will become available to perform the task allocation process.

2.4.1 Agents' interaction

The model operates similarly to a taxi company: the managers are passengers, and the executors are taxis. When a task enters the system for its execution, a passenger (manager) is created at the task's initial point, asking all available taxis to transport it to its destination. Each taxi (executor) replies to the call with the distance they would need to travel to reach the passenger and its destination (in that order). The passenger collects the replies and selects the one with the lowest cost as the winner. The winning taxi travels to the passenger's position, picks it up, carries it to its destination, and drops it there. Figure 2.1 shows the communications between three agents while completing a task. The manager starts by sending a *call-for-proposals* message (CFP), with its position (S_i) and destination (D_i), to all executors available and collects their answers for a while. Upon receiving a CFP, an available executor plans the shortest route to perform the task and sends the distance as the bidding value in a *proposal* message (PRP, *bid*). The manager evaluates all received proposals, sends an ACCEPT to the closest one, and waits for its arrival. This whole process receives the name of Primary Auction. When an executor receives an ACCEPT, it begins to travel to S_i, and when it arrives, it stops and sends a READY message to the manager. The manager loads the items inside the vehicle and sends an ON message to the executor. The latter travels to D_i and, upon arrival, sends a DONE message to the manager. This one, in turn, takes the items out of the vehicle and sends an OFF message to the executor, finishing the task.

When a task manager has an assigned executor and is waiting for its arrival, it may use the reallocation mechanism in our system. During this time, the manager may launch a Secondary Auction to check how close all the available mobile agents are. If it finds a vehicle closer than its current executor, it declares it as the winner and releases the previously allocated robot. However, if the current mobile agent is the fittest for the job, the manager does not send any notifications since it is already on its way. Figure 2.2 shows three agents communicating during a re-auction. A manager waiting for its corresponding

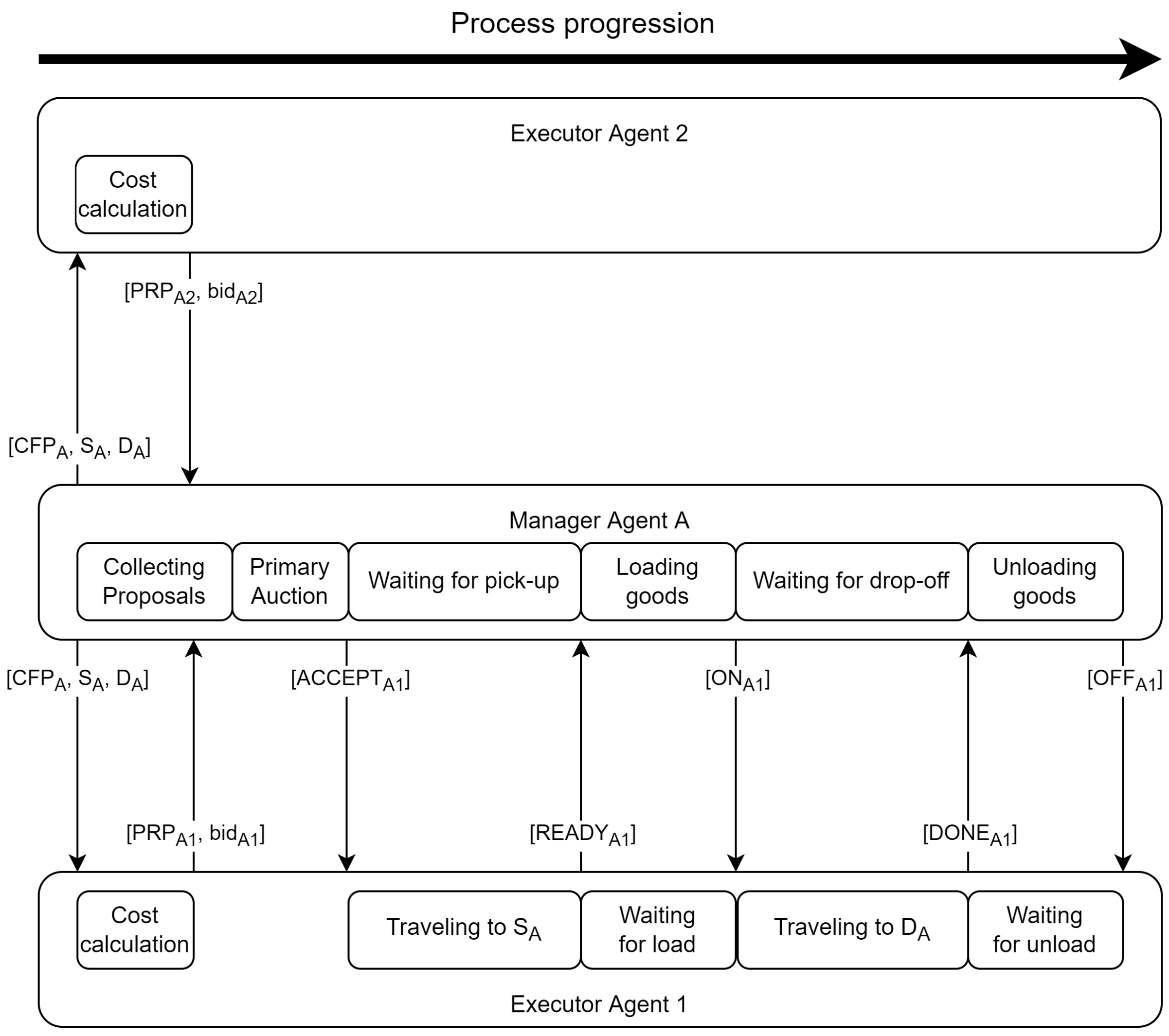


Figure 2.1: Agents' communication during the execution of a task

robot broadcasts a CFP message to all executors in the system and collects their answers. All available executors, and the assigned one, send their bidding value in a PRP message to participate in the auction. If the closest mobile robot is different from its current one, it sends an ACCEPT message to the winner and an ABORT message to the previously assigned executor. After the conclusion of the Secondary Auction it waits for the agent's arrival and may launch a new one if the reallocation mechanism requires it. When an executor receives an ABORT, it cancels its travel and becomes available to carry out any task.

If at any given moment an executor determines that it cannot perform a task assigned to it, it informs the corresponding manager with a FAIL message. This situations can occur because the robot encounters a a problem during the task execution or because it receives the assignment while carrying out another order.

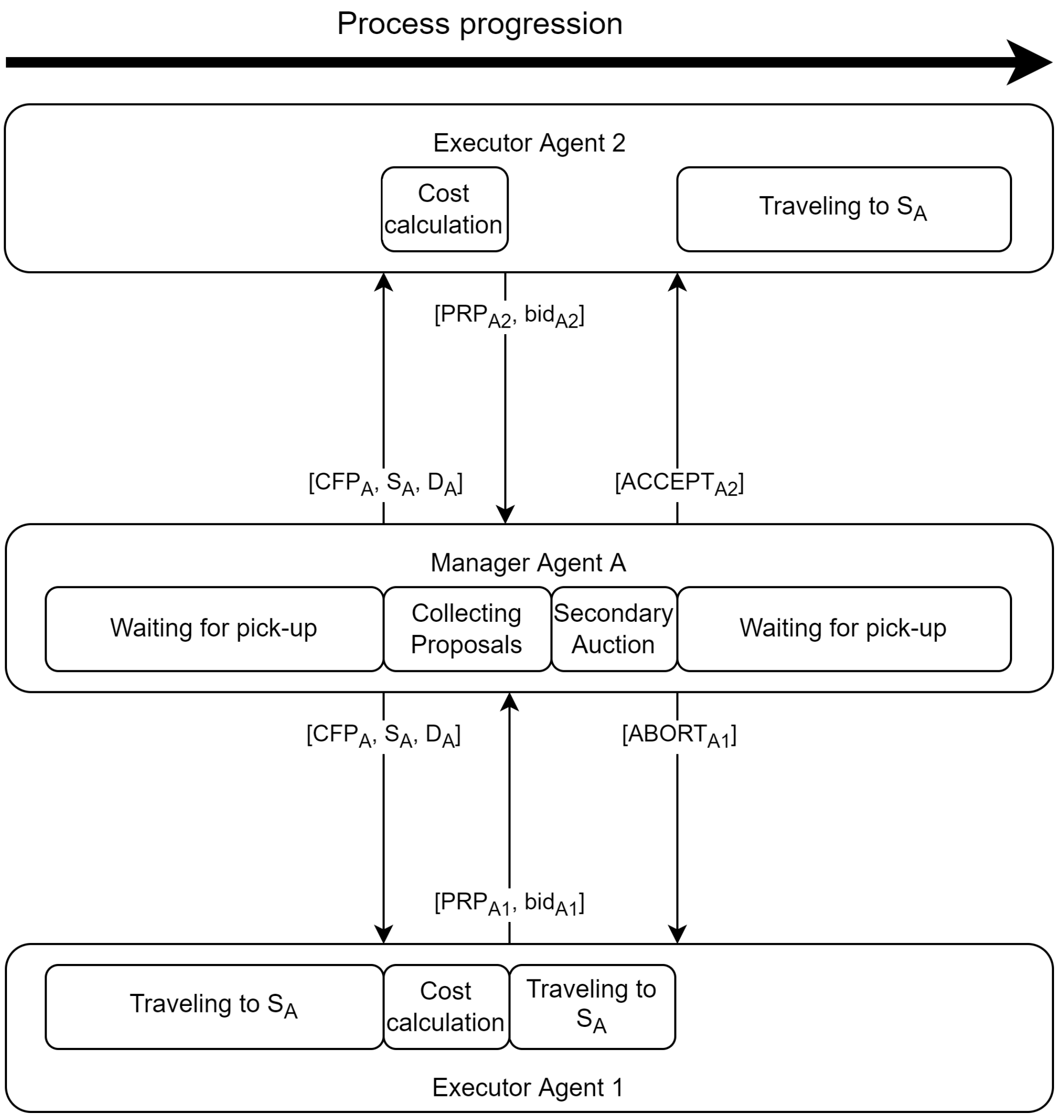


Figure 2.2: Agents' communication during a Secondary Auction

2.4.2 Operations example

As an example, we present how our MRTA system solves an allocation problem in the environment shown in Figure 2.3. This example is extracted from [Riv22b]. The three upper ports are charging stations, which means that they are the starting positions for the mobile agents. The four ports at the bottom are the pick-up and drop-off points. All the roads are unidirectional, with traffic flow in the bottom circuits going clockwise and counterclockwise in the top one. For more precision, the leftmost vertical road and the one in the upper right corner go up while the rest go down, the mid-horizontal street goes to the right, and the other two go to the left.

The system has three tasks to assign, one from each starting port (P1, P2, and P3). All tasks go to the same delivery zone, which are the leftmost ports on the map (D). All orders start simultaneously, having all mobile agents at the charging area (C). Figure 2.3 shows the starting distribution of task managers and executors for this experiment.

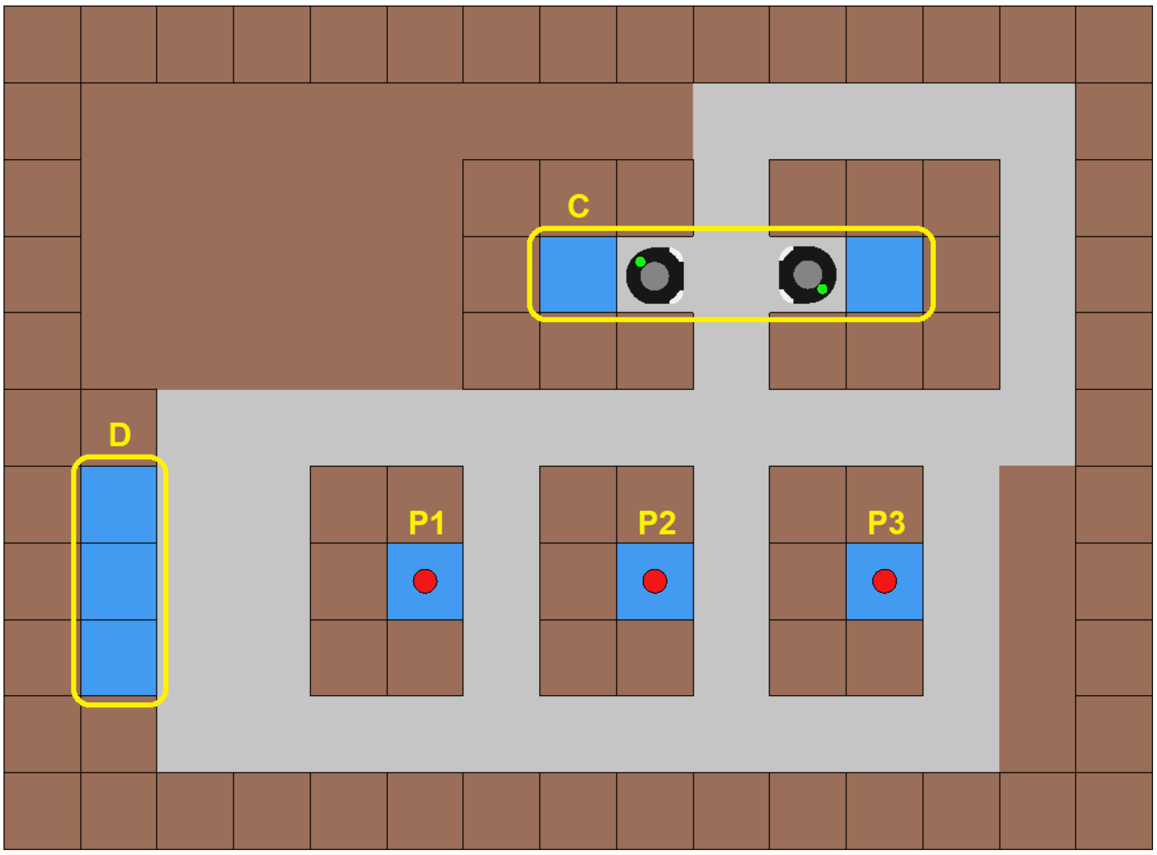


Figure 2.3: Example layout and starting distribution for task managers (red dots) and executors (black and gray objects)

Each task cost consists of two parts, a fixed value and a variable one. The fixed part is the distance between the starting and final ports, which will always remain the same. The variable section of the cost is the distance from the executor to the starting port, which depends on the vehicle's position at the assignment time. Since all tasks have separate starting ports but share the destination, their fixed costs have different values. Given that the executors will only be available at their starting positions (before starting any task) or at the delivery port (after finishing an assignment), the variable cost of each order will have only two possible values. With these particularities, Table 2.1 contains the cost values for each travel segment.

We explain the allocation procedures and interactions for two agents working simultaneously. When all three managers send their CFP, mobile agents A1 and A2 will answer them with PRP messages containing the same bidding values (29, 15, and 21, respectively). Since each auctioneer receives the same bid values, they select the first executor (A1) as the winner and send an ACCEPT message to it. Since task T2 has the lowest total cost in this situation (15), A1 keeps it and sends a FAIL message to the other two managers. When this happens, T1 and T3 broadcast CFP messages again, and since A2 is

Table 2.1: Costs combinations for the tasks

Task	Starting port	Destination port	Fixed cost	Variable cost	
				from C	from D
T1	P1	D	7	22	7
T2	P2	D	10	5	10
T3	P3	D	13	8	13

the only one that sends proposals (A1 answers with REF), both auctioneers pick A2 as the winner. Then, A2 keeps the one with the smaller cost value from the two (T3 with 23) and sends a FAIL to T1. At this point, T1 repeats its auction, receives zero proposals, and goes to the RESTART state for a while.

After all this deliberation, both robots begin to execute their respective tasks. When A1 arrives at P2, it sends a READY to T2 for it to get inside the vehicle. Once this happens, T2 sends an ON message to A1, which starts its travel toward the D ports. Upon arrival, it will send a DONE to T2, who will answer with an OFF after getting out of the mobile robot. A similar exchange happens between A2 and T3 as the robot executes the task. If T1 launches an auction during this time, it remains unattended since both mobile agents would reply with a REF. However, when A1 becomes available after finishing T2, this triggers the reallocation mechanism in T1, causing it to broadcast a CFP, to which the mobile agent will answer with a PRP containing a bid value of 14. Then, A1 becomes the auction's winner and proceeds to complete the task.

The utility of the reallocation mechanism can be exemplified with the same scenario, but considering that the tasks do not start simultaneously but gradually: T2 first, then T1, and finally T3. Let's assume that the two mobile agents are at the charging station when task T2 enters the system. As before, A1 wins the auction and proceeds to complete the assignment. Then, let's assume that manager T1 launches its Primary Auction when A1 is 5 meters away from the D port (and finishing T2). In this case, A2 will be the only executor to bid in T1's auction, making it the winner. However, if A1 attends T1 after finishing T2, the solution cost decreases since A1 only needs to travel 5 meters to port D and 14 to complete T1 from there, while A2 needs to travel 29 meters to do the same from the charging area. So, when A1 drops T2 and becomes available, T1's reallocation mechanism launches a Secondary Auction in which both A1 and A2 participate. At this point, A1 will propose a cost value of 14, while A2's bidding value will be 24 or a little less, depending on how much it advanced during T2's drop-off operations. So, T1 declares A1 as the winner, notifies it with an ACCEPT, and releases A2 with an ABORT. From there, A1 executes T2, and A2 can return to the charging area or attend to T3 when it enters the system.

2.5 Agents' structure

Due to the number of operations both types of agents need to perform, it is easier to design their respective controllers as a controllers' stack. Each controller, or layer, is designed for a specific task and can communicate directly with the other controllers in the same agent.

On each Manager agent, there are two controllers. On the top is the Deliberative Controller (DLB), which takes care of the communications with the Executors, either while performing auctions or at the different phases of the task completion. The Reactive Controller (RCT) measures the pass of time and informs DLB when it is time to perform a new auction or any other operation that depends on a timeout.

On the other hand, Executors have five controllers organized in layers. From the top, they are the Communications Controller (L4), the Path Planner (L3), the Path Follower (L2), the Traffic Manager (L1), and the Physical Controller (L0). The top layer, L4, performs the communications with the Task Managers from the moment they ask for transportation for their tasks to the drop-off at the destination point. Right below it, L3 calculates the shortest route to get to the goal position for obtaining the cost to execute the order during an auction or to get to the points of interest while performing the task. The next layer, L2, segments the planned path into shorter travels that are easier to perform and controls their execution. Second-to-last, L1 resolves the traffic conflicts among vehicles when traveling the environment. And at the bottom of the controllers' stack, L0 controls the vehicle's movement, turning, and obstacle detection.

The interactions between the different controllers of each agent can be seen in Fig. 2.4 and Fig. 2.5. Each controller layer usually sends commands to the layer right below it and responds to the one directly on top of it. However, each controller can access the data from other controllers in the agent whenever needed.

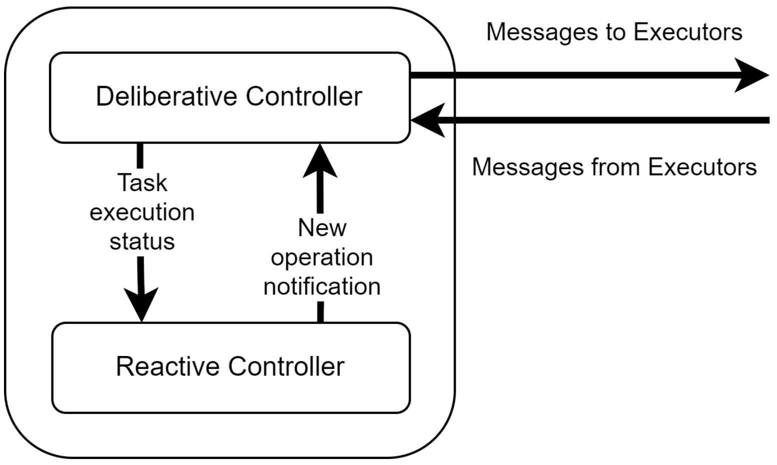


Figure 2.4: Controllers' structure inside a Task Manager

Inside the Manager Agent, DLB and RCT exchange signals to notify each other of the result of their operations. Figure 2.6 shows an example of such an exchange. DLB starts the process by launching an auction. However, since no Executor was available, DLB notifies RCT with a *No-bids* signal. This signal causes RCT to start a countdown (to allow the Executors to finish their current assignments) before sending a *Call* signal to DLB for the latter to restart its operations and perform another auction. During this second auction, some Executors answer the CFP, and DLB assigns the task to one of them. Now DLB sends a *Taken* signal to RCT for the latter to wait for the pick-up. When this happens, DLB receives

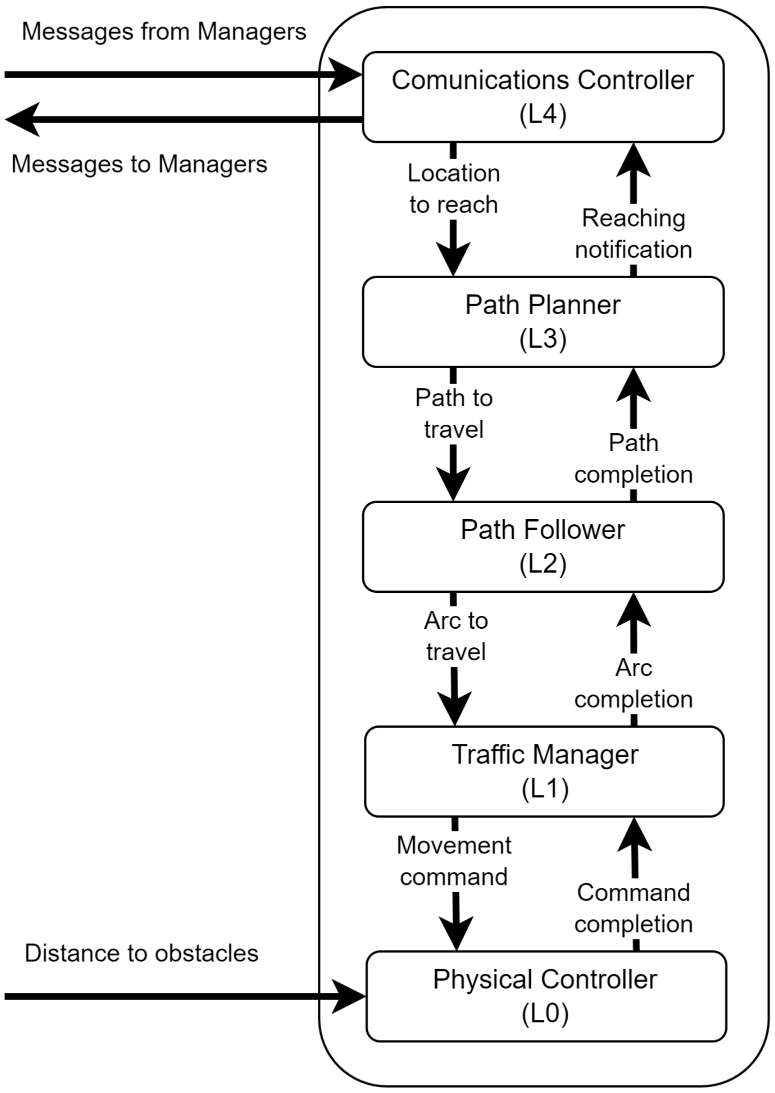


Figure 2.5: Controllers' structure inside an Executor

a READY message from the Executor and sends an *Arrived* signal to RCT for it to start the loading process. Once the goods are in the vehicle, RCT notifies DLB with an *On* signal. Then, this one sends an ON message to the Executor. When the travel finishes, the Executor sends a DONE message to DLB, which sends an *Arrived* signal to RCT to begin the unloading process. After the unload, RCT sends an *Off* signal to DLB for it to notify the Executor with an OFF message, which successfully fulfills the management process for the task.

On the other hand, an example of the interactions inside an Executor agent is shown in Figure 2.7. Since it comprises five controllers, we will explain the interactions between adjacent layers, going from top to bottom in the controllers' stack.

At first, L4 receives a CFP message, saves the Manager's position (S_i) and destination (D_i), and asks L3 for the minimum distance to travel to those places. Then sends this number back to the Manager

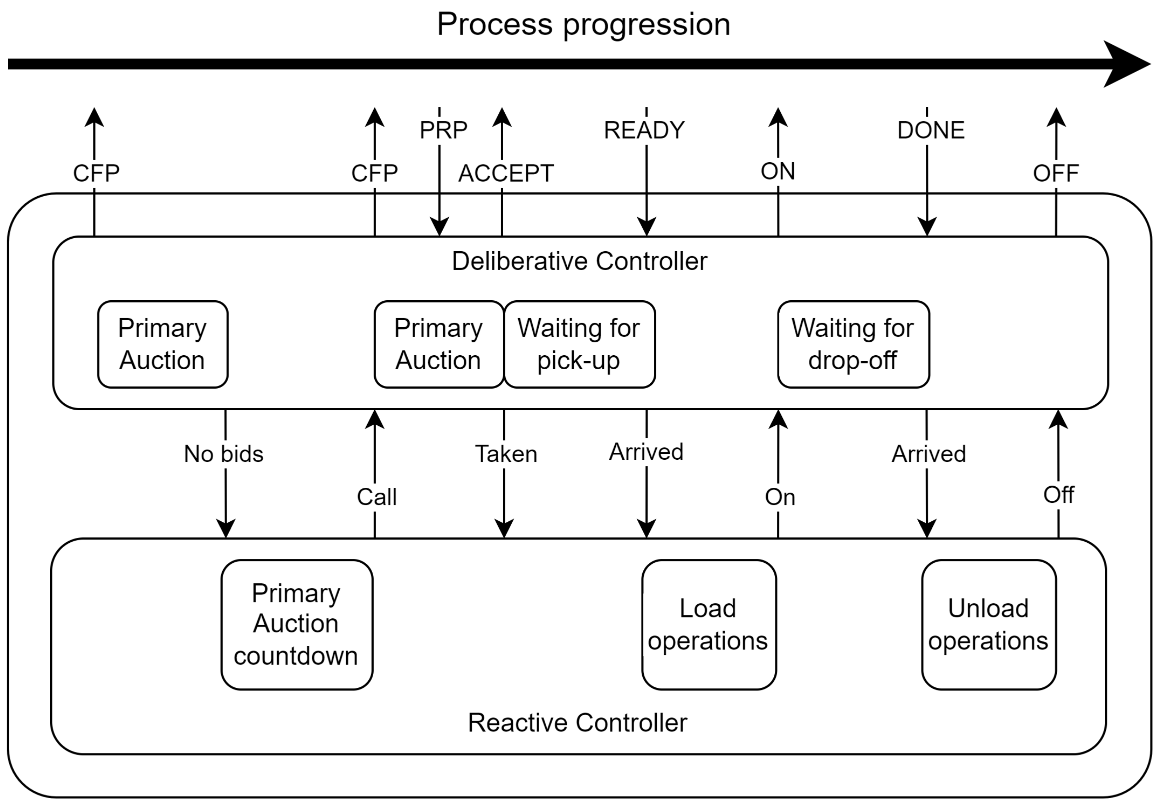


Figure 2.6: Controllers' interaction inside a Task Manager

in a PRP message. When the agent wins the auction (receives an ACCEPT message), L4 sends a GO command, with S_i, to L3 for it to calculate the shortest route there and order its execution. At the end of the travel, L3 sends an OK signal to L4, L4 informs the Manager, and the loading process starts. After these operations, L4 sends a new GO command to L3, but this time with D_i. At the travel culmination, L3 notifies L4 with an OK signal, and this one informs the Manager. Once the task ends, L4 sends a *Free* signal to L3 for the vehicle to proceed with its operations.

Whenever L3 receives a GO command, with a goal location, from L4, it calculates the shortest route to the goal and sends this path (list of connected nodes) to L2 for execution. If an immobile obstacle appears along the way, L2 sends a KO signal, with the blocked *arc*, to L3. When this happens, L3 calculates a new path, avoiding the blockade, and sends it to L2 in a GO command. After the Executor finishes traveling the route, L2 answers L3 with an OK and waits for further instructions. When an assignment ends, L3 notifies L2 with a *Free* signal.

Every time there is a movement to execute, L2 sends a GO command to L1, containing the *arc* to travel (denoted by the node the agent is currently at and the node it wants to visit). L1 waits for the target node to be available and then orders L0 to move there. If the objective node is unavailable for too long, L1 may notify L2 with a KO, for it to re-calculate the route. A similar thing happens if L0 detects an immobile obstacle in the way: L1 sends a KO signal to L2. Otherwise, at the end of the command execution, L1 notifies L2 with an OK.

Before executing any movement from one node to another, L1 checks that the target node is available, which means that no other vehicles are visiting it. If another agent is at the node, L1 waits for the node to become available before sending a GO command to L0 with the *angle* the agent needs to turn and the *distance* to advance to reach the target node. When the command execution finishes, L0 sends an OK to L1 and awaits further instructions. However, if L0 detects an immobile obstacle along the way, it sends a KO signal to L1.

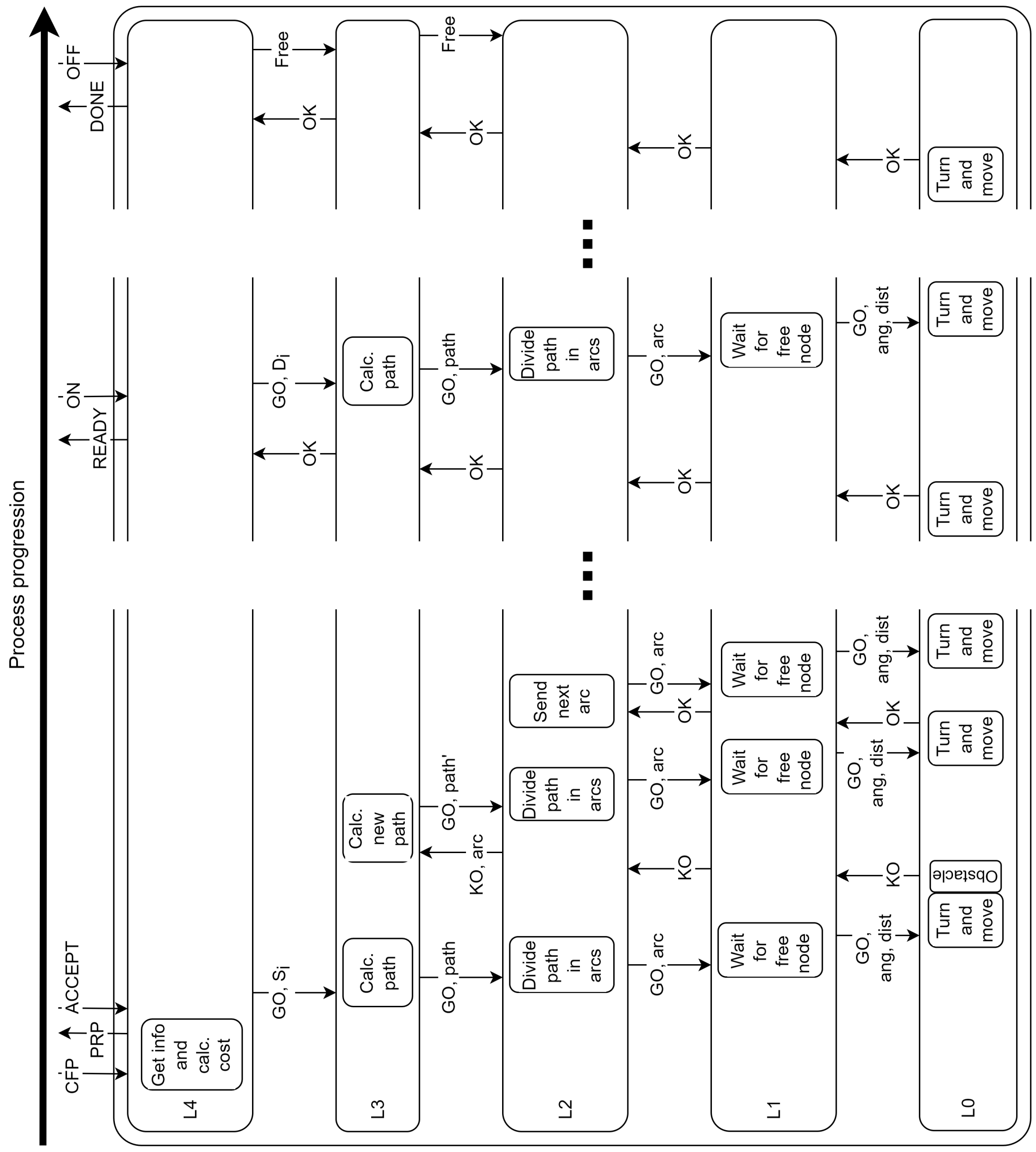


Figure 2.7: Controllers' interaction inside an Executor

2.6 Model representation and formalization

To perform those interactions, the agents need controllers capable of making, modifying, and aborting plans without difficulties. Deterministic procedures to plan transportation are hard to apply in today's industrial and warehousing facilities. So, we intend to develop these plans from a formal description of the system's behavior to preserve predictability and verifiability.

Our approach consists of using state machines to formalize and parameterize a TO management model as a multi-agent system (MAS) integrated with a physical simulator. This method ensures a good interaction of our system with industrial plants, which are distributed systems and populated with equipment whose controllers are, essentially, state machines [Coh17]. The latter constitutes the most widely used method to specify the dynamic behavior of a system [Bou14], and some may even represent the behaviors of simple belief-desire-intention (BDI) agents [Riv18]. Development environments, specifically those based on standards IEC61131 [Lin13] and IEC61499 [Vya11, Pat15, Ian16], include, or allow the inclusion of, tools to synthesize state machines [Ian16, Pos09, Hag08].

Many works exploit the formal background and ease of implementation that state machines provide, which makes them very helpful in the systems' verification and implementation processes [Pop22, Li18, Zho17]. These characteristics lead to error minimization during the programming, easier debugging and more straightforward explanation process. Also, this type of language eases the design, analysis, and customization of the operations in a controller [Coh17]. Furthermore, state machines maintain synchronicity among the variables' values, which allows the user to pinpoint the system's situation at any time during its operation.

We paired these graphical representation methodology with a procedural code synthesis system which differs from others (e.g. [GS22, Coh17, Bru16, Jen12, Bou14, Lin08, Sam00]) in including several types of state machines including one designed in our research group. The code generator takes diagrams drawn with draw.io and outputs source code that adheres to the proposed software architecture. Currently, generates code for Lua (www.lua.org) and Netlogo though it is not limited to these languages. Automated code synthesis increases productivity [Jen12, Hub03, BÖ3, Str08] and minimize architecture drift [Coh17, Bou11] during this phase. These tools also make it straightforward to complement those machines with error control mechanisms, which facilitates the production of more robust controllers.

With the models' formalization, emerges a group of parameters that define the agent's and the system's behaviors. Such parameters may set the waiting time for an event to occur, the number of repetitions of a certain action, etc. Many of these parameters do not have intrinsically an strict value, unless defined in a particular study case. This means that each parameter provides a degree of freedom to the system's behavior. So, some of them can be set to an specific value, to comply with the given restrictions of a particular case, and the rest can be modified to find the combination that returns the best desired results in terms of efficiency, waiting times or any other benchmark.

2.6.1 Extended Finite State Machines

State machines can be described as a five-tuple $(Q, \Sigma, \Omega, \Delta, q_0)$, where Q is the set of states, Σ is the set of inputs, Ω is the set of possible outputs, Δ represents the transition function $\Delta : Q \times \Sigma \rightarrow Q \times \Omega$, and q_0 is the initial state. In Extended Finite State Machines (EFSM), there is an additional set of variable valuations V in the form $V : N \rightarrow D$, where N is the set of variable names and D is the set of values that variable can attain. So the transition function is $\Delta : (Q \times V) \times \Sigma \rightarrow (Q \times V) \times \Omega$ [Pto14].

Input format

For the procedural code generation we use diagrams with general shapes and connections of draw.io to represent state machines. Graphs for EFSMs use circles to represent states and arrows for the transitions, having a direct correspondence between graph components and diagram elements. Computation blocks inside circles define operations that take place at the corresponding states and arrow labels define those expressions that activate corresponding transitions when true [Riv18].

Source code synthesis

The code generation process takes uncompressed XML files of graph descriptions from draw.io and writes code that follows the style of a state-based program [Sha14], i.e. with an explicit main loop [Pon02] to simulate the evolution of the state machines within the system.

In our proposed model, state machines are separated from the execution engine, that is, from the main loop that calls them. In this way, the execution control takes place within the "engine" and the state machines are in charge of their own behavior. The execution engine loop may include, for example, a time management mechanism and conditional calls to the methods or functions that execute the state machines behavior. Time management can range from a simple cycle counter to a system that conveniently updates a variable with the simulated time. Conditional calls to the state machines' associated functions allow, for example, executing only those whose inputs have changed at the beginning of the current instant. That is to say, they allow establishing an execution directed by events (event-driven). In any case, each state machine or module behavior is organized in different methods or functions, with a standard name that facilitates its identification, namely, *init()*, *read_inputs()*, *step()*, *update()*, *write_outputs()*, and *active()*.

The *step()* function calculates the next state from the content of the input signals and the variables in the current state, as well as the contents of the rest of the variables for the next iteration, and method *update()* assigns the current values of the variables and all the output signals.

Note that variables can act as input and output signals. To avoid dependencies of the execution order of variable updates, all variables, whether or not they correspond to the modules output signals, will be tuples of two values: the current cycle value and in the next cycle value. The rest of the signals will only have a single value: that of the current cycle.

To differentiate the values of a variable, e.g. *X*, within a diagram, the next cycle value has a plus (+) sign at the end of the variable name: *X*+. In the code, variable *X* contains two fields: *X.curr* for saving its value in the current cycle and *X.next* for saving the value it will take in the next cycle. At the end of each cycle, the *update()* function assigns each variable's *.next* value to their *.curr* counterpart (i.e. *X.curr* = *X.next*). So, as long as the users follow this data structure (all reading operations do it from the *.curr* values and all the value assignments do it to the *.next* one) the controller has execution order independence.

This execution mechanism separates all reading operations from the writing ones in same cycle. This is beneficial because many controller's designs contain some variables which values are read and modified in the same cycle. This event can lead to a misinterpretation of the situation by the controller, causing it to react inappropriately. This problem is usually tackled by executing all operations that depend on reading a certain variable before any operation that modifies such variable in that same cycle. In simple controllers or programs this can be done manually without great effort. When programming complex controllers however, especially when modifying them, to maintain a proper execution order results quite difficult and labor intensive.

2.6.2 Extended Finite State Stack Machines

An Extended Finite State Stack Machine (EFS^{2}M) is an EFSM with a stack of states [Riv18]. So, it is a tuple $(Q, V, U, \Sigma, \Omega, \Delta, q_0, v_0, u_0)$ where Q is the set of states, V is the set of variable valuations, U is a string of symbols from Q (the stack), Σ is the set of inputs, Ω is the set of possible outputs, $\Delta : (Q\times V\times U)\times\Sigma \rightarrow (Q\times V\times U)\times\Omega$, q_0 is the initial state, v_0 is the initial variable valuation, and u_0 is the initial string of stack symbols. In a sense, EFS^{2}M are similar to pushdown automata (PDA) [Mul85] whose stacks can only be filled by states. Differently from PDA, EFS^{2}M have graphical representations for all model operations, including those that affect the state stack, making representation of machine behaviors easier.

EFS^{2}Ms are supersets of EFSMs that can model procedural agents as well as of Believe-Desire-Intention (BDI) type. The ability to push states onto the stack enables the creation of plans for the EFS^{2}M to follow. Procedural BDI agents have a degree of complexity that usually cannot be represented with conventional EFSMs. EFS^{2}Ms overcome this difficulty and can model BDI agents with representations that are usually simpler than the ones that would be done with conventional EFSM [Riv18]. In fact, all intention operations performed by BDI agents have their EFS^{2}M counterparts. [Riv19b]

Input format

EFS^{2}M diagrams use circles and arrows, like EFSM, but also small circles and hexagons to represent the operations with the stack. Empty small circles at the beginning of arrows mean popping states out of the stack while at their tips mean pushing them into it. In the last case, empty circles imply stacking the current state, to stack any other state, the circles must specify the state name. On the other hand, hexagons at the beginning of arrows mean to empty the state stack, and, at their endings, designate the top of the stack as the next state.

Source code synthesis

The synthesis procedure adds a stack attribute to the state so to be able to generate pushes (small circles at the end of arrows and pops (hexagons and small circles at the beginning of arrows). Hexagons at the end of arrows also become stack pops to obtain the value for next state.

2.6.3 Controllers' formal representation

For our MRTA system modeling, we used EFS^{2}M. Figure 2.8 shows a simplified version of the deliberative controller (DLB) from a manager agent using EFS^{2}M. Using the stack operations, the DLB can plan and adapt as required. Following the example we present in [Riv22b], the controller starts in the INIT state, sends a CFP to all executors, and pushes the current state (INIT) and PRI_AUC into the stack before going to the GET state, where it collects the *proposals*. With this procedure, the controller can advance, after the collection, to the Primary Auction state (PRI_AUC) with a pop operation. Also, if at some point it needs to start a new auction, it can pop the INIT state from the stack to do so, e.g., if DLB is at WAIT and the assigned executor sends a FAIL message because it is unable to perform the task.

The managers collect the proposals from all available executors for as many cycles as the value in the CD (collect deadline) parameter establishes. Then, it evaluates them, keeps the *best* one, and goes to the state at the top of the state's stack, which in this case is PRI_AUC. Now the DLB notifies the winner of the auction results with an ACCEPT message and saves its id before going to WAIT until the robot arrives. While waiting, the controller might launch some Secondary Auctions when the *re-call()* function returns *true*. This function is a simplified representation of the reallocation mechanism that returns *true* every time it considers there are possibilities of reassignment. In these cases, the controller broadcasts a new CFP message, pushes the SEC_AUC state into the stack, and returns to GET. After the collecting, the DLB goes to the state at the top of the stack, which now is SEC_AUC. In a Secondary Auction, DLB verifies if the victor coincides with the currently assigned mobile agent. If they do, it goes to WAIT without performing any more actions. But if they are different, the DLB saves the id of the new winner, notifies it with an ACCEPT message, and releases the previously assigned robot with an ABORT message. The mobile agent sends a READY message to the manager when it arrives at its position. The manager puts the products inside the executor, sends it an ON message, and goes to the RIDE state. During the ride, DLB waits for a DONE message from the executor, which means that they have arrived at the destination. At this point, the manager unloads the products from the robot, sends it an OFF message to release it, and goes to the IDLE state. Since the task is complete, the manager does not perform any more operations after reaching this state. When there is no winner during a Primary Auction, it means that all mobile agents are unavailable. In these cases, DLB goes to the RESTART state, where it remains for as long as the PD (primary deadline) parameter establishes (or until the *re-call()* function

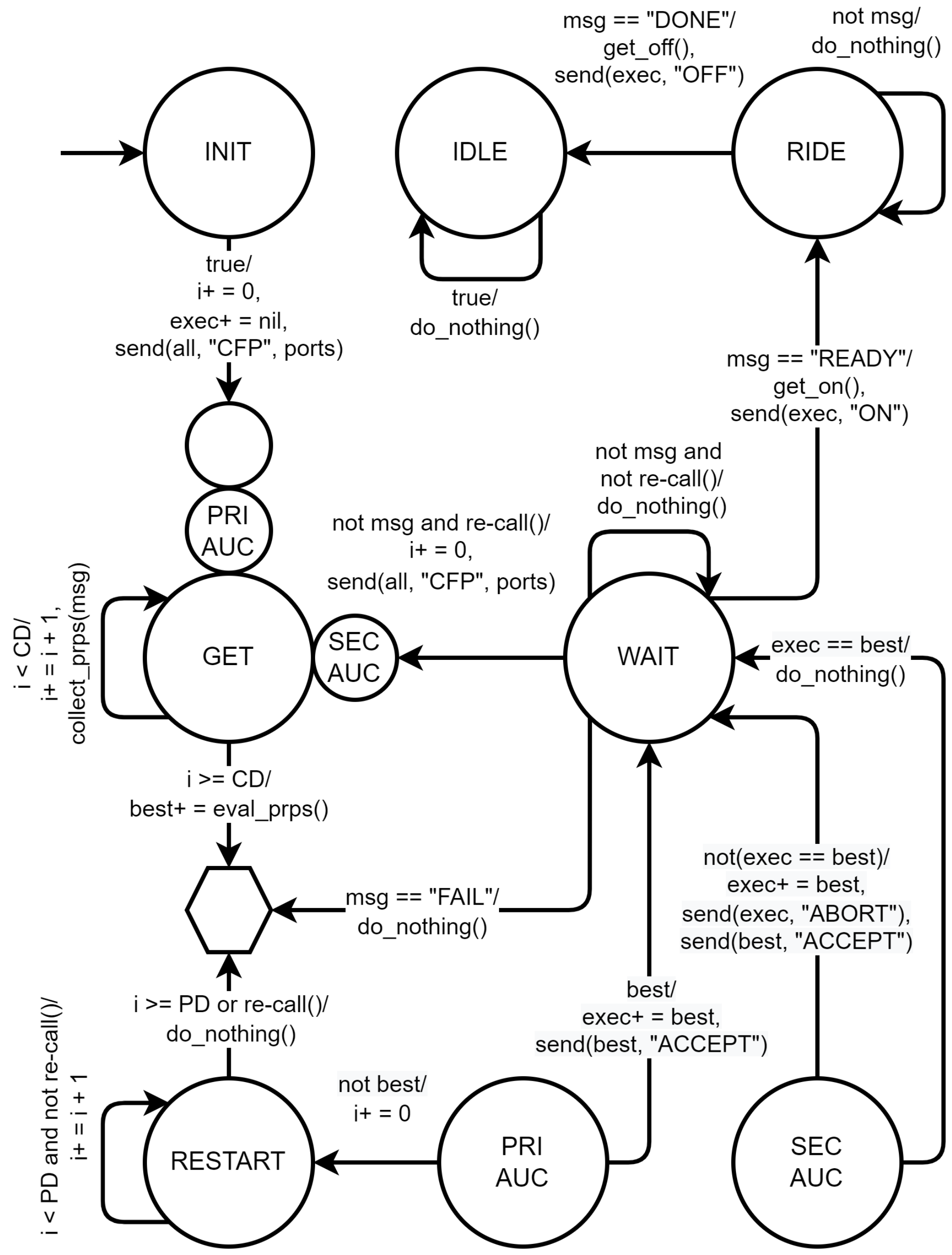


Figure 2.8: Simplified EFS^{2}M diagram of the deliberative controller for a task manager agent

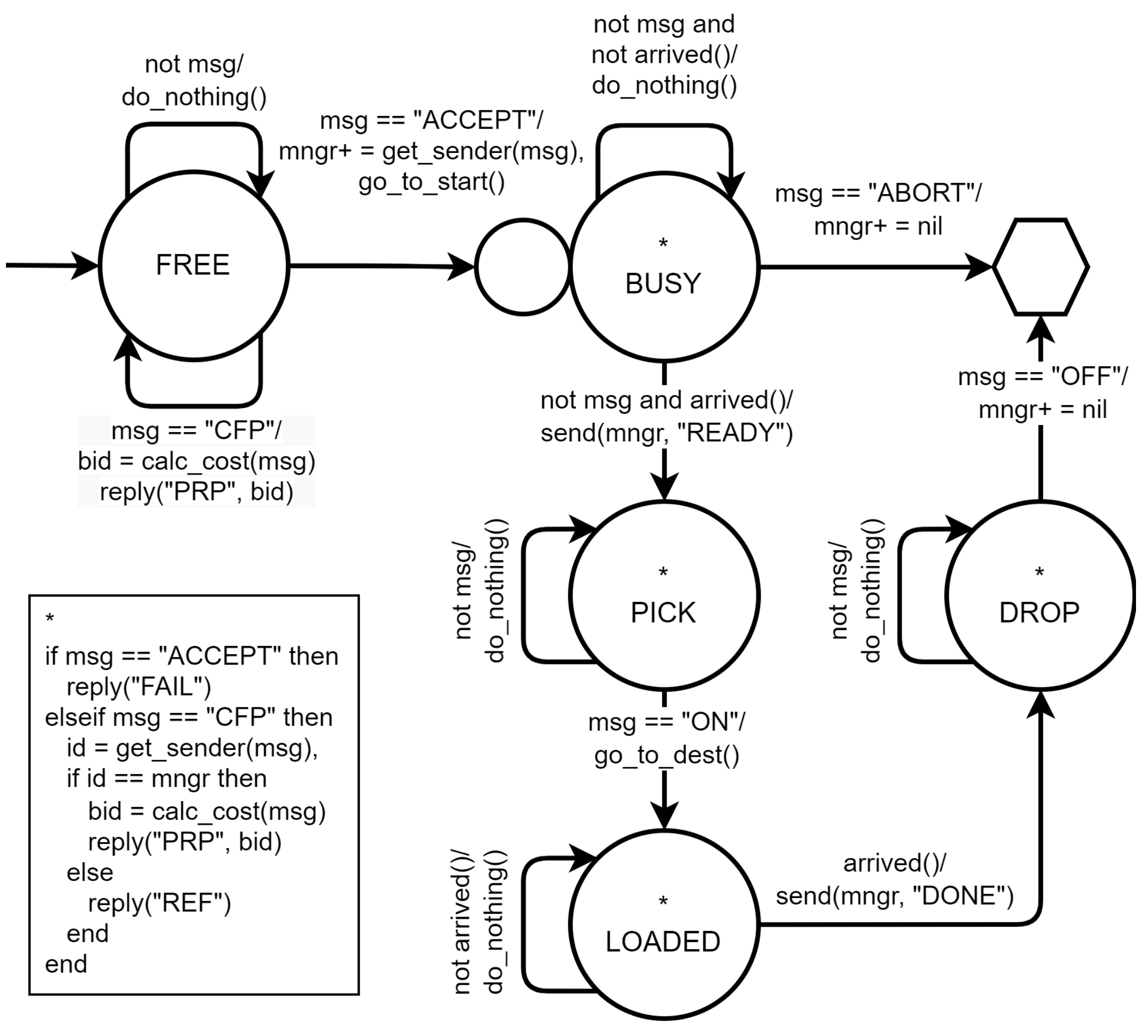
returns *true*). In any of those situations, it goes to INIT to launch a new auction.

The executors' counterpart for DLB is the Communications Controller (L4). Figure 2.9 shows that this controller starts at the FREE state and remains there until winning an auction. However, it performs some operations in this state when it receives a CFP message containing the location and destination of the manager. Since, at the moment, it is not executing any task, it replies to these calls with PRP messages containing the distance it would have to travel to complete the corresponding task. However, when it is busy with an assignment, it will only send *proposals* to the calls from its allocated task manager (the square in the figure describes this behavior). It will *refuse* the calls from any other DLB with a REF message. After winning an auction, the executor receives an ACCEPT message. In this case, L4 saves the sender's id, orders the vehicle to travel to the task's starting position, pushes the current state (FREE) into the states' stack, and goes to the BUSY state. It remains there until the end of the travel, which is signaled by the *arrived()* function, or until receiving an ABORT message from the manager, which means that it found a better transportation option. When the assignment gets canceled, the controller deletes the assigned task's id and returns to the FREE state. However, when it is not substituted and arrives at the pick-up location, it sends a READY message to the DLB and goes to the PICK state. In this state, L4 waits for the manager to get the products inside the vehicle and send an ON message. Upon this confirmation, L4 orders the mobile robot to go to the task's destination port, and it goes to the LOADED state to wait for the end of the travel. Once the *arrived()* function returns *true*, the controller sends a DONE message to the task manager and goes to the DROP state. There it waits for the OFF message at the end of the unloading operations, to which the L4 reaction is to delete the manager's id and return to the FREE state.

Given that several tasks can take place concurrently, some managers may perform their auctions at the same time. In this situation, there is the possibility that more than one DLB declares the same mobile agent as their auction winner. When this happens, L4 keeps the task to which it presented the smaller bid as its assignment and sends a FAIL message to the rest of the task managers that sent an ACCEPT.

2.7 MRTA system and physical simulator integration

The simplified Deliberative controller diagram in Figure 2.8 shows that the whole auction process (from the CFP to announcing the winner) takes several cycles. As we explain in [Riv23], this means that by executing the controller with a regular physics simulator's execution engine, time will pass during the whole operation. Under these conditions, after the controller completes its operations for the current cycle, it will have to wait for another controller to finish a movement before proceeding with its actions. This situation can cause a mismatch between the environmental information a controller is working with and the actual environment, which can cause severe problems in the task assignment. For example, imagine that three free robots (R1, R2, and R3) are passing, one right after the other, in front of the manager for a transport task, represented by a red dot in Figure 2.10 (left), right at its starting time. The



```
*
if msg == "ACCEPT" then
  reply("FAIL")
elseif msg == "CFP" then
  id = get_sender(msg),
  if id == mngr then
    bid = calc_cost(msg)
    reply("PRP", bid)
  else
    reply("REF")
  end
end
```

Figure 2.9: Simplified EFS^{2}M diagram of the communications controller for an executor agent

less costly solution in this situation would be to assign the robot right in front of the starting point (R2) since it is the closest one. However, since the auction lasts more than one cycle, the robots will keep moving, and R2 will pass the pick-up point, represented with a green dot in Figure 2.10 (right)before the winning notification reaches it. At this point, R2 will have to go around the block to pick up the task (given that the traffic flow does not allow it to go backward). In this situation, it becomes the worst candidate to pick the task up: R3 is the closest one, and R1 is ahead of R2 in the route around the block, represented by a red arrow in Figure 2.10 (right). This incorrect assignment can greatly degrade the system performance in environments with a high concentration of robot agents, where this situation will be more common.

Our approach avoids this problem by executing immediate tasks like planning and communications, which have no time consumption, separated from those that do consume time. So, for the case described previously, there would be no delay between asking for transportation to the executors and informing the winner. In this case, R2 stays in front of the task's starting point from the moment it receives the CFP to the instant when it receives the assignment since no time passes between the two events, and it just needs to stop to pick up the goods.

Algorithm 1 shows the execution of this time model. Each simulation starts by executing *init()* to initialize the model elements: agents, communications, etc. Then the simulation executes the main loop. Instead of a single loop structure that calls all functions cyclically, our model relies on a nested loop inside the main one for operation differentiation. Each execution of the main loop implies the passing of time, denoted by the increment of the simulation time (*simTime*) by a time step (*dt*) at the end of the main cycle. The internal loop, which only contains immediate operations, can execute them many times without time advancement. The internal loop keeps on executing while there are changes inside any controller. So, when there is a change in any internal variable value, input, or output of any of the controllers, the time stops until all internal operations finish. When a controller has no more internal changes, we say it reached a stability point. There are two ways to achieve this condition: the controller is waiting for some external event, like a signal from another controller or a timeout, or the controller must change the environment and waits for time to flow, to do it. When all controllers reach a stability point, the internal loop execution stops, and time starts to run again. Since time does not pass during the internal loop execution, any operation that depends on time, like the interactions between the agents and the environment, happens outside the internal loop. *read_environment()* encapsulates all sensing operations from the environment, while all actuation operations are in *write_environment()*.

The internal operations loop only contains functions that do not consume time. Through *read_inputs()*, all controllers register the data sent by other controllers. However, all controllers do not send their signals the same way. Some maintain the signal value for some time, while others send it only for one time step. In this last case, if time stops right at the reception of the signal, the controller may believe that it is a new notification. To avoid this, the controllers must treat the inputs differently when

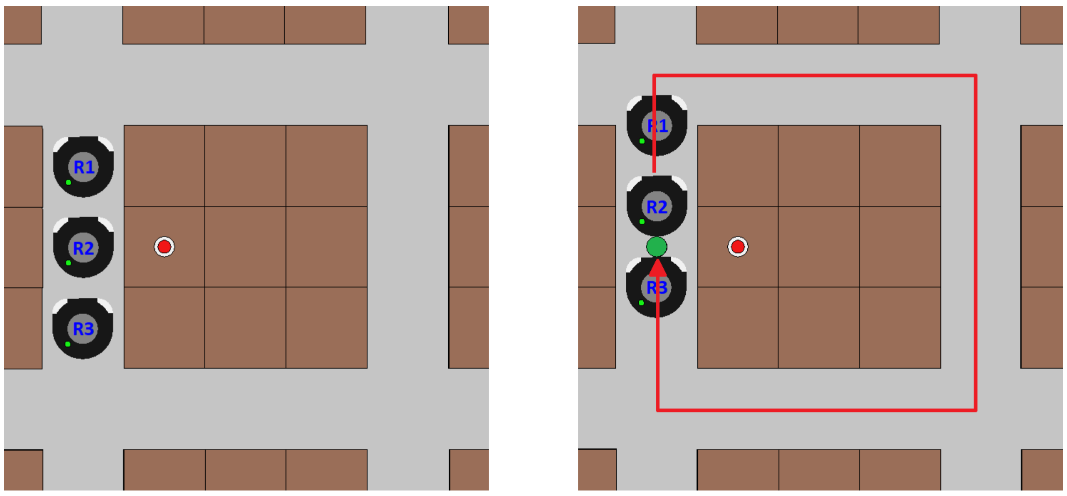


Figure 2.10: TA solution degradation due to time consumption during the assignment. Left: Agents' positions when the proposals collection happens. Right: Agents' positions when R2 wins the assignment

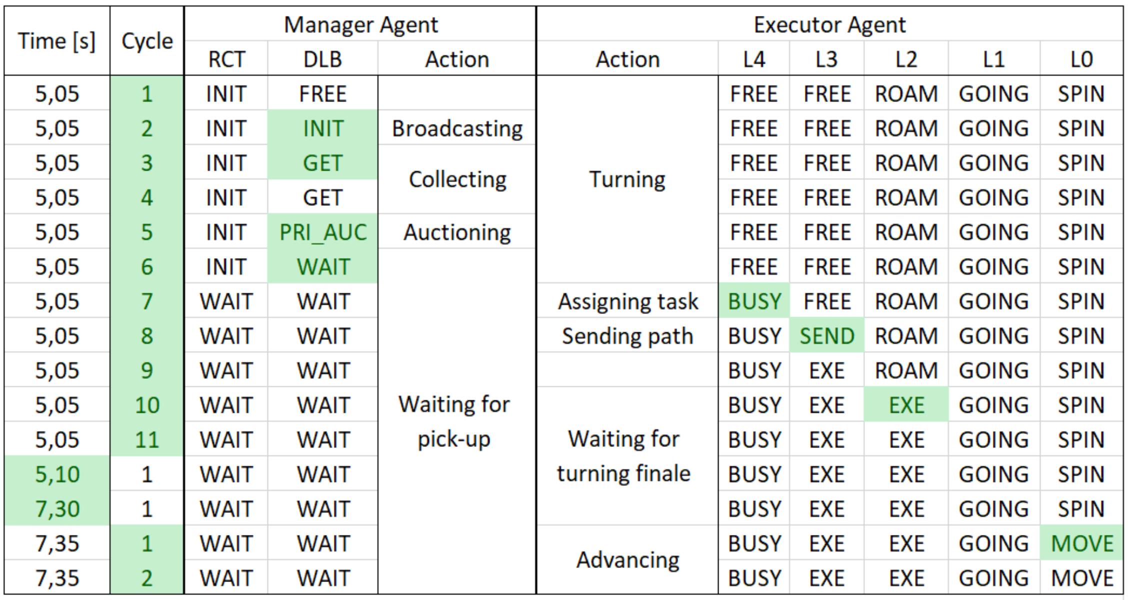
time is flowing than when it is still. Given that the internal loop executes once in a time loop when time flows and more times when time stops, *read_inputs()* receives the parameter *first* to differentiate these situations. The *step()* function determines the values of all variables for the next cycle, according to the current values of the inputs and variables. Also, all controllers send signals and data to other controllers through *write_outputs()*.

Finally, *update()* assigns the values determined by *step()* to the corresponding variables. Since all writing operations become valid during *update()*, the variables maintain the same value for all reading operations in *step()*. Such distribution ensures that all variables remain the same during a cycle, which allows generating the code with order independence between the instructions [Riv18]. The return value of *update()* tells if the internal loop execution must go on or not by comparing all variables' current values with their previous ones. If at least one value is different, the loop keeps executing. If all controllers report no changes, the system is at a stability point, and the main loop execution may proceed.

To show how our Time Model works, we performed a simulation using only one Executor and one Manager. During the execution, the system registered the simulation time, internal cycle count, and state for each controller at each internal operations cycle. A fragment of that register appears in Figure 2.11, where the Manager agent performs an auction and assigns the task to the Executor, which is currently making a turn. Following our design, from the moment the controllers inside the Manager begin to perceive changes in their variables and states, the time progression stops, and the internal cycle count increases. This situation persists until the assignment notification reaches L2. Then, all controllers achieve a stability point, and the time starts to flow again. When a new change happens, due to L0 finishing a turn, time stops again until L0 is ready to begin its movement, when the time starts to flow again.

Algorithm 1 Time model execution

```
$dt \leftarrow 0.05$                          // Simulation time step
$simTime \leftarrow 0$                        // Simulation time
$init()$                                      // Initialization
while $true$ do                               // Main simulation loop
    $int\_ops \leftarrow true$                // Internal operations flag
    $first \leftarrow true$                   // First internal cycle indicator
    $read\_environment(simTime)$              // Sense the environment
    while $int\_ops$ do                       // Internal operations loop
        $read\_inputs(first)$                 // Receive information
        $step()$                              // Calculate next values
        $write\_outputs()$                    // Send information
        $int\_ops \leftarrow update()$        // Establish next values
        $first \leftarrow false$              // Finish the first internal cycle
    end while                                 // End of internal operations loop
    $write\_environment()$                    // Modify the environment
    $simTime \leftarrow simTime + dt$         // Advance the simulation time
end while                                     // End of main simulation loop
```



Time [s]	Cycle	Manager Agent			Executor Agent					
		RCT	DLB	Action	Action	L4	L3	L2	L1	L0
5,05	1	INIT	FREE		Turning	FREE	FREE	ROAM	GOING	SPIN
5,05	2	INIT	INIT	Broadcasting		FREE	FREE	ROAM	GOING	SPIN
5,05	3	INIT	GET	Collecting		FREE	FREE	ROAM	GOING	SPIN
5,05	4	INIT	GET			FREE	FREE	ROAM	GOING	SPIN
5,05	5	INIT	PRI_AUC	Auctioning		FREE	FREE	ROAM	GOING	SPIN
5,05	6	INIT	WAIT	Waiting for pick-up		FREE	FREE	ROAM	GOING	SPIN
5,05	7	WAIT	WAIT		Assigning task	BUSY	FREE	ROAM	GOING	SPIN
5,05	8	WAIT	WAIT		Sending path	BUSY	SEND	ROAM	GOING	SPIN
5,05	9	WAIT	WAIT			BUSY	EXE	ROAM	GOING	SPIN
5,05	10	WAIT	WAIT		Waiting for turning finale	BUSY	EXE	EXE	GOING	SPIN
5,05	11	WAIT	WAIT			BUSY	EXE	EXE	GOING	SPIN
5,10	1	WAIT	WAIT			BUSY	EXE	EXE	GOING	SPIN
7,30	1	WAIT	WAIT			BUSY	EXE	EXE	GOING	SPIN
7,35	1	WAIT	WAIT		Advancing	BUSY	EXE	EXE	GOING	MOVE
7,35	2	WAIT	WAIT			BUSY	EXE	EXE	GOING	MOVE

Figure 2.11: Time flow during an auction

Our time model ensures that all controllers operate with environmental information adjusted to the current situation in the model and also allows a representation of the modeled transportation system closer to reality, where different processes consume different amounts of time. This capability would make it easier to integrate our model with a digital twin of a physical system. The digital twin would provide precise data from the physical system, like travel speeds, waiting times, and task distribution, which would allow a highly accurate representation of the system with our model. This model could later be used to study how different changes in the system (like the number of agents, routes, task distribution, etc.) could affect its performance.

2.8 Obstacles and traffic management

2.8.1 Traffic conflicts

To guarantee a good traffic flow, the control system in automated logistics systems based on mobile robots must deal with their interactions with the environment and between themselves. A good practice is to provide the system with tools to manage the detection of obstacles in the road or conflict situations when two or more vehicles try to access the same location simultaneously. The lack of procedures to deal with these situations can lead to traffic jams that may paralyze the whole plant and, in the worst cases, cause damage to the robots due to collisions with other robots or objects in the environment.

2.8.2 Collision avoidance

Our system provides the mobile agents with two mechanisms to avoid collisions. The most basic one relies on a distance sensor in the front of each vehicle. When the robot detects an object is too close, it stops its movement. If the obstruction remains after a while, the agent considers it a persistent obstacle and tries to find an alternative route that avoids it.

While this method prevents collisions with obstacles or vehicles coming from the front, it does not perform very well in intersections or roundabouts, where other agents may come from the side and get too close without even being detected. To handle these situations, we added a service list to each node to determine the order in which the vehicles could access them.

Whenever an agent plans to go from one node to another, it puts its ID number at the end of the objective node's services list. So, the list contains the ID of all the vehicles that want to travel there, chronologically ordered. However, only the robot with its ID at the top of the list can access the node. When a vehicle detects that this happens, it deletes its ID number from the node it is currently at, allowing the next robot on the list to access it, and starts to travel to the goal node. This mechanism avoids collisions between robots since only one can access a node at a time and allows for solving simple traffic conflicts.

2.8.3 Deadlocks detection and resolution

While our two collision avoidance methods solve most of the conflicts that may appear during the system operations, there are some cases they cannot handle. An example is a situation like the one presented in Figure 2.12: four agents in a roundabout, trying to advance to the node in front of them, which already has another robot on it. Each vehicle placed its ID in the corresponding service list and is waiting to be served, which will never happen until one of them changes its goal node and moves, liberating its current node. This kind of situation is known as a deadlock, and our conflict resolution mechanism alone is not capable of solving them.

When facing this type of problem, many researchers opt for adapting their systems to make deadlocks infrequent and avoid the situation altogether [Per19, Kim06, Li09]. However, we opted to include a method to detect and solve these conflicts. This approach allows our system to offer solutions for a broader range of environments without burdening the user with designing layouts that diminish the occurrence of deadlocks. So, we included a similar mechanism to the one used for obstacle detection. When an agent remains too much time waiting for access to a node, it looks for alternative routes that avoid said node. In case it finds one, the vehicle changes its target node and, in case it is free, starts moving there, liberating its current node for the use of the next robot in its service list. In the example previously described, each agent can turn to its right and leave the roundabout, allowing the rest to resume their movements.

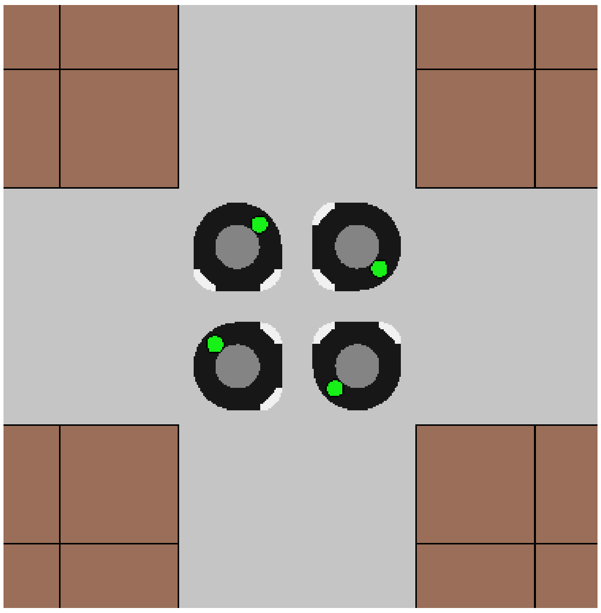


Figure 2.12: Four agents in a deadlock: each one tries to advance to the node in its front, which already has another robot on it

Even though this mechanism can solve deadlocks in most situations, it is not suited for all circumstances. The method resolves the problem by sending one of the robots anywhere but the node it is currently trying to go, without much regard for which vehicle it is or where it is going. This approach works fine in environments with low traffic density, where the deadlocks do not involve many agents. However, in cases with high road saturation, it is more suitable to use a mechanism that analyzes the whole conflict and provides a solution that allows most vehicles to resume their travels rather than lead to a new deadlock.

2.9 Controllers' description

After explaining, in previous sections, the general purpose of each agent in our system, as well as their controllers' structure, objective, and interactions, we will provide a more detailed explanation of each controller's operations. We will start with the controllers inside an Executor, from the bottom of the controllers' stack to the top layer. Then we will proceed to explain the two controllers in a Task Manager, which have a more intricated design. Each layer usually communicates with the ones directly on top or below itself. Between two controllers, usually the one on top sends commands for the other to perform and the latter answers them reporting the results of those executions. The communication variables used for those exchanges contain the name (or part of the name) of the controller sending the messages, along with a "c" when it sends commands, or an "a" when it sends answers. For example, L1 uses *L1c* for sending commands to L0 and *L0a* to receive its answers. The messages containing these commands and answers can have several elements. The first one is a tag that defines the type of message and the rest, if any, is complementary data. Many of the controllers need to be aware of the pass of time, for that reason they use the *time()* function, which returns the simulation time at that instant. To clear the value of a certain variable the controllers assign it the *nil* value, which corresponds to the null value in Lua programming language. When the data structure to clear is a list or an indexed table, the value assigned is a pair of empty braces ().

2.9.1 Executor's model

Each executor contains 5 controllers, with each one of them attending a different kind of operations. From top to bottom they are the Communications Layer (L4), the Path Planner (L3), the Path Follower (L2), the Traffic Manager (L1), and the Physical Layer (L0).

Physical layer

The bottom controller is the Physical layer or L0, which controls the execution of the vehicle's movements and turns, as well as the detection of physical objects in the environment. L0 only alters the position and orientation of the robot when L1 commands it, specifying how much the agent should turn and advance to reach a target position nearby. When the command arrives, L0 proceeds to change the vehicle orientation and position, in that order, according to the values stated in the command. As it executes a command, it checks the readings from a distance sensor in its front in case an obstacle appears in the way. If this happens, L0 pauses the robot's movement and waits for the obstacle to go away before resuming them. When it finishes the movements, or encounters a persistent obstacle in the way, it reports it to the controller above it, to signal that it is ready to receive a new command.

Figure 2.13 shows the EFS^{2}M diagram for L0's controller. The diagram is color coded for operations' differentiation. The elements in black define the base movement operations, while the purple ones define the obstacle management procedures. The values highlighted in red are the parameters defined by the user that can tune the controller's behavior.

The inputs for this controller are *L1c*, *ang*, *spc*, and *oDist*. Where *L1c* is the command from L1, *ang* and *spc* are the angle turned and distance advanced, respectively, during the current time step, and *oDist* is the distance to the nearest object detected by the distance sensor. On the other side, the outputs for this machine are L0's answers to L1's commands (*L0a*), the angle to turn during the next time step *da*, and the distance to advance *ds* in that same time. L0 also has several internal variables. Regarding rotation, *turned* contains the angle turned during the current command, *aInc* is the angular increment to the orientation for the next time step, and *A* is the total angle to turn during the current command. For linear movement, it has similar variables, *moved* for the distance moved during the current command, *sInc* for the linear increment to the position in the next time step, and *S* for the total distance to travel during the command. The parameters to adjust the controller are the safety distance to avoid collisions (*Dsafe*) and the time to wait before considering an obstacle as persistent (*BD* from blocked deadline).

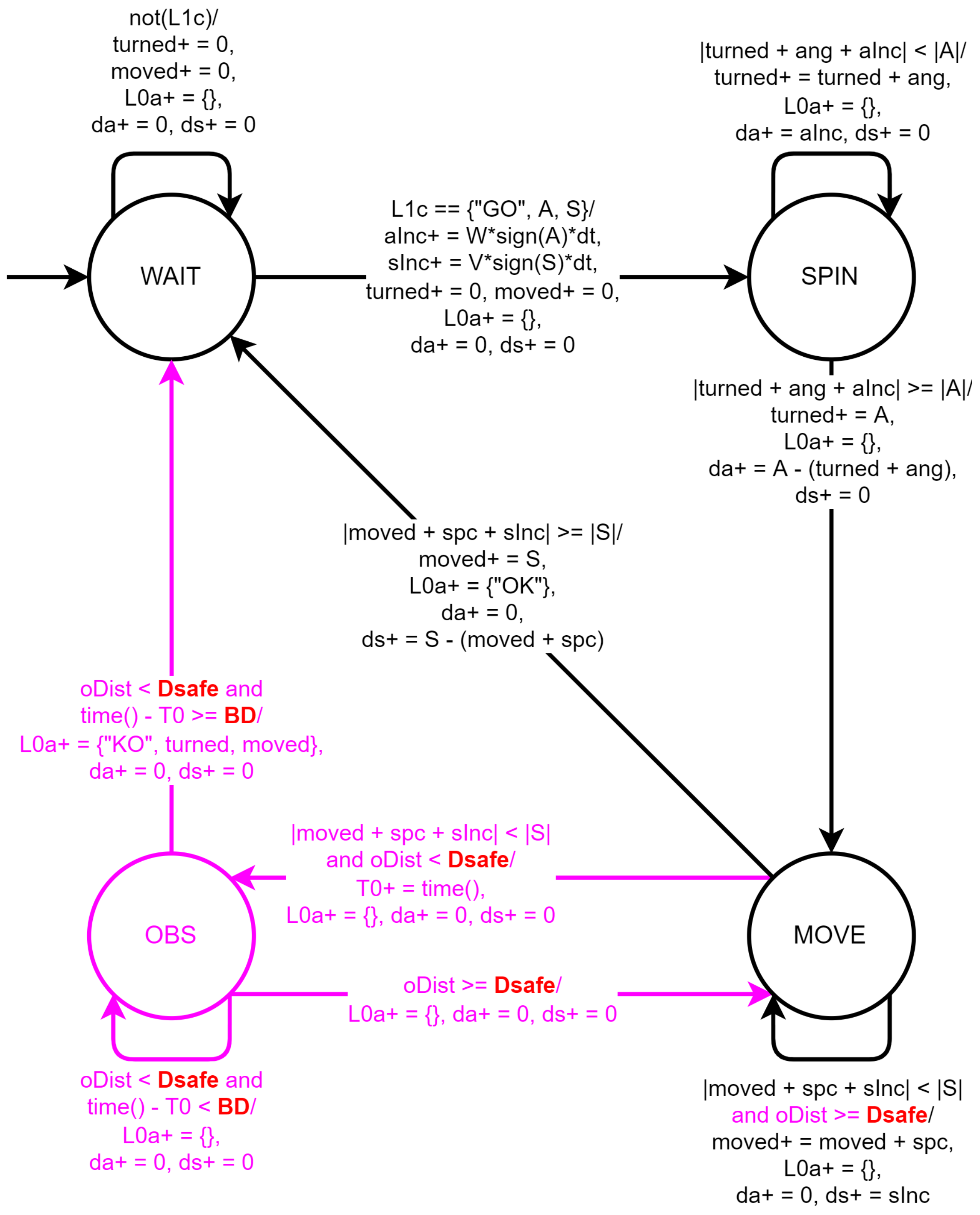


Figure 2.13: Executor's Physical Layer, in EFS^{2}M. Black: movement control. Purple: Obstacle management. Red: Custom parameters

The controller has two predefined constants, which are the angular velocity (*W*) and the linear speed (*V*) of the vehicle. Finally, *dt* contains the size of the time step used in the simulation.

Regarding its base operations, L0 starts them in the state pointed by an arrow coming out of no other state, which in this case is WAIT. While no commands arrive from L1, the machine remains there, maintaining the position and orientation modification outputs (*ds* and *da*respectively) to zero, so the vehicle does not move. When the Executor requires to move, L1 sends a GO command containing the angle to turn *A* and the distance to advance *S*. Upon receiving such command, L0 calculates the angular and linear increments (*aInc* and *sInc*) according to the turning and moving directions, respectively, and goes to the SPIN state.

L0 handles the turn execution by assigning the *aInc* value to the *da* output and incrementing *turned* by the small angle turned at each time step (*ang*) until it reaches the target value (*A*). At this point, L0 puts in *da* the difference between its current orientation and the target one and goes to the MOVE state. Like in the previous state, L0 puts in *ds* the *sInc* value for the vehicle to advance and controls the movement by saving in *moved* the progressive linear increments (*spc*). Once the total distance is closer than *sInc*, it commands the vehicle to advance only the difference between *S* and the advanced distance, notifies L1 of the successful execution with an *OK* answer, and goes to WAIT until the reception of the next command.

For obstacle management, L0 checks the readings from the distance sensor continuously during the linear movements. If an obstacle gets closer than the safety distance (*Dsafe*), it pauses the movement and goes from MOVE to OBS. The machine stays there for as long as the obstacle is present or the *BD* parameter specifies. If the obstacle is no longer close, L0 returns to MOVE to resume the travel. However, if the time limit gets surpassed, the controller reports it to L1, before going to the WAIT state, for receiving a new command that avoids it, if possible. The report has the form of a KO answer, together with the *turned* angle and the *moved* distance.

Traffic manager

The next controller in the stack, from bottom to top, is L1 or Traffic Manager, which ensures that the Executor only moves to locations that are not occupied by other vehicles. So, this layer controls the agent's interaction with the traffic, solving any conflict between several vehicles trying to access the same spot, and helps avoiding collisions in road junctions or roundabouts, where the robots may approach each other from different directions and the frontal sensing for obstacles is insufficient.

All the agents in our MRTA system have a topological description of the layout where they perform their operations. This description is storage as a map divided in cells containing information on the elements inside them, like a port or a road. Each road cell acts as a node with weighted arcs to nearby road cells that can be directly reached from it. The general traffic management in the system establishes that a road cell can only have one vehicle using it. For that reason, each mobile agent must be granted access to a node before traveling towards it. So, before starting to move, an executor needs to request access to the node it plans to go to, and only start its movement when having access. Also, when leaving a node, it must release its access so other vehicles may use it.

Since the focus of our research is the task allocation mechanism, we decided to elaborate simple methods for the mobile agents to deal with objects or other vehicles interrupting the traffic flow. Even when these mechanisms cannot solve every obstruction in a logistics environment, they allow the vehicles to keep performing their operations in most common conflicting situations. For that reason, we made some assumptions to simplify the traffic obstruction problems. The simplification is the following: the logistics environments simulated on our model only contain vehicles that are continuously operating and should not stop indefinitely in a place. This condition allows our system to assume that there will not appear permanent obstacles in the paths. That is to say that any detected obstacle will be a vehicle that will eventually move. Meaning that if re-routing is impossible, the blocked mobile agent only needs to wait for the object to move, to proceed with its path.

Re-routing implies taking a different route from the originally planned to avoid an obstacle, so it requires taking a different arc than the one having the obstruction. The vehicles may change their planned route only when they are over a node with several out-going arcs. So, re-routing depends on when and where the obstacle is detected. If a robot detects obstacles after starting its movement from one node to another, it cannot reroute. Something similar happens if it is over a node with only one outgoing arc, every new route computed will pass through the blocked arc and there is no point in re-routing. In summary, an executor will only attempt to reroute if it encounters an obstacle while being right over a node; otherwise, it will wait for the obstacle to move.

To comply with all these specifications, whenever a mobile agent requires to move to a nearby location, it informs its L1, which calls for access to the corresponding node. After the request, it waits for the node to serve the agent, then releases the access to the node it is currently at, and commands L0 to execute the required movements. Once the bottom controller informs having finished the command, L1 informs this to the controller above itself (L2). To control the access to the nodes, each one of them has a *service* list with the id number of the vehicles requesting access, in chronological order. The entry at the top of the list corresponds to the agent allowed to use the node and the rest are the ones waiting for their turn. To request access, a vehicle puts its id at the bottom of the list and waits to be attended. Once its number reaches the top of the list it can travel to the node and perform other operations, like loading or unloading goods. When leaving the node, the robot deletes its request from the node, granting access to the next vehicle in the list.

In case a goal node remains too much time without attending the Executor, and the current node has connections to other nodes, L1 informs this situation to upper layers for them to find a route that avoids the occupied node, if possible. A similar situation happens if L0 detects a persistent obstacle, however L1 only informs the need for re-routing if the vehicle did not advance, during the current movement command, before encountering the obstacle. The reason for this differentiation is that the routing operations in our system start and end at a node in the topology. So, re-routing a vehicle that already advanced in a certain direction, leaving its starting node behind, would imply that vehicle to go backwards in a road that only admits one traffic flow direction. Since this situation could cause obstructions and collisions between agents, the system avoids it by only re-routing those robots that detect an obstacle before starting to move away from their starting nodes.

Figure 2.14 contains the EFS^{2}M diagram for L1. As with L0's, the diagram is color coded for operations' differentiation. The elements in black describe the base movement operations, the purple ones define the obstacle management procedures, and the values in red are the parameters tune the controller's behavior.

The inputs for this controller are *L2c*, *L0a*, and the value returned by the *served()* function. Following a similar notation to the one in L0's diagram, *L2c* is the command from L2 and *L0a* is the answer from L0. The *served()* function takes as input a node and only returns *true* when the id at the top of that node's service list coincides with the executor's id. On the other side, the outputs for this machine are the answers from L1 to L2 (*L1a*), the *addReq()* function, which adds the executor's id at the bottom of the service list of the designated node, and the *delReq()* function, which deletes said id from the top of the list. The internal variables for this controller are: *A* and *S* for storing the angle to turn and distance to advance, respectively, to reach the target node, *T0* to mark the time at which the controller starts waiting for service, *ori* to save the global orientation of the vehicle, *d* for storing the distance traveled by L0 during a command, and *alt* for retaining if there are possible re-routing alternatives.

L1 starts its operations in the WAIT state, where it remains until L2 sends it the *arc* to travel, inside a GO command. Every *arc* contains two nodes, the starting one (*arc.ini*) and the goal (*arc.fin*). Upon receiving a GO from L2, L1 goes to INIT after passing the *arc* and the vehicle's orientation (*ori*) to the *command()* function to calculate the angle to turn and distance to advance to go from one node to the other. If the movement command requires changes to the agent's position (*S ¿ 0*), the controller adds the executor's id number at the bottom of *arc.fin*'s service list, and goes to CHECK. In this state, the machines verifies if the executor's id is at the top of *arc.fin*'s service list with the *served()* function. When the *arc.fin* node starts attending the executor in question, its L1 controller deletes its service request from the starting node (*arc.ini*), using the *delReq()* function, sends a GO command to L0 (along with the angle to turn and the distance to advance), and goes to the GOING state to wait for the command completion. While at the INIT state, if the movement operations to reach the next node require no change to the position (*S == 0*), there is no need to verify the node service, because the vehicle is already at it. In this case, L1 sends the angle and distance to L0 in a GO command and goes directly to GOING. Upon receiving an OK from L0, when this controller completed the movement command, L1 updates the orientation of the vehicle in *ori*, sends an OK answer to L2, and goes from GOING to WAIT, where it stays until the reception of a new command.

While at CHECK, if the *arc.fin* node does not provide service to the executor for a period greater than the *ND* parameter (not served deadline), L1 goes to OBS after verifying if the *arc.ini* node has direct connections to other nodes using *outs()*. This function returns *true* if from the input node a vehicle

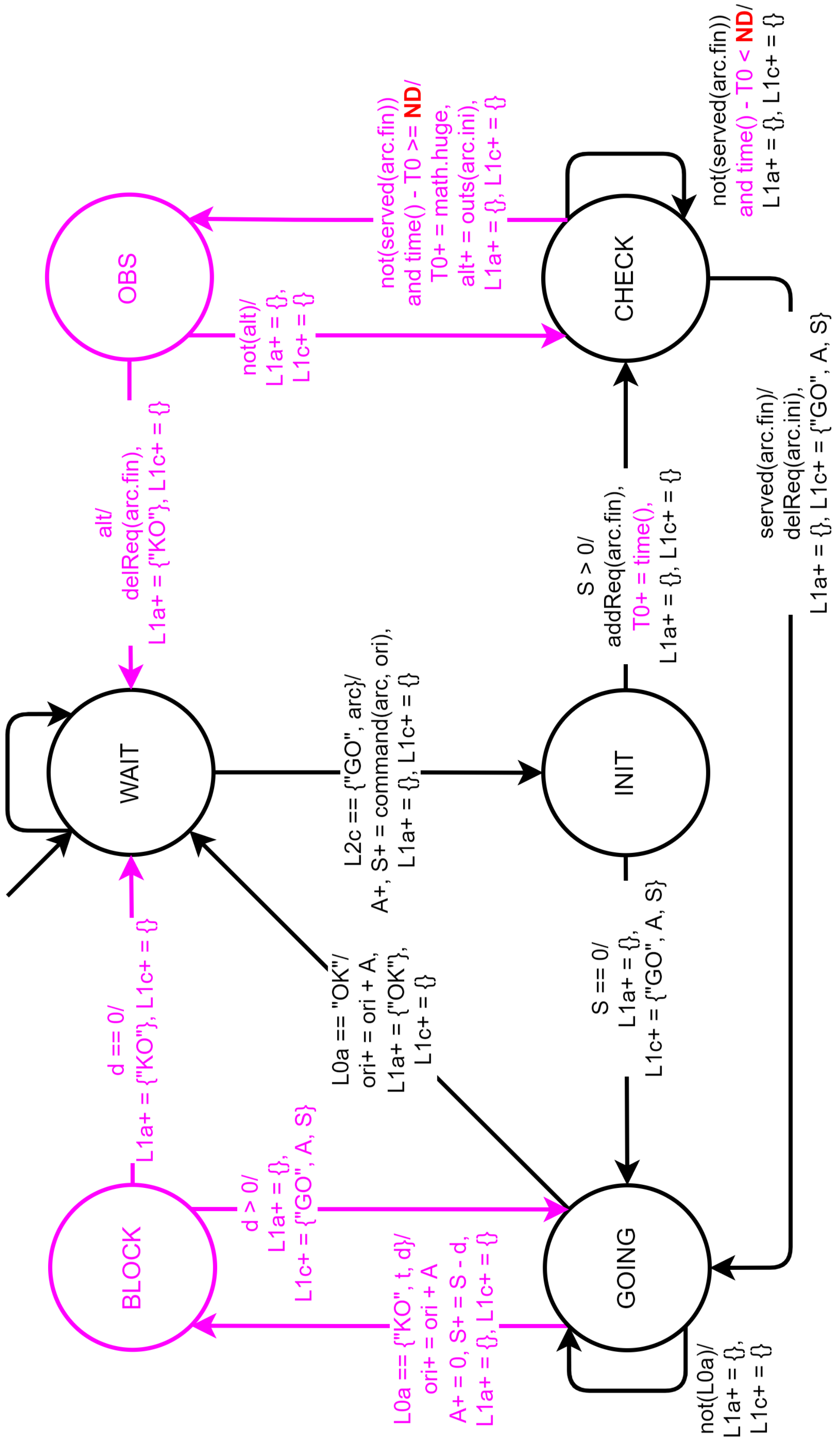

OBS
CHECK
WAIT
INIT
BLOCK
GOING
not(served(arc.fin)) and time() - T0 >= ND/ T0+ = math.huge, alt+ = outs(arc.ini), L1a+ = {}, L1c+ = {}
not(served(arc.fin)) and time() - T0 < ND/ L1a+ = {}, L1c+ = {}
not(alt)/ L1a+ = {}, L1c+ = {}
alt/ delReq(arc.fin), L1a+ = {"KO"}, L1c+ = {}
served(arc.fin)/ delReq(arc.ini), L1a+ = {}, L1c+ = {"GO", A, S}
not(L2c)/ A+ = 0, S+ = 0, L1a+ = {}, L1c+ = {}
S > 0/ addReq(arc.fin), T0+ = time(), L1a+ = {}, L1c+ = {}
L2c == {"GO", arc}/ A+, S+ = command(arc, ori), L1a+ = {}, L1c+ = {}
L0a == "OK"/ ori+ = ori + A, L1a+ = {"OK"}, L1c+ = {}
S == 0/ L1a+ = {}, L1c+ = {"GO", A, S}
d == 0/ L1a+ = {"KO"}, L1c+ = {}
d > 0/ L1a+ = {}, L1c+ = {"GO", A, S}
L0a == {"KO", t, d}/ ori+ = ori + A A+ = 0, S+ = S - d, L1a+ = {}, L1c+ = {}
not(L0a)/ L1a+ = {}, L1c+ = {}

Figure 2.14: Executor's Traffic Manager, in EFS^{2}M. Black: Traffic management. Purple: Obstructions management. Red: Custom parameters

can go directly to more than one other node. This result implies that there is the possibility to find an alternative route, starting from *arc.ini*, that does not pass by *arc.fin*. If there is no possibility for an alternative route (*alt*), the controller returns to CHECK to wait for gaining access to *arc.fin*. However, if *alt* is *true*, L1 deletes its service request from *arc.fin*, sends a KO to L2 for it to find another route, and goes to WAIT for receiving the new arc to travel.

A similar procedure takes place if, during GOING, L0 sends a KO after finding a persistent obstacle. In this case, L1 updates the values of *A* and *S* (to complement the actions carried out by L0) and goes to BLOCK. If the robot did not advance before encountering the obstacle (*d == 0*), there is the possibility for re-routing and L1 sends a KO to L2 for it to re-calculate the route. However, if L0 moved away from the staring node before finding the obstruction (*d ¿ 0*), the re-routing mechanism would need the vehicle to go backwards, to the starting node, which is not allowed by the traffic flow rules. In these cases, L1 sends a new GO command to L0 (with the angle and distance updated for the vehicle to reach the goal node from its current position) and returns to GOING to wait for the movement completion.

Path follower

The Path Follower, or L2, lies directly above L1. This controller receives a full route to follow, from L3, and passes it to the layer below for its step-by-step execution. Each path consists of an ordered list of nodes to visit to go from one location to another. L2 divides them in several arcs that it passes to L1 orderly whenever the latter completes a movement. At the culmination of a travel, the L2 reports this to L3 and waits for a new path to follow. Whenever the executor is not attending a task, it wanders around the environment. To accomplish this, whenever the agent is not busy, L2 selects a nearby node and orders L1 to go there. It keeps repeating this process, every time L1 finishes a movement, until L3 sends it a new route to follow.

This wandering movement around the environment may not depict accurately the operations in a real scenario, where a free agent simply remains idle or returns to a charging station or any other established position. However, given that a dynamic logistics environment receives many unplanned tasks that appear at random times and locations, we considered it could be convenient in some cases to have the mobile agents moving randomly through the environment rather than staying going to a fixed location outside of the main traffic flow area. Nevertheless, we also designed a group of controllers that do not include this wandering behavior. The description of the one corresponding to the L2 layer appears at the end of this section.

If L1 demands a re-routing while the robot is wandering, L2 sends a different neighboring node, if possible. If the same thing happens while following a route, it reports the problem to L3 for it to re-calculate the route, avoiding the troubling location, if possible.

It is possible that during the Executor's travel to pick up a task, L1 cancels it due to delays or finding a better option. In these cases, L2 aborts the route execution and resumes its wandering behavior.

L2's EFS^{2}M diagram appears in Figure 2.15. Like for the previous controllers, the diagram is color coded to differentiate the operations. The elements in black describe the base path following and wandering operations, the purple ones define the re-routing procedures, and the sections in blue correspond to the re-allocation processes.

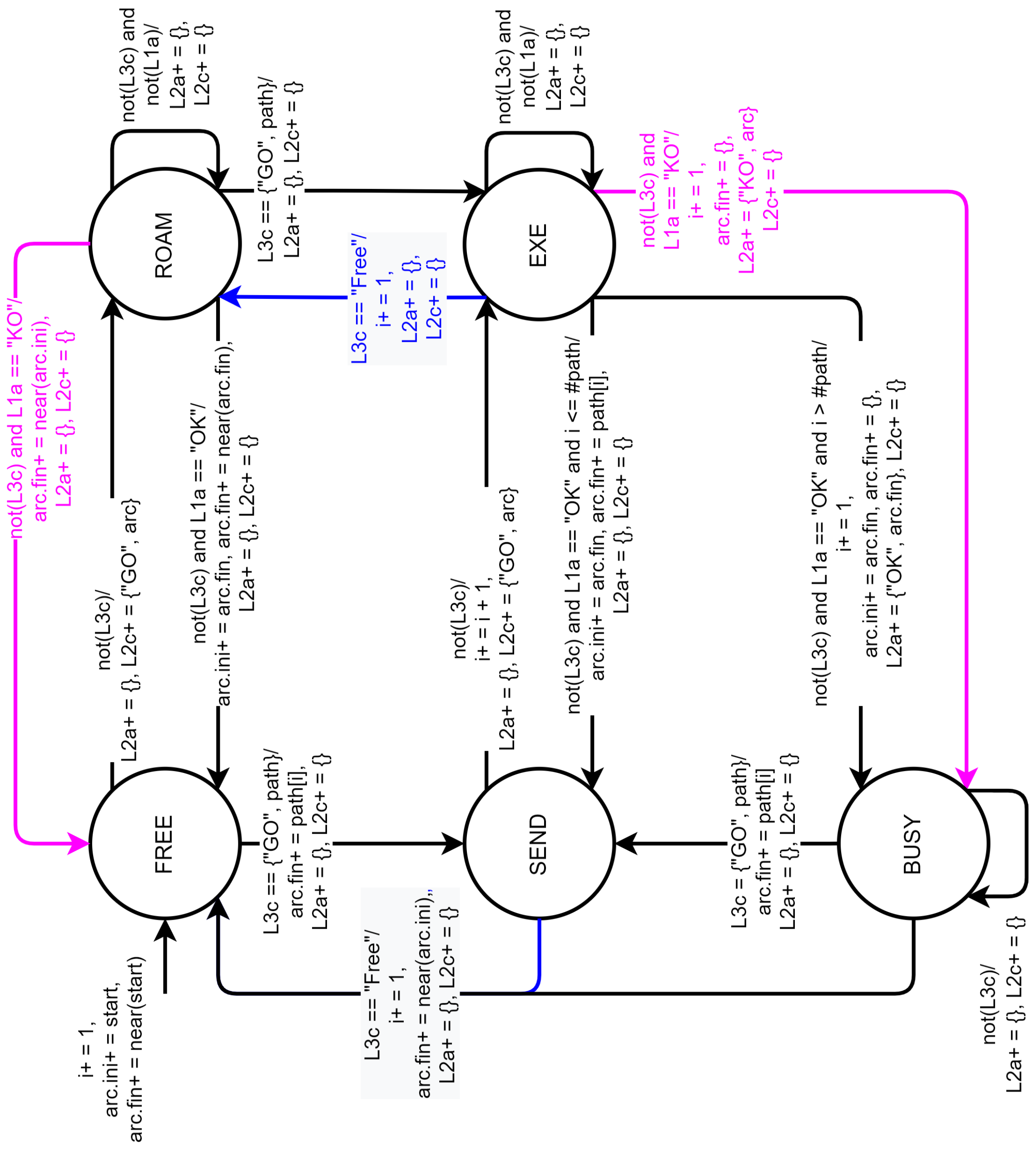

ROAM
EXE
FREE
SEND
BUSY
not(L3c) and not(L1a)/ L2a+ = {}, L2c+ = {}
L3c == {"GO", path}/ L2a+ = {}, L2c+ = {}
not(L3c) and not(L1a)/ L2a+ = {}, L2c+ = {}
not(L3c) and L1a == "KO"/ i+ = 1, arc.fin+ = {}, L2a+ = {"KO", arc} L2c+ = {}
L3c == "Free"/ i+ = 1, L2a+ = {}, L2c+ = {}
not(L3c) and L1a == "KO"/ arc.fin+ = near(arc.ini), L2a+ = {}, L2c+ = {}
not(L3c)/ L2a+ = {}, L2c+ = {"GO", arc}
not(L3c) and L1a == "OK"/ arc.ini+ = arc.fin, arc.fin+ = near(arc.fin), L2a+ = {}, L2c+ = {}
not(L3c)/ i+ = i + 1, L2a+ = {}, L2c+ = {"GO", arc}
not(L3c) and L1a == "OK" and i <= #path/ arc.ini+ = arc.fin, arc.fin+ = path[i], L2a+ = {}, L2c+ = {}
not(L3c) and L1a == "OK" and i > #path/ i+ = 1, arc.ini+ = arc.fin, arc.fin+ = {}, L2a+ = {"OK", arc.fin}, L2c+ = {}
L3c == {"GO", path}/ arc.fin+ = path[i], L2a+ = {}, L2c+ = {}
L3c = {"GO", path}/ arc.fin+ = path[i] L2a+ = {}, L2c+ = {}
i+ = 1, arc.ini+ = start, arc.fin+ = near(start)
L3c == "Free"/ i+ = 1, arc.fin+ = near(arc.ini),, L2a+ = {}, L2c+ = {}
not(L3c)/ L2a+ = {}, L2c+ = {}

Figure 2.15: Executor's Path Follower (with wandering behavior while free) in EFS^{2}M. Black: Base behavior. Purple: Re-routing management. Blue: Reallocation management

This controller has only two inputs, *L3c* and *L1a*, and two outputs *L2c* and *L2a*. Following a similar notation to the other controllers, *L3c* is the command from L3 to L2, *L2c* is the command from L2 to L1, *L1a* is the answer from L1 to L2, and *L2a* is the answer from L2 to L3. As internal variables, this controller has the *arc* to travel (compose by the starting node *arc.ini* and the goal *arc.fin*), *path* for storing the list of nodes to follow, and *i* as a counter to mark the progression of the route execution. The function *near()* returns a random node that can be directly accessed from the input one.

L2 starts its operations at FREE with *arc.ini* containing the starting node for the Executor and *arc.fin* storing a neighboring node. If there is no command from L3, L2 sends L1 a GO command containing such *arc* and goes to ROAM to wait for an answer from the layer below. When L1 informs the command completion with an OK, the machine updates *arc.ini* with the goal node, *arc.fin* with a neighbor to it and returns to FREE. L2 maintains this behavior until L3 sends it a GO command containing the full list of nodes to visit (*path*). If received while at FREE, L2 stores in *arc.fin* the first node of the *path* and goes to SEND. In this state, it sends a GO command to L1 (with the *arc*), increases the counter *i* and goes to EXE to wait for the command culmination. In the case that the GO command from L3 arrives while at ROAM, L2 goes from there to EXE to wait for L1's culmination of the last command sent to it. Whenever L1 completes an execution from the *path*, L2 verifies if that culminates the travel by comparing the command counter *i* to the *path*'s length. If the journey is incomplete, the machine updates the *arc* values accordingly and returns to SEND. When there are no remaining commands to follow, L2 restarts the counter, updates the *arc*, informs L3 with an OK answer (specifying the current node), and goes to BUSY to wait for the next command. If the Executor needs to go to another location, L3 sends a new GO command to L2 and this one starts it execution by returning to SEND. When the travel leads to the culmination of a task, the mobile agent is no longer occupied, and L3 sends a *Free* command to notify it. The reaction from L2 is to store a neighboring node in *arc.fin* and go from BUSY to FREE to start wandering again.

If L1 requests a re-routing while wandering, L2 updates *arc.fin* with a new neighboring node and goes from ROAM to FREE to then send the new *arc* in a GO command. If something similar happens while following a route, L2 informs L3 with a KO answer specifying the *arc* presenting difficulties and goes to BUSY to wait for further instructions.

Regarding the reallocation procedures, a travel cancelation at this level comes in the form of a *Free* command from L3 while L2 is either at SEND or EXE (in the middle of the path's execution). In the first case, since the lower layers are not performing any movements, the controller goes to FREE while storing in *arc.fin* a neighboring node. If the cancelation arrives while the vehicle is moving, L2 goes from EXE to ROAM to wait for the culmination of the movement and then proceed to wander.

Our MRTA system has the option of deploying mobile agents that do not wander when not executing a task. In those executors L2's behavior follows the diagram in Figure 2.16. The machine works very similar to its wanderer variant but in this case has no FREE state and starts its operations at BUSY. Since the culmination of every travel, either by cancellation, obstruction, or completion, ends up in BUSY, we use the states' stack capabilities of EFS^{2}M for returning to that state. When L2 receives a GO command from L3, while at BUSY, it pushes that state into the stack before going to SEND (small circle at the tip of the arrow signaling that transition). This operation ensures that whenever the machine needs to return to BUSY it can simply go to the state at the top of the stack (small hexagon at the tip of an arrow). In this controller there are five different conditions that lead to a transition to BUSY, which we divided in two groups for making the representation easier to follow. If we were to use EFSM for the representation, those five transitions would end up in the BUSY circle along with the starting arrow, the one to remain in the state, and the one to go to SEND. This overcrowding of the area with too many transitions would make the resulting diagram harder to understand.

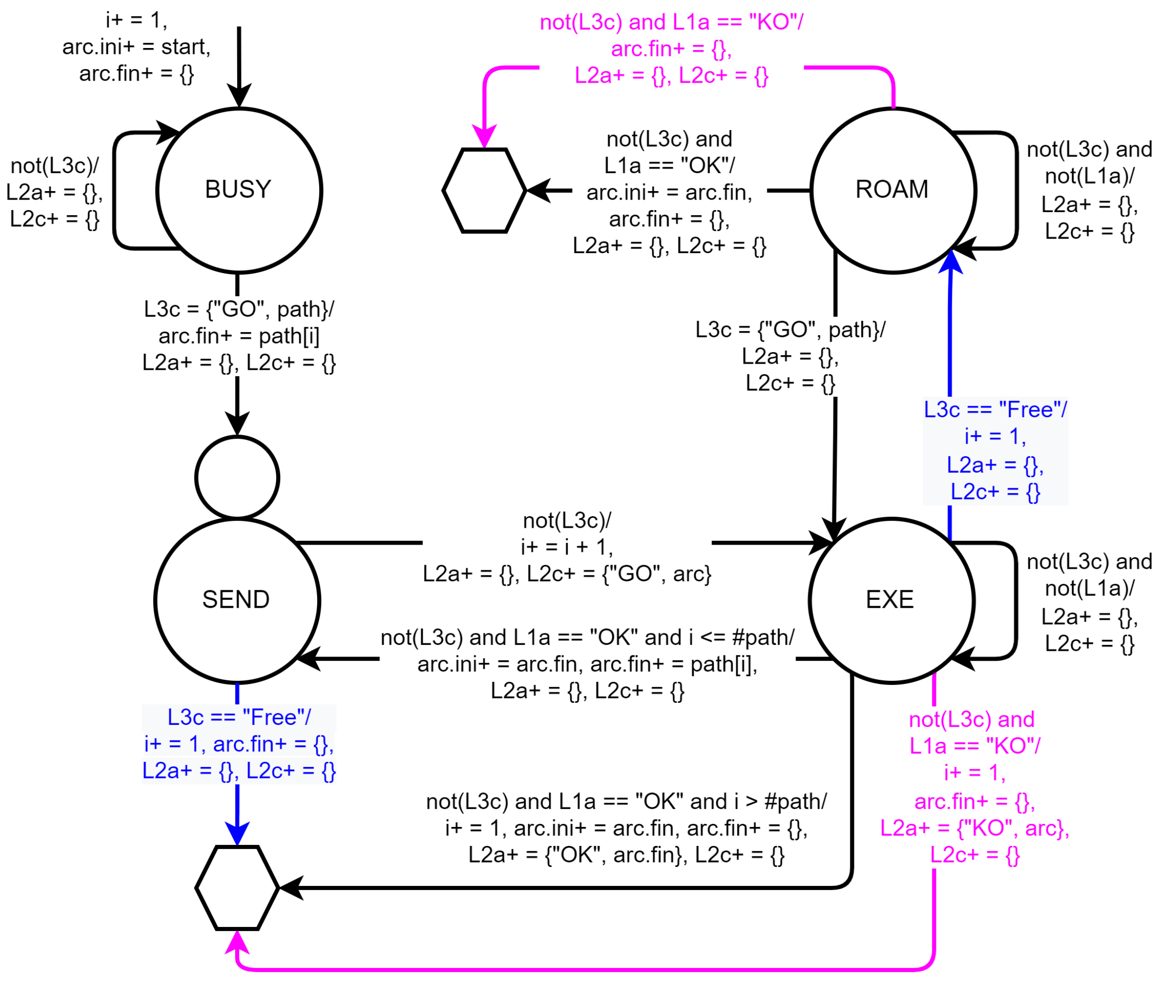


Figure 2.16: Executor's Path Follower (with stationary behavior while free) in EFS^{2}M. Black: Base behavior. Purple: Re-routing management. Blue: Reallocation management

Path planner

The next controller in the list is the Path Planner (L3), which has the purpose of finding the shortest path between two locations while following the imposed traffic rules. Whenever the Executor needs to go to a specific location, L3 calculates the shortest route to it and sends it to L2 for its execution. Once completed, L3 informs L4 and waits for further instructions, which could be a new location to reach or the notification that the mobile agent is no longer has an assignment. If the latter happens at travel completion or during its execution (due to the reallocation mechanism), L3 informs of the situation to L2 for it to follow its behavior while free. In case L2 requests a re-routing, L3 performs the necessary calculations and returns the new path to the layer below.

Figure 2.17 shows L3's EFS^{2}M diagram. Like in previous sections, the diagram is color coded to differentiate the operations. The elements in black describe the path planning operations, the purple ones define the re-routing procedures, and the sections in blue correspond to the re-allocation processes.

Like the one before, this controller has only two inputs, *L4c* and *L2a*, and two outputs *L3c* and *L3a*. *L4c* is the command from L4 to L3, *L3c* is the command from L3 to L2, *L2a* is the answer from L2 to L3, and *L3a* is the answer from L3 to L4. The internal variables for this controller are the *path* to follow, the mobile agent's current node (*pos*), and the *goal* to reach. The *route()* function returns the list of nodes that conform the shortest path from the first input node to the second. It has an optional third input parameter which is a list of nodes to avoid, if possible, during the routing calculations. The shortest route calculations use the Dijkstra algorithm [Cor90].

The L3 machine starts at the FREE state, where it remains while no commands arrive from L4. When L4 sends a GO command containing the *goal* position to reach, L3 calculates the shortest route there using the *route()* function, pushes the current state into the stack, and goes to SEND. For the calculations, the machine gets the node where the agent currently is (*arc.ini*) using L2's function *getPos()*, stores it in *pos*, and uses it to calculate the *path*. In SEND, L3 sends a GO command to L2 containing the *path* to follow and goes to EXE to wait for its completion. When L2 sends an OK message containing the node reached, L3 saves it in *pos*, sends an OK answer to L4, and goes to BUSY to wait for more commands. If the task assigned to the executor requires visiting another location, L4 sends a new GO command to L3 for it to calculate the *path* there and send it to lower layers from the SEND state. When the task finishes, L4 instead sends a *Free* command, L3 transmits it to L2 and goes to the state at the top of the stack, which in this case is FREE.

If during a travel L2 sends a KO answer to report the need for re-routing, L3 uses the *arc* information in the message for the calculations. It uses *arc.ini* as starting point for the route, maintains *goal* as the objective, and uses *arc.fin* as the node to avoid during the calculations. The *route()* function will try to find a path avoiding this last node but, in case that is not possible, it will send a route containing it.

In case that the Task Manager being attended by the Executor cancels the travel, L3 operates similarly as when the task gets completed. Whether it receives a *Free* command while at SEND or EXE, it transmits it to L2 and goes to the state at the top of the top of the stack, FREE.

This L3's programming is intended for working with the wanderer version of L2 explained previously. As soon as the Executor is released from a task, L2 starts to make the vehicle wander and L3 just waits for the next target position to reach. However, when using the stationary version of L2, it is recommended that the vehicle has some mechanism that prevents it from remaining idle in the middle of the environment indefinitely. For that reason, we created another version of L3 that reacts when the

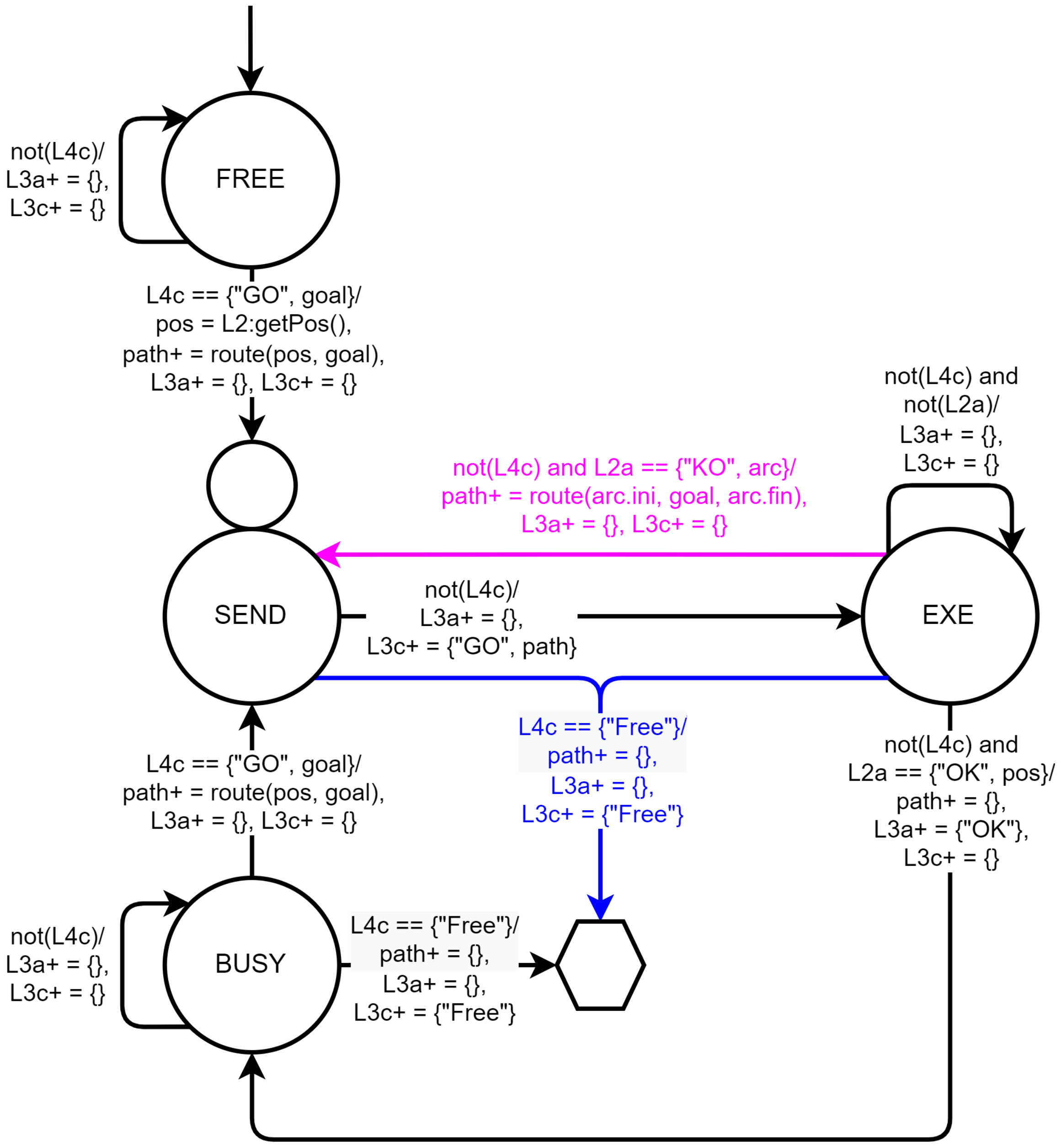


Figure 2.17: Executor's Path Planner in EFS^{2}M. Black: Planning behavior. Purple: Re-routing management. Blue: Reallocation management

Executor is not busy sending it to a travel to its starting location. Such location could be a place where the mobile agent is not blocking the path for other vehicles or a charging station for the robot to recharge its batteries. The EFS^{2}M description of the L3 controller with this behavior appears in Figure 2.18.

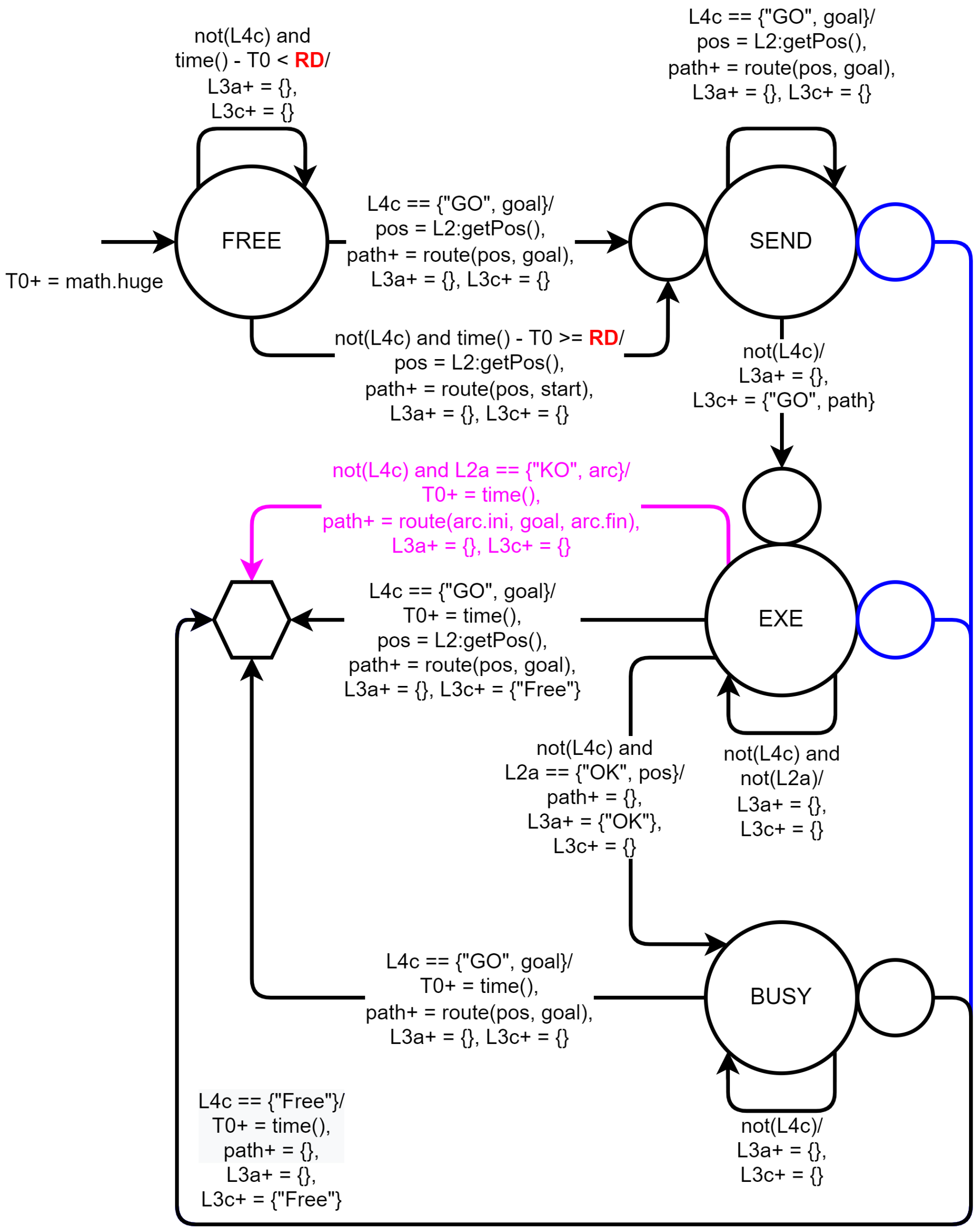


Figure 2.18: Executor's Path Planner in EFS^{2}M (with returning mechanism while free). Black: Planning behavior. Purple: Re-routing management. Blue: Reallocation management

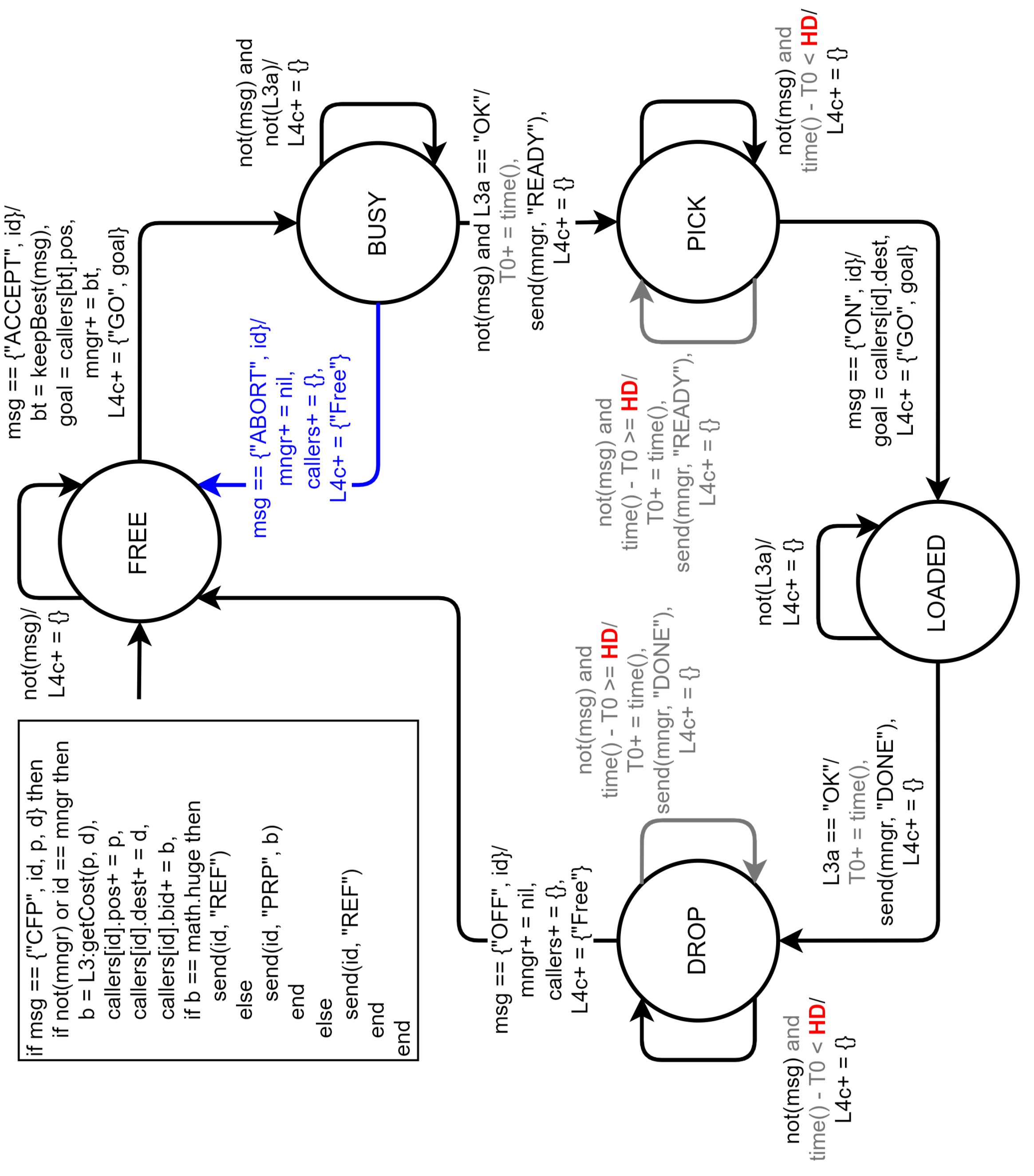
The controller has the same states as the previous version and behaves quite similarly. The main difference is that when L3 is at FREE it measures how much time it remains there. If the time surpasses the preestablished value in RD (return deadline parameter), L3 calculates the *path* to its starting location (*start*), pushes the current state into the stack, and goes to SEND to command the vehicle to follow the route. This modification implies that now a GO command from L4 could arrive at any point during the execution of the returning route. For that reason, all states have new behaviors to react appropriately to such event. In this variant, there are many conditions that require a transition to the SEND state. For that reason, the machine pushes SEND into the stack before going from there to EXE. That way, whenever it requires to return there, it simply goes to the state at the top of the stack. Since the controller uses the same mechanism to return to FREE, which now is not on the top of the stack, the transitions returning to that state require to make a pop operation (small circle at the base of an arrow) first, to remove the SEND state from the stack, and then go to the state at the top, which will be FREE.

Communications layer

The controller at the top of the stack is the Communications Layer or L4, which interacts with the Task Managers for participating in their auctions and during the different stages of the task completion process. When the executor is not attending any task and receives a call-for-proposals (CFP) message from a Task Manager, L4 answers it with a proposal (PRP) containing its bidding value for the auction. If the mobile agent wins, the controller sends the task's starting position to L3 for it to calculate the route and for the lower layers to execute it. Once there, L4 informs the Task Manager with a READY message, which answers with an ON when the loading operations finished. At this point, L4 commands L3 to go to the task's destination and, upon arrival, notifies its counterpart with a DONE message. The Task Manager unloads the goods from the vehicle and sends an OFF when finishes. This signals L4 that the executor is free, and it informs so to the rest of the controllers. While the executor is attending a task, it will only participate in the auctions launched by its corresponding Manager. In case the Task Manager cancels the travel with an ABORT message, L4 notifies this decision to the rest of controllers, for the agent to perform the operations determined by its behavior while not attending tasks.

Figure 2.19 shows L4's EFS^{2}M diagram. Like before, the diagram is color coded to differentiate the operations. The elements in black describe the base operations to fulfill a task, the sections presented in blue define to the re-allocation procedures, and the ones in gray involve communications redundancy operations.

This controller has one output to send commands to L3 (*L4c*) and one input to receive the answers from it (*L3a*). It also has a message exchange mechanism to communicate with the other agents in the system, where every message contains the sender's id number, the receiver's id number, a performative indicating the nature of the communication, and any complementary data required (e.g., the bidding value inside a PRP message). Its internal variables are the id of the task it is attending (*mngr*) and a table containing the information corresponding to all the *callers* that contacted the mobile agent since the last time it stopped attending a task. The *callers* table associates the task's id number (*id*) with its staring position (*pos*), destination (*des*), and the *bid* presented in its last auction. As for the custom parameters, L4 only has *HD* (handling deadline), which determines how much time the controller should wait until



```
if msg == {"CFP", id, p, d} then
  if not(mngr) or id == mngr then
    b = L3:getCost(p, d),
    callers[id].pos+ = p,
    callers[id].dest+ = d,
    callers[id].bid+ = b,
    if b == math.huge then
      send(id, "REF")
    else
      send(id, "PRP", b)
    end
  else
    send(id, "REF")
  end
end
```

Figure 2.19: Executor's Communications Layer in EFS^{2}M. Black: Task execution. Blue: Reallocation management. Gray: Communications redundancy. Red: Custom parameters

assuming there is a communication error during the loading and unloading operations and resend its last message. The *send()* function takes as inputs a list of id numbers corresponding to the receivers, a performative, and a table containing complementary data. With that information it elaborates a message and sends it to each agent in the list.

Since many Task Managers operate parallelly in these environments, it is usual that the available Executors participate in more than one auction at the same time. This situation could lead to a mobile agent becoming the winner in more than one auction. In those situations, L4 uses the *keepBest()* function to determine which task to perform. This function compares the auctioneers that declared the Executor as winner (those that sent an ACCEPT message) by the bidding values presented by the mobile agent in their corresponding auctions. The function returns the id of the task with the smallest bid (lowest cost) and sends a FAIL message to the rest of the involved Task Managers, for them to look for another vehicle.

The L4 controller starts in the FREE state, where it remains until winning an auction. In case of receiving a CFP from a Task Manager (containing its position p and destination d), it answers it with a PRP message with its bidding value (b). The value in b corresponds to the total distance that the mobile agent would have to travel to complete the task. For the computation, L4 uses L3's function *getCost()*, which calculates the shortest path from the vehicle's position to p and from there to d, and returns the sum of both distances. If L3 finds no possible route, because the Task Manager and the Executor are in separated areas within the environment, it returns the value *math.huge*, which corresponds to infinite in Lua. In these cases, instead of sending a PRP message to the Task Manager, L4 refuses the call with a REF message.

When the robot receives an ACCEPT message, it means that it won the auction corresponding to the Manager sending it. Whether it only receives one or many, it uses the *keepBest()* function to obtain the id number of the costless one. It saves the id number in *mngr*, gets its position from the *callers* table, and sends it to L3 in a GO command for the vehicle to go there. Then it goes to wait at BUSY until L3 answers with an OK, indicating the end of the travel. At this point, L4 sends a READY message to the Task Manager and goes to PICK. When all the goods are inside the vehicle, the Executor receives an ON message. L4's response is to send a GO command to L3 containing the task's destination as *goal* and go to LOADED. Once the robot reaches the destination, L4 sends a DONE message to the Task Manager and goes to DROP. The latter answers with an OFF message after finishing the unloading operations. When this happens, L4 clears the *mngr* variable, clears the *callers* table, informs L3 that the robot is *Free*, and returns to the FREE state.

If during the Executor's travel to the Task Manager's position, the latter cancels the order with an ABORT message, L4 behaves similarly to when completing a task: it clears both the *mngr* variable and the *callers* table, informs L3 that the robot is *Free*, and goes to the FREE state. If the controller receives a CFP while attending a task, it verifies if it comes from its assigned Task Manager. If it does, it calculates its new bidding value and sends it in a PRP message. In other cases, it simply refuses the call by sending a REF.

In case there are any communications' errors between the agents, the reallocation mechanism helps to overcome them during the auctions' operations. However, the rest of the task execution also relies on message exchange and requires some procedures to reinforce those communications. For that reason, whenever L4 is waiting solely for its assigned task manager to finish some operations (namely, during the loading and unloading operations), it measures how much time it spends in the corresponding states (PICK and DROP). If the time waiting surpasses the *HD* value, the controller repeats the message that triggers the operations that the task manager should be doing already (READY to start the loading operations and DONE to begin the unloading ones). This procedure is repeated until the task manager confirms that it finished the corresponding operations.

2.9.2 Task Manager's model

In our distributed model, each task manager contains two controllers: the Deliberative Controller (DLB) on top, for deliberation and planning, and the Reactive Controller (RCT) at the bottom, for responding to any external event. Due to the various actions these agents must perform while managing the execution of their corresponding tasks, the resulting controllers contain several states and transitions between them. The EFS^{2}M diagrams modeling their behavior were color coded to ease the identification of the different protocols in the state machines. The states, transitions, and operations related with the Primary Auctions, and the completion of the transport order present a black coloring. The green sections manage the absence of available executors during an auction, while the orange elements represent the procedures to follow when the mobile agent takes too much time to retrieve the goods. The blue portions depict the operations to perform Secondary Auctions and the elements in red show the parameters that can adjust the task manager behavior.

The controllers' diagrams in this section describe the reallocation mechanism including two triggering methods. One of them is reactive, launching the auctions and re-auctions when the task managers are notified that there is a change in the allocation scenery (e.g., an executor becomes available to perform a new task); while the other entails that the agents autonomously determine when to launch a new auction, having a more active role in the process. While both methods can be used together, the active part would be a communications redundancy over the reactive, since the latter already launches the auctions when there is a change in the assignments, hence a possibility for reassignment. So, it is recommended to use them separately. The reactive mode has the advantage of trying to allocate or reallocate the active tasks only when there is a real chance of changing the assignment. However, since it relies more on the communications between the agents, it is more susceptible to its failures. Also, since all unattended or waiting tasks launch their auctions every time there is a change, the communications distribution along the time happens in discrete bursts that may saturate the communications system if the number of tasks is too high. On the other side, the active mode depends heavily on the established time separation between auctions. If too distant, the mobile agents may remain too much time idle, while waiting for a new call, and if too close they may saturate the communications system unnecessarily. In any case, this method is more robust before communication errors, since each manager repeats its auctions autonomously, and produces a more uniform communications distribution along the time.

Deliberative controller

The DLB handles all the communications that take place during the execution of its corresponding task. It starts by calling all available executors to participate in the task's Primary Auction. Then, it collects all the incoming bids, chooses the closest vehicle available (the one with the smallest cost) as the winner, and informs it of the auction's results. When the mobile agent notifies its arrival, DLB commands RCT to start the loading operations, and once these are complete it tells the robot to start the travel towards the destination. Upon arrival, DLB notifies RCT for it to unload the goods from the vehicle. After these procedures, the DLB releases the executor from the task, and remains idle.

Thanks to the re-allocation mechanism, the DLB may re-auction its task while it is waiting for a mobile agent to pick the goods up. This method can be triggered by a message announcing a change in the allocation scene or because the RCT commands it. In any case, the DLB launches a Secondary Auction where it only announces the winner if its different from the currently assigned executor. If during any of the auctions there are no mobile agents available, the controller remains unassigned until the RCT commands it to launch a new auction. The task managers also have a mechanism that prevents it from getting stuck waiting for a vehicle that may not arrive. While the DLB is waiting for the arrival of the mobile agent, if the RCT determines that the robot is never going to come, it commands DLB to dump the assigned executor and start a new Primary Auction.

Figure 2.20 shows the EFS^{2}M diagram for DLB. The diagram is color coded for operations' differentiation. The elements in black define the base task managing operations, the blue ones describe the re-auction procedures, the green components manage the executors' unavailability, the ones in orange deal with the vehicles' delays, and the gray sections correspond to the communications redundancy procedures.

This controller has one input to receive commands from RCT (*Rc*) and one output to send its answers to it (*Da*). It also has a message exchange mechanism to communicate with other agents in the environment. The controller's internal variables are: *exec*, where it keeps the id number of its assigned executor; a cycle counter (*i*); *best*, to store the winner of an auction; and *prps*, a list containing all the proposals received during an auction. The controller uses the *clear_prps()* function to empty the proposals' list whenever its going to start a new auction. The *add_prp()* function exams the incoming messages and if any of them is a PRP, saves the bidding value and the executor's id in the *prps* list. The *send()* function takes as inputs a list of id numbers corresponding to the receivers, a performative, and a table containing complementary data. With that information it elaborates a message and sends it to each agent in the list.

The *eval()* function is the core of the manager's auctions. It evaluates all received proposals and returns the id number of the executor with the lowest bidding value. In the case of a tie between two or

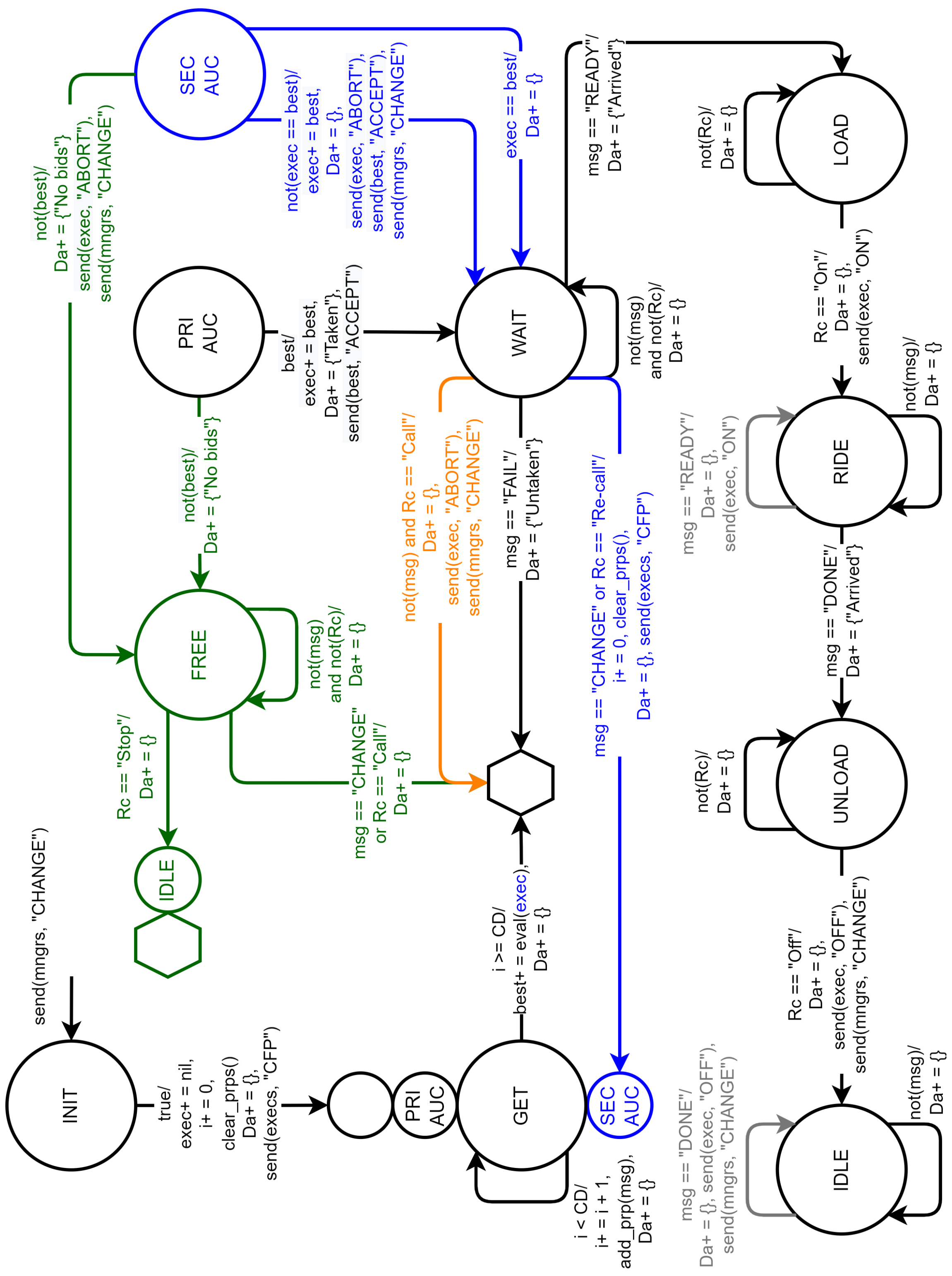

INIT
send(mngrs, "CHANGE")
true/ exec+ = nil, i+ = 0, clear_prps() Da+ = {}, send(execs, "CFP")
PRI AUC
GET
SEC AUC
i < CD/ i+ = i + 1, add_prp(msg), Da+ = {}
i >= CD/ best+ = eval(exec), Da+ = {}
IDLE
FREE
Rc == "Stop"/ Da+ = {}
not(msg) and not(Rc)/ Da+ = {}
msg == "CHANGE" or Rc == "Call"/ Da+ = {}
PRI AUC
not(best)/ Da+ = {"No bids"}
best/ exec+ = best, Da+ = {"Taken"}, send(best, "ACCEPT")
SEC AUC
not(best)/ Da+ = {"No bids"} send(exec, "ABORT"), send(mngrs, "CHANGE")
not(exec == best)/ exec+ = best, Da+ = {}, send(exec, "ABORT"), send(best, "ACCEPT"), send(mngrs, "CHANGE")
exec == best/ Da+ = {}
WAIT
not(msg) and Rc == "Call"/ Da+ = {}, send(exec, "ABORT"), send(mngrs, "CHANGE")
msg == "FAIL"/ Da+ = {"Untaken"}
msg == "CHANGE" or Rc == "Re-call"/ i+ = 0, clear_prps(), Da+ = {}, send(execs, "CFP")
not(msg) and not(Rc)/ Da+ = {}
msg == "READY"/ Da+ = {"Arrived"}
LOAD
not(Rc)/ Da+ = {}
Rc == "On"/ Da+ = {}, send(exec, "ON")
RIDE
msg == "READY"/ Da+ = {}, send(exec, "ON")
not(msg)/ Da+ = {}
msg == "DONE"/ Da+ = {"Arrived"}
UNLOAD
not(Rc)/ Da+ = {}
Rc == "Off"/ Da+ = {}, send(exec, "OFF"), send(mngrs, "CHANGE")
IDLE
msg == "DONE"/ Da+ = {}, send(exec, "OFF"), send(mngrs, "CHANGE")
not(msg)/ Da+ = {}

Figure 2.20: Task Manager's Deliberative Controller in EFS^{2}M. Black: Primary Auctions and task management. Green: Executors' unavailability management. Orange: Executors' delay management. Blue: Secondary auctions management. Gray: Communications redundancy

more mobile agents, it selects one at random as the winner. This function accepts as input value the id number of the currently assigned executor (for performing Secondary Auctions). Whether the input is a valid number or *nil* (which means that the task manager does not have a mobile agent assigned), the function still returns the id of the closest vehicle. However, if there is a tie and the inserted id is among the winners, the function returns that same value rather than a random one from the winners' list. In case that no executor answered the call and there are no proposals to evaluate, the function returns *nil*.

The controller has one constant value, which is the number of cycles to remain collecting proposals (*CD*, from collect deadline). In this case the process progression is measured in cycles because we assume that the deliberative process is instantaneous. So, every auction, from the starting call to the winner's announcement takes places inside the internal operations loop of the time model (section 2.7), without time progression. For that reason, *CD* needs to have a value greater or equal to the number of cycles required for the mobile agents to react to the call and reply their bidding values. It would be interesting to design a Deliberative Controller for a model where these processes are not instantaneous. In those cases, *CD* could be a variable parameter that would determine how much time the task managers provide the executors to react to their calls. In those cases, a greater time window could increment the number of proposals received by a task manager, which is an advantage similar to the one provided by the repetitive execution of Secondary Auctions.

The DLB starts at INIT, but first, it announces its going to launch an auction by sending a CHANGE message to all active task managers. This should cause them to also launch new auctions if they are unassigned or waiting for their corresponding vehicles' arrival. Then, DLB proceeds to clear its *prps* list, erase the id number of the assigned vehicle (which at the start is none), send a CFP to all available executors in the area, push the current state and the PRI_AUC one into the stack (in that order), and go to GET. The controller stays there, for as many cycles as *CD* establishes, using the *add_prp()* function to save the information from any incoming PRP message. Once the collecting finishes, it saves the *best* candidate for executing the task and goes to the state at the top of the stack, which in this case is PRI_AUC. If there is a winner, DLB informs RCT that the task is *Taken*, communicates the auction's result to the winning agent with an ACCEPT message, and saves its id number. After that, DLB goes to WAIT until the mobile agent announces its arrival with a READY message. When this happens, DLB informs RCT that the robot *Arrived*, so the task manager's bottom controller starts the loading operations. Meanwhile, DLB waits in the LOAD state until RCT confirms that everything is inside the vehicle with an *On*. DLB then send an ON message to the executor and goes to RIDE until the end of the travel to the destination. At this point, the mobile agent sends a DONE to DLB and this one sends an *Arrived* to RCT before going to UNLOAD. Once all the goods get moved to the destination port DLB receives an *Off* from RCT, and its reaction is to release the executor with an OFF message, announce such release to other task managers with a CHANGE message, and go to the IDLE state, where it does not perform any other operations. If the mobile agent determines that it is unable to complete the task before arriving to the task's starting position, DLB receives a FAIL message while a t WAIT. In this case, it informs RCT that the task was

Untaken and goes to the state at the top of the stack, INIT, to launch a new Primary Auction.

If during its time at WAIT, DLB receives a CHANGE message from other task manager or a *Re-call* command from RCT, it means that it should launch a Secondary Auction. To do so, it clears its *prps* list, broadcasts a CFP to all available mobile agents, pushes SEC_AUC state into the stack, and goes to GET to collect proposals. After the collecting, it goes to the state in the top of the stack, which now is SEC_AUC. The difference between a Primary and a Secondary Auctions is that in the latter, if the *best* candidate is the executor already carrying out the task, the controller goes from SEC_AUC to WAIT without performing any operations. However, if the winner is another agent, DLB saves its id, informs it of the result with an ACCEPT message, releases the previously assigned vehicle with an ABORT, and announces all task managers that there is a free vehicle, with a CHANGE message.

In case that during any of the auctions there were no executors available, the *eval()* function returns a *nil* value. When this happens during a Primary Auction, DLB reports it to RCT with a *No bids* answer and goes to the FREE state to wait for a while, before launching a new Primary Auction. This happens when DLB receives a CHANGE message from another task manager or a *Call* command from RCT. In both cases the controller goes to the state at the top of the stack, which is INIT. However, if RCT determines that the executors unavailability is not going to change, it may send an *Stop* command to DLB for it to refrain from launching any other auctions. In this situation, DLB pushes the IDLE state into the stack and then goes to the state at the top of the stack, which results in going to IDLE and leave the task as incomplete. This mechanism is not designed to be part of the allocation solution since it leads the system to leave tasks unattended. Its intention is to highlight if the logistics system is not working as intended. For example, let us assume that a user wants to design and configure a logistics system where the tasks managers should not have to wait more than a certain amount of time to be attended. Having a fixed time separation between Primary Auctions, this requirement could be met if the task managers do not need to perform more than x Primary Auctions. So, setting $AL = x$ during the simulations will show the percentage of tasks that required a higher amount of auctions. This result allows the user to know what configurations comply with its requirements and which ones do not.

If the task manager launches a Secondary Auction (which implies that it already has a dedicated executor) and does not receive any proposals, it means that there is some error in the communications or that the mobile agent had a fatal failure that was not able to communicate. In any case, DLB assumes that the assigned vehicle is not coming so it cancels that travel by sending an ABORT message to the executor. It also informs RCT with a *No bids* answer and the rest of task managers with a CHANGE message. Then, the controller goes to the FREE state to wait before launching a new auction. It is important to note that if the assigned executor is not able to participate in a Secondary Auction, but other mobile agents send their proposals, the re-auction winner will be a different robot and the re-allocation mechanism will change the assignment to it without needing the initial mobile agent to report that it is no longer capable to reach the task manager's position.

The final mechanism in this controller is the one that prevents the task manager from waiting indefinitely for a mobile agent that may never arrive. When the RCT controller determines that the assigned executor has taken so long to arrive that it can be considered as unable to do so, this controller sends a *Call* command to DLB, which is at WAIT state. When this happens, DLB sends an ABORT to the corresponding mobile agent, informs the release to all task managers with a CHANGE message, and goes to the state at the top of the stack, INIT, to launch a new Primary Auction.

While the reallocation mechanism increases the communications' robustness of the logistics system during the auctions' execution, the task's completion process also needs some procedures to reinforce the communications that take place during those phases. If the executor does not receive the ON message sent after the loading operations, it will wait for a while before sending a new READY message to the task manager. In this case, DLB will receive this message while at the RIDE state and, since the products are already inside the vehicle, it will simply reply with an ON message for the vehicle to start its travel towards the destination. A similar mechanism is used at the IDLE state in case the executor has troubles receiving the OFF message. Given that the goods are already out of the mobile agent, DLB simply sends an OFF message to robot and informs the rest of the managers with a CHANGE message.

Reactive controller

The Reactive Controller or RCT is the part of the task manager that deals responds to any external event that involves its corresponding task. It starts its operations waiting for the DLB to inform the results of its first Primary Auction. When the task gets a vehicle assigned, RCT starts measuring the time the task manager remains at that stage. When the mobile agent arrives, RCT loads the goods in it so the travel to the destination port can begin. Once there, it moves all the products to the from the vehicle to the port and finishes its operations. If the executor takes too much time to pick the task manager up, RCT can tell DLB to launch a re-auction or even to release said vehicle and launch a new Primary Auction. In the case that there are no mobile agents available to perform the task, RCT determines how much time to wait before launching a new auction, so that the executors finish their current assignments and become available.

The EFS^{2}M diagram for RCT appears in Figure 2.21. The operations are differentiated by color. The elements in black dictate the base procedures, the blue ones define the re-auction operations, the green elements manage the executors' unavailability during auctions, and the orange elements correspond to the management of vehicles' delays. The red values are the parameters to tune the controller's operations.

This controller has only one input and one output. It sends commands to DLB through *Rc* and gets its answers through *Da*. Its internal variables are: *T0* and *T1* to measure the time at different stages, and *aucs* to count the number of auctions carried out. As for the adjustable parameters, this controller is, by far, the one with the highest count in our model. It has *PD* (primary deadline) to control the time separation between Primary Auctions, *SD* (secondary deadline) to do the same for the Secondary ones, *AD* (arrival deadline) to limit the time waiting for the vehicle the pick the products up, and *AL* (auctions

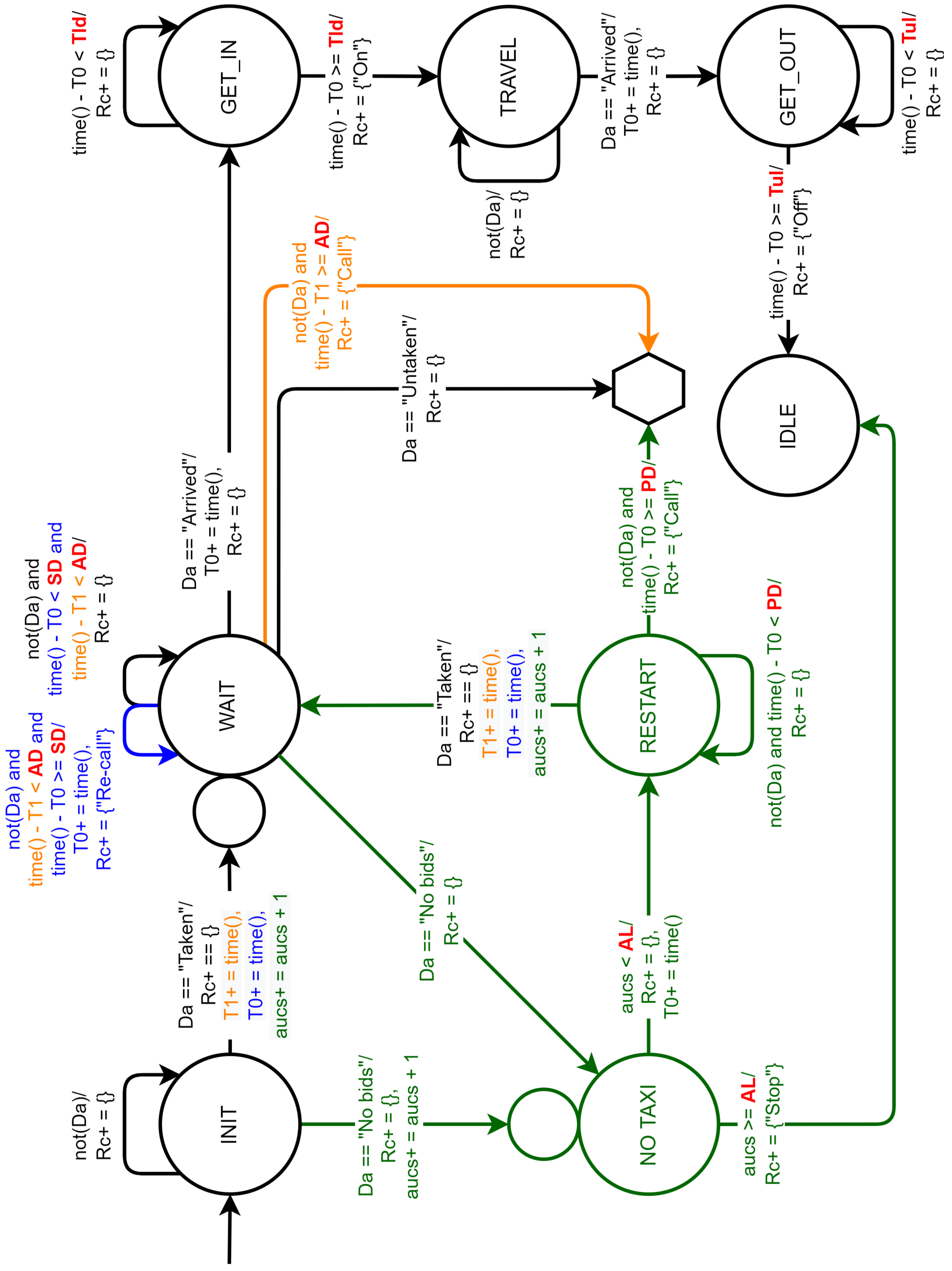

INIT
not(Da)/ Rc+ = {}
Da == "Taken"/ Rc+ == {} T1+ = time(), T0+ = time(), aucs+ = aucs + 1
WAIT
not(Da) and time() - T1 < AD and time() - T0 >= SD/ T0+ = time(), Rc+ = {"Re-call"}
not(Da) and time() - T0 < SD and time() - T1 < AD/ Rc+ = {}
Da == "Arrived"/ T0+ = time(), Rc+ = {}
GET_IN
time() - T0 < TId/ Rc+ = {}
time() - T0 >= TId/ Rc+ = {"On"}
TRAVEL
not(Da)/ Rc+ = {}
Da == "Arrived"/ T0+ = time(), Rc+ = {}
GET_OUT
time() - T0 < TuI/ Rc+ = {}
time() - T0 >= TuI/ Rc+ = {"Off"}
IDLE
not(Da) and time() - T1 >= AD/ Rc+ = {"Call"}
Da == "Untaken"/ Rc+ = {}
not(Da) and time() - T0 >= PD/ Rc+ = {"Call"}
Da == "Taken"/ Rc+ == {} T1+ = time(), T0+ = time(), aucs+ = aucs + 1
RESTART
not(Da) and time() - T0 < PD/ Rc+ = {}
Da == "No bids"/ Rc+ = {}
aucs < AL/ Rc+ = {}, T0+ = time()
Da == "No bids"/ Rc+ = {}, aucs+ = aucs + 1
NO TAXI
aucs >= AL/ Rc+ = {"Stop"}

Figure 2.21: Task Manager's Reactive Controller in EFS^{2}M. Black: Task management. Green: Executors' unavailability management. Orange: Executors' delay management. Blue: Secondary auctions management. Red: Custom parameters

limit) to prevent the task manager from performing auctions indefinitely when there are no available mobile agents to carry out the task. Finally, we have *Tld* and *Tul* for loading time and unloading time, respectively, which determine how much time the task manager dedicates to getting the goods into the vehicle and out of it. In a real case, this would be determined by the physical operations of moving the products from one place to another. But, for simplicity, our model simply waits for a while. However, it would be simple to replace those parameters with a couple of functions that started those operations and informed the controller once they are completed. The values of all these parameters can strongly affect the overall performance of the logistics system. For example, *SD* determines how often the task managers will launch Secondary Auctions. If it is too low, the re-auctions will be too close to each other, and the busy executors will not have had enough time to complete their current assignments and become available. This situation would lead to an unnecessary increase in the number of messages exchanged without a visible reduction in the time to complete a task or the distance traveled in the process. On the other side, a high *SD* value will diminish the number of messages sent, but will limit the exploitation of the Secondary Auction's capabilities.

To perform its base operations, RCT starts at the INIT state, where it remains until DLB shares the auction's results. When the task is *Taken*, RCT saves the current time in *T0* and *T1*, increases the auctions counter (*aucs*), pushes the INIT state in the stack, and goes to WAIT. Once in there, if the executor is not able to perform the task it informs DLB, and this one sends an *Untaken* answer to RCT. In this case, the latter goes back to INIT (the state at the top of the stack) while DLB launches a new Primary Auction. If, on the contrary, the mobile agent does reach the task manager's location, RCT receives an *Arrived* from DLB, saves the time in *T0*, and goes to the GET_IN state. It remains there for as long as the loading operations last and then informs DLB with an *On* before going to TRAVEL. During this state, the controller simply waits for DLB to report that they *Arrived* to the destination. Once this happens, it goes to GET_OUT and stays there while the unloading operations take place. When all the products are on the destination port it sends an *Off* to DLB and goes to IDLE to culminate its operations.

If the time staying at WAIT surpasses the *SD* value, RCT stays in the same state but sends a *Re-call* command to DLB and updates the value at *T0*. In case the time spent in this state surpasses the value in *AD*, RCT sends a *Call* command to DLB (for it to start a Primary Auction again) and goes to the state at the top of the stack, which is INIT.

When the task manager receives no proposals during a Primary Auction, RCT receives a *No bids* answer while at INIT state. In this case, it increases the auctions counter, pushes INIT into the stack and goes to NO_TAXI state. If the executors' unavailability occurs during a Secondary Auction, the machine receives the *No bids* signal while at WAIT. When this happens, it will simply go to NO_TAXI because it already increased the auctions counter and pushed INIT into the stack when going from that state to WAIT. In NO_TAXI, RCT checks if the number of Primary Auctions carried out surpasses the established limit (*AL*). If it does, the controller sends a *Stop* command to DLB and goes to IDLE state to finish its operations. While the limit amount is not reached, RCT saves the current simulation time in *T0* and goes to RESTART state to wait the amount of time determined by *PD*. When the wait is over, it sends a *Call* command to DLB (so it starts a new Primary Auction) and goes to the state at the top of the stack, which is INIT. If during this wait DLB launches an auction (triggered by a CHANGE message from another task manager), and the task gets assigned to an executor, it will notify RCT with a *Taken* value in the answer channel. In this case, RCT will increase the auctions counter, save the simulation time in *T0* and *T1*, and go from RESTART state to WAIT.

Chapter 3

Operational scenarios and performance evaluation

3.1 Execution Scenarios

One of the base elements our system needs to run, besides a set of transport tasks and a set of mobile agents, is an environment to place them. Since our target is to model an internal logistics system, the environmental layouts should be similar to warehouses, workshops, and such. Ideally, they would replicate existing designs for more realistic representations.

3.1.1 Input method

Our program obtains the layout characteristics by reading a plain text file (*.txt*) with the environment description. Figure 3.1 shows an example of the contents of such a file. For its outline, the layout is divided into cells with different characteristics according to the elements in that area. A cell can be a transit zone (a road), a no-traffic area, or a port (for loading and unloading operations), among others. The text file contains a group of alphanumeric characters representing those cells according to their characteristics. For example, to describe a road cell where the traffic flows upward, we use an **‘A’** as a simple representation of a bolt pointing up. By the same logic, a downwards road is a **‘V’**, a leftwards one is a **‘<’**, and one flowing rightwards is a **‘>’**. To denote docks our system uses **‘@’** characters, and to indicate a no-traffic area it can use a **‘#’** or a **‘.’** (a point). The difference between both cells is purely aesthetic: a **‘#’** cell will have a cube-shaped landmark, while the **‘.’** one will not.

Right after the grid, the file contains a list with the ports’ names and destinations. The first element in each item is the symbol ‘@’ and a number for its position in the ports list. The second element is the port name and the rest are its destinations. In Figure 3.1 the first port is called ‘A’ and has port ‘C’ as its destination. The destinations list can have more than one element, all separated by blank spaces. This notation informs the program that every task originating in port A should go to port C. When a port has

```
######
#V<<<#
#V@@A#
#V.@A#
#>>>A#
######
@1 A C
@2 B
@3 C
```

Figure 3.1: Input text file content for layout description.

an empty destinations list (like ports B and C), the destination of any task that starts from there is not restricted to any port subgroup.

Our program uses this data to generate a map with more details. The map consists of a data structure with explicit information extracted from the distribution of the cells in the text file. The model saves the information from each cell in patches and arranges them by their coordinates on the map. Every **patch** contains data regarding itself and its surroundings. For example, every port is a no-traffic cell, so they require at least one gate, a cell able for traffic, where the mobile agents stop to perform the load and unload operations in said port. So, every port patch marks every road patch adjacent to itself as one of its gates. Based on these data, the system generates a graphical representation of the layout, like the one in Figure 3.2.

Regarding traffic flow, each road patch checks its surroundings to determine the possible directions in which a vehicle can go from it to adjacent road patches. These directions, or exits, allow the mobile agents to know where they can go without breaking the established traffic rules. In general, every road patch will have an outway pointing to an adjacent patch if the latter is also a road and going in that direction does not oppose the traffic flow of either patch. For example, imagine a patch pointing upwards and establishing its exits. It will have one pointing up if the upper patch is a road aiming anywhere but down. If there is a road to its right and it is not pointing leftwards, our patch will also have an exit pointing to the right. The opposite happens with the patch to the left. Finally, there will be no exit going down since it would oppose the traffic flow of the patch itself.

The built map structure is also the base to estimate the minimum distance between two locations.

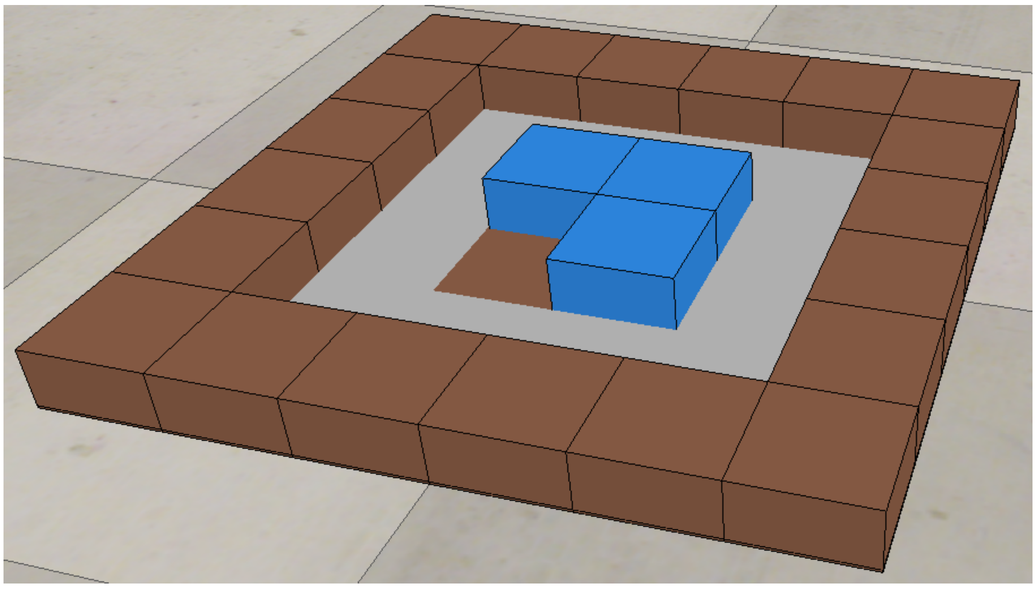


Figure 3.2: Graphic representation of the layout described in Figure 3.1.

Our system makes those calculations with the Dijkstra algorithm [Cor90], which uses a network of arcs and nodes to determine the shortest route between two nodes. Since all road patches contain information on which other patches a vehicle can visit from them, they serve as the base for the Dijkstra algorithm to work.

To validate our system capabilities, we created several environments. The layouts had diverse characteristics, depending on the focus of our tests.

3.1.2 Small square layout

The first test scenario we built mimics a warehouse with a Manhattan distribution, as shown in Figure 3.3. It contains four blocks, in a 2-by-2 distribution, with bidirectional roads on their sides. Each block has a port on its most external corner (represented in blue) for pick-up and drop-off operations. Each port has gates on each adjacent road (represented by black dots), where the mobile agents stand to interact with the port.

A map with these characteristics allowed us to perform quick tests to check the behavior of each one of the system's components. Regarding traffic flow, its several types of crossing bidirectional roads allowed us to check how the mobile agents executed basic maneuvers like going forward, turning to each side, and reacting to obstacles or vehicles approaching from different angles to avoid collisions. Concerning route calculations, the four ports with two gates each produce various possible combinations for going from one port to another. This characteristic allows us to assess the elaboration of all possible routes and the election of the shortest.

We also used this environment to quickly evaluate how modifying a parameter impacted the overall

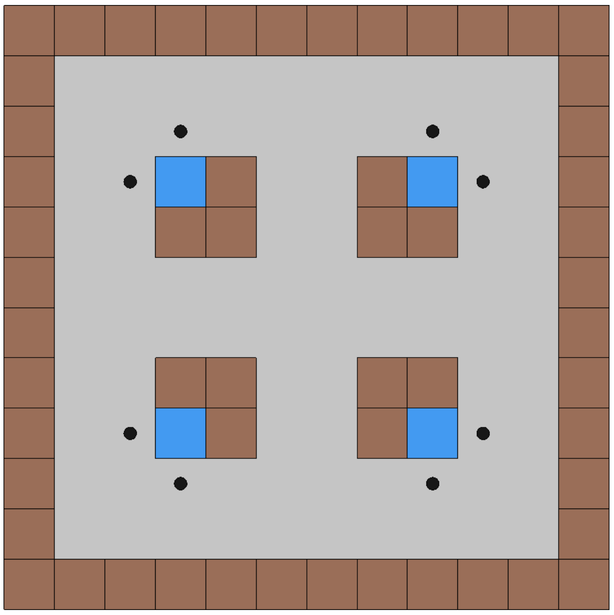


Figure 3.3: Graphic representation of our small square layout.

system performance. For example, we assessed the time to complete all tasks increment due to communication errors during the auctions in [Riv23].

3.1.3 Small linear layout

Besides testing basic operations, we needed to verify that our simulation model provides rational solutions to the MRTA problem. To that end, we designed an environment simple enough that it is feasible to calculate all possible allocation solutions manually. With that data, it is possible to assess how close the solutions provided by the model are to the optimal ones. Figure 3.4 shows a graphical representation of the layout created. The three upper ports function as charging stations, which means that their gates are the starting positions for the mobile agents. The four ports at the bottom part function as pick-up and drop-off points. Namely, the leftmost port is the delivery point for all goods, and the other three ports are the retrieval points.

All the roads in the environment are unidirectional, with the general traffic flow going clockwise. To that end, the leftmost vertical street goes up while the rest go down, the top horizontal channel goes right, and the one on the bottom goes to the left.

This distribution provides the layout with three convenient characteristics for our test:

- All tasks have only one path for their execution.
- Every task cost is different.
- No mobile agent is closer to any of the tasks than the others at the start of the operations.

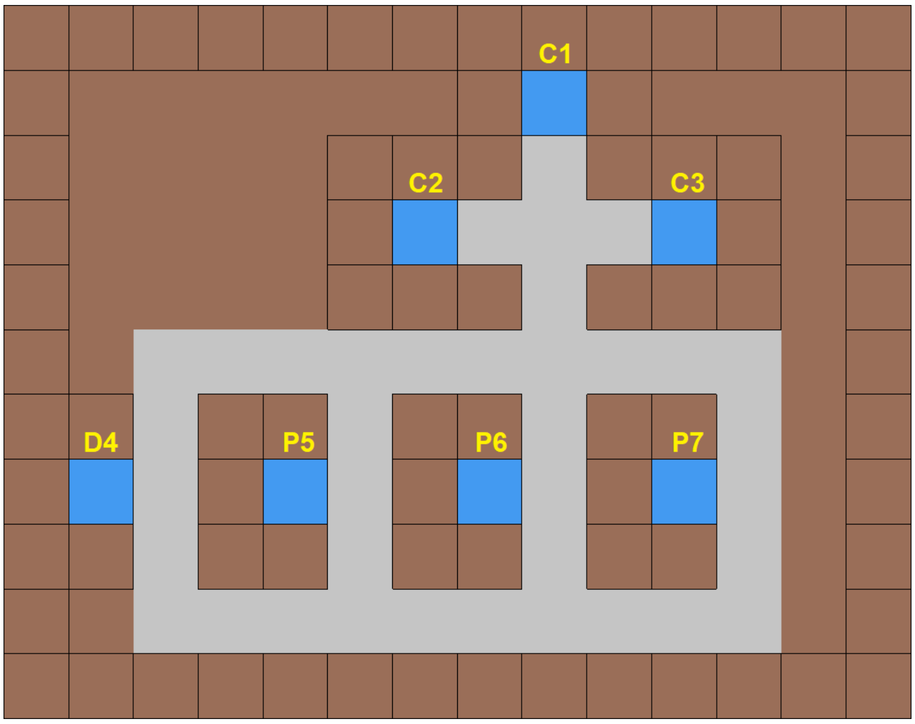


Figure 3.4: Graphic representation of our small linear layout.

The fact that there is only one path for executing each task makes it easier to calculate their cost manually. Since all tasks' costs are different, there is no ambiguity in which one is more appropriate for an agent to execute at any point. Finally, given that no agent has a cost advantage over the others at the start, they are all equally suitable to perform each one of the tasks.

We designed a Mixed Integer Linear Programming model to obtain optimal or near-optimal solutions to the MRTA problem. The objective of this model was to assess the quality of the solutions found by our physical simulation approach. This new model's capabilities needed testing and validation, which was also one of the reasons we created this small linear layout.

3.1.4 Realistic warehouse

Once all base features verification was complete, we had to test our model's performance when dealing with realistic situations. To that end, we designed a new setting based on the warehouses described in [Tan21]. In general, those setups had parking areas for the AGVs to remain while unused, a storage area containing the racks to transport, and a picking area with several stations where the racks were to be taken to and later retrieved.

In our design, the starting ports for mobile agents are in the upper right corner. The storage area contains 18 groups of six racks distributed in a grid of three rows and six columns. There are nine picking stations, three on the left side of the map and six at the bottom. All roads are unidirectional; inside the

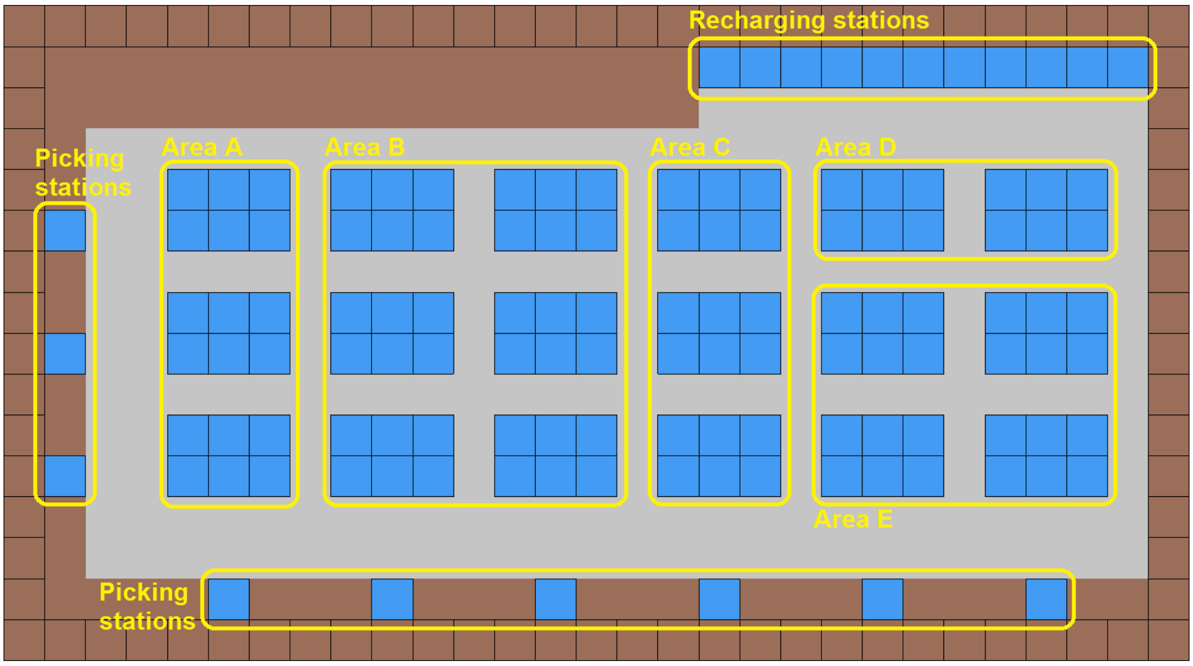


Figure 3.5: Graphic representation of our realistic warehouse layout.

storage area, they only admit one vehicle at a time, while in the picking and charging areas, they are wide enough to admit up to two mobile agents at a time. The storage area contains five sections (named from A to E) with different task arrival rates between them. Figure 3.5 shows the graphical representation of the environment.

3.1.5 Realistic workshop

Since our system intends to provide allocation solutions for internal logistics problems in warehouses and factories, evaluating its performance in those scenarios is essential. The main difference between warehouse and factory layouts is that the latter tend to have more linear road grids with fewer crossroads, which reduces the route options for going from one location to another. Also, the transport tasks tend to be more sequential and dependent on each other. The reason is that usually, the goods must receive several treatments or operations in a sequence. Even if some items can skip one or another step, the transportation tasks keep having a well-defined flow direction. Inside a warehouse, however, the traffic flow is more irregular.

For testing, we recreated a semiautomated workshop from a company that has worked with members of our research group. Figure 3.6 shows how our program interprets the workshop. The environment has two separate groups of roads, meaning that not all mobile agents can attend to all tasks. All streets are unidirectional and only allow one vehicle at a time. Both areas have several loop roads with few intersections between them, which means that there are few options when choosing the path to go from one point to another.

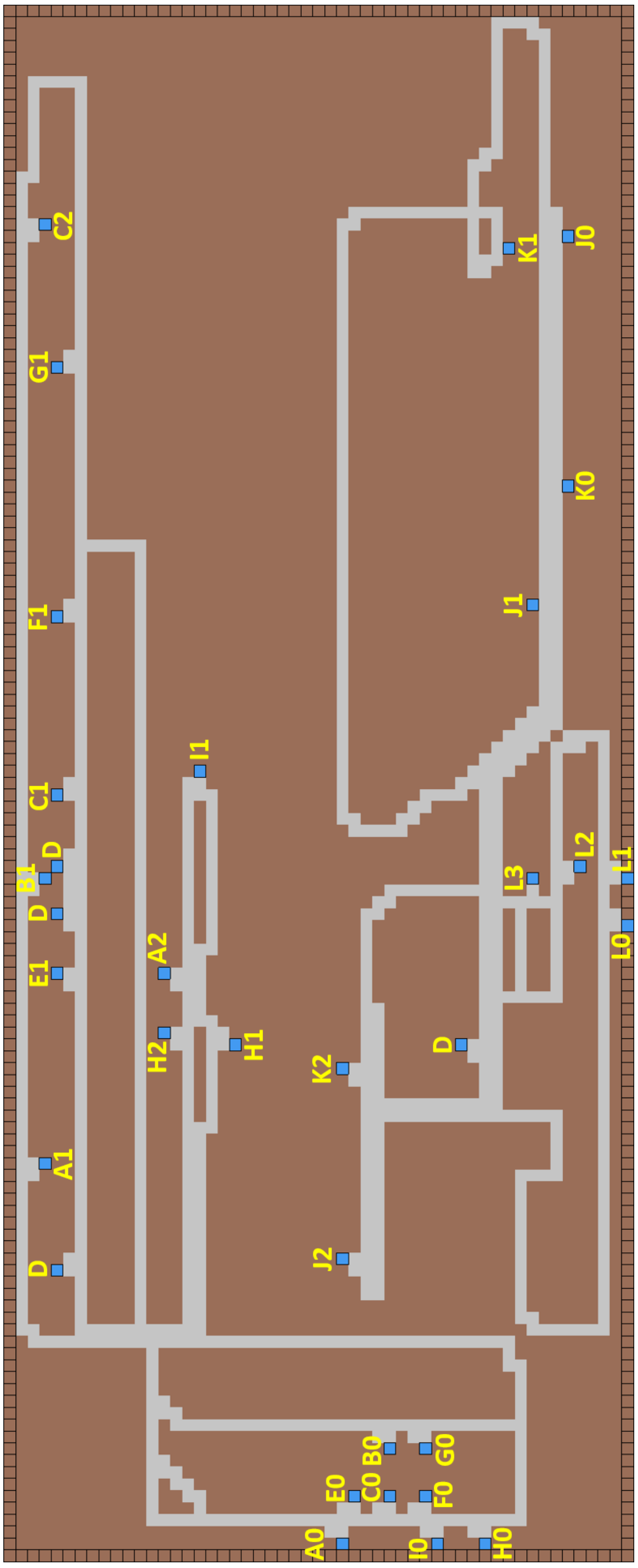


Figure 3.6: Graphic representation of a realistic factory workshop layout.

For clarity, the names of the ports appear near their locations. The letter indicates the group to which they belong and the number, the sequence that the tasks must follow. For example, all operations related to group **'A'** start at port **'A0'**, the storage or retrieval port for that group. The first transport task will go from **'A0'** to **'A1'**, where the transported item receives some treatment. Once the operations finish, a new transport task appears, from **'A1'** to **'A2'**, the final destination for the product. Ports with the **'D'** tag are not used.

3.2 Performance characterization

3.2.1 State of the art

A critical part of managing a logistics system consists in evaluating its execution. A good performance can have different meanings depending on the application and the user's interests. For example, the focus of a system that mainly receives urgent tasks, like distributing medicines or food supplies to survivors after a catastrophic event, will be on minimizing the time required to perform each task. However, when dealing with a system that distributes non-essential goods inside a warehouse, there may be more interest in a slower performance that minimizes the distances the robots must travel to perform their tasks. For that reason, many studies employ different metrics to evaluate the performance of their systems. In most cases, to measure the system efficiency by identifying the number of resources required to perform a single task, a group, or all the tasks received. What changes from one approach to another is the resource they focus on primarily.

Some studies measure the consumption of those resources with an objective monetary value, like the **fuel, battery, or energy consumption** during the system operations [See20,Sch17,Kim12]. In [Kim12], they propose to evaluate the resource consumption per task by dividing the *total resource consumption* by the *number of completed tasks*. However, these measurements are only possible with a very detailed system model or a physical system to perform the tests and readings since this consumption depends on the physical characteristics of the mobile agents, the scenario, and the elements to carry. It is not always possible to carry out these experiments with a physical system since it would have to stop performing its regular operations while the tests last. That is why most approaches quantify energy consumption indirectly by recording the time consumed by each task or the distance traveled to carry it out and inferring the energy consumption. Equation (3.1) shows an example of how to do so, where the energy dedicated to an operation (E_{op}) is equivalent to multiplying the *time spent in that operation* (T_{op}) by an *energy factor* defined for it (F_{op}). Even though that is not a precise measurement of resource consumption, it helps to compare the system performance in different configurations.

$$E_{op} = T_{op} * F_{op} \tag{3.1}$$

Since all logistics applications are usually time-sensitive, most performance evaluations take time into account. Their usual objective is to find solutions that minimize the time consumption of the operations they measure. An intuitive way to evaluate the whole system is by measuring the **time required to complete all given tasks**, which is why it is a metric used in many approaches [Tep22, Ott19, See20, Sar18b, Sch18, Sch17]. This metric is suited for systems that receive transportation tasks in clusters, which the system must complete before the next batch arrives. However, in systems that operate continuously, like a warehouse of a major distribution company, it makes no sense to talk about the total number of given tasks since there is no limit on the amount they can receive. In those cases, it is preferable to measure the **average time required to complete a task**, which explicitly indicates the resource consumption per task, regardless of the total amount [See20, Sar18a, Zha16].

Given that all transportation tasks consist in picking up an item at a location and dropping it off at another, we can divide the time to complete an assignment into two parts: one while the goods are waiting for pick-up and the other while they are traveling to their destination. In systems where all the mobile agents are equal, there is no way of reducing the traveling time through task allocation. Since the fleet is homogeneous and the starting in finishing points are invariable, the traveling time will be the same regardless of the assigned agent. The waiting time, however, does depend on the vehicle's starting position and is directly affected by the task allocation. For that reason, some approaches focus on measuring the **average waiting time for each task** rather than the time to complete the whole task [Tur18a, Che18, Tur18b].

Another popular metric to evaluate a task allocation solution is the *total distance traveled by all the agents to complete all the tasks*. Some authors call this measurement **solution cost** [Ott19, See20, Sar18b, Sch17, Zha16] since it can be linked directly to the energy consumption of the solution.

Even though the consensus is to reduce the traveled distance, there are several approaches on the way to accomplish it. Some studies focus on diminishing the *maximum distance traveled by any agent* [Ott19, See20, Sch17, Zha16]. They call this metric **solution quality**, and its minimization ensures a good task distribution among the agents, preventing one of them from getting overcharged, which may cause a faster deterioration of the robots.

As in the case of measuring time consumption, there are situations where it is more convenient to measure the **distance traveled per task** rather than the total distance traveled by all the agents [Whi19, Zha16]. In [Sch17], they minimize the **assignment distance spread** (*DS*) to ensure a good task distribution. To calculate it, they subtract the *minimum distance of an agent to any of its assigned tasks* ($min(tasks_{dist})$) from the *maximum distance of that agent to any of its assigned tasks* ($max(tasks_{dist})$), equation (3.2).

$$DS = max(tasks_{dist}) - min(tasks_{dist}) \tag{3.2}$$

While the logistics systems in warehousing and production are likely to function continuously, there are other applications where these systems work on well-defined and restrictive time windows. Especially in those cases, researchers focus on maximizing the **number of completed tasks** [Ott19, See20, Kim12]. For a matter of perspective, when comparing different scenarios or configurations, some studies calculate the **task completion ratio**, which is the division of the *number of completed tasks* by the *total number of tasks received* [See20, Kim12]. Another way to measure the workload balance among the agents is to divide the *maximum number of tasks completed by any agent* by the *total amount of tasks completed*, equation (3.3), which is called **task distribution rate** (*TD*) [See20].

$$TD = \frac{max(tasks_{agent})}{tasks_{total}} \tag{3.3}$$

A critical element for designing and configuring a logistics system is how many agents to employ. Purchasing and maintaining mobile agents increases the operations cost of any logistics system. Also, having more agents in the work environment can increase the number of traffic conflicts and make the system more prone to congestion, which deteriorates its overall performance. That is why many studies look to minimize the **number of agents** to use in the solution [See20, Li17]. A way to establish the best number of agents is by calculating the **average agent utilization rate**. This rate corresponds to the fraction of *time that the agents are attending to their tasks* from the *whole time they were active*. This value ranges from 0 to 1, and users usually prefer allocation solutions that maximize it to ensure that they are getting the most from the deployed agents.

Since time is crucial when dealing with emergencies, many approaches focus on how quickly the allocation mechanism finds the best solution to a given situation. An intuitive way to determine the system response speed while a single entity carries out the decision process is to measure **how much time the controller consumes to calculate the solution** [Ott19].

When the mechanism depends on a group of controllers that must deliberate and find the solution together, there are several approaches to determine how fast it is. If the dialogues are cyclical until all controllers converge to the same conclusion, many authors measure the number of iterations required to reach such convergence [See20, Tur18a, Sch18, Tur18b]. In cases where the algorithm allows task reassignment to better-suited agents than those previously assigned, they employ two metrics. The first is counting the **number of reassignments**, and the second is dividing that number by the *number of completed tasks* to obtain the **reassignment rate** [Ott19].

Finally, many authors use several formulas to determine how close a given solution is to a known optimal solution for a specific case. Some divide their *solution cost* by the *optimal solution cost* to calculate the **cost ratio** (*CR*), equation (3.4) [Sar18b]. Since the objective is cost minimization and the optimal solution is the global minimum, the goal is to minimize this parameter's value as close to 1 as possible.

$$CR = \frac{cost_{solution}}{cost_{optimal}} \tag{3.4}$$

Other studies use metrics focused on the difference between the values obtained by them and the optimal ones. For example, [Tep22] calculates the **objective gap** (*OG*) for a specific metric by subtracting the *optimal value* from *the one they got* and dividing that amount by their *obtained value*, equation 3.5.

$$value_{OG} = \frac{value_{obtained} - value_{optimal}}{value_{obtained}} \tag{3.5}$$

Many approaches compare themselves with known optimal solutions because, while they cannot best the optimized metric value, they may present better values in other areas. For instance, a system that offers a solution with a near-to-optimum traveled distance for all agents may be more appealing than a system that delivers an optimal one if the first can calculate its solution many times faster than the latter.

3.2.2 Solution data extraction and analysis

Our system performs most of the measurements we found in the bibliography and others of our own for describing less common factors like traffic congestion. This capability, combined with the option to show the detailed allocation solution in writing, increases our tool's usefulness when evaluating and comparing different setups according to the user's priorities and requirements.

The executions data extraction comes from three different sources: the tasks' managers, the executors, and the global environment. At the end of each run, the execution system compiles all the data collected during the simulation.

From the task managers, the model saves the **assignments completion sequence**, linked to their **id number** for clear differentiation. Each **task completion status** (completed or not) is recorded for **task completion ratio** calculations, along with their **starting and destination positions**. Regarding time measurement, the system records the **time stamp** of every decisive event for all managers: the **insertion** into the system, the first **assignment** to an executor (changed from unassigned to assigned), the executor's **arrival to the starting position**, the **start of the travel** to the destination (end of the loading operations), the executor **reaching the goal position**, and the **task's completion** (end of the unloading operations). The model also keeps the **maximum time to attend the assignment** to analyze how close was the completion time to that limit.

With these time records, we can easily calculate several metrics for the solution evaluation. The first would be the **completion time**, which goes from the moment of the first assignment to the task's completion. The second would be the **satisfaction time**, which lasts from the insertion instant to when the order reaches its destination. Then, there is the **unassigned time**, which starts when the order enters the system and ends when it gets assigned for the first time. Following this is the **waiting time**, lasting from the assignment to the instant when the executor reaches the task's starting position. After this comes the **getting-on time**, which covers all the loading operations into the vehicle. Then, there is the **traveling time** from the end of the loading process to the arrival at the destination. Finally, we have the **getting-off time**, on which the goods get unloaded from the mobile agent.

Since our model allows task reassignment, the system records the number of reassignments for every task. Our tool logs the **id number** of each executor to participate in fulfilling an order. It also saves the vehicle's **position when assigned and de-assigned** (either due to reassignment or task completion), along with the **time at which they occurred**, which allows a deeper analysis of the solution. Regarding cost examination, each assignment entry contains the auction-winning bid as the **estimated cost** and the distance traveled during the assignment as the **real cost**. The bid of the previously assigned executor that causes it to lose the re-auction is also kept as the **estimated cost of the previous winner**. With these data, it is possible to calculate how much the **cost estimation improved** due to the task reassignments by subtracting the winning bid value from the bid of the losing agent. We can also know and how far the **cost estimation** was from the **real cost**, with the difference between them.

Even though the task managers provide precise information about their execution, some data are more accessible from other sources, like the executors. From these agents, our model extracts the **number of tasks completed** (carried out until completion) and the **amount they lost** (reassigned to another executor) for analyzing **task distribution**. The system records each vehicle's **starting and final positions** to provide more details about the allocation solution. Regarding time, the model saves how much **time each mobile agent was active** and how much **time it was assigned to any task** to calculate the **utilization rate**. For distance-related analysis, each executor provides its **total traveled distance** and the **distance range** between its completed tasks with the highest and the lowest cost. Regarding traffic saturation and hindering, the system keeps track of the **amount of re-routing attempts** by the executors, either due to persistent obstacles detected by their sensors or due to continued lack of service from the nodes they try to access.

Since each executor can intervene in the execution of several tasks, they keep a list containing all their assignments. Each entry on the list has the manager id number for differentiating it and its **completion status** for tracking **task completion distribution**. For solution replication, each list element contains the **time and location of the agent** when the assignment started and ended. For cost analysis, each list item includes the **traveled distance to complete the task**.

When computing the solution performance, the system does not include any assignment the executor does not complete unless the agent completes any task after those lost assignments. The logic behind this procedure is that any order taken away from the mobile agent will only be relevant to the allocation solution regarding how much its position changed while traveling to the indicated location. If the agent wins an auction after losing an assignment, it is reasonable to assume that the travel influenced the auction result. If the executor completes the task, it is relevant to the solution, and we include it in its description. However, if the executor loses its final assignment, the related travel will not affect any task's completion, and the model removes it from the allocation description. This procedure keeps repeating until the last assignment for every mobile agent, if any, is a completed task.

On a more general note, our model also records how long every simulation lasted and the time needed to calculate the allocation solution. It also registers the **number of messages sent and received** per run, classified by performative. These two amounts differ because when any agent broadcasts information, it counts as a single sent message and several received messages, depending on the number of active agents. While the number of received messages better describes the amount of information shared during a run, the number of sent messages can be helpful for those concerned about energy consumption due to communications. For that reason, we keep both numbers.

3.2.3 Allocation solution description

With all the data collected, our model can provide many details and statistics of the simulations it performs. The system can show all the details of the allocation solution or in a resumed format that includes only the most crucial data for evaluating the execution. Both presentation formats show the data with a tabulation meant for an Excel file. Currently, the program cannot export these data directly to this type of file (*.xlsx*), but it can save them in a plain text file (*.txt*) for the user to copy and paste into an Excel file. Another option is to copy the data directly from the program's console.

When offering the resumed version of the solution, the system presents two tables, one contains the information gathered by the task managers, and the other has the data collected by the executors.

The task managers' table lists the tasks in the sequence they were completed and includes the following fields: the task manager's id number, whether the assignment reached its completion or not, the id number of the executor that completed the task, the times at which the executor won the auction and finished the order, the task's total estimated cost, the task's total actual cost (which is the traveled distance to complete it), and how much the cost estimation improved due to task reassignment. If more than one vehicle participated in a task's execution, the entry for that task will have several lines. The top line will have the information on the agent that culminated the order, as well as the task's estimated and actual costs. The entry will have one additional line per executor involved (including the one that completed it), with their respective costs.

On the other side, the executors' table starts with the executor's id number, how long it remained active, how much time it remained assigned to any task, and the total distance it traveled while completing orders. Regarding its assignments, it contains the id number of those tasks, whether they reached completion or not, the time at which the executor won their corresponding auctions, and the moment when it was released from them (either by substitution or completion). This table also presents several lines per entry, according to the number of tasks assigned to the each executor.

While this information describes many aspects of the allocation solution, the program can also provide an extensive execution overview. If the user chooses the complete solution outline, the program expands the two before-mentioned tables with much more data.

The first elements included in the task managers table are the coordinates and names corresponding to the starting and destination ports. The table presents all the measurements described previously regarding the time consumed on each operation (the completion time, the satisfaction time, the unassigned time, the waiting time, the getting-on time, the traveling time, and the getting-off time).

The table also contains the time at which any major event for the task happened for temporal description. Namely, its insertion into the system, its first assignment to an executor, the executor's arrival at the starting position, the start of the travel to the destination, the executor's arrival at the destination, and the task completion. The last time element is the maximum time to attend the assignment.

Regarding all the executors that intervened in the task completion, the table shows first the total number of reassignments for each order. Then, it lists the positions and times at which each executor won its auction or lost its assignment to the task.

The final columns for each entry are cost-related. First comes the cost estimated by the executor (its auction bid), followed by the proposal presented by the previous winner, if any. Finally, there is the real cost incurred by the executor while performing the task and the difference between the real and estimated costs.

As with the task managers, the executors' data table experiences significant growth when showing the full solution description. The table now exhibits the time and position of each mobile agent when they started their operations and when they finished. Task completion-wise, it shows the number of completed tasks and the number of de-assigned ones. All executor entries include their respective assignments, containing the position at the start and end of each order, along with its estimated and real costs. The difference between the highest and lowest cost values per mobile agent appears as the cost range. Finally, regarding traffic congestion, the program shows all the re-routing attempts of the vehicle, classified as obstacle triggered and due to lack of service.

The program offers the resumed description by default to avoid overwhelming the users with the data. The latter can be modified with minor adjustments to show more data from the complete characterization.

3.2.4 Single run statistics

For a general description of the execution, our program can show the statistics of the run. One statistic category may present several fields, one for the total value, one for the average value, and one for the standard deviation. Our model starts by showing the total simulation time and the total processing time. They correspond to how much time it took to complete all the tasks and how much time the system spent calculating the allocations.

The rest of the statistics related to time consumption during task completion are the average and standard deviation values for the task completion time, the task satisfaction time, the unassigned time, the waiting time, the getting-in time, the traveling time, and the getting-off time. It also shows the time-consumption-to-task-completion ratio, which corresponds to the total simulation time divided by the number of completed tasks.

Regarding task completion, the system shows other metrics: the number of tasks inserted in the simulation, how many of them got assigned to an executor, and which reached completion. It also provides the task completion ratio, which corresponds to the division of the number of completed tasks by the number of inserted ones.

In the reassignments' ambit, the model offers the total number of reassignments during the run along with the average number of reassignments per task, with its standard deviation. It also calculates two ratios: one between the number of reassignments and the number of assigned tasks and another between the number of reassignments and completed tasks. Both ratios provide more perspective when comparing runs with different amounts of orders regarding the number of reassignments.

Regarding operation costs, our software brings the total distance traveled by all agents as the solution cost. The solution quality, or maximum distance traveled by any agent, also appears. For more details, it includes the average cost per task and its standard deviation. Furthermore, the model shows the average cost range per executor by calculating the difference between its tasks with the highest and lowest costs.

The system presents similar data for cost estimation and its improvement. It shows the sum of the estimated costs for each task as the solution estimated cost, its enhancement due to reassignments, with the average and standard deviation values for the estimated cost improvement per task. For comparing the estimated and actual costs, the model offers the difference between the global values and the average difference per task (with its standard deviation).

From the executors' side, the software shows two general values: the number of agents in the system and the task completion distribution value, which is the maximum number of assignments completed by any executor over the total number of completed tasks. In addition, it presents the average and standard deviation values for the number of tasks completed and the number of assignments lost per executor.

In terms of agent occupation, the model provides three metrics (in terms of average and standard deviation values): the time the executor was operative, the time it remained assigned to any task, and the occupation ratio, which is the division of the occupied time by the active time. For traffic congestion analysis, it also provides three metrics: the average value of all re-routing attempts, those caused by an obstacle, and those caused by lack of service. All these measurements come with their corresponding standard deviation values.

Finally, the software provides data regarding data transfer during the execution. It divides this information into messages sent and messages received. These global categorizations get divided according to the message performative. For each category, the system offers the total amount and the average and standard deviation values per agent (including task managers and executors).

3.2.5 Multiple runs statistics

Since our system can perform several runs in a row, it can also provide statistics for those executions. The data presentation starts with the common elements from all the runs: the number of executors in the scene, the maximum number of tasks to complete, and the time limit for the execution. After this heading, the model shows the statistics per run in a table, which allows us to easily calculate the average and standard deviation values for each metric (across all executions) to compare the system performance in different configurations.

The first metrics shown are related to the operations' cost. The first ones are the real cost of each solution, which corresponds to the total distance traveled, and the average cost of executing each task, with its corresponding standard deviation. Next comes the expected cost of the whole solution, followed by the average and standard deviation values of this metric for each task. These values are the sum of all the winning bids in the simulation and the winning bids for each task. Accompanying those results are the expected cost improvement for the whole solution and for each task. These values represent the diminution in the expected costs due to the tasks' reassignments. The last distance related element included is the solution quality, which is the maximum distance traveled by any executor while performing its assigned tasks.

The next block of results contains details about time consumption. It starts with the simulation time consumption of the whole solution, along with the average simulation time to complete each task. Then, it provides the time consumed in processing operations (deliberation, planning, and calculations) for the whole solution and the average per task. Regarding task completion, the system returns the total amount of completed tasks and what percentage they represent from all the tasks inserted during the simulation. The following values are the average number of tasks completed per executor, their occupation ratio, and their cost range. The latter metric corresponds to the difference between the tasks with the highest and lowest costs per executor. The next two values presented are the total number of reassignments in each solution and the average number of reassignments per task. They are followed by four metrics regarding information exchange, namely: the total amount of messages received by all agents, the average amount of messages received per agent, the total amount of messages sent by all agents, and the average amount of messages sent per agent. The last two metrics give an indication of the solutions' traffic congestion, they are the total number of re-routing attempts in each solution and the average number of re-routing attempts per mobile agent.

3.3 Performance assessment

To assess the quality of the allocation solutions provided by our Multi-Agent System it is necessary to compare them with other solutions. There are two ways to perform reliable comparisons. One of them would be to replicate in our model the situations described in other studies and compare our results with the ones they provide. The second way is to simulate an environment in our system, replicate its characteristics in the software developed in other approaches, and compare the results.

The major obstacle to the first method is obtaining all the necessary details to reproduce the environments and situations described in other studies. Replicating this kind of experiment requires detailed information about the plant distribution and physical characteristics, the robots, and the tasks to perform. For example, even to replicate the operations of a simple logistics system composed of a small warehouse with a Manhattan distribution and a homogeneous fleet of agents, we still would require more details. To name a few, we would need, from the physical environment, the pathways' dimensions, the traffic flow direction for each road, and the position of the pick-up and drop-off locations. From the agents, their linear movement speed, angular velocity, as well as the time they require to pick up and drop off the products. And from the tasks to perform we require data regarding how frequently they enter in the system, which starting and destination locations they will use, and whether they are sequential (one triggers another) or independent of each other.

The second comparison method requires access to the software and models employed in other works, which is not always easy to obtain. Also, it demands mastering those systems enough to be capable of replicating the simulated system characteristics across all platforms, which could become problematic.

With these consideration factors, we opted for using Mixed Integer Linear Programming (MILP) to model the same scenarios presented to our Multi-Agent System. We presented this optimization model in section 2.3, it provides allocation solutions that look to minimize the total distance traveled, guaranteeing optimal, or at least near-optimal, solutions. With that data, it is possible to objectively evaluate the quality of the results provided by our multi-agent model. For the MILP implementation, we used the IBM ILOG CPLEX Optimization Studio 22.1.0.

3.3.1 Optimization testing

To verify the effectiveness of our optimization model, we provided it with data from a small environment and a set of tasks. The presented allocation problem is simple enough that it is possible to calculate the solution manually, allowing us to evaluate if our system behaves correctly. The test environment was the Small Linear Layout described in section 3.1.3.

We included three transport tasks for this scenario, one from each retrieval point (the three rightmost ports at the lower part of the map). They all shared the same delivery point, which was the leftmost port

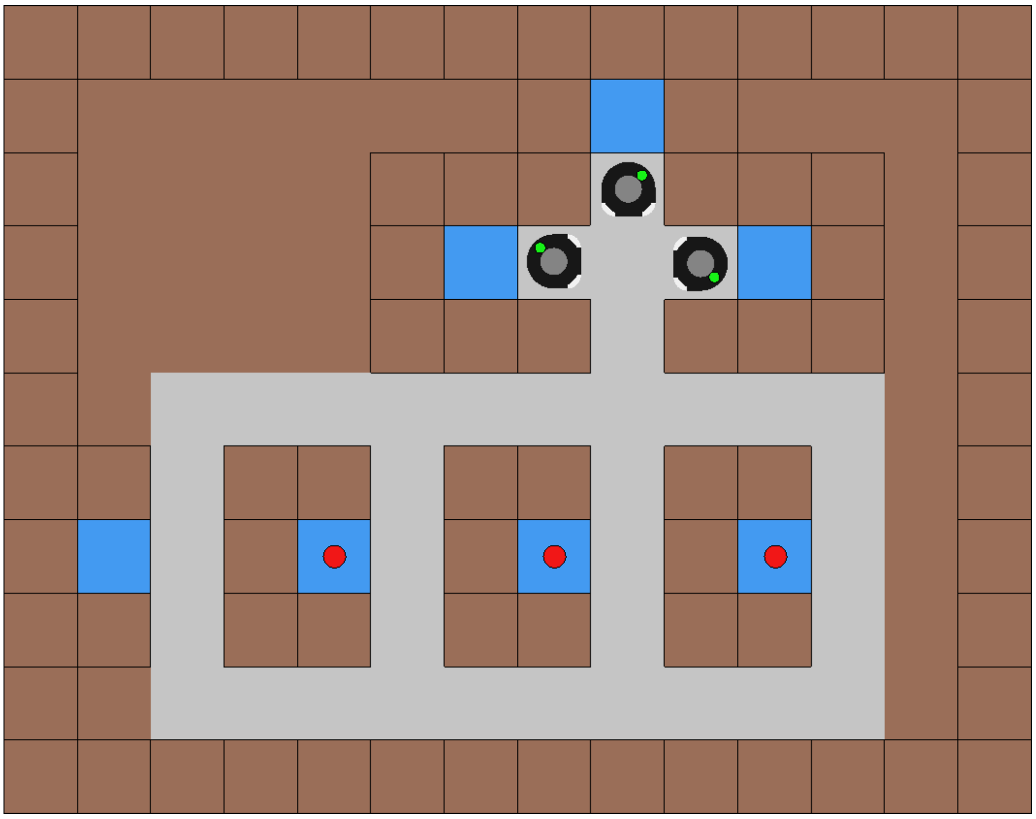


Figure 3.7: Starting conditions for the operations in the small linear layout. Red dot: task manager. Black object: executor

on the map. All orders started simultaneously, with all mobile agents at their corresponding charging stations. Figure 3.7 shows the starting distribution of managers and executors for the experiments.

Each task cost consists of two parts, a fixed value and a variable one. The fixed part is the distance between the starting and final ports, which will always remain the same. The variable section of the cost is the distance from the executor to the starting port, which depends on the vehicle's position at the assignment time. Since all tasks have the leftmost port as the destination, their individual fixed costs are different from the others. Given that a free executor will take an unassigned task as soon as possible, they will do so at their starting positions (when taking the first task) or at the delivery port (after finishing an assignment). So, the variable cost of each order will have only two possible values. With these particularities, Table 3.1 contains the cost values for each travel segment.

We solved the allocation problem manually for one, two, and three agents working simultaneously and evaluated the solutions according to their cost (total distance traveled). The allocation remains the same for three or more vehicles since there are enough agents to assign all tasks at the start of the operations. With only one robot operating, there are three combinations, depending on which assignment it takes first, Table 3.2. Following the traffic rules, the furthest port from the charging stations is P5, so if the agent starts by Task 1, the total cost would be 75 meters. However, if it begins with any of the other

Table 3.1: Travel cost for each task segment.

Task	Starting port	Destination port	Fixed cost	Variable cost	
				From charging ports	From destination port
T1	P5	D4	7	22	7
T2	P6	D4	10	5	10
T3	P7	D4	13	8	13

Table 3.2: All possible allocation solutions for 1 agent.

Agent	Tasks execution sequence	Individual cost	Solution cost
A1	T1, T2, T3	75	75
A1	T2, T1, T3	55	55
A1	T3, T1, T2	55	55

two, the total distance would be only 55 meters since their ports are closer to the charging stations.

A similar thing occurs when two agents work together: if one starts by taking T1, it will travel 29 meters, while the other will travel 41 meters performing tasks T2 and T3, regardless of the order, Table 3.3. These combinations have a solution cost of 70 meters. Nevertheless, if their starting tasks are T2 and T3, the one with T2 will finish first and take T1 right after. With this allocation, the total distance traveled is 50 meters.

Finally, for three executors, each one takes a different task at the beginning, and they all travel a total of 65 meters, Table 3.4. It is interesting that even when the three robots operate concurrently, they do not even get to finish all tasks faster than the best solution with only two of them. The reason is that the agent taking T1 at the latter solution travels the same distance as the one taking T2 and T1 in the best solution for only two agents. So, both solutions would take practically the same time, with one using

Table 3.3: All possible allocation solutions for 2 agents.

Agent	Tasks execution sequence	Individual cost	Solution cost
A1 A2	T1 T2, T3	29 41	70
A1 A2	T1 T3, T2	29 41	70
A1 A2	T2, T1 T3	29 21	50
A1 A2	T3, T1 T2	35 15	50

Table 3.4: All possible allocation solutions for 3 agents.

Agent	Tasks execution sequence	Individual cost	Solution cost
A1	T1	29	
A2	T2	15	65
A3	T3	21	

fewer agents, with a better solution cost and the same solution quality (maximum individual cost for any agent).

When executing our optimization model, it finds the optimal allocation solution in only 0.11 seconds. This solution contains the following specifications:

- Solution cost: 50
- Number of agents: 2
- Maximum number of tasks per agent: 2
- Active arcs:

  (0, 2), (2, 1), (1, 0)

  (0, 3), (3, 0)

Since the active arcs denote which task followed which and considering that the 0 marks the start and end of a route (depending on which part of the arc it is), this information means that one agent performed task 2 and then task 1 while the other only carried out task 3. All these details correspond with the best possible solution from all the ones we manually obtained. So, this proves that the model can determine the optimal solution for a simple case in a minuscule amount of time.

Chapter 4

Experimental results

We designed several experiments to assess the performance of our MRTA model in different configurations and scenarios. These tests cover several aspects, like the system's adaptability to changes in the assignments and its robustness when dealing with errors or disturbances. To obtain valid statistical data from our system, we ran 100 executions for each configuration described in this section and calculated the average value for each metric. We also constructed various experiments to evaluate the effectiveness of our MILP model and compare the results obtained from both models.

All agents in our multi-robot approach were coded in Lua and executed in the CoppeliaSim [Roh13] simulator. As for the MILP model, it was developed using the IBM ILOG CPLEX Optimization Studio 22.1.0. The code files for executing both models, a brief explanation of each controller's functions and operations, and the EFSSM diagrams modeling their behavior, can be found in the public repository [Riv22a].

4.1 Performance in a warehouse scenario

The allocation problems in this section include several transportation tasks in a warehouse environment like the one described in section 3.1.4. A transport order in those facilities entails carrying a rack from the storage area to a picking station for an employee to extract its contents. Then, the rack must return to the shelf it came from in the storage area. The system behaves likewise when dealing with empty racks that need filling. Since those facilities receive over 250 tasks per hour, it is not reasonable to manually create and enter them and their specifications into the system. For that reason, we used a procedural mechanism of our own to generate all required tasks. Our generation method requires several inputs to operate, like the number of assignments, the time range in which they must appear, the time separation between the orders starts, the list of possible starting positions, and the list of potential destinations. Following those entries, our algorithm will create as many tasks as possible inside the time frame while complying with the time separation until meeting the required amount or there is no more capacity in the time window. It is also possible to indicate the mechanism to use the destinations in the starting ports instead of providing a list.

Even though our warehouse setting is smaller than the original, it also contains five storage areas, named from A to E, with 96, 75, 45, 12, and 24 tasks per hour, respectively. After each order from the storage area to the picking area finishes, the generation system schedules a new one to return the rack to its storage position. These new tasks start after the time for picking the contents (25 seconds) passes. This combination generates 504 transport orders per hour, 252 towards the picking area and 252 towards the storage area. During their operations, all mobile agents moved at an average speed of 1 m/s and took 5 seconds to load or unload a rack. We kept our model executing under these conditions until the completion of 252 tasks, which should be equivalent to over a half hour of work.

4.1.1 Optimal Performance Comparison

To obtain a perspective of the quality of the results obtained by our MRTA system, we compared its results with the optimal values returned by our MILP model. For the experiments, we used the warehouse distribution described before, with different numbers of tasks and mobile agents to execute them. For the optimization, we provided our MILP model with the 118 ports in the scene and the minimal distances between them. This data is extracted automatically by a mechanism of our design, which analyzes the ports' distribution in the environment and, following the imposed traffic rules, calculates the minimal routes between them. The mechanism provides distance information in a matrix format ready for our MILP model to read, along with the information on the tasks to perform.

For the comparison, we generated several groups of tasks and introduced them in both models. Each group contained 25, 50, 100, and 252 simultaneous tasks, which means that all are available from the beginning, and both models can elaborate their plans and routes using any of them as the first ones. This situation emulates a non-dynamic system where the tasks come in batches: all at once, and there are no further additions before all assignments get completed. Even though this type of system is not the exact target of our study, we made this simplification to reduce the complexity of the allocation problem, which should allow the optimization algorithm to find the optimal solutions in a relatively short time. This simplification would be equivalent to grouping all incoming orders for a while and presenting them to the allocation mechanism in batches. This methodology is applicable as long as the products do not require immediate transportation, meaning that they can wait the time that the grouping phase involves before the actual execution of the tasks. Also, when using this approach, the allocation solution calculations must not take longer than the grouping time. Since the tasks arrive periodically, if the allocation mechanism takes more time to find a solution than the established grouping time, the number of orders received will be larger than intended. If the system keeps the batches' size constant, it creates a cumulative overflow of unattended orders. If it increases the batch size to include the extra tasks, the allocation algorithm will take longer to find its solution. Which, in turn, will make larger the next batch, and so on.

We executed the MILP model first to obtain the number of vehicles required to achieve the optimal solution cost value. Then, we launched the executions in the MRTA system with the best combination of tasks and executors found by the MILP model. To measure how far the results of our MRTA model from the optimal values found by the exhaustive exploration algorithm were, we calculated the variation percent for the solution cost using equation 4.1. This formula indicates how much a metric varies regarding its base value. To do so, it first calculates the difference between the value of interest ($value_I$) and the base value ($value_B$), then divides it by the base value and multiplies it by 100. So, the formula returns what percent the difference between the two values represents from the base value. A positive percent value means that the metric grew over the base value, and a negative percent represents a decrement in the measurement.

$$VP = \frac{value_I - value_B}{value_B} \times 100 \tag{4.1}$$

Since the MILP model bases its results on the minimal distance between the ports and the number of tasks per mobile agent, its solution costs imply ideal executions where the agents do not interfere with each other during their travels around the environment. To evaluate both models on equal ground, we compared the solution costs obtained by the MILP system with the estimated solution costs from the MRTA model. The estimated solution cost is the sum of the winning bid for each task in the system. So, it's the minimum distance the winning agent expects to travel to complete the task without considering how other elements in the road may affect it, just like the cost used by the MILP model. For the comparison, we used results obtained by our base MRTA mechanism, where the task managers can only perform Primary Auctions. Figure 4.1 shows the variation percent for the MRTA's estimated cost, using the MILP results as base values. The graphic shows that our MRTA system never reaches the optimal values, finding, in some cases, solutions that are, on average, almost 17% greater than the global minimum. However, as the number of assignments increases, our distributed model gets results closer to the optimal ones, reducing the gap by almost half from the worst cases (25 tasks) to the best ones (252 tasks). It is important to note that for all the scenarios, except the simplest one (25 tasks), the optimization algorithm could not find the allocation solution with optimal objective value within an hour and returned the best solution it could find in that period. So, even in these simplified scenarios, the MILP method struggles to find an optimal solution for the MRTA problem. Considering that, in a real case, those task quantities would enter the system at a much faster rate (25 tasks every five minutes, 50 every 10 minutes, 100 every 20 minutes, and 252 every hour), a method taking so long to find a solution results ineffective. We chose this time limit value for several reasons: it corresponds to the period in which the maximum amount of orders considered enters the system, it is large enough for the solver to explore many combinations, it could not find the optimum for higher values, and in those cases, the quality of the results barely improved. Meanwhile, our agent-based system needed just over 2 minutes on average to solve the most complex TA problems presented (252 tasks). These timing results highlight a major disadvantage of the MILP systems: the high amount of computation time they require for solving complex problems.

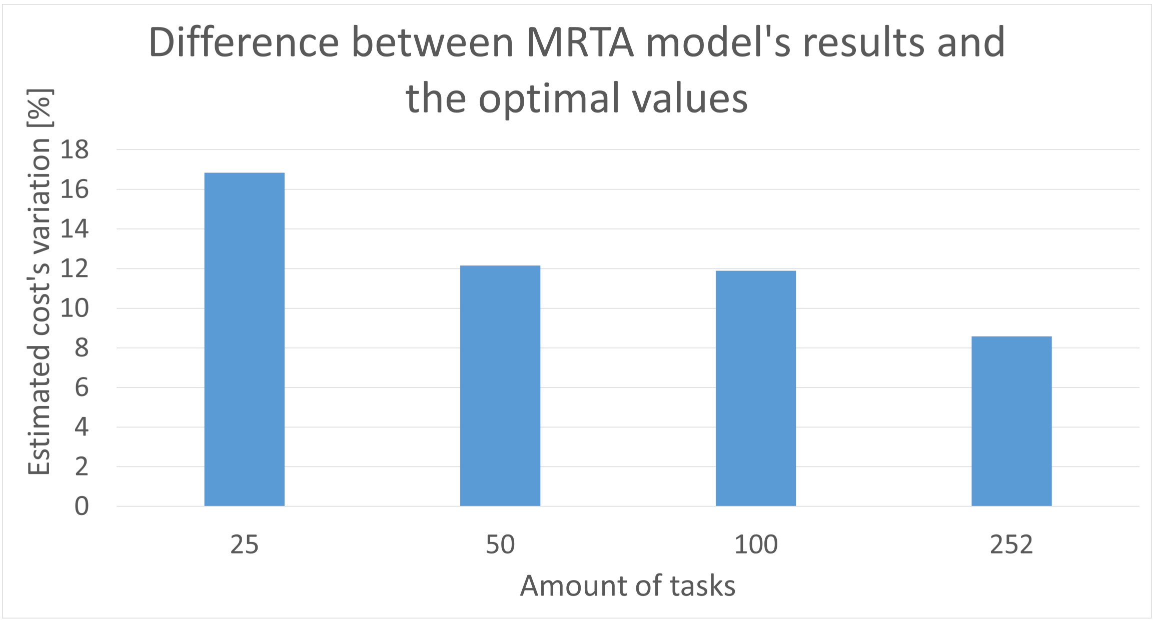


Figure 4.1: Comparison between the MRTA model's results and the MILP model's results

Given that our distributed system uses a greedy allocation mechanism, it is logical that its results are not close to the optimal values. Our solution seeks to minimize local costs (for each task) rather than global ones, so it will never provide results as good as a thorough exploration algorithm like MILP. Nevertheless, we compare our agent-based method's results with the optimal (or near-optimal) values for each case to provide a point of reference for our approach results' quality. The comparison shows that our greedy solution gets closer to the exploration algorithm's solution for higher amounts of tasks in the system. This tendency is encouraging since our work looks to provide allocation solutions in scenarios with high task density.

Since the optimization objective of our MILP model is the minimization of the solution cost and the maximum number of tasks per mobile agent, it provides results that comply with those requirements. However, they may not be appealing solutions to the users due to their rapid increment in the number of agents used. Table 4.1 shows the best results found by the optimization tool within an hour for different amounts of tasks and the number of agents required to achieve them. The results show that even for a relatively small amount of tasks, like 25, the solution with optimal cost requires up to 13 mobile agents, with the number increasing until reaching the established limit (27) for higher task concentrations. Given that, for all cases, the solutions require vehicles' amounts that represent over a 20% of the elements to transport (the only exception is the 252 because it reaches the constraint to the number of agents), they may not be very appealing in a real scenario with hundreds or thousands of transportation tasks. Using too many robots can cause them to remain unoccupied most of the time, which many users consider a waste of resources. So, even when the comparison to the MILP solutions shows that our MRTA model solutions have a lot of ground to improve, it should not be the only factor to evaluate the quality of the results.

Table 4.1: MILP solution cost values and agents required to achieve them

Tasks	Agents	Cost
25	13	712
50	17	1391
100	23	2695
252	27	7333

From these results, we conclude that the cost of the solutions provided by our MRTA model gets closer to the MILP system results as the amount of tasks grows. This tendency indicates that our distributed system, while greedy and sub-optimal, presents better results for larger logistics systems, which are the target of our study.

4.1.2 Reallocations' Impact

Now that we know how far our base MRTA model's results are from optimal or near-optimal values in static situations, we must assess its performance in dynamic environments and how much those results improve when using the different capabilities of our system. Since one of the main features of our TA system is the task reallocation mechanism, we assessed its impact on the logistics system performance, specifically in terms of solution cost, average task cost, simulation time, and average task completion time. To do so, we configured our system to solve the same allocation problem in two ways: one without reallocations and the other with them.

The first configuration variant only uses Primary Auctions for the allocations. So, after an executor obtains a task assignment, it will keep it until its completion for as long as it takes. The other setting launches Primary and Secondary Auctions every time there is a change in the allocation scene, like a new task arriving or the release of an executor. These releases might happen because the mobile agent completed its assigned order, was substituted by a robot better fitted for the task, or took too long to pick up the goods. For these experiments, we returned to the configuration described at the beginning of this section, where 504 transportation tasks appear over an hour, and the execution stops once 252 of them reach completion.

To measure how much the use of reallocation impacted the system performance, we calculated the variation percent for each metric mentioned before, using equation 4.1. Figure 4.2 shows the results when comparing the allocation solution without reallocations (base value) and the solution with them (value of interest). Since most TA solutions look to minimize these metrics, the fact that their values decrease when the system uses reassignment mechanisms is a positive result. The results show a positive impact in all metrics for different amounts of mobile agents, reducing the average task cost by over 8% for the case with ten robots.

Since the number of assignments surpasses the number of available vehicles, the latter will be busy most of the time. This situation causes that when a new order appears, many mobile agents will not an-

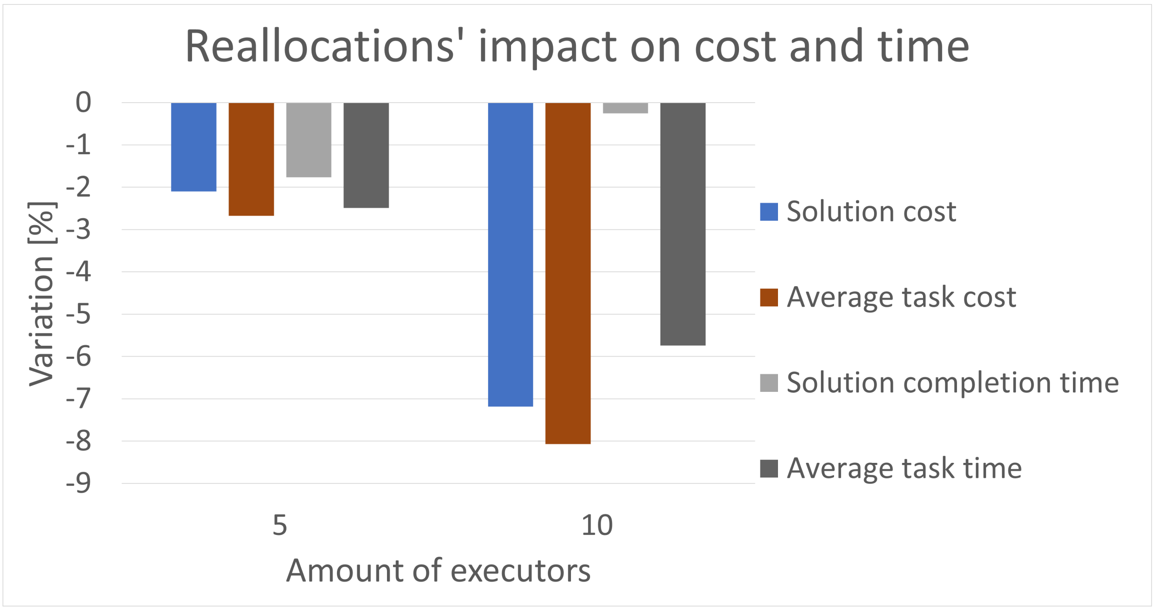


Figure 4.2: Reallocations' impact on different metrics

swer its CFP, and it will have to choose from a limited sample of participants. However, the higher the number of concurrently working executors, the higher the concentration of them finishing their assignments in the same period. The latter means that whenever an already attended task manager launches a Secondary Auction, it will be more likely to find a better-suited candidate for its transportation task.

An event that draws our attention is that all metrics decreased their values when using reallocation mechanisms with a higher amount of mobile agents, except the solution completion time, with a 0.25% diminution for ten executors. Such occurrence seems odd when the average task time consumption for that case decreases by over 5%, and both metrics are strongly linked. The explanation for this behavior is that, for this scenario, ten vehicles complete the tasks in around 50 seconds, while for five mobile agents, the average task time consumption is close to 130 seconds. Considering that the time separation between the assignments can take values between 35 and 300 seconds (depending on the storage area), we realize that, for a high number of vehicles, the total simulation time depends more on the time separation between tasks than on the time it takes to complete the tasks.

In conclusion, the results described in this subsection show how the inclusion of Secondary Auctions improved the performance of our MRTA system, reducing the time required to complete the orders and the distance traveled to fulfill them. The variations are case-dependent, and several elements can influence them, but they offer a better result in general than their non-flexible counterparts. These improvements lead us to think that the use of Secondary Auctions may be able to compensate for the difference between our base model results and those from the MILP model, bringing our solutions closer to the optimal ones.

4.1.3 Auction triggering mode's impact

Our reallocation mechanism has two triggering modes: reactive and active. In the first one, the task managers only launch new auctions when notified of a change in the allocation scene, which usually means that an executor just became available. With the active one, however, they autonomously decide when to perform their corresponding auctions.

On the one hand, the reactive mode, which is the one we used for the comparison in the previous section, guarantees that the new auctions and re-auctions only get launched when there are realistic opportunities for allocation or reallocation, depending on the case. This way, all executors remain continuously occupied while there are unassigned tasks in the system. However, if a task manager fails to receive the change notification, it will not launch its corresponding auction. This situation makes the allocation system more vulnerable to communication errors with this triggering method. With this methodology, every time there is a change announcement, all waiting task managers will broadcast their CFP messages, and all the available executors will answer them with their PRP or REF messages. Due to this behavior, the communications happen in discrete bursts, which may saturate the communication system with too many simultaneous messages.

On the other hand, while in active mode, the task managers determine when to launch their corresponding auctions, eliminating the need for an announcement system. This behavior makes the system less susceptible to communication errors. However, it does not guarantee a continuous exploit of the executors working in the environment, e.g., an executor may finish a task and become available but will remain in that state until at least a waiting task manager decides to launch an auction. While determining when to launch a new auction can consider many factors, in our current implementation, the task managers wait for some time before launching a new auction or re-auction. This behavior produces a more uniform message distribution over time than its reactive counterpart, reducing the probability of saturating the communications system with high message concentrations.

So, after confirming that our reallocation mechanism improves the system's performance, we verified how different was the system behavior when changing from reactive to active mode. For these experiments, we used the warehouse environment described at the start of the results section and the same tasks as in the previous experiment: 504 tasks distributed over an hour, with the simulation finishing once half of them get to completion. To assess how much the mode change impacted the system performance, we measured the variation percent (equation 4.1) for several metrics and presented those values in Figure 4.3. In this case, the base values are the results of the reactive mode, and the interest values are the results of the active one. The graphic shows that using the reactive configuration ensures better overall system performance, with variations from case to case. However, using the active mode drastically reduces the average amount of messages received per agent (data transference), going from more than 445 messages in the reactive configuration to barely 66 messages while in the active mode.

The obtained results confirm our expectations about both triggering methods and provide a quantita-

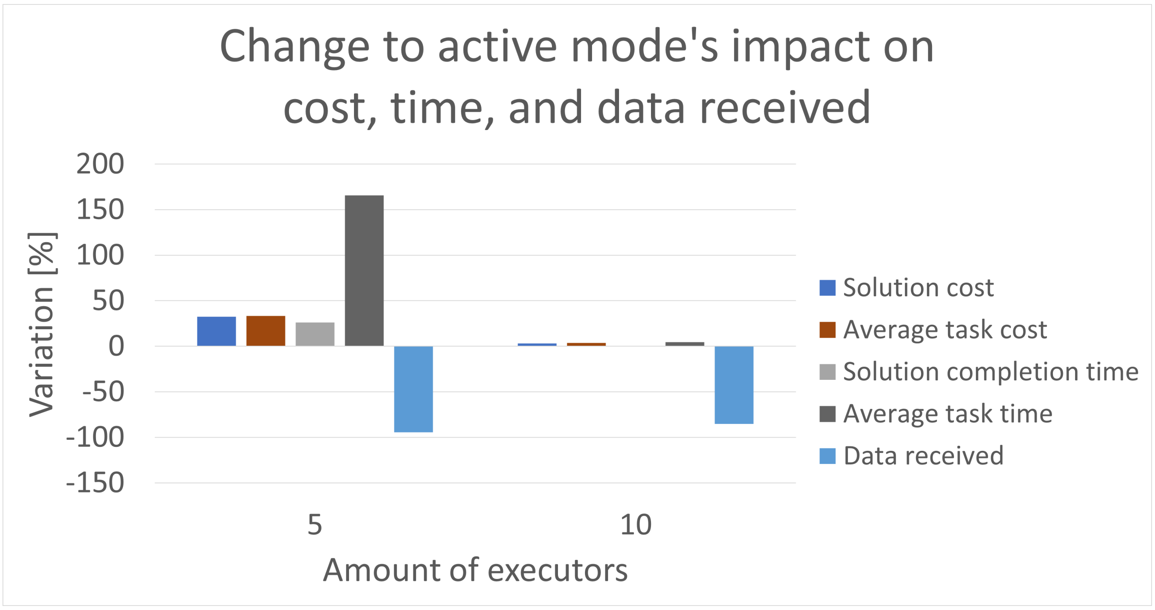


Figure 4.3: Active mode's impact on different metrics

tive differentiation between them. Both seek to satisfy the possible needs of the clients when configuring the task allocation system in different situations. The quality of their results varies due to the different quantities of resources they use to find and execute their solutions. In conclusion, they provide the users with different ways to reach the compromise value between system performance and communications' limitations that best fits the allocation problems they are trying to solve.

Parameters manipulation's impact

When triggering the auctions autonomously, the task managers space the broadcast of CFP messages over time. As explained in section 2.9.2, the time separation between the auctions depends on the values of specific parameters in the controllers. Since the active mode strongly depends on these parameters, like other mechanisms inside our system, we assessed how manipulating those values impacted the system performance. Figures 4.4 and 4.5 show the variation percent of different metrics when changing the time between auctions. In these experiments, the base value for the variation percent formula were the ones from the reactive mode, and the interest values were the ones from the active configuration with different parameters' values. To maintain all values in the graphics relatively similar in terms of scale, the metrics that experienced the more significant changes had their values divided by ten (which appears in the graphics' legends).

The time separation parameters have no limit on the values they can take. The only limitations come from a functional point of view. If the value is too low, the auctions will happen more frequently and may not give enough time for the allocation scene to have significant changes. This situation would cause the triggering of many unnecessary auctions. However, if the parameter is too high, the auctions will be more

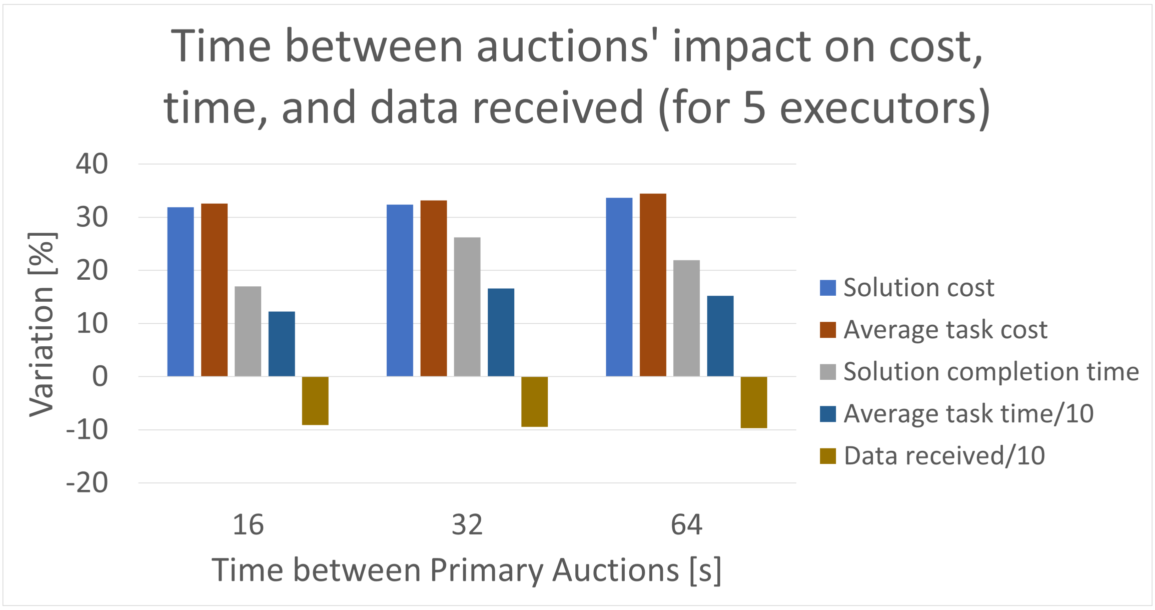


Figure 4.4: Time between auctions manipulation's impact on different metrics (for 5 executors)

sporadic, which could cause the mobile agents to remain unoccupied for a higher percentage of the total time. It could also cause the task managers to perform few or no re-auctions, missing the benefits of the reallocation mechanism. For those reasons, we used the results from the reactive mode simulations as a reference for selecting the time separation values. Considering that roughly half the average completion time of each task is spent waiting for an executor and the other half on the travel to the destination, we established that value as the upper limit for the time separation between Primary Auctions. So, the first value was the upper limit, the second was approximately half that amount, and the third was around a quarter from it. We made re-auctions happen even more frequently to ensure that the task managers used the reallocation mechanism. To that end, in all experiments, the time between Secondary Auctions is half of the time separation between Primary Auctions.

The results show how larger separations between the auctions decrease the amount of data received and usually lead to higher costs and time consumption. However, that is not always the case. For example, in the five executors' simulations, both cost metrics roughly increase their values by 1% for each time separation increment. However, that is not the case for the time consumption metrics, which reach their maximum values for the 32 seconds time separation. Even when the experiments in the middle perform approximately two times the number of auctions (and re-auctions) carried out by the ones on the right, their solution completion time is almost 5% bigger, and their average task time is even higher, reaching 15%. These results indicate that the auctions' separation parameters, as the others present in our model, have a heavy impact on the system performance, and the tunning of their values is case specific, according to the problem the user seeks to solve.

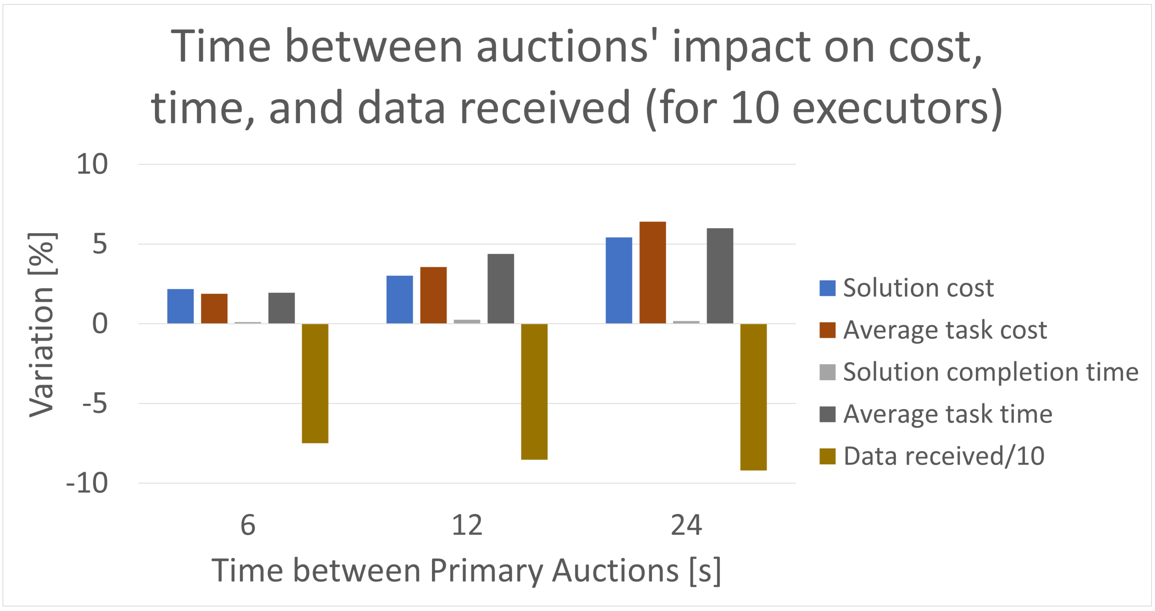


Figure 4.5: Time between auctions manipulation's impact on different metrics (for 10 executors)

4.1.4 Unexpected Events' Impact

The resolution of TA problems in dynamic environments depends not only on the capability of finding the best allocation solution but also adapting it when facing unexpected events. An agent presenting errors or dealing with other vehicles in the environment are two common causes for these contingencies. For that reason, we evaluated how those situations would affect the performance of our MRTA model.

Regarding the insertion of obstacles in the system, we designed several experiments where the mobile agents shared the warehouse environment with different amounts of vehicles that did not participate in the solution. These vehicles move through the warehouse without control or supervision and may obstruct the execution of transportation tasks. All the experiments contained ten active agents performing 252 tasks. The number of non-participant agents had values of 10, 20, and 50. To assess the systems' resistance to errors, we replicated communication errors by eliminating a percentage of the messages exchanged during the assignment process. The error probability had values of 30% and 60%.

For the evaluation, we compared the mentioned executions with the results obtained from a scenario without communication errors or vehicles that do not participate in the allocation solution. Figures 4.6 and 4.7 show how much the presence of these elements affected the system performance in terms of total simulation time and average task time, which are the parameters most affected by these issues.

Both figures show little variations in the total completion time for all experiments, with the maximum value increment slightly over 2.5% for a communications error probability of 60%. The impact in this metric from a 30% error probability is almost indistinguishable (0.1%). The cause is that if more than one executor is available when a task manager performs an auction, the error probability for that manager

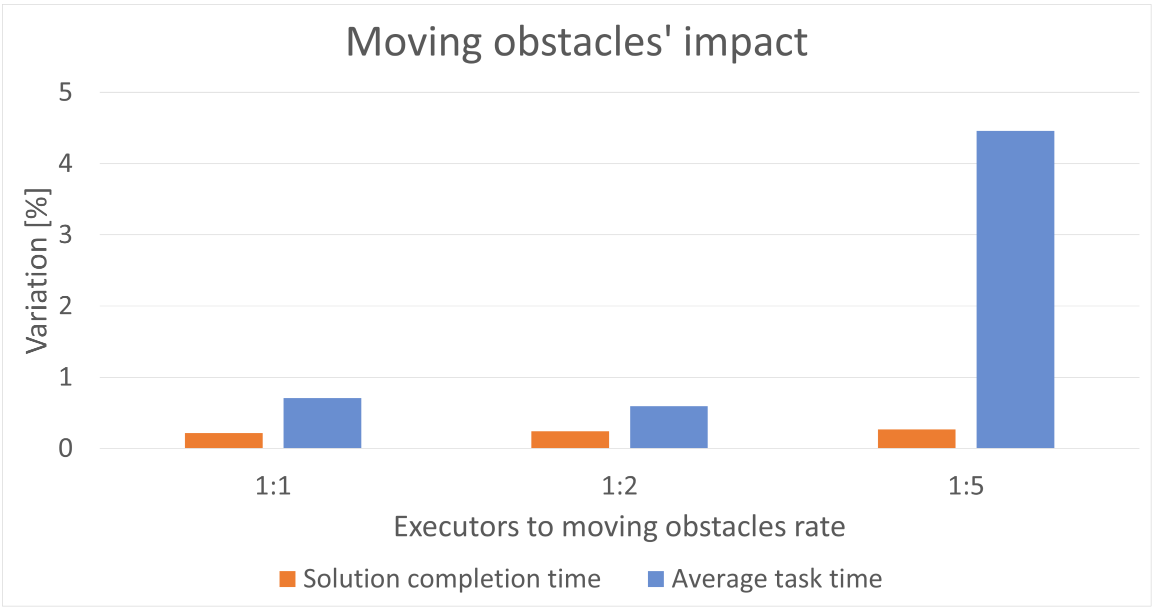


Figure 4.6: Blocking vehicles' impact on time consumption

not to receive a proposal, and wait before launching another auction, is the joint error probability of those executors, which diminishes it significantly. However, higher probability errors deteriorate the solution visibly, especially regarding the average task time, which increases its value by more than 35%.

Regarding the effect of including uncontrolled vehicles in the environment, Figure 4.6 shows that their presence increases the simulation time and the average task time. However, those increments are not large enough to consider that a scenario crowded with non-participant agents could cripple the capability of our MRTA system to find an allocation solution and execute it. The cause for the difference in the magnitude of the affections to the simulation time and average task time is the issue explained in the reallocations' impact evaluation (section 4.1.2). Using ten mobile agents in this scenario reduces the average task time enough for the simulation time to depend more on the time separation between tasks than on the task completion time.

These experiments demonstrate that the developed MRTA model is both adaptable to changes and robust against errors and disruptions. The introduction of communication errors and moving obstacles in the system degraded the system performance, but the model maintains its capability to find and execute a TA solution. Since our TA algorithm is distributed among the agents, there is no single point of failure in our model; if any agent (mobile or manager) stops working, the rest of the system adapts and proceeds with the execution of the allocation solution. These characteristics make our model capable of providing allocation solutions in dynamic environments, where new tasks are introduced in the plan frequently, and there are several elements out of the control of the allocation system.

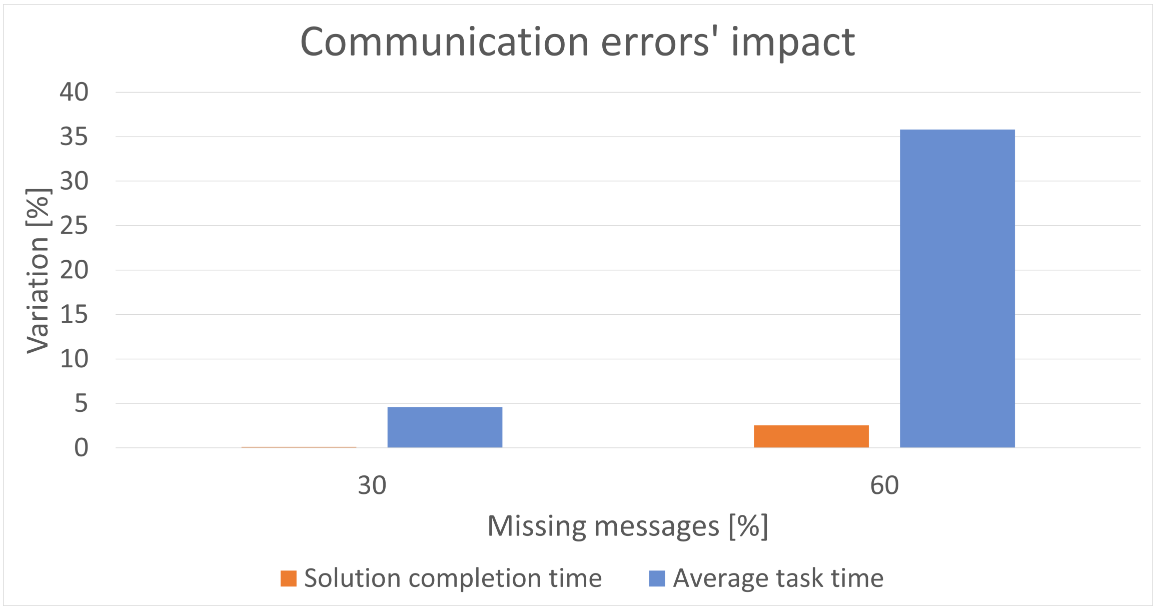


Figure 4.7: Missing messages' impact on time consumption

4.2 Performance in a workshop scenario

For our experiments in a factory or workshop-like scenario, we replicated the allocation problem inside the workshop environment described in section 3.1.5. The products that arrive at this facility need processing in several phases, which implies that each item must visit two stations or more in an orderly manner. So, the transportation system must carry the different products to all the stations (ports) that are part of the same process in a specific order. The first part of each station's name is the process it belongs to, and the second part is its position in the sequence of operations of that procedure. For example, every product that requires the treatments from process A will enter the system through port A0, asking to be carried to A1 for further processing and then to A2 for continuing the treatment and exiting the system. These characteristics make the transport tasks inside the workshop more sequential than those in the warehouse setup.

The presented layout contains two separate areas, each receiving a new product every 20 minutes. With this in mind and considering that the operations at each processing station last around 7 minutes, the tasks in this environment will be more spaced in time than the ones seen in previous sections.

For the tasks generation, we executed our tasks generation algorithm twice, once for each area. On each occasion, we only included the ports in that area containing a zero ('0') on their name as possible starting locations. As destinations for these tasks, we used the destination ports stored in the starting ports' description, which comes from the layout description. That way, we ensure that each product goes to the corresponding port to follow the processing it must receive. After the culmination of a task, if the reached port had a destination defined, the task generation algorithm scheduled a new order to go there.

This new task should take place after the processing time for the product passes, which means that it starts 7 minutes after the delivery of the item in its starting port.

For executing their tasks, all mobile robots traveled at an average speed of 4 m/s and spent 5 seconds loading or unloading the items in the ports. In all our experiments, we inserted 200 products in the system (100 in each area), where approximately half of them required only one transportation task, and the rest needed two or three of them. This combination should be equivalent to 33 hours of continuous work.

4.2.1 Reallocations' Impact

After evaluating how much our reallocation mechanism increased the performance of logistics systems in a warehouse scenario, we evaluated its impact in a workshop scenario. Even when both applications are similar, they have many characteristics that make their allocation problems intrinsically different. For that reason, we assessed the reallocation's impact on the logistics system performance by measuring how the solution cost, average task cost, simulation time, and average task completion time changed due to its application. Like in the warehouse environment, we made our system solve the allocation problem using only Primary Auctions in the first group of experiments and including the re-auctions for the second group. Figure 4.8 shows the changes in all metrics expressed as a percent of their base values, which correspond to using only Primary Auctions.

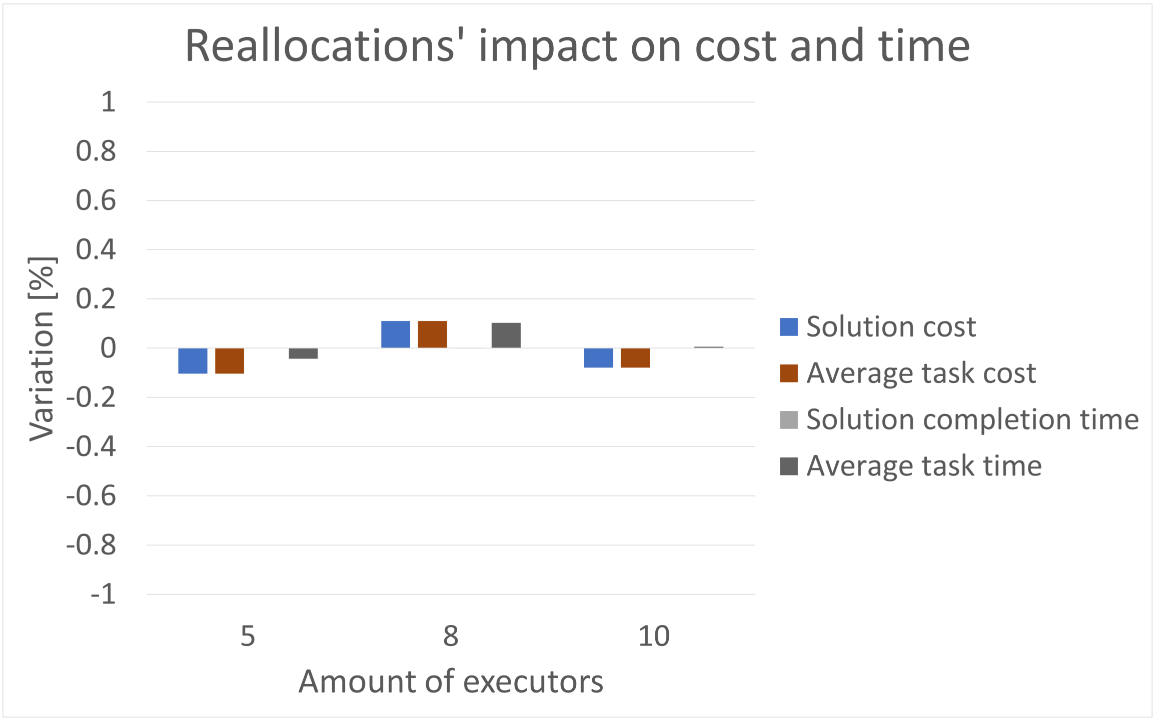


Figure 4.8: Reallocations' impact on different metrics

The results show almost no impact across all metrics, with the highest variations barely getting over 0.1% of the base values. The reallocation mechanism had practically no effect on the results because only

1.89% of the executions where Secondary Auctions were allowed actually performed task reassignments, having only one or two in each case. In those cases, the cost improvement was negligible since they did not even reach 0.03% of the total cost. So, we can assume that the reallocation mechanism was not used during the workshop simulations and the results' variations come from the intrinsic variability of the executions. One of the main reasons for that variability is the mobile robots obstructing each other during their travels, which could cause different delays in the execution of the tasks, depending on the magnitude of the obstruction. If this affection were to happen to an order that triggers others, it would postpone the insertion of its follow-up tasks, significantly changing the general plan from one simulation to another.

The reason for such a big difference between the simulations in this environment and the ones inside a warehouse is that the latter has some conditions that motivate the use of reallocations, like having many concurrent transport orders and a road network that usually is a very interconnected mesh. In a factory environment, however, the transportation tasks tend to be more distanced in time, and the layout usually has few path options for going from one point to another. With these characteristics, it is unlikely to find better-suited robot after allocating a task because all mobile agents will have to follow a similar path, if not the same, to reach it. So, our results in these scenarios coincide with what we expected to obtain. However, the executions show our MRTA model's capability to solve the allocation problem in these scenarios, which is the other main objective of these tests.

In conclusion, the results obtained in a workshop environment show how the inclusion of Secondary Auctions had almost no impact on the logistics system performance because our MRTA mechanism barely had any opportunities to use them. In any case, even without reallocations, our model remains capable of solving the task allocation problem in these environments with their distinctive characteristics regarding layout and task distribution. These solutions still conserve the rest of the features verified in the previous sections, like robustness before errors and obstructions, adaptability to changes on the fly in the planning and the mobile robots to use, and good overall performance without the need for previous planning.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This document presents our research on an automated system for solving the Multi-Robot Task Allocation (MRTA) problem. The contribution of this thesis lies in the formalization of a distributed reallocation model that adapts its solution on the fly when facing any changes in the planning, errors in the agents' behavior, or disruptive elements in the environment. The algorithm is based on a Multi-Agent System (MAS) where its agents deliberate to determine the distribution of the transportation tasks between them. Those deliberations happen as a series of auctions where agents managing the tasks' execution offer them for bidding, and the mobile agents act as bidders. The reallocation mechanism allows the task managers to launch new auctions, even when they are already assigned to a robot, to find a better-suited mobile agent for their transportation [Riv22b, Riv19b, Riv19a].

To model as many MRTA problems as possible and elaborate solutions for them, we designed a framework that contains our task allocation algorithm integrated into a physics simulator [Riv23]. To define the programming of the controllers, we used Extended Finite State Stack Machines, which are ideal for modeling their behaviors and describing them formally [Riv18]. This formal description of the system behavior ensures predictability and verifiability for the model. Combining the physical simulator with the full agent representation allows our system to reproduce all the situations and interactions in a logistics environment, from the high-level control functions regarding tasks allocation and communications to the physical actions and interactions of the agents at the ground level. With this detailed depiction of the operations in the facility, the TA mechanism can make and modify its planning using information gathered during the tasks' execution, making it more exact than when using estimated data. Thanks to this, the user can introduce changes or disturbances of various kinds into the model and evaluate their impact on the system performance.

However, due to the nature of their operations, some state machines require that their transitions occur without time passing, which is impossible with the conventional execution engines of physics simulators. To solve this problem, we developed a time model that separates the execution of instantaneous operations from those that consume time [Riv23].

The framework contains several layers that attend to different aspects of the control of the whole system. This structure allows our framework to generate the allocation solution at various control levels and act as an automated manager for physical agents of diverse complexities. Also, these layers operate as individual modules which can complement other execution systems. Our solution can work as a standalone decision support tool by executing the allocation solution with the physical simulator and obtaining data on the system's behavior and performance under different conditions.

We created a MILP model that searches for the allocation solution that offers the minimal linear combination of distance traveled by all the mobile agents in the system and the maximum number of tasks per vehicle [Riv22b]. The objective behind designing this model was to have another set of results to compare to the ones from our MAS approach. Such comparison would offer a more objective evaluation of the quality of the obtained results.

We performed several experiments in warehouse-like and workshop-like environments with different levels of complexity to assess the quality of the results of our models. The experiments show that our MRTA model offers results closer to the optimal values for scenarios with large amounts of transportation tasks. Using a reallocation mechanism improves the performance of auction-based systems in general. Especially in scenarios with high task density and multiple routes from one point to another in the environment. Regarding changes in the allocation problem, the experiments show that our algorithm can adapt its solution to the apparition, on the fly, of new transportation tasks that were not in the original planning. The system has several parameters whose values influence the agents' behavior. The tuning of those parameters can adjust the task allocation algorithm operations' to better fit the requirements and limitations of the different logistics scenarios. Also, inserting communication errors affected the overall system performance, but the system remains capable of solving the allocation problem and attend the tasks. All this is possible thanks to having agents who deliberate and continuously adapt their assignments as the situation evolves. These characteristics turn our system into a suitable solution for the MRTA problem in static and dynamic environments, which leads us to think that our agent-based model can improve the process of transportation and delivery tasks in internal logistics environments. Also, given that our model controls the simulated robots during the execution of the tasks, it is possible to adapt the system and use it as a digital twin to control and manage real-life logistics systems.

5.2 Future Work

There are several areas we would like to improve in our distributed model. First, we would like to include new behaviors during the deliberation process that may enhance the allocation solution. The most significant one is to substitute the greedy algorithm used for task allocation for one that explores the different allocation options for all the tasks having auctions simultaneously and finds a solution that minimizes the general cost in that situation. We believe that this modification would bring our model's results even closer to the minimal possible solution cost without deteriorating the other beneficial features of our model.

Another modification we want to examine is to allow mobile agents to participate in the auctions even when they are on their way to pick up some products and even change their assignment if a less expensive task appears. This mechanism would allow more participants in each auction and may improve the system's performance. Also, regarding the mobile agents, we would like to explore the effects of increasing the time window for them to collect and evaluate incoming ACCEPT messages (which signal that the robot won a task auction). Currently, the bidder evaluates only the results from simultaneous auctions. An extended period would allow it to assess delayed offers that may be more beneficial for it to execute.

Finally, we would like to improve our deadlock resolution method. Our current approach offers any possible re-routing solution that does not involves going through the location in dispute. This mechanism helps solve simple traffic conflicts in the system but is not very helpful in situations involving many vehicles. For that reason, we plan to develop a deadlock resolution mechanism that analyses the conflict more deeply and offers a solution more beneficial for all the mobile agents involved rather than for individual gains.

Bibliography

[BÖ3] Egon. Börger, Robert. Stark, *Abstract State Machines: a Method for High-Level System Design and Analysis*, Springer Berlin Heidelberg, 2003.

[Bou11] Amar Bouali, "Scade: Industrial success of a scade: Industrial success of a synchronous language and its future challenges", *Anniversary Workshop in honour of Gérard Berry and Jean-Jacques Lévy*, pag. 24, 2011.

[Bou14] Brahim Bousetta, Omar El Beggar, Toufiq Gadi, "A model transformation approach for code generation from state machine diagram", *IADIS International Journal on Computer Science and Information Systems*, Vol. 9, pags. 1–15, 2014.

[Bru16] Sebastian G. Brunner, Franz Steinmetz, Rico Belder, Andreas Dömel, "Rafcon: A graphical tool for engineering complex, robotic tasks", *IEEE International Conference on Intelligent Robots and Systems*, Vol. 2016-November, pags. 3283–3290, 11 2016.

[Buc19] N. Buckman, J.P. How, H.-L. Choi, "Partial replanning for decentralized dynamic task allocation", *AIAA Scitech 2019 Forum*, 2019.

[Cac20] Hernan Caceres, "Cvrp-cplex · github", `https://github.com/industrial-ucn/jupyter-examples/blob/master/optimization/cvrp-cplex.ipynb`, 2020.

[Cha16] Ismael Chaile, Lluís Ribas-Xirgo, "Running agent-based-models simulations synchronized with reality to control transport systems", *Automatika*, Vol. 57, pags. 452–465, 2016.

[Che18] Xinye Chen, Ping Zhang, Fang Li, Guanglong Du, "A cluster first strategy for distributed multi-robot task allocation problem with time constraints", *2018 WRC Symposium on Advanced Robotics and Automation, WRC SARA 2018 - Proceeding*, pags. 83–89, 12 2018.

[Che19] X. Chen, P. Zhang, G. Du, F. Li, "A distributed method for dynamic multi-robot task allocation problems with critical time constraints", *Robotics and Autonomous Systems*, Vol. 118, 2019.

[Coh17] Curtis Cohenour, "Teaching finite state machines (fsms) as part of a programmable logic control (plc) course", *ASEE Annual Conference & Exposition*, 2017.

[Cor90] Thomas H. Cormen, Charles Eric. Leiserson, Ronald L. Rivest, *Introduction to algorithms*, MIT Press, 1990.

[Dai19] Min Dai, Dunbing Tang, Adriana Giret, Miguel A. Salido, “Multi-objective optimization for energy-efficient flexible job shop scheduling problem with transportation constraints”, *Robotics and Computer-Integrated Manufacturing*, Vol. 59, pags. 143–157, 10 2019.

[Das18] Pragna Das, Lluís Ribas-Xirgo, *Adaptive multi-robot control through on-line parameter identification at system level*, PhD Thesis, UAB, 2018.

[Dra20] Ivica Draganjac, Tamara Petrović, Damjan Miklić, Zdenko Kovačić, Juraj Oršulić, “Highly-scalable traffic management of autonomous industrial transportation systems”, *Robotics and Computer-Integrated Manufacturing*, Vol. 63, pags. 101915, 6 2020.

[GS22] Miguel Ángel González-Santamarta, Francisco Javier Rodríguez-Lera, Camino Fernández Llamas, Francisco Martín Rico, Vicente Matellán Olivera, “Yasmin: Yet another state machine library for ros 2”, *ROBOT2022: Fifth Iberian Robotics Conference*, pags. 528–539, 5 2022.

[Hag08] Nils Hagge, Bernardo Wagner, “Implementation alternatives for the omac state machines using iec 61499”, *2008 IEEE International Conference on Emerging Technologies and Factory Automation*, pags. 215–220, IEEE, 9 2008.

[Hub03] Engelbert Hubbers, Martijn Oostdijk, “Generating jml specifications from uml state diagrams”, *In Forum on Specification & Design Languages FDL’03*, pags. 263–273, 2003.

[Ian16] Niccolo Iannacci, Matteo Giussani, Federico Vicentini, Lorenzo Molinari Tosatti, “Robotic cell work-flow management through an iec 61499-ros architecture”, *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pags. 1–7, IEEE, 9 2016.

[Ism19] Zool Hilmi Ismail, Nohaidda Sariff, “A survey and analysis of cooperative multi-agent robot systems: Challenges and directions”, *Applications of Mobile Robots*, 3 2019.

[Jen12] Asger Jensen, Joseph R. Kiniry, *Code Generation from an Abstract State Machine into Contracted C#*, PhD Thesis, IT University of Copenhagen, 2012.

[Kaw12] Takehito Kawakami, Shozo Takata, “Battery life cycle management for automatic guided vehicle systems”, *Design for Innovative Value Towards a Sustainable Society*, pags. 403–408, Springer Netherlands, 2012.

[Kim06] Kap Hwan Kim, Su Min Jeon, Kwang Ryel Ryu, “Deadlock prevention for automated guided vehicles in automated container terminals”, *OR Spectrum*, Vol. 28, pags. 659–679, 10 2006.

[Kim12] Min Hyuk Kim, Sang Phil Kim, Seokcheon Lee, “Social-welfare based task allocation for multi-robot systems with resource constraints”, *Computers & Industrial Engineering*, Vol. 63, pags. 994–1002, 12 2012.

[Kor13] G. Ayorkor Korsah, Anthony Stentz, M. Bernardine Dias, “A comprehensive taxonomy for multi-robot task allocation”, *International Journal of Robotics Research*, Vol. 32, pags. 1495–1512, 10 2013.

[Lee20] C.W. Lee, W.P. Wong, J. Ignatius, A. Rahman, M.-L. Tseng, “Winner determination problem in multiple automated guided vehicle considering cost and flexibility”, *Computers and Industrial Engineering*, Vol. 142, 2020.

[Li09] Zhi Wu Li, Meng Chu Zhou, “Deadlock resolution in automated manufacturing systems”, *Deadlock Resolution in Automated Manufacturing Systems*, 2009.

[Li17] Zhenping Li, Xueting Li, “Research on model and algorithm of task allocation and path planning for multi-robot”, *Open Journal of Applied Sciences*, Vol. 07, pags. 511–519, 2017.

[Li18] Wei Li, Alvaro Miyazawa, Pedro Ribeiro, Ana Cavalcanti, Jim Woodcock, Jon Timmis, “From formalised state machines to implementations of robotic controllers”, *Springer Proceedings in Advanced Robotics*, Vol. 6, pags. 517–529, 2018.

[Lin08] Felix Lindlar, Armin Zimmermann, “A code generation tool for embedded automotive systems based on finite state machines”, *2008 6th IEEE International Conference on Industrial Informatics*, pags. 1539–1544, IEEE, 7 2008.

[Lin13] Dirk Van Der Linden, Wim Ploegaerts, Georg Neugschwandtner, Herwig Mannaert, “Towards evolvable state machines and their applications”, *International Journal on Advances in Systems and Measurements*, Vol. 6, pags. 335–352, 2013.

[Lin23] Yishuai Lin, Yunlong Xu, Jiawei Zhu, Xuhua Wang, Liang Wang, Gang Hu, “Mlatso: A method for task scheduling optimization in multi-load agvs-based systems”, *Robotics and Computer-Integrated Manufacturing*, Vol. 79, pags. 102397, 2 2023.

[Liu19] M. Liu, H. Ma, J. Li, S. Koenig, “Task and path planning for multi-agent pickup and delivery”, *AAMAS ’19: Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, Vol. 2, pag. 1152–1160, 2019.

[Mic08] Nathan Michael, Michael M. Zavlanos, Vijay Kumar, George J. Pappas, “Distributed multi-robot task assignment and formation control”, *Proceedings - IEEE International Conference on Robotics and Automation*, pags. 128–133, 2008.

[Mul85] David E. Muller, Paul E. Schupp, “The theory of ends, pushdown automata, and second-order logic”, *Theoretical Computer Science*, Vol. 37, pags. 51–75, 1 1985.

[Nun17] Ernesto Nunes, Marie Manner, Hakim Mitiche, Maria Gini, “A taxonomy for task allocation problems with temporal and ordering constraints”, *Robotics and Autonomous Systems*, Vol. 90, pags. 55–70, 4 2017.

[Ott19] Michael Otte, Michael J. Kuhlman, Donald Sofge, “Auctions for multi-robot task allocation in communication limited environments”, *Autonomous Robots 2019 44:3*, Vol. 44, pags. 547–584, 1 2019.

[Pat15] Sandeep Patil, Victor Dubinin, Valeriy Vyatkin, “Formal verification of iec61499 function blocks with abstract state machines and smv – modelling”, *2015 IEEE Trustcom/BigDataSE/ISPA*, pags. 313–320, IEEE, 8 2015.

[Per19] Florent Perronnet, Jocelyn Buisson, Alexandre Lombard, Abdeljalil Abbas-Turki, Mourad Ahmane, Abdellah El Moudni, “Deadlock prevention of self-driving vehicles in a network of intersections”, *IEEE Transactions on Intelligent Transportation Systems*, Vol. 20, pags. 4219–4233, 11 2019.

[Pol15] Rahul Shivaji Pol, M. Murugan, “A review on indoor human aware autonomous mobile robot navigation through a dynamic environment. survey of different path planning algorithm and methods”, *2015 International Conference on Industrial Instrumentation and Control (ICIC)*, pags. 1339–1344, IEEE, 5 2015.

[Pon02] Michael J. Pont, *Embedded C*, Addison-Wesley, 2002.

[Pop22] Martin Poppinga, Marc Bestmann, “Dsd - dynamic stack decider: A lightweight decision making framework for robots and software agents”, *International Journal of Social Robotics*, Vol. 14, pags. 73–83, 1 2022.

[Pos09] Moacyr C. Possan, Andre B. Leal, “Implementation of supervisory control systems based on state machines”, *2009 IEEE Conference on Emerging Technologies & Factory Automation*, pags. 1–8, IEEE, 9 2009.

[Pto14] Claudius Ptolemaeus (ed.), *System Design, Modeling, and Simulation using Ptolemy II*, Ptolemy.org, 2014.

[Riv18] Daniel Rivas, Pragna Das, Joaquín Saiz-Alcaine, Lluís Ribas-Xirgo, “Synthesis of controllers from finite state stack machine diagrams”, *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, pags. 1179–1182, 2018.

[Riv19a] Daniel Rivas, Joan Jiménez-Jané, Lluís Ribas-Xirgo, “Auction model for transport order assignment in agv systems”, *Advances in Physical Agents*, pags. 227–241, Springer International Publishing, 11 2019.

[Riv19b] Daniel Rivas, Lluís Ribas-Xirgo, "Agent-based model for transport order assignment in agv systems", *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, Vol. 2019-September, pags. 947–954, 9 2019.

[Riv22a] Daniel Rivas, "Adaptable mrta system with task reallocation code repository", `https://github.com/DanielRivas88/Adaptable-MRTA`, 2022.

[Riv22b] Daniel Rivas, Lluis Ribas-Xirgo, "A multi-robot dynamic task allocation framework for indoor logistics", *Robotics and Computer-Integrated Manufacturing*, 2022.

[Riv23] Daniel Rivas, Lluís Ribas-Xirgo, "Integrating state-based multi-agent task allocation and physical simulators", *ROBOT2022: Fifth Iberian Robotics Conference*, pags. 576–587, Springer International Publishing, 2023.

[Roh13] Eric Rohmer, Surya P.N. Singh, Marc Freese, "V-rep: A versatile and scalable robot simulation framework", *IEEE International Conference on Intelligent Robots and Systems*, pags. 1321–1326, 2013.

[RX12] Lluís Ribas-Xirgo, Antonio Miro-Vicente, Ismael F. Chaile, A. Josep Velasco-Gonzalez, "Multi-agent model of a sample transport system for modular in-vitro diagnostics laboratories", *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, pags. 1–8, IEEE, 9 2012.

[RX13a] Lluís Ribas-Xirgo, Ismael F. Chaile, "An agent-based model of autonomous automated-guided vehicles for internal transportation in automated laboratories", *ICAART 2013 - Proceedings of the 5th International Conference on Agents and Artificial Intelligence*, Vol. 1, pags. 262–268, 2013.

[RX13b] Lluís Ribas-Xirgo, Ismael F. Chaile, "Multi-agent-based controller architecture for agv systems", *2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA)*, pags. 1–4, IEEE, 9 2013.

[RX21] Lluís Ribas-Xirgo, "Multi-agent system model of taxi fleets", *Advances in Intelligent Systems and Computing*, Vol. 1285, pags. 123–134, 2021.

[Sam00] Miro Samek, P. Montgomery, "State-oriented programming", *Embedded Systems Programming*, pags. 22–43, 1 2000.

[Sar18a] Chayan Sarkar, Sounak Dey, Marichi Agarwal, "Semantic knowledge driven utility calculation towards efficient multi-robot task allocation", *IEEE International Conference on Automation Science and Engineering*, Vol. 2018-August, pags. 144–147, 12 2018.

[Sar18b] Chayan Sarkar, Himadri Sekhar Paul, Arindam Pal, "A scalable multi-robot task allocation algorithm", *Proceedings - IEEE International Conference on Robotics and Automation*, pags. 5022–5027, Institute of Electrical and Electronics Engineers Inc., 9 2018.

[Sch17] Eric Schneider, Elizabeth I. Sklar, Simon Parsons, “Mechanism selection for multi-robot task allocation”, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 10454 LNAI, pags. 421–435, 2017.

[Sch18] Philipp Schillinger, Mathias Bürger, Dimos V. Dimarogonas, “Simultaneous task allocation and planning for temporal logic goals in heterogeneous multi-robot systems”, *International Journal of Robotics Research*, Vol. 37, pags. 818–838, 6 2018.

[See20] N. Seenu, R. M. Kuppan Chetty, M. M. Ramya, Mukund Nilakantan Janardhanan, “Review on state-of-the-art dynamic task allocation strategies for multiple-robot systems”, *Industrial Robot*, Vol. 47, pags. 929–942, 10 2020.

[Sha14] Anatoly Shalyto, “Automata-based programming and automata-based control”, 2014.

[Ste19] Roni Stern, “Multi-agent path finding – an overview”, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 11866 LNAI, pags. 96–115, Springer, 2019.

[Str08] Detlef Streitferdt, Georg Wendt, Philipp Nenninger, Alexander Nyßen, Horst Lichter, “Model driven development challenges in the automation domain”, *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pags. 1372–1375, IEEE, 2008.

[Tan21] Hongtao Tang, Xiaoya Cheng, Weiguang Jiang, Shouwu Chen, “Research on equipment configuration optimization of agv unmanned warehouse”, *IEEE Access*, Vol. 9, pags. 47946–47959, 2021.

[Tep22] Erich Christian Teppan, “Types of flexible job shop scheduling: A constraint programming experiment”, *ICAART Proceedings*, 2022.

[TL16] Jonatan Trullas-Ledesma, Negar Zakeri Nejad, David Quiros-Prez, Lluís Ribas-Xirgo, “A study on applied cooperative path finding”, Servicio de Publicaciones de la Universidad de Málaga (ed.), *XVII Workshop of Physical Agents*, pags. 115–121, 2016.

[Tur18a] Joanna Turner, “Distributed task allocation optimisation techniques”, *AAMAS ’18: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, 2018.

[Tur18b] Joanna Turner, Qinggang Meng, Gerald Schaefer, Amanda Whitbrook, Andrea Soltoggio, “Distributed task rescheduling with time constraints for the optimization of total task allocations in a multirobot system”, *IEEE Transactions on Cybernetics*, Vol. 48, pags. 2583–2597, 9 2018.

[Vya11] Valeriy Vyatkin, “Iec 61499 as enabler of distributed and intelligent automation: State-of-the-art review”, *IEEE Transactions on Industrial Informatics*, Vol. 7, pags. 768–781, 11 2011.

[Whi19] A. Whitbrook, Q. Meng, P. Chung, “Addressing robustness in time-critical, distributed, task allocation algorithms”, *Applied Intelligence*, Vol. 49, 2019.

[Woo18] Bradley Woosley, Prithviraj Dasgupta, “Integrated real-time task and motion planning for multiple robots under path and communication uncertainties”, *Robotica*, Vol. 36, pags. 353–373, 2018.

[Zha16] Wanqing Zhao, Qinggang Meng, Paul W.H. Chung, “A heuristic distributed task allocation method for multivehicle multitask problems and its application to search and rescue scenario”, *IEEE Transactions on Cybernetics*, Vol. 46, pags. 902–915, 4 2016.

[Zho17] Haotian Zhou, Huasong Min, Yunhan Lin, Shengnan Zhang, “A robot architecture of hierarchical finite state machine for autonomous mobile manipulator”, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 10464 LNAI, pags. 425–436, 2017.