
UNIVERSIDAD DE SEVILLA

Trabajo Fin de Máster
Máster en Ingeniería Electrónica, Robótica y Automática

Aerial co-workers: a task planning approach for multi-drone teams supporting inspection operations

Autor:
Álvaro Calvo Matos

Tutor:
Jesús Capitán Fernandez
Associate Professor

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Máster: Aerial co-workers: a task planning approach for multi-drone teams supporting inspection operations

Autor: Álvaro Calvo Matos
Tutor: Jesús Capitán Fernandez

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Acknowledgment

To my advisor, Jesús, for guiding me in this project, for trusting in me to join the research group to which he belongs, for supporting me in my decision to join a doctoral program, for always seeking the best for us despite his preferences and for his kindness.

To all those department colleagues who have helped me every time I have needed it and all those who have ever volunteered to support me. In particular, I would like to thank Fran and Arturo for all the time they have dedicated to helping me out.

To Damián, for accompanying me in the easy and difficult moments, but above all, for being my friend, and for being there unconditionally for whatever I needed.

To my classmates, who despite being a difficult year with social distancing, have been as close as ever.

To all my friends, for being such good friends.

To my entire family, for their unconditional love and support, and for their patience and understanding.

Thanks for everything

Álvaro Calvo Matos

Sevilla, 2021

Abstract

This master's thesis has addressed problems steaming from the recent increase in the applications of cooperative Unmanned Aerial Vehicle (UAV) teams, which are their autonomy to operate over a long period of time with robustness to possible failures, and the ability to enhance the team with cognitive capabilities so that they are able to operate in dynamic environments with humans.

Many of these applications are currently being executed by humans, making the activities much more expensive, time-consuming and, in some cases, even dangerous. This is why there is currently a great deal of interest and effort being put into developing solutions to the problems posed.

The aim of the work in this thesis was to develop cognitive planning techniques for coordinating fleets of quadrotors to assist human operators in inspection and maintenance tasks on high-voltage power lines. These techniques should also extend the autonomy of the system, ensure that safety requirements between UAVs and human workers are met, and ensure the success of the mission.

A software architecture has been proposed based on a central planner and a distributed behaviour manager. To carry out mission planning, cost functions for each incoming task have been defined. Thus, tasks are assigned to UAVs efficiently taking into account their battery constraints. Moreover, to control the behaviour of the UAVs and ensure the safety of the aerial equipment, behaviour trees have been implemented.

As a result, it has been possible to develop a software architecture capable of dynamically planning missions while ensuring the safety of the equipment involved. This provides a good base that can be easily adapted and from which more complex planners could be developed in the future. Compared to the typical way of implementing behaviour managers, involving complex finite state machines that are difficult to read, reuse and extend, the use of behaviour trees is a great improvement and will allow the creation of increasingly complex behaviours.

Resumen

Este Trabajo de Fin de Máster ha afrontado problemas que surgen del reciente aumento de las aplicaciones de equipos cooperativos de UAV, los cuales son la autonomía para operar de forma prolongada en el tiempo con robustez ante posibles fallos, y la dificultad de aportar al equipo capacidades cognitivas para poder operar en entornos dinámicos con humanos.

Muchas de estas aplicaciones están siendo ejecutadas actualmente por humanos, haciendo las actividaded mucho más costosas, lentas, e incluso en algunos casos, peligrosas. Es por eso que actualmente existe un gran interés y se están destinando muchos esfuerzos para desarrollar soluciones para los problemas planteados.

El objetivo del trabajo en este TFM es desarrollar técnicas cognitvas de planificación para coordinar flotas de UAVs que asistan a operarios humanos en tareas de inspección y mantenimiento en líneas eléctricas de alta tensión. Estas técnicas deben además extender la autonomía del sistema, garantizar que se cumplan los requisitos de seguridad entre UAVs y trabajadores humanos, y asegurar el éxito de la misión.

Se ha propuesto una arquitectura de software basada en un planificador central y un gestor de comportamientos distribuido. Para llevar a cabo la planificación se han definido costes para las distintas tareas existentes. De esta forma, se asignan a los distintos UAVs de manera eficiente, teniendo en cuenta sus restricciones de batería. Por el otro lado, para controlar el comportamiento de los UAVs y asegurar la seguridad de los equipos aéreos, se han implementado diferentes árboles de comportamiento.

Como resultado, se ha conseguido desarrollar una arquitectura de software capaz de realizar la planificación de las misiones de forma dinámica asegurando mientras tanto la seguridad de los equipos involucrados. Esto constituye una buena base que se puede adaptar fácilmente y a partir de la cual se pueden desarrollar futuros planificadores más complejos. Comparado con la forma típica de implementar gestores de comportamiento, ivolucrando complejas máquinas de estados finitas difíciles de leer, reutilizar y ampliar, el uso de árboles de comportamiento supone una gran mejora y permitirá la creación de comportamientos cada vez más complejos.

Short Outline

Contents

1 Introduction

The use of UAVs has grown considerably in recent years for numerous civil applications, including real-time monitoring, search and rescue, providing wireless coverage, security and surveillance, precision agriculture, package delivery and infrastructure inspection [**?**]. With the rapidly developing technology in this area, and demonstrations of what UAVs can do, there are increasing efforts to bring this technology to other applications. With the expected increase in applications for this technology, new problems and challenges arise, including autonomy, safety, obstacle avoidance and coordination of multi-UAV teams. Developing technology to solve these problems is a major effort, but as UAVs have proven to be critical in situations where humans are at high risk or highly inefficient, and they have proven their capacity to evolve and develop even more potential in the short term, companies are investing in developing all sort of UAV-based solutions.

1.1 Motivation

With the increase in global electricity demand, a challenge has arisen for electricity supply companies to maintain and repair power grids in a way that minimises the frequency of outages. According to [**?**], one of the main causes of power outages is damage to transmission lines due to bad weather or inefficient inspection campaigns.

Figure 1.1 Operators getting off the helicopter during a maintenance mission.

The strategy often used by electric companies to reduce power outages is to schedule periodic maintenance operations on active lines. This is the most suitable method if the correct functioning of the system is to be ensured and when replacing a circuit is unacceptable [**?**]. These maintenance missions are carried out by experienced crews on board helicopters and equipped with safety suits and harnesses among other things that prevent the operators from receiving an electric shock (see Figure **??**). The problem with this solution is that these activities are dangerous for the operators, as they are working at high altitude and on electrified lines, are extremely time-consuming and expensive ($1500 per hour) and are subject to human error [**?**].

These are the reasons why distribution companies have the need to develop more efficient and safer maintenance methods. Multiple solutions have been proposed to automate this task [**?**], but the best seems to be the use of UAVs, due to their flexibility and ability to inspect at different levels [**?**]. To achieve this, there are still some important barriers to overcome, such as the limited autonomy of these devices, the strong electromagnetic interference to which they would be subject due to being close to power lines, and the ability to detect and avoid obstacles of different nature that could be found in this type of environment [**?**]. Providing UAVs with the cognitive capability to operate autonomously in such dynamic environments and with the presence of humans, and providing them with a rapid on-line planning method [**?**], is key to address these complexities and to safely and successfully accomplish the assigned mission with UAV fleets.

A versatile and reliable software architecture is be essential to integrate and interconnect all the heterogeneous components that compose these cognitive multi-UAV systems. In [**?**], as part of the AERIAL-CORE European project[1], a multi-layer software architecture is presented for carrying out such missions cooperatively between human operators and a fleet of quadrotors. One of the software components involved is a high-level task planner. Its function is to coordinate the entire fleet of UAVs to generate high-level behaviours in order to efficiently, safely and successfully complete the maintenance or inspection mission. This type of work has the characteristic of being dynamic, since it is not possible to know in advance what the outcome of the inspection as such will be in order to plan offline, but rather, as the mission develops, new tasks will arise that the fleet will have to attend to. Therefore, the task planner should be able to react to unexpected events (new tasks, failure of a UAV, loss of connection, less autonomy than calculated, etc.) and to replan online. Thus, this high-level planner will be the main cognitive block of the system [**?**].

1.2 Objectives

The overall objective of this project is to develop a cognitive task planner in charge of governing the behaviour of multi-UAV teams for the inspection and maintenance of power lines in a collaborative way with human operators, being one of the software layers that compose the aforementioned software architecture [**?**] developed for the AERIAL-CORE European project. The fleet of governed UAVs acts as aerial co-workers and can perform various tasks such as delivering a tool to an operator, inspecting regions of the power line, or monitoring a worker while operating to ensure his safety. The planner receives both high-level input and feedback from the different platforms that make up the fleet, and processes all the information to elaborate a plan to manage the UAV team or modify it in reaction to an unforeseen event. To achieve this, the following objectives were defined:

- Ensure that resources are used and tasks are executed efficiently.
- Comply with all security requirements and ensure the integrity of aerial platforms and mission success.
- Be able to replan online to react to unforeseen events.

[1] AERIAL-CORE European project homepage: *https://aerial-core.eu/*

- Implement the software layer in Robot Operating System (ROS) and manage the necessary communication with the rest of the software layers and modules that make up the whole architecture.
- Carry out Software In The Loop (SITL) simulations to prove that the algorithm is able to govern the behaviour of the fleet efficiently and safely, and that it is able to react to unforeseen events dynamically, demonstrating cognitive capabilities.
- Design the task planner in such a way that it is easy to maintain, modify or extend, seeking to make it modular and reusable so that it can serve as a basis for the construction of planners for other applications.

2 Preliminaries

This chapter focuses on the current state of the art of those technologies related to this project, as well as on the software tools used for its development. In addition, related works in the literature are also described to establish the context of the project.

2.1 Current technology

Although in the last decade the use of UAVs has spread to a large number of applications, the origin of this technology dates back to 1898 with the invention of radio control and the appearance of the first unmanned aircraft, baptised with the name of drone. These were mainly used for military purposes (see Fig. **??**).

Figure 2.1 General Atomics MQ-1 Predator. A Remotely Piloted Aircraft (RPA). Source: Wikipedia.

Later, with the development of technology, computers were designed with sufficient size and computing power to run the software necessary to operate a UAV autonomously, and even to control aircrafts with more complex and even unstable dynamics (gliders [**?**, **?**], airships [**?**], quadrotors [**?**, **?**, **?**, **?**], multirotors [**?**], flapping wings [**?**, **?**, **?**], etc.) (see Fig. **??**). Even though computational capacity was still insufficient for some applications, the development of UAV systems was made possible by performing calculations on the ground. What was done was to run the critical and most important systems for autonomous flight on the on-board computer (controls, data acquisition, obstacle avoidance, etc.), and to run the more demanding calculations that are not necessary in real time on the ground computers [**?**].

Figure 2.2 GRIFFIN's flapping wing robot [**?**].

For an aerial vehicle to operate autonomously, it is necessary to acquire data from the environment and process it in real time. A large number of different sensor configurations as well as numerous data acquisition and processing techniques can be found in the current literature [**?**, **?**, **?**].

Once UAV technology reached sufficient capacity and autonomy, the first applications for both single [**?**, **?**, **?**] and multi-UAV [**?**, **?**, **?**] systems began to appear. There is a great interest in the latter, as they can be configured in different ways [**?**] and collect and process data in a distributed way, increasing the computational capacity of the system [**?**, **?**] and, generating global collective behaviour that emerges from interactions between a large number of UAVs that individually are relatively simple, known as swarming [**?**, **?**, **?**].

Current applications often require human presence to carry out certain decisions, with the human pilot overseeing that everything runs smoothly and providing the cognitive capacity to analyse the generally dynamic environment and react to unforeseen situations [**?**, **?**, **?**]. This is because providing a UAV with sufficient cognitive capacity to operate fully autonomously in dynamic environments is a very complicated task and requires a great deal of processing power. In recent years, UAV technology has evolved rapidly, benefiting from advances in computing and artificial intelligence. As processors are becoming more powerful, efficient and smaller, UAVs are becoming more and more powerful without increasing their weight or compromising their autonomy. With the increase in the number of operations per second that UAVs can perform, this opens up the possibility of using UAVs for previously unthinkable applications, applications that require a large amount of processing and usually have to be performed in real time [**?**, **?**]. At the same time, advances in artificial intelligence mean that the perception, analysis and sensory fusion capabilities of UAVs are getting better and better. Advances in technology are breaking down one of the barriers preventing UAV technology from achieving this level of autonomy, and with it, more and more research effort is being devoted to breaking down another barrier, i.e., developing software that enables UAVs to have cognitive capabilities.

Regarding related research, the recent European project AERIAL-CORE [1] is worth mentioning. In that project, major European robotics teams are jointly participating with the aim of developing a fully autonomous robotic system with sufficient cognitive capabilities to work together with human operators in inspection and maintenance work on electrical networks [**?**]. The European project PILOTING [2] aims to develop a complete inspection platform that will provide its users with the information they need to draw up maintenance plans for structures [**?**]. HYFLIERS [3] is another European project that focused on the inspection of long pipe arrays in hard-to-reach areas. This,

[1] AERIAL-CORE European project homepage: *https://aerial-core.eu/*
[2] PILOTING European project homepage: *https://piloting-project.eu/*
[3] HYFLIERS European project homepage: *https://www.oulu.fi/hyfliers/*

unlike the previous two projects, is not fully autonomous, but needs for a pilot to indicate the inspection points along the pipes, and to supervise the aerial robot while it operates [**?**]. It is also worth mentioning a recent NASA (National Aeronautics and Space Administration) achievement, which is the first flight of an UAV outside the Earth [**?**, **?**]. This is specifically the Martian helicopter called Ingenuity, whose mission was simply to take off, move around and land in the Martian atmosphere with the added difficulty that, due to the distance between the two planets, this had to be done completely autonomously.

2.2 Related work

According to the literature review conducted by Hazim Shakhatreh et al. in 2019 [**?**], the market value of UAVs for civil infrastructure inspection is expected to be more than $45 billion, representing 45% of the total UAV market. The development of heterogeneous UAV fleets and efficient algorithms for their communication and coordination is important to have multi-UAV teams capable of carrying out a successful inspection and maintenance mission. If the inspection equipment is to be fully autonomous, so that it can be operated by personnel not specialised in the piloting of aerial vehicles, a module capable of reacting to any unforeseen event and modifying the plan in real time if necessary is required. This module, which could be called a task planner, is usually part of a larger software architecture in charge of fleet management and which tries to provide intelligence to the system. The task planner developed in this thesis consists of two distinct modules: the task planner itself and the behaviour manager.

2.2.1 Task planning in multi-UAV teams

There are numerous proposals for solving the task planning problem, each with its strengths and weaknesses. Jiang et al. [**?**] compared the performance of different types of planners in the literature. The conclusion they reached was that Planning Domain Description Language (PDDL)-based planners are better on problems requiring long solutions, while Answer Set Programming (ASP)-based planners are less susceptible to domain object augmentation. When complex reasoning is required, ASP-based solutions can be considerably faster than PDDL-based solutions. In [**?**], they present a standardised integration of probabilistic planners in ROSPlan, which is a framework for task planning in ROS. By integrating RDDL (Relational Dynamic Influence Diagram Language) into ROSPlan, they allow combining deterministic and probabilistic planning within the same framework.

An example of probabilistic planning is presented in [**?**], where they use an improved multi-objective particle swarm optimisation algorithm to solve a task allocation problem for multiple UAVs. The system consists of two phases with which they try to accelerate convergence and avoid the algorithm falling into local minimum. The results shown by this study reveal a good performance in solving this kind of problems. Using a completely different approach, [**?**] propose an intelligent task planner focused on fuzzy neural networks. This system selects the best action from a set of possible actions. Time-constrained planning is also something that has been researched, e.g., [**?**] incorporate time constraints into task planning for multi-agent systems. The proposed solution consists of two phases, first all execution times are pre-computed in a decentralised way and then the possible configurations are tested so that the collective execution time is guaranteed. Last, the work in [**?**] is worth mentioning. They propose a final planning done by humans, but also a series of control interfaces and coordination algorithms that assist humans in the decision-making process and compute the shortest path for each UAV, taking into account the task priorities and the type of UAV required for each of them. The human commander can modify the plan or include constraints, such as assigning a task to a particular UAV, and he has to review and accept the plan suggested by the algorithm once it is finished.

The task planner proposed in this thesis tries to integrate together positive features such as robustness to failures, the capacity to react and replan online in case of any unforeseen event, the incorporation of capacity restrictions like the type of UAV, task priorities or the battery level of each UAV, etc. All that in an automatic way without the need for human supervision.

2.2.2 UAV behaviour management

In this thesis, *UAV behaviour manager* refers to what would commonly be controllers and safety modules executed at different times and levels. Once the aerial vehicle has concrete orders of what to do, i.e., once it has an assigned plan, it is time to execute the controls in charge of carrying out that plan. On the one hand, it will be necessary to execute some control in charge of supervising and ensuring the integrity of the aircraft. In [**?**], a Finite State Machine (FSM) is used in which one of the states is in charge of managing emergency situations, to which it transitions when another module detects and communicates the emergency condition. On the other hand, a high-level controller has to be executed in charge of calling the corresponding low-level controllers at any given moment. In this last study, they develop the complete flight control using FSM that transition from one state to another depending on the information provided by the sensors. The use of finite state machines is in fact the most common. In [**?**], for example, an automatic landing system programmed in this way is presented. Vitor de Araujo et al. [**?**] present a solution for the control of UAVs in search and rescue tasks based on a Parallel Hierarchical Finite State Machine (PHFSM) with which they claim to achieve many improvements with respect to other typical implementations.

The problem with this approach is that state machines are difficult to scale and reuse. Moreover, there comes a point where it becomes even difficult for a human to interpret. There is therefore a problem with further increasing the capabilities of UAVs using state machines. As discussed in [**?**], Behaviour Trees (BTs) are an alternative that provide, among other advantages, scalability, modularity and readability, and they can also be used for UAV mission management. In [**?**], they also emphasize the advantages of using this type of system for UAV control.

Although BTs are already widespread in the videogame industry, there are still not so many proposals that use them for the management of autonomous systems. Therefore, the module in charge of controlling the behaviour of each UAV in this project has been developed using BTs.

2.3 Tools

This section discusses the most relevant software tools used in this project and briefly explains which role they have played in it.

2.3.1 ROS

The Robot Operating System (ROS) [4] is more a framework than an operating system. It is a collection of tools, libraries and conventions that aim to facilitate the development of software for robots. The aim of this tool is to enable research teams from all over the world to collaborate with each other and to take advantage of each other's work. ROS is present in most robotics projects today. In this project, it forms the basis on which everything else is coded, as it allows the use of other useful tools developed by the community, such as those mentioned below.

2.3.2 Gazebo

This is a simulation tool usually used by the robotics community to accurately simulate and test their systems. In addition to being open-source, it allows the entire system to be tested, from the design

[4] ROS homepage: *https://www.ros.org/*

of the robot itself to its programming. In this project, Gazebo has been used to test the developed software safely both in the development phases and in the final stages of testing and verification.

2.3.3 Rviz

Rviz is a tool for visualising information in 3D. It also allows to graphically represent the information received by a robotic system. It can be integrated with ROS applications, being very useful for debugging tasks. Rviz is used in this project to visualise the position of each of the UAVs as well as different details about the simulation for monitoring and debugging purposes.

2.3.4 UAL

As its name suggests, UAV Abstraction Layer (UAL) [5] is a software layer that simplifies the process of developing and testing high-level algorithms for aerial robots by standardising and simplifying the interfaces with these robots. In addition, UAL can work with both simulated and real platforms [**?**]. This software is also available in ROS for free use in any robotics project. Although in this thesis a higher level software layer has been developed, which does not have to communicate directly with the autopilot of any of the aerial vehicles, it has been necessary to communicate with them in order to test the correct functioning of the task planning approach in simulation. Therefore, UAL has been used in this project to programme at a low level the movement of the UAVs, ignoring which autopilot they will incorporate in the future.

2.3.5 Behaviour Trees

BehaviorTree.CPP [6] is a C++ 14 library for creating behaviour trees. Its design features include flexibility, ease of use and speed. This library, among other things, allows the creation of asynchronous *actions*, enables reactive behaviours that execute multiple *actions* at the same time, permit to load trees at runtime, and provides a type-safe and flexible mechanism to do *dataflow* between *nodes* of the tree. This library has been chosen over others that fulfil the same function because of its extensive documentation [7] and support. Its function in this project has been the creation of BTs for the programming of the UAV behaviour manager in a modular, scalable and easily reusable way.

2.3.6 Groot

Groot[8] is a graphical editor to create BTs (see Fig. **??**). It is compliant with the library used in this project to create behaviour trees, BehaviorTree.CPP. This graphic interface can also be used to monitor a running behaviour tree. For programming BTs in this project, a simple XML file editor has been used, so Groot's role was to monitor the execution of the BTs during testing.

[5] UAL repository: *https://github.com/grvcTeam/grvc-ual*
[6] BehaviorTree.CPP repository: *https://github.com/BehaviorTree/BehaviorTree.CPP/*
[7] BehaviorTree.CPP documentation: *https://www.behaviortree.dev/*
[8] Groot repository: *https://github.com/BehaviorTree/Groot*

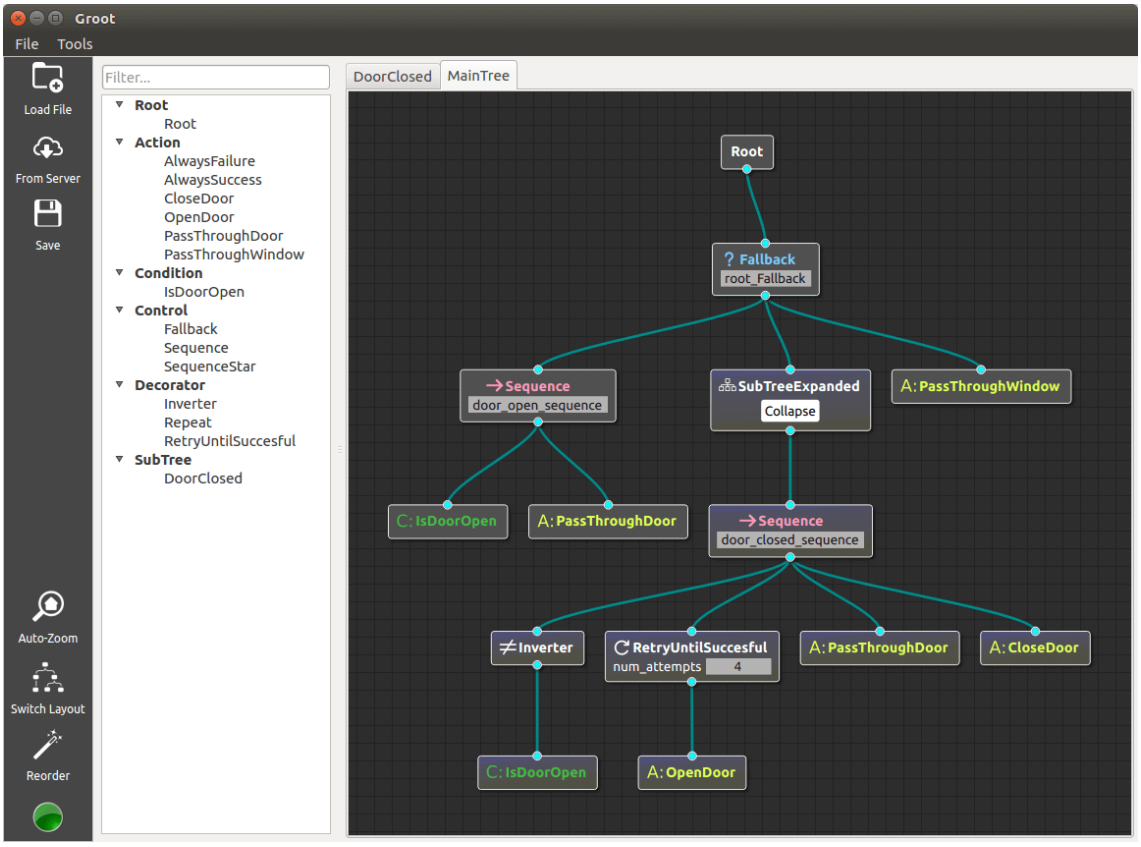

Figure 2.3 Groot graphical interface for editing behaviour trees. Source: Github.

3 Problem Formulation

As mentioned in Chapter **??**, the context around which this cognitive task planner is being developed is the inspection and maintenance of electrical networks. Although one of the objectives is to build a task planner whose characteristics allow its easy reuse and adaptation for other applications, it is relevant to state the problem for which it is being originally prepared.

As already mentioned, the AERIAL-CORE project aims to develop different technologies for the use of multi-UAV system in inspection and maintenance tasks in high-voltage electrical installations. In particular, one of the technologies proposed is the use of Aerial Co-Worker, i.e. small teams of cooperative UAVs to safely support maintenance workers while working at height on power lines. These systems would have to interact with humans (see Figure **??**) to inspect certain parts that are indicated to them, monitor worker safety during operation and deliver tools or other light equipment, in order to make the work more efficient and safer. In addition, to have a greater impact, the system would need to operate over extended periods of time, being able to autonomously deal with certain faults or recharges.

Figure 3.1 Multi-UAV team supporting an operator. Source: Aerial-Core website.

Three types of ACWs are referred, each intended to provide different functionality: *Inspection-ACW*, *Safety-ACW*, and *Physical-ACW*. The use case scenarios can be summarised as follows:

- *Inspection*, where a fleet of ACWs (i.e., *Inspection-ACWs*) carries out a detailed investigation of power equipment autonomously, helping the human workers to acquire views of the power tower that are not easily accessible (see Figure **??**);

- *Safety*, where a formation of ACWs (i.e., *Safety-ACWs*) provides the supervising team with a view of the humans working on the power tower in order to monitor their status and to ensure their safety (see Figure **??**);
- *Physical*, where an ACW (i.e., *Physical-ACW*) physically interacts with the human worker and provides physical assistance to it, i.e., while in contact with the human it flies stably, reliably, and accomplishes the required physical task (e.g., handover of a tool) without becoming harmful for the human worker (see Figure **??**).

Even if there is a specific type of ACW for each of the tasks (inspection, monitoring and tool delivery), this does not mean that a UAV can at any given time undertake a task for which it is not the best. It will therefore be the planner's task to take into account which ACWs are best suited for each task, which are not but could perform it without problem, and which do not have the capacity to perform it at all. As a consequence, the number of ways in which the mission planning can be carried out multiplies, thus considerably increasing the difficulty of the problem that the task planner has to solve.

This mission planning problem with multiple UAVs with battery constraints can be posed as an optimisation problem, the solution of which indicates the most efficient way to allocate the different tasks and plan recharges. To react to possible failures, one of the most widespread options is to come up with dynamic methods that can replan in real time as certain events occur. Although there are many variants, most formulations for missions where multiple vehicles visit multiple locations to inspect or make deliveries give rise to NP-hard optimisation problems and, therefore, the most widespread approach is to solve them using heuristic algorithms.

Planning methods based on uncertainties are appropriate for adding cognitive capabilities to a system that has to interact with humans in dynamic environments, as they allow optimising plans by predicting the most likely intentions of humans and the outcomes of future actions. The main problem is their computational complexity, as the plan search space would grow exponentially with the number of UAVs and with the future time horizon over which planning is to take place.

It is in this context and with these ideas in mind, the cognitive task planner in this thesis was developed. As this cognitive planner is a module part of a bigger software architecture to tackle the whole problem, the complete picture of that architecture is briefly introduced. Mainy, which information exchanges exist between the upper and lower layers of the software architecture, the interfaces by which this information travels, and the interactions between layers to activate low-level controllers. In the following section this information is presented by individually explaining the different tasks contemplated in the project.

Additionally, a review will be made of other important considerations that the planner has to take into account, such as battery recharges, connection losses and task rescheduling; analysing the different situations in which each of them can occur and their different causes.

3.1 Description of tasks

As mentioned above, three different types of tasks are envisaged in the project. These tasks are requested at any time by human workers through gestures. There will be a higher level software layer that processes the information contained in the gestures so that the planner receives an asynchronous communication from the upper layer containing the specifications of a new task. At this point, it is the planner's job to process the new information together with the information it already had in order to elaborate and implement a new plan. The same planner is also in charge of calling the low-level controllers when necessary and ensuring the safety of the UAVs and the fulfillment of the mission. Each task is explained in detail in the following sections.

3.1.1 Inspection tasks

This task can be performed by all three types of ACWs. It is the second highest priority task, with the tool delivery task being the only one that exceeds it. It consists of carrying out a detailed inspection of the specified areas of the power equipment (see Fig. **??**). The layer immediately above the task planner is responsible for passing it a list of waypoints (WPs) that define the inspection task, and the planner is responsible for deciding how many ACWs it recruits to execute the task and which of the available ACWs it assigns it to. Dividing the total WP list to be inspected into subsets and assigning each one to one of the ACWs selected for the task is the job of a low-level controller. Therefore, once the planning is executed, the tasks of this type are transmitted to the lower level layers with the total list of WPs to be re-inspected and a list with the identifications (IDs) of the selected UAV.

All the aforementioned communications will be carried out asynchronously, as the creation of the task by the workers, which triggers the whole sequence of actions, is done in this way.

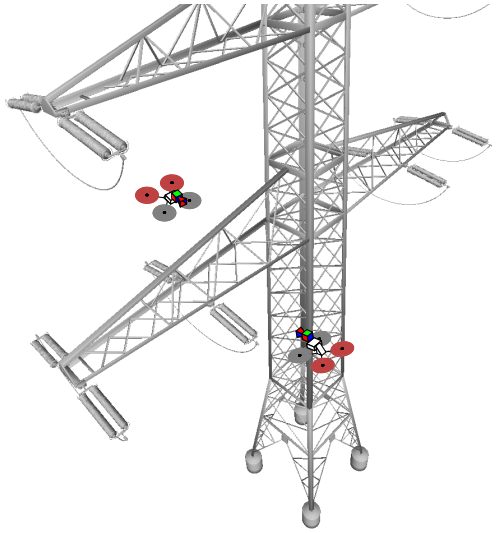

Figure 3.2 *Inspection-ACW* carrying out an inspection task.

3.1.2 Monitoring tasks

This task can also be executed by all three types of ACWs. It is the lowest priority task. Monitor worker's safety consists of providing the supervisory team with a view of the people working in the power tower to monitor their status and ensure their safety (see Fig. **??**). The layer immediately above the task planner communicates this time the ID of the worker to be monitored, the number of UAVs desired and the distance they should keep from the worker. It is the task planner's responsibility to decide once again which of the available ACWs to assign to this task and the formation they should maintain during the flight. Once the planning has been carried out, the tasks of this type are passed on to the lower level layers with both the original information and the information resulting from the planning.

The aforementioned communications will also be carried out asynchronously for the same reason.

3.1.3 Tool delivery tasks

This task can be performed only by *Physical-ACW* UAVs, as special hardware is required to perform the physical interaction with the low-weight objects and the human. This is the highest priority task. Delivering a tool consists of picking up a tool and transporting it to the worker, with whom a physical interaction will take place through which the delivery of the tool will take place (see Fig.

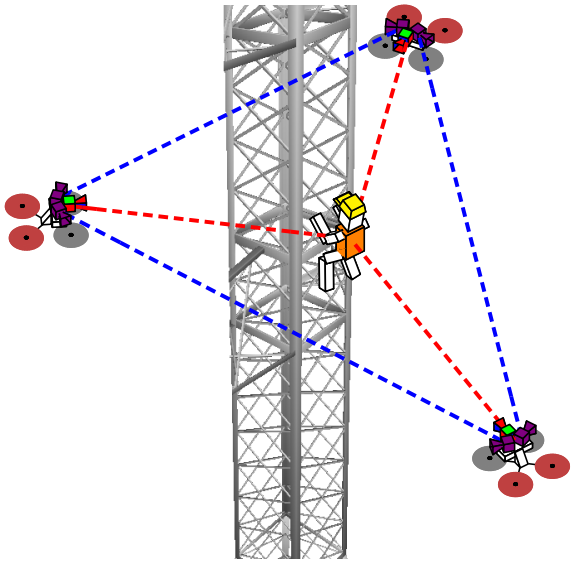

Figure 3.3 *Safety-ACW* carrying out a monitoring task.

??). Low-level controllers will have to be especially precise and careful not to hurt the worker. This time, the layer immediately above the task planner communicates the ID of the worker to whom the tool is to be delivered and the ID of the requested tool. Again, the task planner's mission is to decide to which of the available ACWs assign this task. Once the planning is carried out, tasks of this type are passed on to the lower level layers with the same information as originally.

The aforementioned communications, once again, will be carried out asynchronously.

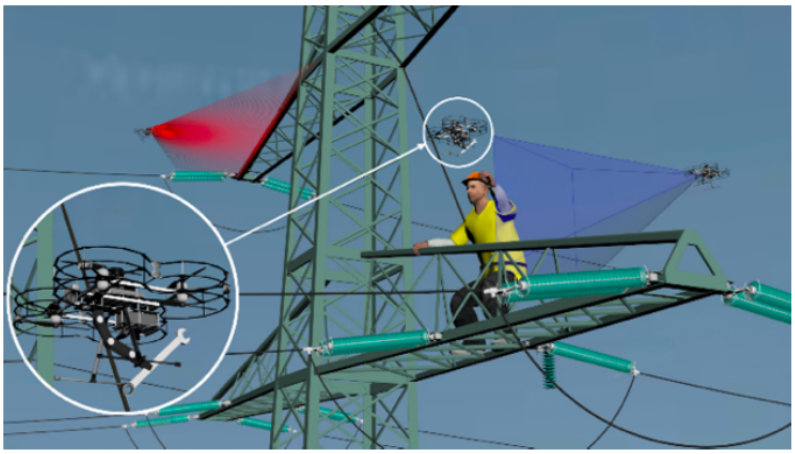

Figure 3.4 *Physical-ACW* carrying out a tool delivery task.

3.2 Battery recharges

Given the current autonomy problem with UAV technology, eventually each of the ACWs involved in the mission will run out of battery power. The moment when the battery of each one will run out can be estimated from the mission planning itself, so the planner can take this into account when distributing the tasks so that the ACW itself anticipates this event. Recharging does not necessarily have to take place when the UAV is about to run out of battery, nor does it have to take place until the battery reaches its maximum, but both will be parameters to be taken into account during the mission planning and optimisation process.

Besides, it is possible that the calculations may fail for some reason, and the battery may run out sooner than expected. It will therefore be necessary to periodically read the battery status and perform emergency recharging and replanning if necessary, reacting to unexpected faiulres. One possible scenario is that the aerial robot runs out of battery during a loss of connection. Since the planner is centralised at a ground station, there should also be a battery check and action module on board each aerial vehicle, and an emergency protocol in case this happens.

In the absence of specifications, it is assumed that battery recharging does not occur instantaneously (it is not a battery change), so reaching the desired battery level takes a certain amount of time that should be considered in the plan.

Also, the task planning algorithm has to be able to handle without blocking situations where all ACWs are simultaneously without sufficient battery power and therefore there is no UAV with which to execute a task immediately.

3.3 Connection losses

Another important consideration is the possible loss of connection between the centralised planner, where most of the cognitive capacity is concentrated, and one of the ACWs. Since a loss of connection is an unforeseen event, it is most likely that the planner will recalculate the optimal task allocation once the UAV fleet is updated, so that the tasks previously assigned to the disconnected ACW will be executed on another one. This is a potentially dangerous situation because the disconnected UAV could act autonomously according to its last plan and cause an accident with the rest of the agents that are still online.

It is therefore important: (i) to implement a system to detect disconnections from both sides of the communication, and (ii) to establish a common action protocol so that both modules know how the other is going to act, thus ensuring the integrity of all vehicles and the safety of the workers.

3.4 Task replanning situations

Once the mission is underway, any unforeseen event has the potential to completely change which the optimal plan is. Therefore, even if there is a possibility that the event will not affect the mission at all, it will always be necessary to execute a mission replanning in case of an unforeseen event.

The following is a list of the unforeseen events that have been contemplated in this work:

- Arrival of a new task.
- Modification of a task's parameters.
- Connection of a new ACW.
- Disconnection of an ACW.
- Unplanned insufficient battery in one of the ACWs.
- Battery drain faster than expected and therefore will not be enough for the current plan.
- An ACW finishes recharging ahead of schedule.
- A task is successfully completed.
- A task is completed unsuccessfully.

Note that some of the events considered are not really unexpected. Successful completion of a task is what is desired, for example, so it should not imply a change of plans. However, this event is included in the list because it is a good moment to check if there is a better plan and to modify the current one if necessary. As the planner is pursuing the optimal plan, the result of the replanning will keep the previous plan unchanged if it is still optimal.

4 Design of the proposed solution

This Chapter provides the details about the implementation of the solution to the problem in Chapter **??**: block diagram, pseudocode and inter-module communications. All the code is available online [1], and was developed under the Ubuntu 18.04 operating system and ROS Melodic.

The solution proposed follows a hierarchical approach, with a high-level planner in charge of activating different low-level controllers. The high-level planner detects the tasks required by the operators, and distributes them from the ground in a centralised way among the available ACWs, planning the necessary recharges throughout the mission. In addition, this planner reacts in real time to possible events by reassigning tasks. The low-level planners are on board each UAV and are responsible for executing contingency plans for these events while the central planner calculates and communicates the new plan. They will also be in charge of controlling the movement of the ACWs to execute the different tasks assigned by the higher-level module (e.g., flying to a location to be inspected or to the position of an operator waiting for a tool). From now on, the low-level module on board each UAV will be called the *Agent Behaviour Manager*, and the centralised module on the ground will be called the *High-Level Planner*. Together, these modules will provide cognitive capabilities to interact with humans efficiently in a dynamic scenario.

4.1 Block diagram

As stated in Chapter **??**, the developed task planner is part of a software architecture consisting of different layers, being the main cognitive block the central layer, the *high-level cognitive task planner*. Figure **??** shows a schematic of the software architecture from the perspective of the module implemented in this thesis, including the different blocks and their interfaces. The part of the diagram in grey would be the complete software architecture, including from the high-level module in charge of analysing the gestures made by the operators to extract the tasks from them, to the low-level controllers in charge of executing those tasks. The software layer corresponding to this thesis, in charge of high-level decision-making, is marked in blue-green. It is composed of the *High-Level Planner*, which is centralised and runs on a ground station (in orange) and the *Agent Behaviour Manager*, distributed on board each ACW (in lime).

In the software architecture scheme, although some communications are bidirectional, it can be seen that there is a main flow of information. Starting with the information arriving at the module *Gesture Recognition*, this propagates to the last layer, where the *Lower-Level Controllers* use the already processed information to command the ACWs. Table **??** shows the type of data that each of the modules in Figure **??** receives as input and the type of data that each of them sends as output. Additionally, Table **??** explains the details of the data types.

[1] Human aware collaboration planner source code: *https://github.com/grvcTeam/aerialcore_planning*

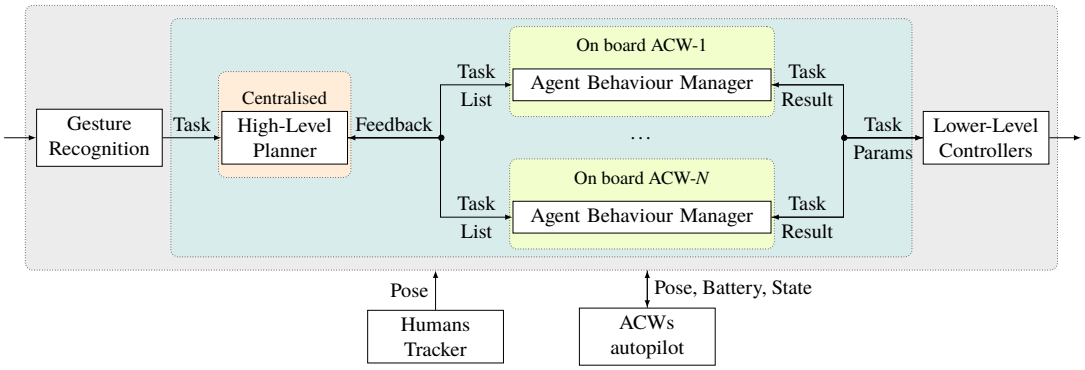

Figure 4.1 Software architecture: blocks and interfaces. Block diagram from the high-level cognitive task planner perspective.

Table 4.1 Description of the data interfaces for each software module.

Module Name	Input Data	Output Data
Gesture Recognition	Images	**Task, defined by:** Task ID, Task Type, Monitoring Distance, Monitoring Number, WP List, Tool ID (some task parameters will be ignored depending on Task Type)
High-Level Planner	Task, Feedback (Task Result, BatteryEnough, BT info), Humans' Pose, ACWs' Pose, Battery and State, and Agent Beacon	Task List adding to each one its extra parameters result of the planning (Formation and/or List of ACWs' IDs) and Planner Beacon
Agent Behaviour Manager	Task List, Low-Level's Result, Human Pose, ACWs Pose, Battery and State	Params needed by Low-Level Controllers (depending on Task Type), Feedback (Task Result, BatteryEnough, BT info) and Agent Beacon
Low-Level Controllers	Params (depending on Task Type)	Result
Humans Tracker		Pose
ACW autopilot	Low-Level orders	Pose, Battery and State

The first module is constantly checking the images captured by the UAVs for a gesture that is indicating a new task or the modification of an existing task. When this occurs, it asynchronously sends out a task, which will be picked up by the centralised planner. As shown in Table **??**, this communication includes the unique ID that differentiates this task from the others, the type of task, and the parameters that define it.

The *High-Level Planner*, when it receives this information, proceeds to re-evaluate the optimal plan taking into account the task received, the information it receives from the *ACWs' autopilot*, and the position of the operators, which is periodically published by the *Human Tracker*. These data constitute the input for the *High-Level Planner*, together with the feedback coming from each *Agent Behaviour Manager*. Its output is a list of tasks for each ACW.

On board each ACW there is an *Agent Behaviour Manager*. This module is in charge of collecting the corresponding task list provided by the centralised planner. With this input and the information coming from the *Human Tracker* and the *ACW's autopilot*, this module is in charge of calling the *Lower-Level Controllers* to carry out the execution of the assigned plan. The information emitted by the *ACW's autopilot* is also used to check that everything is working correctly and to execute the security protocols in case they are necessary. If this happens, the corresponding communication would be issued back to the *High-Level Planner* in order to calculate a new plan. Thes modules

Table 4.2 Description of data types.

Data name	Data type	Comment
Task ID	String	Unique identifier of each task
Task Type	Integer	Task type indicator: m/M, i/I or d/D
Human Target ID	String	Unique identifier of each human worker. The position of the human target and other needed info is supposed to be known and accessible via its ID.
Monitoring Distance	Float	Distance from which the ACW surveil the worker during a safety monitoring task
Monitoring Number	Integer	Number of ACWs that are required in formation for a certain safety monitoring task
WP List	List of 3 float tuples (x, y, and z)	List of waypoints to be inspected
List of ACWs' IDs	List of Strings	List of the unique identifiers of the ACWs that have been selected for a task that requires multiple ACWs
Formation	Integer	Indicates which of the predefined types of formations should be used for monitoring (e.g., circle, triangle)
Tool ID	String	Unique identifier of the tool to be delivered
ACW's Pose	geometry_msgs /PoseStamped	ACW's Position and orientation
ACW's Battery	sensors_msgs /BatteryState	Percentage of battery in the ACW
Task Result	String, Boolean	First one is the task unique ID and second one its result once it's finished
Battery Enough	Boolean	Result of computing if an ACW will have enough battery for its current task
BT info	String list	Status of each BT's node in its last execution (Running, IDLE, SUCCESS or FAILURE)
Agent Beacon	String, String	First one is the ACW's unique ID while the second one defines ACW's type (SafetyACW, InspectACW, or PhysicalACW). It is used as heartbeat and to detect new ACWs in Planner
Planner Beacon	Time	ROS::Time message containing the time when the beacon was sended. It is used to check the status of the connection from Agent's side.
Lower-Level's Result	Boolean	Result of the Lower-Level Controllers once they have finished after being called

also receive the *Lower-Level Controllers*' result after calling each of them, and publish it back to the *High-Level Planner* as feedback.

In addition to these communications, the modules *High-Level Planner* and *Agent Behaviour Manager* periodically exchange beacons that are used to detect both the connection of a new ACW and its disconnection in case of failure. Moreover, there is an asynchronous communication that is broadcast to all components indicating the end of the mission when this happens.

Finally, it is worth mentioning that the *Gesture Recognition* module does not have a communication aimed at modifying parameters of a task already contemplated within the *High-Level Planner*. However, this is possible because tasks have a unique identifier. Once a task has been delivered to the *High-Level Planner*, in order to change any of its parameters, the *Gesture Recognition* module just has to submit the task again, keeping the same task ID and updating only the desired parameters. Thus, within the function that is executed when a new task is communicated, another function is called to update the parameters of tasks that have already been registered.

4.2 Centralised module: High-Level Planner

As mentioned above, the *High-Level Planner* is a centralised module running on a ground station and constitutes the main cognitive module of the software architecture. Its purpose is to plan the mission in an optimal way, i.e., to distribute the pending tasks among the available ACWs by specifying the order in which they are to be executed, taking into account the time it takes to complete each one, the type of each UAVs, the distance each one will have to travel, the battery they have available, the task each one was executing, the priority of each task, the battery consumed by each task, the recharges that will be needed, and when it is best to carry out those recharges.

The general pseudocode for this component from launch to termination is depicted in Code **??**.

Code 4.1 General operation of *High-Level Planner*'s code.

```
1. Read from a ros::param the address of the configuration file.
2. Read from the configuration file all necessary information.
3. Configure ROS communications (Publishers, Subscribers and
   ActionServers).
4. Set the loop rate.
5. Main "while" loop. While ros::ok() and not mission over do:
  5.1. Check the timeout of the Agents' beacons.
  5.2. Publish a new Planner beacon.
  5.3. Check for pending incoming communications (ros::spinOnce).
  5.4. Sleep the remaining time to send the next beacon.
6. Wait until all UAVs have finished and disconnected. While there is
   any agent connected do:
  6.1. Check the timeout of the Agents' beacons.
  6.2. Check for pending incoming communications (ros::spinOnce).
  6.3. Sleep for a while.
```

Since the environment in which the UAVs operate is dynamic, this module has been programmed in such a way that it can react to unforeseen events and recalculate the optimal plan. As it can be deduced from Code **??**, everything works through callback functions. Every time a communication arrives from another ROS node, a response is triggered on this node. The information contained in the message is analysed and it is decided whether a replanning is necessary or not. The situations in which a replanning has been deemed necessary are listed in Section **??**. The communications summarised in Tables **??** and **??** and in Figure **??** are sufficient to detect these unforeseen events and to be able to respond to them in the best possible way.

Code 4.2 Task callback pseudocode.

```
1. If the task already exists:
  1.1. If the new task's type is the same as old one's type:
    1.1.1. Update parameters, perform a task planning and return.
  1.2. Else: Warn operators that a pending task is going to be deleted
       and delete old task.
2. Read the type of task and the parameters that apply to it.
3. Add the new task to the pending task list.
4. Perform a task planning.
```

There is a callback that is executed when the node *Gesture Recognition* sends a task, which in case the given task is correct, always ends up calling the function in charge of calculating the

optimal plan (see Code **??**); the mission over callback, whose only action is to change the value of a variable so that the node exits the main while loop; and finally the agent's beacon callback, which is executed every time a UAV beacon is received and whose pseudocode is in Code **??**.

Code 4.3 Agent's beacon callback.

```
1. Read the information contained in the beacon.
2. If it is a connection of a new UAV:
  2.1. Register it in the database.
  2.2. Perform a task planning.
3. Else, if it is the heartbeat of an already known UAV:
  3.1. Reset the timeout timer.
```

The action carried out by the agent's beacon callback varies depending on whether it is the beacon of a new UAV or the heartbeat of a known UAV. For each agent there will be an object in the database that will contain another series of callbacks that will be in charge of receiving the messages coming from the ACWs and respond accordingly.

Code 4.4 Callback that runs when an *Agent Behaviour Manager* sends battery feedback.

```
1. Update the value of the internal flag associated with the battery.
2. Perform a task planning.
```

The *Agent Behaviour Manager* block only sends communications messages indicating the battery status when it is due to an unplanned event. This event can be either an early battery depletion or a faster than expected recharge. In both cases, the callback function, whose pseudocode is Code **??**, updates the value of an internal variable used during planning, and recalculates the optimal plan.

The other possible communication coming from a node of type *Agent Behaviour Manager* with the ability to trigger a reaction in the planner is due to the termination of a task. When a task finishes successfully, it is simply removed from the list of pending tasks. In this case, the emphAgent Behaviour Manager block also removes the task from its queue, which is the only case where it does so. In addition, this moment is used to re-evaluate the optimal plan. It is expected that the mission is still within the optimal plan, so in that case the planning result should be the same as the plan that was already being executed. If, instead, conditions have changed since the last planning and there exists a better plan now, it is at this point that the plan is updated. If the task ends with a failure, the callback action will depend on the causes of the failure (note that the interruption of a task will result in a failure). If the interruption is due to the UAV battery, it may be planned, in which case no action is required, or it may be unexpected, in which case the corresponding actions are taken by the battery callback. Once it has been verified that the task has not finished due to the battery, a check is made to see if the task was at the beginning of the queue. If so, a failure has indeed occurred, so the operators are warned, the task is removed from the list, and a replanning is executed. Otherwise the task in question would have been moved from the top of the queue due to a change of plans and therefore no action would have to be taken either. The pseudocode corresponding to what has just been explained is in Code **??**.

Code 4.5 Callback that runs when an *Agent Behaviour Manager* sends a task result.

```
1. Read the information contained in the task result.
2. If the task result is SUCCESS:
  2.1. Delete it from the pending tasks list.
  2.2. Perform a task planning.
```

```
3. Else, if the task result is FAILURE:
  3.1. If the task has been halted because of not having battery
     enough:
    3.1.1. Return.
  3.2. Else, if the task is on the front of that ACW's task queue:
    3.2.1. Notify operators that a task has failed and is going to be
       deleted.
    3.2.2. Delete task from the pending tasks list.
    3.2.3. Perform a task planning.
  3.3. Else:
    3.3.1. Return.
```

The other two communications received by the *High-Level Planner* from the ACWs are sensor readings corresponding to the UAVs' position and battery percentage. In both cases the only action of the corresponding callback is to update the information with the new values.

The last function that remains to be explained of those that can potentially request a replanning of the mission is the one in charge of checking the timeout of the agents' beacons. As shown in Code **??**, this function is not a callback like the previous ones, instead it is executed periodically in the main while loop. Its operation is shown in Code **??**. Basically, for each agent connected, it checks that the timeout amount of time has not elapsed since its last beacon was received. If a timeout has occurred, that ACW is considered disconnected and is removed from the centralised node data. If, after checking all agents, the number of connected UAVs has decreased, i.e. if any of the previously connected UAVs has disconnected, a mission replanning is executed.

Code 4.6 Beacons' timeout check function.

```
1. For each agent connected:
  1.1. If the elapsed time since the last beacon is greater than the
     timeout time:
    1.1.1. Add that agent's ID to the list of disconnected agents.
2. While the list of disconnected agents is not empty:
  2.1. Take first ID from the list.
  2.2. Erase from the block's data all information related to that ID.
3. If any agent has been disconnected:
  3.1. Perform a task planning.
```

The pseudocode that is executed when one of these functions deems it necessary to perform a new task planning is summarised in Code **??**. It is important to remember that some tasks have a higher priority than others, and this depends only on the type of task. To simplify the process, it has been decided to allocate the tasks in order of arrival, assuming that between two tasks of the same type, the one that arrived first will have priority. When a new task is received, it is stored both in a *std::map* that contains all the pending tasks to facilitate the access to the information, and in a *std::vector* with task types, where the order of arrival is maintained. What this simplification allows is to assign tasks one at a time. By having a prioritised list of tasks and assuming that no task can be assigned before a task with a higher priority, the mission planning problem is reduced to calculating the cost of each task individually for each UAV with the ability to execute it and assign it to the one with the lowest cost. For monitoring-type tasks, the selection of the required number of agents is strictly cost-based. The N agents that cost the least to execute the task are selected. The same is a little more complex for the tasks of type inspect, where the number of agents to select is a parameter to be defined by the planner itself. This value is first set according to the number of points to be inspected. Up to three points, a single ACW is selected; up to six points, two are

selected; and from seven points onwards, three agents are selected, this being the maximum number imposed by the low-level controller. Moreover, as the low-level controller in charge of this task works, all the ACWs selected for this task are required to start executing it simultaneously, so a second approximation of this number is made according to the number of idle UAVs. Thus, if they are assigned this as the first task, they will start executing it simultaneously. Academically, this simplification seems to deviate from the optimal solution, but it should be recalled that this work is part of a software architecture that will operate in real situations. In such situations, it is not expected that there will be a large number of UAVs connected simultaneously, nor a long list of pending tasks. In such simplified scenarios, this assumption makes sense without deviating too much from the optimal solution. Finally, the number of agents to be selected will be the smaller of the two above, being equal to one when there is no UAV idle and zero in case there is no ACW with enough battery. In the latter case, the task would be assigned after recharging. Once the number of agents to be selected has been defined, the agents that have the least cost to execute the task are selected from among those that meet the conditions described. Having selected the ACW that will carry out the task, all that remains is to distribute among them the WPs to be inspected. Although the algorithm in charge of performing the optimal distribution is in the low-level controller of this task, as the rest of the modules that make up the software architecture are not yet available, it has been necessary to program a distribution algorithm in order to be able to carry out the experiments. More details on this will be given in Section **??**.

The cost for each UAV is calculated as the weighted sum of three different types of costs. A first cost assesses the type of ACW and penalises the assignment of tasks to those UAVs designed for another type. It penalises especially the assignment of lower priority tasks to agents designed to perform higher priority tasks. The second cost evaluates the total distance the UAV will have to travel from where it is at the beginning of the task to where it should be by the end of the task. This cost is an approximation of the expected battery consumption, although it does not take into account intermediate travel and hoovering times during the mission. The last cost penalises the interruption of the task that was being executed according to the previous plan and rewards the assignment of the same task. This cost is intended to ensure that a task is preferentially assigned to an idle UAV, to an UAV that is executing a lower priority task, or even to an UAV of a different type, rather than interrupting a task unnecessarily just because that ACW has to travel a shorter distance, for example.

Code 4.7 Task planning function's pseudocode.

```
1. If there is any agent connected:
  1.1. For each agent connected:
    1.1.1. Make a copy of the current task queue.
    1.1.2. Empty the task queue.
  1.2. For each Tool Delivery task:
    1.2.1. Compute the cost of the task for each PhysicalACW that has
       enough battery.
    1.2.2. Assign the task to the agent for whom the task costs the
       least (from those who has enough battery).
    1.2.3. Add the task to that agent's task queue.
  1.3. For each Inspection task:
    1.3.1. Extract from the task parameters the list of WP to inspect.
    1.3.2. For each ACW (any type) that has enough battery:
      1.3.2.1. Compute the cost of the task for that ACW.
      1.3.2.2. Check if that ACW is still idle.
    1.3.3. Calculate the number of agents to select for the task based
        on the number of WP and the number of idle agents.
```

```
      1.3.4. If no agent has enough battery, continue.
      1.3.5. Else, if the number of agents to select is equal to zero,
         assign the task to the agent that costs the least.
      1.3.6. Else, select the calculated number of agents for whom the
         task costs the least.
      1.3.7. Divide the WP to inspect among the selected agents.
      1.3.8. For each selected agent:
        1.3.8.1. Set the remaining task parameters (List of selected
           ACWs' IDs and divided WP list).
        1.3.8.2. Add the task to the agent's task queue.
    1.4. For each Monitoring task:
      1.4.1. Compute the cost of the task for each ACW (any type) that
         has enough battery.
      1.4.2. If the required number of ACWs for the task is zero:
        1.4.2.1. Warn operators that this parameter can not be zero.
        1.4.2.2. Delete task from pending tasks.
      1.4.3. Else, select the requested number of agents for whom the
         task costs the least.
      1.4.4. Set the remaining task parameter (List of selected ACWs'
         IDs)
      1.4.4. Add the task to each selected agent's task queue.
    1.5. For each ACW connected, send the new task queue to its Agent
       Behaviour Manager.
  2. Else:
    2.1. Warn operators that no agent is connected.
```

Once the calculation of the mission plan has been completed, the new task queues are sent to the corresponding distributed modules. Each *Agent Behaviour Manager* will react to this communication and will take care of executing the newly assigned plan. In the meantime, the *High-Level Planner* block returns to the main while loop to continue waiting until an event that triggers a replanning occurs again.

4.3 Distributed module: Agent Behaviour Manager

This component is in charge of executing the plan assigned by the *High-Level Planner*, checking the security of the UAVs at all times, detecting unforeseen events and communicating them to the centralised node so that it can make a change of plans if needed. The *Agent Behaviour Manager* will communicate with the low-level controllers, handing over control when necessary to complete the assigned plan.

The general structure of this module is quite similar to that of the central module. The pseudocode is summarised in Code **??**. Upon initialisation, the *Agent Behaviour Manager* prepares the necessary information to start its operation, configures the necessary communications, declares and initialises the behaviour tree and, once the UAV has finished initialising, starts sending beacons to the central node to notify that it is joining the mission. Once the code finishes initialising and reaches the main while loop, the activity of the *Agent Behaviour Manager* concentrates on the execution of callbacks in response to incoming messages, as in the *High-Level Planner*, and on the execution of the behaviour tree, which directs and supervises the UAV movement.

Code 4.8 General operation of *Agent Behaviour Manager*.

```
1. Read from a ros::param the beacon's content (ACW's ID and type).
2. Read from a ros::param the address of the configuration file.
3. Read from the configuration file all necessary information.
4. Configure ROS communications (Publishers, Subscribers and
   ActionServers).
5. Set the loop rate.
6. Declare the behaviour tree.
7. Initialise each BT node.
8. Start BT loggers to facilitate debugging and monitoring of the node
   's performance.
9. Wait until the ACW fully initialises.
10. Main "while" loop. While ros::ok() and BT status is running:
  10.1. If a timeout of Planner's beacons has not ocurred:
    10.1.1. Publish a new Agent beacon.
  10.2. Check if battery is enough for the current task.
  10.3. Check for pending incoming communications (ros::spinOnce).
  10.4. Sleep the remaining time to send the next beacon.
```

The BTs are who governs the ACWs to perform each of the assigned tasks. Each BT monitors its ACW's battery and task status and reacts to any possible failure or unexpected event, requesting a new re-planning to the *High-Level Planner* in case of need. A BT can be defined as an improved FSM. They are a more advanced mechanism to implement behaviours, especially because of their advantages in terms of scalability, modularity, readability and reusability, facilitating the creation of more complex behaviours with less effort.

Despite this, the process of designing a state machine is quite different from the process of designing a behaviour tree. Designing behaviour trees without ever having done it before is not a trivial task. Moreover, there will be more than one valid implementation to achieve the same behaviour, which makes it more complicated to design this type of solution when you do not still have enough intuition to know which one is better. Taking advantage of the fact that the use of BTs is widespread in the videogame industry, information about them was gathered and studied to try to develop enough knowledge and intuition to design from scratch a BT that meets the needs of the mission. For that, previous examples found in [**?**, **?**, **?**] were very useful.

Before proceeding with the explanation of the designed BT, the types of nodes that can be found in the selected C++ library (see Figure **??**) and the functioning of each of them will be briefly discussed.

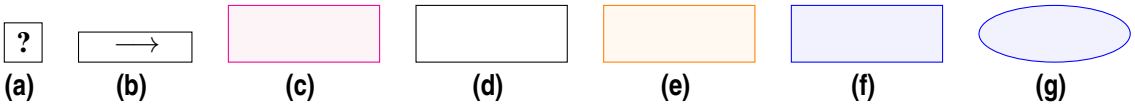

Figure 4.2 Different types of nodes that can be present in an BT.

Behaviour Trees are made up of *Control* nodes, *Decorator* nodes, and *Leaf* nodes. *Control* nodes could be either *Fallback* nodes, represented with a question mark (see subfigure **??**), which try success calling one by one each of their children; or *Sequence* nodes, represented with an arrow (see subfigure **??**), which call their children in order if the previous one has succeeded. On the one hand, *Fallback* nodes return *SUCCESS* if one of its children does it, *FAILURE* if none of them success, and *RUNNING* if one of its children returns *RUNNING*. On the other hand, *Sequence* nodes return *SUCCESS* when all children have been called in order and have returned *SUCCESS*. If any of them returns *FAILURE*, the sequence is broken and the *Sequence* node returns *FAILURE* too. When a child returns *RUNNING*, the *Sequence* node does it too. *Control* nodes are represented in a black

rectangular box when they are the standard ones (see subfigure **??**), but they could also be *Reactive* control nodes, represented by a magenta box (see subfigure **??**), which means that its already called children will be called again in the next iteration. This is very useful for generating behaviours where an action is constantly reattempted, or where it is necessary to check that the required conditions are still met. A *Child* node could be another *Control* node, a *Decorator* node, a *Leaf* node or a whole sub-tree. A *Decorator* node, represented in an orange box (see subfigure **??**), can only have one child (of any type) and its function is programmable (e.g., modifying its child result or retrying calling its child a number of times). *Leaf* nodes, represented in blue, could be *Condition* nodes, represented in a blue elliptical shaped box (see subfigure **??**), that check a condition and return either *SUCCESS* or *FAILURE*; or *Action* nodes, represented in a blue rectangular box (see subfigure **??**), that execute code that takes longer and therefore these nodes could also return *RUNNING*.

4.3.1 Main tree

In general, the design of both the behaviour tree and the *Agent Behaviour Manager* has been made with the aim of concentrating as less intelligence as possible, to ensure the success of the mission and the safety of the UAVs and the workers. That is why the only task of the callback functions that are executed when different messages come in is to update the value of the corresponding internal variables.

However, not all intelligence and decision-making can be placed in the ground station node. There has to be some decision-making capability on board UAVs in case the connection to the central node is lost. That is why there is a predefined protocol to act when this happens or when the battery runs out of power earlier than expected. These two factors are periodically checked in the main while loop (see Code **??**). In the case of the battery, if the function in charge determines that there is not enough battery to complete the current task, what happens is that the task queue is emptied and the value of the internal flag associated with the battery is updated. In addition, the event is communicated to the task planner in case the connection is still alive to generate a new plan. Similarly, if a connection loss is detected, the task queue is emptied and the corresponding flag is updated. In this case the *High-Level Planner* node will execute a replanning when it also detects the connection loss.

The behaviour tree is designed in such a way that, when the task queue is emptied and the respective flags are updated, the corresponding UAV goes to the battery charging station, which is the established emergency protocol. In order to justify this decision, each of the cases will be analysed separately below.

In case the connection between the two nodes is still active but there is not enough battery, the aim of the contingency plan is to eliminate risks to the UAV while the *High-Level Planner* generates new instructions. In addition, the new plan is likely to involve recharging the battery as a first step. Besides, there is a possibility that the connection may be lost at this point.

The danger of connection loss is that the *High-Level Planner* will replan the mission without the disconnected ACW, so that the tasks previously assigned to it will now be executed by others. If in this scenario the disconnected ACW continues with the last assigned plan, collisions could occur. The emergency protocol ensures that the disconnected UAV does not interfere with the new plans. In addition, the time until the connection is re-established is used by recharging the battery, which is positive for the mission.

BTs operate recursively. All nodes, regardless of their type, have a function that executes their content, the *tick* function. When the root node is *ticked* from the main while loop, it propagates the *tick* among its children following the operation rules described in Section **??** until eventually a *leaf* node returns one of three possible responses (*SUCCESS*, *RUNNING* or *FAILURE*), which will be propagated back, to obtain a final result that the root node will return.

Typically, a BT is executed to achieve a goal, and therefore the executor keeps on doing *tick* to the tree root until the response is either *SUCCESS* or *FAILURE*. As the function of this BT is to

control during the whole mission the movement of a UAV, it is of interest that the result of the root is *RUNNING* until the mission ends. This is why a *Decorator* node that always returns *RUNNING* regardless of the result of its child node has been defined. The BT implemented as a solution for the described problem is further divided into several BTs, taking advantage of the modularity offered by this approach. The main tree is represented in Figure **??**, being the node named as *Main Tree* the root of the complete tree.

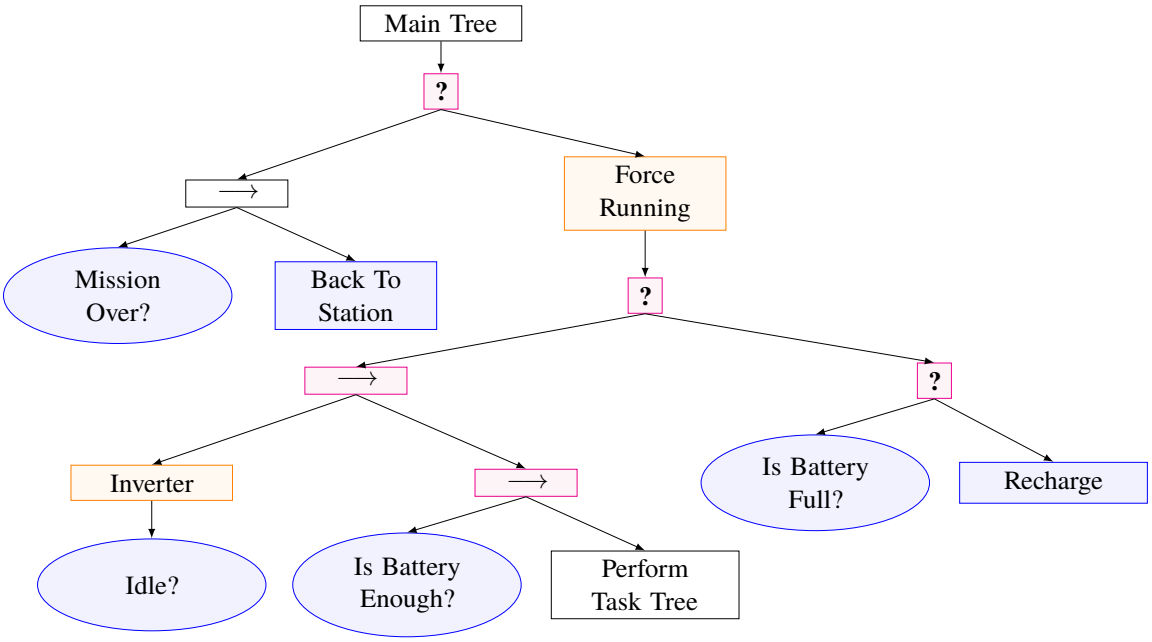

Figure 4.3 Behaviour Tree: Main tree.

This BT checks whether the mission is over (reminder: a mission would represent the working session, not a single task, i.e., whether the ACW is ready to be turned off) and if so directs the ACW back to the base station. If not, the main *Fallback* ticks to the right branch of the tree, entering the *Recursive Fallback* node that controls the mission. This branch checks if any tasks are assigned. If it turns out that the ACW is idle and the battery is not at hundred percent, the ACW is guided to a recharging station [2]. If a task is assigned and the corresponding flag indicates that the battery is enough, it enters directly into the *Perform Task Tree* (sub-tree represented in Figure **??**).

This *Recursive Fallback* is where is coded the behaviour that prepares the BT to be safe against a loss of connection or an unexpected battery event. As both unexpected events are managed flushing the task queue, the ACW reacts recharging, while giving the High-Level Planner control to decide when it is the best time to stop recharging (the High-Level Planner just needs to assign tasks again so that the ACW starts working back). Note that, thanks to the presence of *Recursive Control* nodes, the *Leaf Condition* nodes are constantly being re-evaluated. Thus, in case of any unforeseen event or change of plan, the BT will react by instantly stopping the executing branch and switching to the appropriate branch.

The *Perform Task Tree* checks which is the first task in the queue, which is the task that should be executed at that moment. This tree does not require much more explanation, it simply connects the *Main Tree* with the corresponding *Task sub-tree*. At this point, instead of sub-trees, it would be possible to directly place *Leaf* nodes that give control to the appropriate low-level controller,

[2] Both Safety, Inspection and Physical-ACW provide an input interface to guide the ACW to the charging station. In other words, among the low-level controller capabilities, there is a "reach this waypoint". The location of the charging stations is known in advance or provided as input by the High-Level Planner/Behaviour Tree.

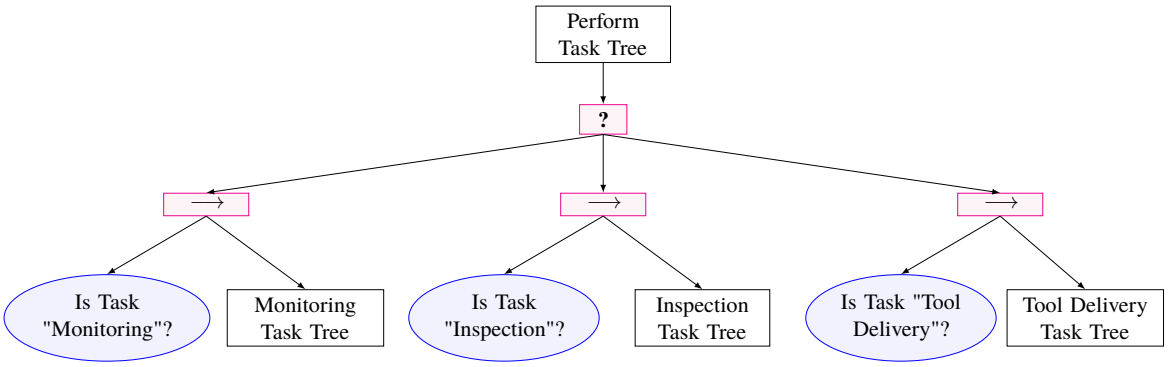

Figure 4.4 Behaviour Tree: Perform Task Tree.

starting to execute the task directly. However, it was decided that the full control would not be given to the low-level controllers until the ACW is close enough to the area where the task takes place.

In Figures **??**, **??** and **??** are depicted the sub-trees that run Safety, Inspection, and Physical tasks, respectively. They all guide the ACW close enough to where the low-level controllers need to be called (e.g., close to a worker to monitor or a place to inspect) and then, control is given to the corresponding one. These low-level controllers run on board the corresponding ACWs and communicate their results (success or failure) asynchronously back to the *Agent Behaviour Manager*, so that the BT can continue running.

In the following, the functioning of each of these sub-trees is described, as well as the details of each of the tasks that have not yet been explained.

4.3.2 Inspection task tree

This BT is quite simple as the task is just to visit a series of points and stop at each one to take pictures (see Fig. **??**).

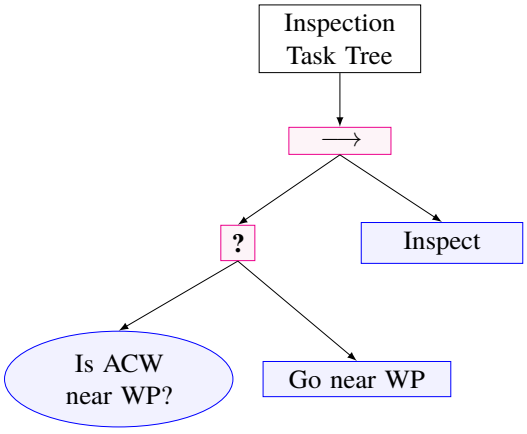

Figure 4.5 Behaviour Tree: sub-tree that controls the inspection tasks.

As mentioned above, control is handed over to the low-level controllers when the ACW is close enough to the area where the task takes place. The *Reactive Sequence Control* node behind the root of this sub-tree is in charge of checking this condition at any moment, stopping the low-level controller if it is no longer fulfilled.

All the information that the BT needs to operate is known in advance by the *Agent Behaviour Manager*, either because it has received it through one of the interfaces represented in Figure **??**, or

because it has calculated it itself in one of the functions that are executed in the main while loop. In particular, to check if the ACW is close to the points to be inspected, the information received from the *ACW's autopilot* is used against the points in the list of WPs to be inspected. If necessary, the *Action* node *Go near WP* is executed, which connects to the low-level controller in charge and gives as target location the nearest WP in the list. Once there, the low-level controller that executes this task is called.

4.3.3 Monitoring task tree

As can it be seen in Figure **??**, this BT follows the same structure as the previous one. The only differences are the *Leaf* nodes.

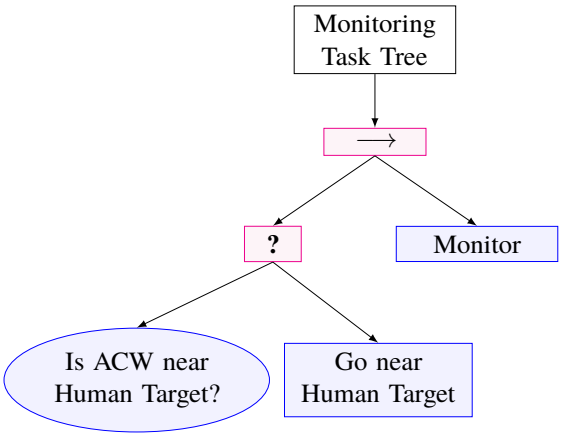

Figure 4.6 Behaviour Tree: sub-tree that controls the safety monitoring tasks.

In this case, the position of the operator to be monitored is known from communications with the *Human Tracker* node (see Figure **??**). If the ACW is not close enough, the BT will call the low-level controller giving it this time the position of the operator as a target. Once the UAV is in a position close to the worker, control is passed to the low-level controller in charge of this task. The information needed by this node is the *Monitoring Number*, the *Monitoring Distance*, the *list of ACWs' IDs*, and the formation to be maintained during the flight, which will be chosen by the *High-Level Planner* from a set of predefined formations based on the monitoring number (see Table **??**). Finally, it is worth mentioning that an ACW could be added or removed from the formation at any time by updating the parameters of the task, or even due to availability.

4.3.4 Tool delivery task tree

This tree is a bit more complex, as the task involves several steps (see Fig. **??**). First, it picks up the requested tool in case the ACW does not have it yet. This part involves travelling to the station to pick up the tool in case the UAV is not already there. The second part of the task is the process of approaching the operator. After all this, the low-level controller can be activated to carry out the delivery of the tool through a physical interaction between the operator and the ACW.

Again, the location of the operator and the position of the ACW is provided by the *ACW's autopilot* and *Human Tracker* blocks. The position of the fixed elements, in this case, the station where the tools are placed, is available among the information read by the *Agent Behaviour Manager* node from the configuration file during its initialisation.

In this case, the low-level controller that delivers the tool will be in charge of requesting the operator's permission to approach. In the meantime, it will have to wait. When it gets the permission, it is time to make the delivery. If too much time elapses without the operator giving the order, the task will be aborted and the tool will be returned to the station.

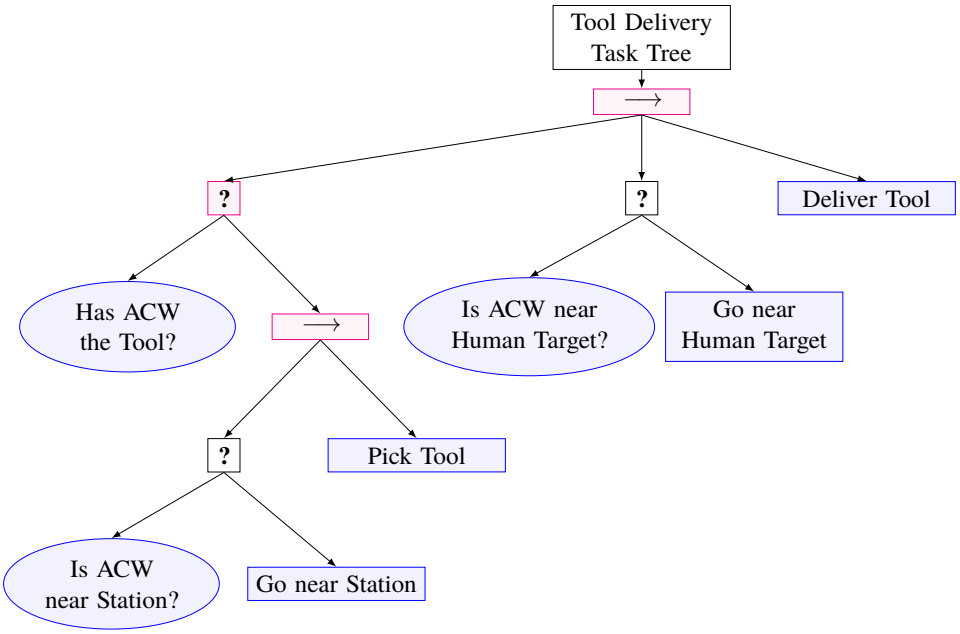

Figure 4.7 Behaviour Tree: sub-tree that controls the tool delivery tasks.

4.4 High- and low-level blocks faking

The work in this thesis consisted of programming one of the software layers that make up a software architecture. As there are still parts of the software architecture that are not yet available for integration, temporary solutions have had to be programmed to simulate their action during testing. This section will discuss those parts of the code whose mission is to fool both *High-Level Planner* and *Agent Behaviour Manager* blocks into believing that they are communicating with the real blocks, or to provide functions necessary for the execution to progress.

Low-level controllers have been faked in different ways. For BT's *Action* nodes of the *"Go To"* type, the *GoToWaypoint* ROS service available in the UAL tool is called instead, which leads the UAV to the entered coordinates (although it does not take obstacles into account). For more complex *Action* nodes such as *Inspect*, *Monitor*, *Deliver Tool* or *Pick Tool*, a function is simply called which sleeps the *Action* node for a while simulating that the low-level controller has been called and is waiting for a response. For these *Action* nodes, the response of the *tick* function will be *RUNNING* while sleeping, and either *FAILURE* or *SUCCESS* depending on whether their execution is halted or not. During sleeping time the rest of the tree continues running. The *Recharge* node also calls some low-level controllers. In this particular case it was decided to ignore the call and simply land the UAV on the charging station.

On the other hand, since UAL does not allow battery control, and the battery level that is available for reading remains static throughout the simulation, it has been necessary to create a block that simulates the evolution of the battery both during flight and during recharging. This block is programmed in such a way that at initialisation it reads the configuration file, thus having the position of the charging stations, and also subscribes to the information published by the *ACW's autopilot* to which it is faking the battery in order to know its position and status. If the UAV is in the air, the battery will be periodically decremented. Otherwise, the battery will remain static unless the UAV is over a charging station, in which case the battery will periodically increment. The battery percentage and charge/discharge rate is externally configurable at any time during the simulation. In addition, for ease of testing, this block allows for different modes of operation that can among other things make the battery static. The false battery level is periodically published in a

similar direction to the one used for the real battery.

Finally, as mentioned in the section **??**, the algorithm in charge of performing the distribution of WPs for an inspection task among the different selected ACWs has been forged. Normally, this algorithm is executed inside the low-level controller itself which is called by the *Inspect Action* node which can be found in the tree of the figure **??**. However, as this *Action* node has been completely forged, a distribution of WPs has been made within the task planning algorithm. It simply assigns, in order, one WP from the list to each selected ACW until the list of WPs is exhausted.

5 Results

This chapter discusses the experiments carried out to validate the software layer built. The simulation software used for the tests was Gazebo. The simulation environment consists of a high-voltage tower and an operator standing on the ground next to the tower, but as the low-level controllers have all been faked, not much attention has been paid to the simulation elements during the development of the tests. Instead, the focus is on the task distribution and the execution of the BTs.

The experiments were divided into two phases. In the first phase, simulations involving a single ACW were carried out in order to test the performance of each element of the system in a controlled manner. On the one hand, it was checked that the *High-Level Planner* performed the mission planning correctly, assigning the tasks as expected according to the specifications and constraints, and on the other hand, it was checked that the *Agent Behaviour Manager* performed its function correctly, both the execution of individual tasks and the ability to detect and act in case of unforeseen events. During this phase, the validation of the distributed block takes centre stage.

In the second phase, the simulations consisted of including multiple ACWs and testing in different scenarios. This phase focuses less on validating the BT, which would be fully validated during the first phase, and more on evaluating the capabilities of the *High-Level Planner*. The situations faced by the system in this phase involve disconnections, reconnections, input of new tasks, modifications of the battery level, etc. In addition, the type of ACWs has been modified from one test to another. Since the BTs are fully validated at this stage, the visualisation of the simulation in Gazebo is not as important. That is why the results of this phase are mostly presented by visualising the BTs with the Groot tool and by means of the information printed by both blocks through the terminal.

AC: En los comentarios está el plan de pruebas que voy a hacer y comentar y entre paréntesis las capturas que pretendo meter.

5.1 Phase I: single ACW simulations

Once the simulation has finished initialising, tasks can start to be requested. To do so, the fake *Gesture Recognition* block has to be executed indicating the type and parameters of the task. As the simulations during this phase consist of a single ACW, the behaviour of the planner is quite predictable, so in addition to validating the BT, these tests can be used to make a first verification of the *High-Level Planner* block.

First, a series of tasks were requested and executed without interruption in order to check that the system worked well under favourable conditions and was able to complete all tasks. Next, the responsiveness of the *Agent Behaviour Manager* block to unforeseen events was evaluated. In this part the behaviour of the planner remained predictable without the need for any calculations. It

started by requesting, in that order, a *Tool Delivery* task, an *Inspection* task and a *Safety Monitoring* task. The expected order in which the tasks will be assigned is the same order.

Figure 5.1 BT starting point displayed with Groot: battery charged and no task queued..

At the beginning, the UAV has the battery charged, is landed on the charging station and has no task queued, then the BT is returning *SUCCESS* except for the *Force Running* node, which keeps the BT running by always returning *RUNNING* (see Fig. **??**). Once the first task arrives the ACW starts running. The BT now checks which task it is and proceeds with its execution (see Fig. **??**). At this point the other two tasks were communicated. The *High-Level Planner* performed a replanning for each event, but as the *Tool Delivery* task remained at the top of the queue, the BT did not notice the change. Figure **??** shows both the mentioned replanning and the parameters selected by the planner for each of the tasks.

subfloat[] subfloat[]

(a) **(b)**

Figure 5.2 BT transition from idle to *Tool Delivery Task Tree*.

Figure 5.3 Feedback of the task planning process and communications between *High-Level Planner* and *Agent Behavior Manager*.

The *Tool Delivery Task Tree* functioned perfectly well. In Gazebo, the movement of the ACW from one side to the other was observed as the BT traversed the nodes of the subtree of this task (see Fig. **??**). In addition, the task result was checked for correct communication with the centralised block, which used the moment to re-evaluate the plan to see if it is still within the optimal plan. As the plan did not change, the BT continued to execute the queued tasks. Figure **??** shows the evolution of the simulation during the execution of the *Inspection* task, for which the BT also proved to work perfectly.

subfloat[] subfloat[]

(a) **(b)**

Figure 5.4 Evolution of the simulation during the execution of the *Tool Deliverry Task Tree*.

Finally, when this task was finished, the BT proceeded to execute the *Safety Monitoring* task, which was also carried out without problems (see Fig. **??**). This task ends when the operator requests it, so the fake block pretending to be the low-level controller for this task is programmed so that the task never ends. This task remains in the queue until the mission is completed or it is overwritten by a task of another type with the same ID, which is a contemplated case for which the planner is prepared and warns the operators that an unfinished task is going to be deleted. Since during planning, tasks are assigned in order, the non-completion of this task is not a problem for the execution of other tasks, but an opportunity to study in a controlled way the emergency protocols and the response of both blocks to different unforeseen events.

5.2 Phase II: multi-ACW simulations

AC: He pensado que en esta fase puedo modificar el código para que funcione sin la simulación (porque ahora mismo el nodo Agent espera que el UAL/State signifique "Ready") y hacer pruebas con muchos drones sin que explote mi ordenador. Como además las capturas de Gazebo no darán mucha información, me parece una buena opción. Creo que no habría problema luego con las

subfloat[] subfloat[]

(a) **(b)**

Figure 5.5 Evolution of the simulation during the execution of the *Inspection Task Tree*.

subfloat[] subfloat[]

(a) **(b)**

Figure 5.6 Evolution of the simulation during the execution of the *Monitoring Task Tree*.

llamadas a los servicios de UAL. Tengo que comprobarlo, si da error por no tener ningún nodo UAL arrancado sí que va a ser un lío.

6 Conclusions and future work

6.1 Conclusions

6.2 Future work

List of Figures

List of Tables

List of Codes