Aalto University
School of Electrical
Engineering

Master's programme in ICT Innovation

Onboard Mission- and Contingency Management based on Behavior Trees for Unmanned Aerial Vehicles

Matteo Albi

Master's Thesis
2023

Aalto University
School of Electrical Engineering

Author Matteo Albi

Title Onboard Mission- and Contingency Management based on Behavior Trees for Unmanned Aerial Vehicles

Degree programme ICT Innovation

Major Autonomous systems

Supervisor Prof. Quan Zhou

Advisor Dipl.-Ing. Simon Schopferer

Collaborative partner DLR German Aerospace Center

Date 25 September 2023 **Number of pages** 79 **Language** English

Abstract

Unmanned Aerial Vehicles (UAVs) have gained significant attention for their potential in various sectors, including surveillance, logistics, and disaster management. This thesis focuses on developing a novel onboard mission and contingency management system based on Behavior Trees for UAVs. The study aims to assert if behavior trees can be effectively applied to this domain and how they perform with respect to other modelling architectures. Furthermore, this document explores which tree structures are more efficient, good-design practices and behavior tree limitations. Overall, this thesis addresses the challenge of autonomous onboard decision-making of UAVs in complex and dynamic environments, particularly in the context of delivery missions in off-shore wind farms. The developed architecture is tested in a simulated environment.

The research integrates a Skill Manager, a Mission Planner, and a Mission and Contingency Manager. The architecture leverages Behavior Trees to facilitate both mission execution and contingency management. The thesis also presents a quantitative analysis of key performance indicators, providing a comparative evaluation against traditional architectures like Finite State Machines. The results indicate that the proposed system is efficient in mission execution and effective in handling contingencies.

This study offers a comprehensive structure targeting onboard planning, contingency management and concurrent actions execution. It also presents a quantitative analysis of Behavior Trees' performance in UAV mission execution and reactivity to contingent situations. It contributes to the ongoing discourse on UAV autonomy, offering insights beneficial for the broader deployment of UAVs in various industrial applications.

Keywords Autonomy, Mission management, Flight management system, Behavior trees, UAV

Preface

I would like to thank Simon Schopferer for his constant support during my project and for the time he sacrificed to help me during the writing process of my thesis. A big thanks to DLR for welcoming me into its company and for the help I received from all my colleagues. In particular, I want to express my gratitude to my fellow students Julius, Lukas, Jan, Simon, Philip and David for the amazing time we spent together in Braunschweig.

I also want to thank my family Nives and Nicola and my girlfriend Clara for being there for me during this time away from home. Also to my friend Pier, for the important suggestions he gave me to write this paper.

Finally, my most important thanks go to my grandfather Giuseppe Monico, who departed during my permanence in Germany, for being my biggest supporter in this experience I had abroad.

Matteo Albi, Braunshweig, 25 September 2023

Contents

List of Figures

List of Tables

Abbreviations

ALC(D)	Attributive Language with Complements and Concrete Domains
BPS	Back-Propagated State
BVLOS	Beyond Visual Line-Of-Sight
C2	Command and Control
CHDS	Controlled Hybrid Dynamical System
CGS	Current Goal State
DLR	Deutsches Zentrum für Luft- und Raumfahrt, German Aerospace Center
EASA	European Union Aviation Safety Agency
EnBW	Energie Baden-Württemberg
FDI	Fault Detection and Identification
FMEA	Failure Modes and Effects Analysis
FSM	Finite State Machine
HFSM	Hierarchical Finite State Machine
HTN	Hierarchical Task Network
LEC	Learning-Enabled Component
MCM	Mission and Contingency Manager
OWF	Off-shore Wind Farms
ROS	Robot Operating System
RTA	Run-Time Assurance
UAS	Unmanned Aerial Systems
UAV	Unmanned Aerial Vehicle
UDW	Upcoming Drones Wind Farm
uORB	micro Object Request Broker
uXRCE-DDS	Micro XRCE-DDS
VLOS	Visual Line-Of-Sight
XML	Extensible Markup Language

1 Introduction

Recently, Unmanned Aerial Vehicles (UAVs), also addressed as Unmanned Aerial Systems (UASs) or drones, have attracted attention as their deployment in several sectors is expanding. Possible application domains span from industry environments and warehouses to urban areas and surveillance. As reported from [1], the military industry is particularly interested in UAV development as well. Applications examples are surveillance and reconnaissance activities. The civil field also benefits from the development of such solutions. Indeed, drones are also employed for prison surveillance, alongside firefighters to detect and locate survivors after natural disasters [2], to monitor natural parks and act against pyromaniacs and poachers, and to provide support during the extinguishing of forest fires [3]. In general, UAS operations can be categorized into two families, in visual line-of-sight (VLOS) and beyond visual line-of-sight (BVLOS). In the former case, the pilot, or a co-pilot addressed as the observer, must be capable of directly and clearly seeing the drone. The latter case includes all situations where the first does not hold, specifically full autonomous flights. While flying under visual VLOS is regulated and allowed both for professional and non-professional users, regulations for BVLOS are particularly restricted and hard-to-get certifications are necessary. For example, the European Union Aviation Safety Agency (EASA), regulating drone operations in European regions, requires specific authorisations for BVLOS UAV operations. Nevertheless, industries are pressing for such operations to be regulated, as current laws represent the main limitation to autonomous UAVs deployment. A UAV use case which attracted particular interest is mid or short-range delivery as industries are increasingly focusing on logistic improvements. A particularly successful application is Zipline [4], which started by developing fixed-wing drones to deliver medicines in Rwanda. Dense urban areas may also benefit from these services. Indeed, as stated in [5], the growing population in metropolitan cities is increasing drastically the traffic. Thus, using UAVs would allow for fast short-range delivery by exploiting free aerial space.

The German Aerospace Center (Deutsches Zentrum für Luft- und Raumfahrt, DLR) reserves particular interest in the development of UAV technologies. An experimental test bed on which the institute is focusing its research is an autonomous helicopter: the "superARTIS". The drone, shown in figure 1.1, is being used to tailor the problem of autonomous delivery in several industrial scenarios. A recent study case that DLR is addressing, in collaboration with the energy provider Energie Baden-Württemberg (EnBW), is the transportation of materials and tools to supply Off-shore Wind Farms (OWFs).

Recently, the German government has been investing major resources in the green energy transition. For this country, wind turbines are one of the main sources of clean energy [6]. These complex machines may be installed on land or at sea. In the first case, the wind park is addressed as an on-shore farm, and in the second case as an off-shore farm. The main benefit of the latter is the presence of more stable and continuous wind currents in the sea environment. However, it poses severe challenges for the maintenance and management of the wind farm. Notably, maintenance figures

as one of the main costs when operating off-shore wind turbines. In fact, the current means of transportation are dedicated helicopters or ships, meaning that companies need to lease expensive resources and hire specialized crews to operate them.

To cope with this problem, DLR has launched the Upcoming Drone Wind Farm (UDW) project [7]. The objective is to employ aerial drones as a transportation channel for the maintenance supply to off-shore wind farms. In this scenario, there will be mainly two actors: the drone delivering the supply and the wind park control tower. The drone must be equipped with deliberative capabilities and a human operator is in charge of its monitoring. Ideally, the drone control station is not equipped with instruments to pilot the UAV manually, but it rather provides an interface to supervise the mission and the UAS activities. The operator should be capable of controlling the drone, or a fleet of drones, by prompting high-level commands in order to change its behaviour. The control tower is in charge of managing maritime and aerial traffic in the wind farm, providing permission to the UAV to enter specific flight zones and managing the wind turbine operations during the delivery process.

Within the delivery mission, four main phases are highlighted in [7]:

1. **Route planning and departure** The UAV plans the route to reach the off-shore sites, along with emergency and contingency routes in case of mission abortion. The operator approves the plan and the mission starts.

2. **Flight to the wind farm** The UAV follows the planned trajectory while sharing its state via a narrow-band data connection. The operator is capable of triggering a mission replanning, abortion and prompting other high-level commands at any moment.

3. **Entering the wind farm** The UAV connects to the wind farm network, updates weather and wind-farm operations data and plans a route to the target location, which must be approved by the operator. Then, the UAV asks the control tower for permission to enter the wind farm. The drone can wait to approach the target site by following specific loitering trajectories.

4. **Approach and landing on the target platform** Before approaching the target, the drone requests detailed information about the wind turbine state and weather conditions. Then, computes a safe approach route and emergency routes, taking into account turbulence and other risk factors. Lastly, the plan is approved by the pilot and the drone enables a different flight controller to accomplish the landing.

Notably, the UAV operates in a sea territory with dense traffic of both aerial and maritime transportation, a dynamic environment situated dozens of kilometres from the coast. Nevertheless, the operator monitoring the mission should be capable of operating several drones at once exploiting their high autonomy. For these reasons, the employed system needs to be resilient and capable of mitigating arising risks or unforeseen situations by executing contingency procedures autonomously.

Indeed, safety and usability pose two major barriers to the deployment of UAS in everyday activities. In different studies [1, 8], experts explain how ensuring the safety

Figure 1.1: Frame of a video taken during a flight test of the superARTIS.

of UAVs is a particularly challenging task given the dynamic environments in which drones operate, combined with the increasing complexity of the current technology. Moreover, in [9], the authors point out how these complex robotic systems require trained experts to be operated. Thus, to reach broad employment, it is necessary to program autonomous systems that allow end-users to easily instruct the drone with the desired activities, e.g. automatically starting a delivery mission or an inspection activity.

To overcome these limitations, experts emphasize the necessity for deliberative architectures [10, 11], i.e. systems able to automate activities and overcome unforeseen hazards through autonomous reasoning, coupled with reactive behaviours and recovery strategies. In [12, 13], researchers underline modularity as a key design characteristic to program deliberative and robust systems. Several architectures have been studied and tested [2, 14], but recently many researchers [8, 15] are exploring the adoption of Behavior Trees. Behavior trees were first developed and applied in the video game industry to program the behaviour of non-playable characters. They are structured as directed trees, where intermediate nodes are control nodes, which direct the information flow inside the tree, and the leaf nodes are conditions checks or actions [16]. This structure allows for high modularity and reusability by encapsulating more complex behaviours into subtrees and using them as leaf nodes. Moreover, several studies [15, 17] state how behavior trees are a promising approach for designing autonomous control architectures. Nevertheless, using proper structure and control flow nodes, behavior trees can endow robustness and safety to control systems, as analytically proved in [18]. Additionally, due to the information flow mechanism used in behavior trees, they can program reactive strategies to handle unexpected events or failures, as presented in [19, 20].

A mission management system represents the highest level in the control architecture

of the drone and is endowed with the task of operating and monitoring the UAS mission and managing hazardous situations. While behavior trees are largely adopted in ground-industrial robots, their application in state-of-the-art UAV mission management systems is more sparse [21]. In particular, the UDW project requires switching between control modes for the different phases of the mission, e.g. takeoff control, trajectory following and high-performance landing controller, used to reach the target platform. The management of concurrent tasks and several communication interfaces is another important requirement for the correct execution of the mission. For example, the UAV is equipped with two broadcasting devices to communicate its motion data and metadata to other participants of the air and maritime traffic. Furthermore, communications with the OWF control tower and the operator overseeing the operation exert a key role in the delivery process. They provide permissions for the execution of several steps of the mission and the concurrent management of these requests-answers would benefit the overall efficiency of the operation. Lastly, the drone must be capable of handling different contingency situations, such as adverse weather conditions, incoming collisions both in the aerial and maritime traffic and internal faults.

Behavior trees present a suitable mechanism for the governance of Unmanned Aerial Vehicles (UAVs), showing potential advantages in modularity and extensibility in contrast to traditional control architectures such as Finite State Machines (FSMs) or Hierarchical Finite State Machine (HFSM) [19, 22]. To validate this hypothesis, this study proposes the design of a novel mission management system for UAVs based on behavior trees, providing proof-of-concept of their applicability in this field. This structure focuses on onboard planning and mission management, while simultaneously equipping the system with reactive behaviours required for contingency management, i.e. the mitigation of operational and safety risks occurring from non-nominal situations. The architecture exploits skill programming, i.e. the definition of abilities that a drone can execute repeatably and their assembly in more complex behaviour, to build a modular system decoupled from the flight management system of the operated drone. Additionally, the tree is designed to facilitate parallel planning and execution of actions. Concurrently, the research aims to discern if the tick method, employed by behavior trees during execution, matches the reactivity exhibited by FSMs, which function through an event-based state transition. The investigation is structured to assess these aspects through an appropriate tree design, extending into a quantitative analysis of the performances of the designed tree structure in terms of execution efficiency and reactivity. Finally, the study identifies best design practices and examines the inherent limitations when applying behavior trees to the specialized use case.

The research is specifically aimed at the application of Unmanned Aerial Vehicles (UAVs) in complex and dynamic environments, with a particular focus on off-shore wind farms. While it explores the use of behavior trees for both mission execution and contingency management in this context, the study does not directly address other potential applications of UAVs, where requirements might be different. This study provides a comparative analysis with traditional architectures like Finite State Machines, but it focuses primarily on behavior trees and does not delve into the details of other possible architectures. Results are generated in a simple simulation

environment, where detailed dynamics of the UAV are neglected. Furthermore, collected data consists of run-time timestamps, thus they depend on the performances of the machine running the simulation and they might differ in a flight test.

The document is organized as follows. Section 2 provides a more detailed explanation of skill programming and behavior trees, along with the existing research in these fields. It also presents an overview of several studies about different control architectures deployed on UAVs and the benefits of using behavior trees as a paradigm to program autonomous drones. In section 3 the framework of the management systems is presented, along with a detailed explanation of the design choices. Section 4 describes the software architecture derived from the proposed method and the simulation environment used to test it. Follows in section 5 a report of the simulation results and their evaluation. The thesis ends with the conclusions about the elaborated study in section 6.

2 Literature review

2.1 Skill programming

Skill programming plays a pivotal role in the development of autonomous systems, serving as the building blocks to execute complex tasks and adapt to dynamic environments. By abstracting tasks into modular and reusable skills, developers can more efficiently program, update, and extend the capabilities of autonomous systems. This modular approach not only streamlines the development process but also enhances the system's adaptability and fault tolerance, making skill programming a valuable tool in the creation of reliable and flexible autonomous systems.

The paper [23], authored by Archibald et al., introduces the concept of "robot skills", which are abilities that a robot can execute repeatably and can be described unambiguously. These skills serve as building blocks for creating more complex robot operations. The paper presents a paradigm for simplifying the programming of robotic systems called Skill-Oriented Programming (SKORP). The SKORP paradigm aims to separate the responsibilities of application specialists and systems programmers, thereby reducing the cost and complexity of programming robotic systems. The paper also discusses the underlying software architecture and how it accommodates sensor-based actions. Skill programming has emerged as a critical area of research in robotics, aiming to simplify the complex task of programming robots for various applications. Also Bøgh et al. in [24] propose that a small set of predefined skills can solve a wide range of industrial tasks, thereby enabling easier programming and greater flexibility in robotic systems. The authors introduce a terminology that distinguishes between tasks, skills, and motion primitives, aiming to simplify the programming process. Tasks are production-related goals, while skills are the building blocks that enable the robot to complete these tasks. The paper emphasizes the importance of pre and postconditions for each skill to ensure its correct functioning. The authors also analyze industrial applications to identify the skills needed in logistics and assistive tasks.

On the same line, the paper [12] by Franz Steinmetz and Roman Weitschat argues that as industries shift from mass production to low-batch, highly customized production, there is a need for more flexible and user-friendly robotic programming. The authors introduce a software architecture that makes human-robot interaction more robust and allows for easier process control. The architecture employs "action blocks", software modules that change the system state and consist of an activity, an exit handler, and condition observers. These action blocks can be nested hierarchically, providing a structured way to handle various conditions and events. The paper also discusses four basic demands for skill parameterization to make the process faster and more intuitive. These include parameterization of the skill, direct parameter application and verification, parameter reduction by automatic derivation, and the use of different parameter sets depending on the user's expertise.

Formalizing skills provides a standardized way to define, describe, and implement them. This is crucial for ensuring that skills are consistently understood and executed, whether by human operators, developers, or the robots themselves. The paper [25],

authored by Lesire et al., aims to bridge the gap between operational and descriptive models of robot skills. Operational models focus on how a skill is executed, while descriptive models focus on what the skill does. The authors introduce a Domain Specific Language (DSL) that allows for the specification of both types of models, ensuring their consistency. The paper elucidates the significance of preconditions, invariants, and postconditions, serving as essential verifications before, during, and after the execution of a skill. Preconditions delineate the requisite circumstances for skill execution. Invariants monitor ongoing adherence to specified conditions, and postconditions, which declare data modifications, are pivotal in affirming the correct realization and conclusion of skills. The paper also discusses tools that can translate these skill models into standard languages or frameworks, facilitating the use of state-of-the-art algorithms and tools.

The study [26] from Albore et al. introduces an executive layer structured using formally defined skills. The authors propose a three-layered structure consisting of a decisional layer for autonomous reasoning, a functional layer for reactive tasks and processing, and an executive layer that bridges the two. The executive layer is of particular interest as it is responsible for the system's robustness and fault tolerance. These skills serve as both an abstraction of the functional architecture and as controllers for the functional layer. The paper also provides tools for verifying these models, generating code, and assessing fault tolerance, thereby contributing to developing more reliable and autonomous robotic systems.

Leite et al. in [14] implements a skill-based approach to address the problem of fault tolerance as well. The proposed architecture incorporates a novel risk assessment and decision-making model. This architecture aims to enable the robot to autonomously decide whether to proceed with or abandon a mission based on various risk factors. They use a mathematical model, specific to each skill, to compute a risk score associated with the current state and skill deviation. The advantage of using a general mathematical function is to be able to evaluate any situation the system might encounter. The output value is then evaluated using a specific threshold and subsequently, a decision-making model computes proper reactions.

In the direction of decision-making algorithms, many studies adopted skills as foundation blocks to compose plans. Crosby et al. in [27] uses predefined skills to automate task planning. The authors propose a framework that automatically generates planning problems based on existing skill definitions, allowing industrial robots to perform new tasks without manual reprogramming. The framework integrates these skills into a world model, which serves as the basis for automated planning. The paper demonstrates the effectiveness of this approach in a simulated industrial environment and confirms its applicability in real-world settings. The authors argue that their method simplifies the task-level programming of robots, making it more efficient and adaptable to changing manufacturing requirements.

Instead, Pane et al. propose in [28] the implementation of reactive and composable skills using an autonomous replanning and acting framework that utilizes constraint-based programming. These skills can be dynamically composed at runtime to adapt to unforeseen disturbances, such as human-induced changes. The framework integrates a task planner that computes a sequence of skills to perform a task, and when

a disturbance is detected, it replans to include a reactive composed skill to address the issue. The paper validates the framework through an industrial assembly task performed by a dual-arm robot system, demonstrating its effectiveness in real-world scenarios. It also provides quantitative results of the experiment, providing interesting insights about the replanning performances of the proposed algorithm.

Skill-level programming emerges as a foundation in the architectural design for drone mission planning and execution. The approach offers a layered abstraction, transforming complex actions into parameterized units, referred to as "skills", which serve multiple functions in mission planning and execution. The use of preconditions, invariants, and postconditions abstracts the skills in a manner that makes them modular and versatile. This modularity provides the system with the capability for autonomous reasoning, where skills become building blocks enabling the system to handle both planned and unplanned scenarios in a dynamic environment. Moreover, skill-level programming acts as a facilitator for autonomous monitoring, providing a framework for validating actions in real time and allowing for swift re-calibrations in the plan when necessary. The integration of skill-level programming is expected to improve the existing models of drone mission planning and navigation by making them more robust, adaptable, and user-friendly.

2.2 Behavior trees

In the context of skill execution management, behavior trees serve as a robust decision-making framework that facilitates the systematic implementation of skills, taking into account both system and environmental states.

The book [16] by Colledanchise and Ögren offers a comprehensive overview of behavior trees and their applications in robotics and artificial intelligence. Behavior trees are a formal, graphical modelling framework used predominantly in robotics and video game design to describe complex, hierarchical behaviours. Unlike FSMs, which represent state transitions, Behavior Trees focus on the hierarchical organization of various tasks, represented as nodes in the tree structure. To evaluate and execute the behaviours encapsulated within the tree, this framework adopts a "tick" system. A "tick" refers to a single update or iteration through the tree, starting from the root node and progressing through the structure based on the logic defined by its nodes. Each node has a specific state that it returns upon the execution of a tick. These states are essential for the parent nodes to decide how to proceed with the tree traversal and are integral to the overall behaviour the tree is designed to execute. The typical states returned by nodes are:

- Success: returned when a tick is processed on a node and the node successfully completes its task.
- Failure: as opposed the the previous type, it is returned when the node fails to complete its task.

- Running: returned when a node needs multiple ticks to complete its task. This is common for actions that take time to execute.

Success and failure states are also addressed as terminal states. In a Behavior Tree, the nodes are categorized into three different families: composite nodes, decorator nodes, and leaf nodes. Composite nodes, also addressed as control nodes, determine the flow of tasks, executing their child nodes based on their returned state. The basic control nodes are:

- Sequence: returns failure whenever a child node returns failure, and the next tick starts by ticking the first child. If a child returns running, the sequence node returns running as well and ticks the same child during the next cycle. When a child node returns success, the sequence node proceeds by triggering the next node. If all child nodes return success, the sequence node returns success as well.

- Selector or fallback: returns success whenever a child node returns success, and the next tick starts by ticking the first child. If a child returns running, the selector node returns running as well and ticks the same child during the next cycle. When a child node returns failure, the fallback node proceeds by triggering the next node. If all child nodes return failure, the selector node returns failure as well.

- Parallel: is the only node that can have more than one child running. Ideally, the parallel node ticks all child simultaneously, keeping track of their returned state. However, in programming implementation, it usually ticks all child nodes one by one. Nodes reaching a terminal state are not ticked again. The parallel node returns either success or failure based on defined thresholds. The success threshold indicates how many nodes must return success for the parallel node to return success as well. The same logic applies to the failure threshold with the corresponding state. If neither of the two thresholds is reached, the parallel node returns running.

Decorator nodes modify the behavior of their child nodes, for instance by repeating an action a specified number of times. Leaf nodes, usually at the tree's extremities, execute specific actions or conditions, forming the base functionality of the Behavior Tree. Notably, conditions nodes are particular as they return either a success or a failure state. Moreover, designers can define new nodes to fit their necessities, e.g. expanding the basic control nodes to achieve different behaviours. A typical example is the memory sequence node, which on the tick cycle subsequent to returning failure, instead of restarting from the first child node, ticks again the last failed node.

To share data among the nodes of the tree, the framework uses a blackboard approach. This is a common approach in several applications [26, 27], as represents a simple, yet efficient solution to achieve global access among several modules. It consists of a data-sharing structure available to all nodes forming the tree. Data is saved as key-value pairs and any node can access the value and update it.

Regarding the programming implementation, it is imperative that the nodes maintain a minimal computational load. This constraint arises from the necessity to preserve the tick frequency of the tree structure from a substantial cut-off. Indeed, if any node running time is too high, it prevents the tree from executing the next tick cycle, thus considerably increasing the ticking period. If circumstances necessitate intensive computations, it is advisable to execute them within a separate thread using the node solely as an interface to start and monitor the process, as suggested in [29]. This approach guarantees the preservation of system reactivity and responsiveness.

Colledanchise and Ögren also discuss the limitations of Finite State Machines and emphasize the need for systems that are both reactive and modular, qualities that behavior trees inherently possess. Their book also explores how behavior trees relate to and generalize earlier control architectures. It outlines design principles, extensions, and formal tools for analyzing properties like safety, robustness, and efficiency. Furthermore, in [18], Colledanchise et al. provide a mathematical proof of the robust and safety features provided by behavior trees. The paper focuses on the formalization and analysis of behavior trees as a framework for designing controllers. The authors present a functional model of behavior trees and discuss how they can be recursively combined while preserving key properties like robustness and safety. The authors also draw parallels between behavior trees and earlier ideas in robotics control theory.

Marzinotto et al. address the problem of behavior tree formalization in [30]. The study introduces a unified framework that is both mathematically rigorous and compact. The framework also offers equivalence notions between behavior trees and Controlled Hybrid Dynamical Systems (CHDSs), providing insights into their representational capabilities. The authors validate their framework through a real-world grasping mission involving a NAO robot.

The study [19] from Ghzouli et al. delves into the application and effectiveness of behavior trees in robotics. In particular, the authors conduct an empirical study to understand how behavior trees are employed in real-world robotic applications, comparing them with traditional behaviour modelling languages like state and activity diagrams. They find that behavior trees offer a pragmatic and extensible language that allows for project-specific adaptations. The paper also highlights that behavior trees are increasingly becoming an alternative to state machines for high-level coordination in robotics, offering modularity and flexibility. Even when compared to Hierarchical FSMs, which were introduced as an extension to address FSM's limited modularity, behavior trees exhibit a lower level of complexity during the design phase. Similar conclusions are reported in the survey [21] conducted by Iovino et al. The authors compare finite-state machines with behavior trees, highlighting the advantages of the latter in terms of modularity and maintainability.

An interesting application of behavior trees, that exploits their descriptiveness and modularity, is the field regarding model checking and safety. In [31] Grunske et al. propose an automated approach that integrates formal and informal methods, utilizing behavior trees and model checking. The authors aim to address the limitations of traditional Failure Modes and Effects Analysis (FMEA) in safety-critical systems. This approach aims to automate the tedious and error-prone aspects of FMEA by

injecting component failure modes into behavior trees and translating them into SAL code for model checking. The paper argues that this method allows for more effective and efficient safety analysis, especially for large-scale systems. It also provides a detailed procedure for translating behavior trees into SAL code, thereby offering a tool-supported process for automating FMEA. Lindsay et al. [32] focus on the use of behavior trees, combined with formal methods, to build a model-checking framework and evaluate the safety of systems with substantial redundancy. The authors employ a case study of the hydraulics systems of the Airbus A320 aircraft to demonstrate their methodology. These systems are designed to handle up to three different component failures without a complete loss of hydraulic power. The paper argues that behavior trees offer a simplified yet sufficiently rich model for capturing a wide range of component behaviours and system safety requirements. The authors also introduce a systematic method for using counterexamples to identify combinations of component failures that could lead to hazardous system failures.

One feature that makes Behavior Trees especially appealing for use in robotics is the ease with which their functionalities can be extended through the creation of custom nodes. Colledanchise et al. address in [33] the challenges of managing concurrent actions in behavior trees. The authors introduce two new behavior tree nodes specifically designed to tackle concurrency issues, thereby enhancing the predictability and performance of autonomous agents. The paper also argues that parallel configurations within behavior trees showcase superior scalability as the intricacy of the desired behaviour increases. This stands in contrast to alternative control architectures where the overall system complexity results from the multiplication of the complexities inherent in its subsystems. Instead, the study [17] by Pereira and Engel delves into the integration of Reinforcement Learning (RL) into behavior trees to enhance the adaptability of constrained agents. The authors introduce a new node in the Behavior Tree, called "learning node," which embeds a local RL model. The learning node can be used either for learning how to perform a task, or which task execution, among a given set, yields the most desirable outcome. This integration aims to maintain the advantages of both behavior trees and RL while minimizing the risks associated with learning. The paper also establishes a relationship between this framework and Options in Hierarchical Reinforcement Learning, thereby ensuring the convergence of nested learning nodes. Empirical validation is provided through agent simulation experiments in a fire control scenario. Lastly, Colvin et al. [34] addresses the limitations of standard behavior trees in modelling time-critical systems. The authors introduce timed behavior trees, an extension of behavior trees, to incorporate the concept of time. They provide operational semantics based on timed automata, which serves as a formal basis for translating timed Behavior Trees into the input notation of the UPPAAL [35] model checker. This enables automatic verification of system-level timing properties. The paper also introduces timed FMEA, a process for identifying cause-consequence relationships between component failures and system hazards in real-time safety-critical systems.

As behavior trees represent a tool to execute plans, researchers have also explored techniques to automatically generate behavior trees as a planning problem. The paper [20] by Segura-Muros and Fernández-Olivares proposes an architecture that

automatically converts plans generated by a hierarchical task network (HTN) planner into executable behavior trees. These behavior trees serve as the foundation for an execution monitoring process that can adapt to unexpected changes in the environment. The architecture is modular and minimizes the need for replanning by incorporating both deliberative and reactive behaviours. The algorithm breaks down the objective into a sequence of skills. Subsequently, an executor generates a behavior tree composed of two sequences, one managing the mission execution and the other handling the contingency actions. Employing this approach, the system initially examines potential triggers for contingency behaviour; if none are detected, it proceeds with the nominal mission execution. The authors propose to endue the user to manually design the contingency actions. Indeed, the faults that may occur during a mission are dependent on the system, its underlying hardware and the specific use case. Thus, a deep knowledge of the application is necessary to adequately program the contingency actions. The system was tested in a simulated scenario, demonstrating its ability to handle complex problems with interleaved goals in a dynamic environment. Another example is provided in [36] by Haotian Zhou et al., where the authors propose an autonomous task algorithm that utilizes behavior trees to enable robots to complete tasks autonomously in dynamic environments. The paper introduces a unified representation model based on behavior trees that facilitates the transformation of abstract commands into specific robotic operations. The algorithm generates a plan according to the user's intention, maps it into a behavior tree model, and executes it. The plan can be dynamically updated based on the status feedback of the execution. The authors validate their approach through simulations and actual experiments, both of which always succeeded. Notably, the resulting behavior tree structure is complex and the algorithm is tested on industrial ground robots. The results do not discuss replanning performances, nor reactive response time, thus the application feasibility of this method on UAVs is not clear.

In conclusion, in academic research behavior trees are highlighted as a versatile framework that effectively balances skill execution management and contingency planning in autonomous systems. Their modular design facilitates real-time adaptability, making them an instrumental tool for both mission-specific tasks and unforeseen challenges. The integration of these features positions behavior trees as an integral component in the current study, bridging the gap between traditional decision-making models and the dynamic requirements of modern autonomous operations.

2.2.1 Reactive behavior trees

In [37], Klöckner introduces the concept of "transient tasks", defined as nodes that do not yield a running state and therefore do not require activation. This implies that condition nodes inherently fall into the category of transient nodes. Adhering to this principle, the paper suggests the execution of entire subtrees or their segments within a single tick, contingent upon encountering a sequence of transient nodes. In typical setups, which involve a prolonged action node coupled with a condition node meant for monitoring purposes, this approach has been recommended to prevent chattering activations of the action node. A similar problem arises when implementing reactive

sequences. Reactive nodes are a specific extension of the standard control nodes used to implement reactive behaviours. In particular, the node of interest is an expansion of the sequence node. The ticking rule of the reactive sequence node is defined as follows: the sequence node always starts ticking the first child, and ticks the next whenever the predecessor returns with a success state. If any node returns failure, the reactive sequence returns failure as well. This ticking rule is particularly suitable for monitoring actions. In this context, the first node is a condition node whose role is checking that the invariant conditions of the skill are met throughout its execution, and the second is the action to be monitored. To achieve a constant monitoring of the action, it is important that its precedent node immediately returns either success or failure, otherwise the ticking frequency of the skill node gets substantially reduced. Thus, as proposed by Klöckner, the reactive sequence executes all children in sequence in a single tick, as long as the nodes can be identified as transient tasks. Only one node should be able to return with a running state, otherwise the subsequent nodes undergo a substantial reduction of their tick frequency, losing the advantages of adopting reactive sequences.

The reactive sequence node is used in the proposed design to implement reactive behaviors. When the conditions overseeing the routine currently running are no longer met, the reactive sequence will halt the action node and return failure. Notably, the action can also be represented by a subtree. This is used as the main mechanism to trigger replanning routines or contingency actions.

2.3 Mission and contingency management in UAVs

Mission management is crucial for UAVs as it orchestrates the various operational elements, ensuring that the vehicle successfully navigates through dynamic environments to achieve its objectives. In the context of UAV mission management, deliberation serves as the mechanism that enables autonomous acting based on real-time data and predefined objectives. It integrates planning, monitoring, and decision-making functions to adaptively manage contingencies and faults, thereby ensuring mission success and system safety.

Ingrand and Ghallab [10] provide a comprehensive overview of deliberation functions in the realm of autonomous robotics. The authors identify five key deliberation functions: planning, acting, monitoring, observing, and learning, and discuss their characteristics, design choices, and constraints. The paper argues that deliberation is essential for autonomous robots operating in diverse and unpredictable environments, as it allows them to act deliberately to fulfil their missions. The authors also emphasize that deliberation in robotics is not limited to task planning but involves a coherent integration of multiple functions.

As stressed by Guiochet et al. [13], key features to be included in the development of robotic systems are fault prevention, fault forecasting, fault tolerance and fault removal. The paper also discusses the different layers of autonomous systems and their potential hazards, as well as the limitations of current safety standards and certification methods.

Several papers address the problem of safety and fault management for autonomous systems, as it represents the primary step to enable a broader employment of UAVs. Zaman et al. present in [38] the development of a novel diagnosis and repair architecture for robot systems operating on the ROS framework. The paper makes three major contributions: it integrates diagnosis and repair into a single system, incorporates both hardware and software components in the diagnostic and repair process, and builds upon the widely-used ROS framework. The research also provides a running example inspired by the RoboCup Rescue competition to demonstrate the effectiveness of the proposed architecture.

The study [1] by Drozeski et al. focuses on enhancing the reliability and autonomy of unmanned aerial vehicles through a fault-tolerant control architecture. This architecture combines fault detection and identification (FDI) with reconfigurable flight control to ensure the UAV's stability and performance. In particular, one objective of the study was to test the feasibility of the control reconfiguration proposed by Osder [39] in a real flight test. The authors employ an adaptive neural network feedback linearization technique for stabilization post-fault detection. However, as mentioned in the document, neural networks are deficient in specific stability properties. Thus, in case the network diverges, the system switches to a PID controller as a fail-safe strategy. The architecture is validated through actual flight tests on an unmanned helicopter, demonstrating its ability to recover from different catastrophic faults of the actuators. Moreover, the neural network never failed to converge during the several flight tests.

In [11] the authors present a structured methodology that integrates specific fault-tolerant mechanisms into an adaptive control architecture. The system consists of an observer, running introspection routines, and a layered control system supervising the robot's operation at three different levels. The supervisors are addressed as global, local and adapter, from the higher to the lower control tier. When a fault is detected, the robot's autonomous behaviour is automatically adjusted to address the issue, triggering the proper supervisor depending on the severity of the fault. Experimental results on a mobile robot illustrate the effectiveness of the proposed autonomy adaptation approach.

Skoog et al. present in [5] the development of a run-time assurance (RTA) network architecture aimed at ensuring the safe operation of highly autonomous aircraft. The paper builds upon the ASTM industry standard F3269-17, proposing an architecture that shifts the responsibility for aircraft safety from the pilot to automated systems. This shift is crucial for the development of future pilotless transportation concepts. The paper discusses the architecture's critical features, such as the use of multiple separate monitors for different safety functions. Indeed, the study mentions that it is advisable to keep safety tasks separated, as their merging into a single behaviour usually results in more intricate algorithms than the addition of the single task complexity. This approach reflects the response of human pilots when confronted with multiple emergencies. They always give precedence to addressing the most pressing threat when the current situation necessitates rapid action.

Always concerning enhancing the safety enhancement of UAVs, the paper [9] by Vachtsevanos et al. introduces a novel framework that focuses on self-organization

and control reconfiguration strategies to improve system resilience in real-time. The authors propose to embed a Learning-enabled component (LEC) to forecast faults and hazards based on past data and to manage the system, combined with particle filtering as a nonlinear filtering method for managing uncertainty in long-term prediction. To achieve self-organization, the system represents internal components relations using a graph and applies graph spectral and epidemic spreading modelling tools. The goal is to reorganize the graph and maximize the connectivity while observing the system constraints. As a reconfigurable control strategy, the paper combines reinforcement learning and model predictive control. Lyapunov stability conditions are used to ensure that the resulting strategies are effective and safe, especially when the system is subjected to severe threats or hazards.

Pippin [2] further delves into the architectural considerations for enabling autonomous flight in UAVs. It emphasizes the need for modular, open, and extensible architectures that can adapt to varying levels of human interaction and onboard data processing. The paper introduces the frontseat-backseat driver paradigm, where the frontseat driver focuses on immediate flight control, while the backseat driver deals with mission-level planning. This separation allows for better fault tolerance and extensibility. The paper also discusses the importance of a message-oriented middleware for inter-process communication, which enables the system to be loosely coupled and easily extensible. It concludes by highlighting the need for rigorous testing and introspection capabilities to ensure the safety of autonomous UAVs.

Castano and Xu discuss in [8] the development of a theoretical framework for rapid and safe decision-making. The authors employ Behavior Trees to design UAS behaviours that can quickly respond to flight anomalies, such as encountering obstacles or experiencing system faults. The framework aims to detect and identify hazards and activate the safest response, which may include fault-tolerant operation, obstacle avoidance, or emergency landing procedures. The FDI process is divided into initial detection, classification and verification, stressing the importance of asserting the correctness of the identification process. Instead, a more recent paper [22] by Stojcsics et al. proposes a Behavior Tree-based autonomy architecture that incorporates a Fault Detection and Isolation Learning-Enabled Component (FDI LEC) and an Assurance Monitor to assert the prediction reliability. Utilizing Inductive Conformal Prediction techniques, the architecture aims to manage real-time contingencies through fault detection, isolation, and system reconfiguration. The paper also discusses the scalability of the FDI LEC and its ability to adjust thresholds for fault coverage and risk. The study contributes to the field by offering a novel system architecture that enhances mission execution robustness against faults and anomalies.

Mission planning and management is another key feature to develop a deliberative UAV. However, when examining current frameworks used in the industry, they present limitations in managing complex missions. An example is the PX4 Autopilot [40] open-source flight control framework. This comprehensive platform encompasses various critical aspects of UAV operation, including flight control, mission planning, sensor integration, and communication. In PX4, missions are typically defined as a sequence of waypoints that a vehicle follows autonomously. As discussed by Junger et al. in [41],

traditional waypoint-based command and control (C2) interfaces are not well-suited for managing tasks at different levels of abstraction. To overcome this, they propose a new C2 interface that employs task decomposition into modular skills. The paper demonstrates the application of this interface in two different UAS mission scenarios, emphasizing its potential to replace existing waypoint-based C2 systems and enhance the capabilities of next-generation autonomous UAS. Skill-based approaches for UAV mission management are explored in other studies as well. Albore et al. propose in [42] a layered approach, central to which is a skill management layer based on formal skill models. This layer serves as an abstraction between the decision-making, based on behavior trees, and functional layers, facilitating the implementation of resilient behaviours in autonomous systems like UAVs. The skill management layer allows for the structuring and testing of the functional layer while providing a simplified interface to the decision layer. The paper also discusses the application of this architecture in scenarios involving sensor failures and communication losses, demonstrating its utility in real-world conditions. The architecture has been particularly used to support the implementation of resilient behaviours in autonomous UAV missions. However, it does not consider the importance of deliberation and automatic planning to achieve increased autonomy.

Exploring more solutions based on behavior trees, the paper [37] by Klöckner focuses on the formalization of mission plans for UAVs using behavior trees and Description Logic. It argues that to rigorously analyze mission plans, they must be translated into a tractable formalism. The study proposes to use the description logic Attributive Language with Complements and Concrete Domains (ALC(D)) as the input formalism for a mission plan based on behavior trees. The paper describes an interface using the safety circuit of an exemplary UAV and demonstrates that the system can be used as a first step towards the verification of mission plans with formal methods. Always regarding behavior trees as a tool for mission planning, the paper [15] by Gao et al. addresses the challenge of automatically generating behavior trees that are both time-efficient and reactive to environmental changes. The authors introduce a new domain knowledge definition method that considers the time efficiency of actions. They propose a novel algorithm that automates the generation of behavior trees with time constraints, encompassing construction, simulation, and online re-planning. The algorithm builds the behavior tree starting from the defined goal state, and back-chaining actions until the preconditions match the starting state. This method yields a tree structure representing the entire mission, where the actions are nested over several layers, each progressively guiding the system's state toward satisfying the precondition of the subsequent planning step. This algorithm is easy to implement and customize. However, the generated behavior tree results in a complex structure developed over several levels. The paper validates the effectiveness of the proposed method through simulation experiments involving UAVs in search and rescue missions. However, the generated structure may result in inefficient tick signal propagation within the tree in case of a more complex mission. Moreover, direct tree generation does not facilitate the mission progressions monitoring. The study contributes to the field by being the first to propose a time-constrained behavior tree synthesis and re-planning framework.

In summary, the comprehensive body of research in both safety and mission management for UAVs underscores a move towards more integrated, adaptive, and real-time systems. On the safety side, there is a strong focus on fault detection and self-repair, while on the mission management side, the trend is towards more flexible and responsive frameworks, incorporating skill-based and behavior-tree methodologies. Both domains aim to enhance the robustness, resilience, and real-world applicability of autonomous UAVs. The presented work advances the existing literature on mission and contingency management in Unmanned Aerial Vehicles (UAVs). The paper focuses on dynamic re-planning and reactive behaviors for contingency management while designing a modular and extensible architecture. The work also addresses the problem of concurrent action execution using specialized tree structures. Unique to this study is its quantitative evaluation of the proposed infrastructure's performance, filling a gap in the current literature. In addition to providing performance metrics, the study also delves into an exploration of the limitations inherent to Behavior Trees and suggests efficient design structures to mitigate these constraints.

3 Methodology

3.1 Top-level Architecture

The proposed architecture follows a structured approach, as suggested in various studies [26, 42], which delineates it into three distinct layers to efficiently manage the drone's operations. A comprehensive overview of the structure is shown in figure 3.1

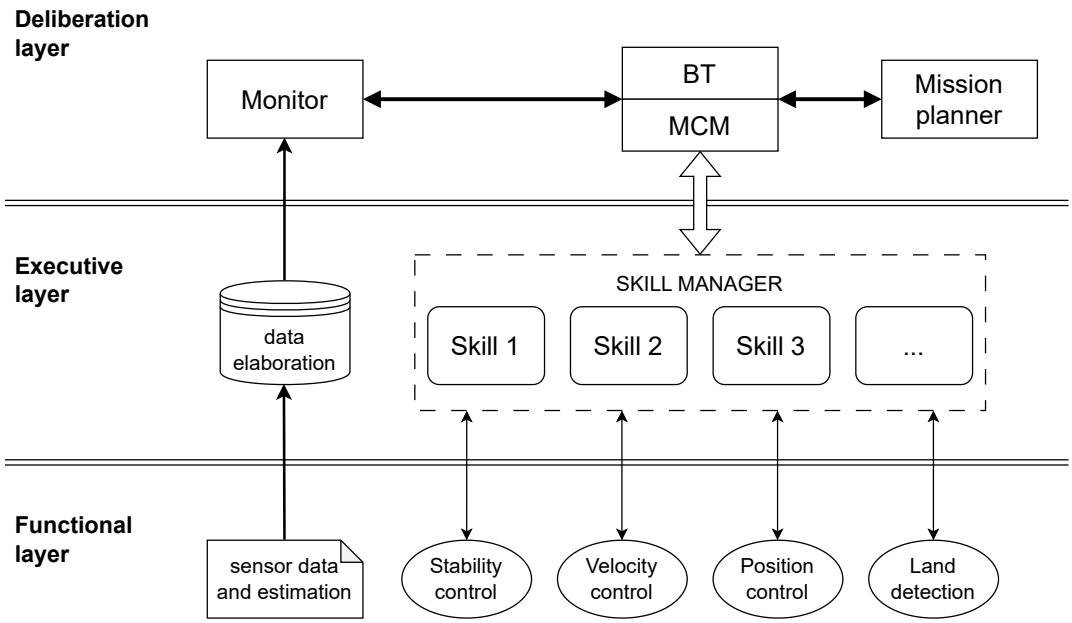


Figure 3.1: High-level architecture of the proposed design.

Deliberation Layer Occupying the top layer, the deliberative reasoning is responsible for orchestrating the platform's actions by dispatching high-level directives to the layer below. Essentially, it serves as the commander of the drone's mission plan, taking into account the dynamic environment and responding to any unforeseen situation that may arise. By issuing specific commands to the Skill Manager, the deliberation layer effectively translates the mission's objectives into concrete actions that the drone executes.

At this level, three main blocks cooperate in order to achieve the desired functionalities. Central to it is the Mission and Contingency Manager (MCM) incorporated into a single behavior tree. It oversees the whole mission execution, consisting of two main duties: Task Management and Contingency Management. The Task Management oversees the execution of the skills to carry out the steps provided in the current mission plan. This is created by the Mission Planner, responsible for examining the mission objectives and producing a sequence of actions to achieve them. The Task Management is also responsible for triggering a re-plan procedure to recover the current mission from unexpected events.

On the other hand, the Contingency Management process works in tandem with the Monitor module. The latter runs an introspection routine providing real-time data about the status of the system and eventual external hazards. The former incorporates decision-making rules based on these data in order to protect the system and the surroundings from contingent situations.

Executive Layer Positioned in the middle, this layer acts as an intermediary, connecting the Mission and Contingency Manager with the underlying hardware platform. Its primary goal is to establish a clear separation between MCM and hardware-related functions. This demarcation ensures that the MCM remains hardware-agnostic and maintains a consistent interface to instruct actions to the UAV. Within the Skill Manager, high-level directives from the MCM are translated into commands suitable for transmission to the functional layer below. Moreover, it runs control routines to process the feedback received from the Functional Layer and monitor the progress of the instructed actions. The Executive Layer also act as the communication bridge between the first and third layer. It's important to note that this portion of the software stack might require reorganization if implemented on different platforms due to inherent hardware differences.

Functional Layer This constitutes the lowest layer in the architecture and plays a critical role in managing the hardware components of the drone. Its responsibilities include collecting essential data, managing communication interfaces, conducting fundamental estimation processes, ensuring stability through control mechanisms, and acting as a flight controller. Most importantly, it encompasses the autopilot system capable of flying and controlling the drone based on different inputs and control strategies. Usually, it provides a collection of controllers which the user can trigger in order to enter different flight modes. However, due to its platform-dependent nature, this layer is not extensively discussed, as it falls outside the study's primary scope. Section 4.2 provides a practical description of the adopted system, which serves as background information to better understand the implementation of the upper layer.

These layers provide a structured and efficient framework for managing the drone's operations, ensuring that high-level mission objectives are translated into precise commands that the drone can execute. The following subsections delve deeper into the functionalities and interactions within the several blocks composing the system.

3.2 Skill Manager

In the proposed architectural framework, the MCM assumes the role of instructing the Skill Manager regarding the actions that must be executed. In this discussion, the terms skill and action are used interchangeably. The Skill Manager operates as the central system responsible for the execution and coordination of the drone's actions and capabilities. Its principal function consists of translating high-level instructions coming from the upper layer into commands understandable by the lower level. This

role provides a higher-level abstraction that shields the MCM from the intricacies of low-level commands expected by the functional layer.

For instance, consider the fly-to-waypoint as an illustrative example. The MCM specifies just one skill parameter, which corresponds to the target position. Subsequently, the Skill Manager acts as an outer-level control loop, calculating velocity set-points to reach the indicated waypoint. These velocity commands are then dispatched to the functional layer, which, in turn, directly governs the robot's actuators through specialized, real-time control loops. This process ensures consistent and effective performances of the executed skill. This oversight is critical, as it involves continuous monitoring of skill execution, ensuring that skills progress as expected and promptly addressing any deviations or errors that may occur during execution. When errors arise during skill execution, it promptly reports these issues to the upper layer. The upper layer then provides directives on how to respond effectively to the current situation, facilitating error resolution. Lastly, resource allocation is a central part of the Skill Manager's responsibilities. It involves the efficient allocation and utilization of various resources essential for the smooth operation and optimal performance of robotic systems. For mobile robots like UAVs, this often entails overseeing motion-related and hardware resources, such as actuators and motors, as well as computational resources, including specific controllers within the functional layer. For instance, the Skill Manager orchestrates multiple manoeuvre requests from the MCM and decides whether to reject the last request or to interrupt the ongoing manoeuvre to execute the latest received command.

In the proposed design, each skill has been implemented using a finite state machine. In this case a FSM is sufficient to represent the skill architecture as no concurrency is necessary and the workflow is fairly simple. Moreover, an event-based transition method accurately fits the model of an action, as the skill must respond both to internal and external stimuli. The resulting structure is shown in figure 3.2. Upon receiving a request for a specific action from the MCM, the Skill Manager initiates the corresponding skill, which enters the state "Process request" (PR). Moreover, the Skill Manager associates a unique identifier, called skill ID, with each request coming from the MCM. When entering the PR state, the skill sends the command to enable the desired controller to the functional layer and waits for the acknowledgement. If the acknowledgement is negative, thus the functional layer has rejected the command, or none has been received after a predefined timeout, the skill terminates and the Skill Manager reports the error to the MCM. Otherwise, if a positive acknowledgement has been received, it means the underlying layer accepted the command and started to execute it. In this case, the state switches to "Execute" (E) and the Skill Manager registers the current skill as running. While in state E, the skill updates the controlling signal to feed to the functional layer to achieve the desired behavior, and the Skill Manager reports the skill advancement to the MCM. Once the action execution terminates, either because successful or because the functional layer reported an error, the skill enters the "Report" (R) state. When in state R, the system communicates the outcome to the deliberation layer, and the skill process terminates. If the MCM prompts the cancellation of the current action, the skill enters in "Cancel" (C) state and forwards the request to the functional layer. The request consists of a switch to a loiter

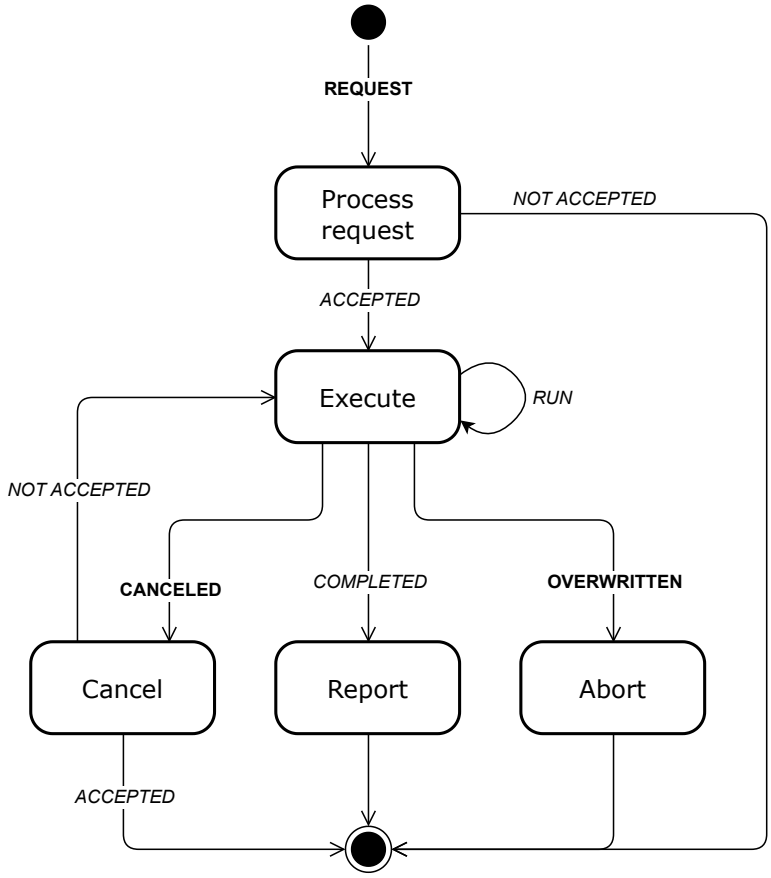


Figure 3.2: Structure of the FSM representing every skilled programmed in the Skill Manager. The filled circle represents the starting point, while the circled filled circle the termination. Transitions in bold are triggered by the Skill Manager, and transitions in italics are triggered internally.

routine to command the drone to enter a neutral state. If the functional layer accepts the instruction, the skill terminates and the successful cancellation is reported to the MCM. Otherwise, if the functional layer rejects the request to enter into the idle state, the skill switches back to state E and the error is communicated to the deliberation layer.

A special case arises when a skill is currently running and the MCM commands the execution of a new skill which requires the same resources as the first. The adopted behavior is to overwrite the current action with the new one requested. Firstly, the Skill Manager triggers the new skill to be executed and the request process unfolds as previously described. In case the functional layer rejects the new command, the new skill terminates and the one currently running is not influenced by the process. Otherwise, the Skill Manager updates the ID of the skill currently registered as running. Upon this update, the running skill recognizes that the skill has been overwritten, thus it switches to the "Abort" (A) state. In this case, the skill terminates its execution

sending to the MCM an abort message. Simultaneously, the new skill switches to state E and starts to command the triggered controller of the functional layer. By using a unique code to identify the current running skill, the MCM can overwrite it with an action of the same type, but with different parameters. This is particularly useful if it is necessary to vary fly direction or speed or to specify a different target point to be reached.

3.3 Mission planner

By incorporating a mission planning module into the behavior tree, the previously rigid structure becomes more flexible. This flexibility enables the addition of adaptability to behavior trees, a challenge highlighted in [17]. Furthermore, it empowers end-users to define missions as sequences of objectives, eliminating the need for manual programming of individual steps in the plan. This aspect is particularly crucial, as it simplifies the utilization of the proposed technology while providing the system with autonomous reasoning.

The proposed solution bases its functionalities on skill-level programming. Skills are parameterized through the use of preconditions, invariants, and postconditions [12, 25]. These conditions pertain to the system state and specify requirements that must respectively hold before, during, and after the execution of an action. This framework facilitates action monitoring, enabling the assessment of whether the system can initiate the skill, identifying any errors during execution, and verifying if the achieved state aligns with the desired outcome. Furthermore, this approach for defining skills finds application during the planning phase as well. In this context, the Mission Planner views the action as a black box, with knowledge limited to its initial and final states. These states are also referred to as start or back-propagated states and goal states, respectively. Consequently, when presented with an objective and the current state, it calculates a plan by chaining the skills through the alignment of their start and goal states. Upon the start-up of the system, the Mission Planner loads a sequence of objectives defined by the user. For each objective, the planner tries to define a plan to accomplish it, which is then handed to the MCM to start its execution. Every objective is marked with either "success", "failure" or "waiting", with the first two cases addressed as terminal states. The first indicates that the Mission Planner was able to define a plan to fulfil the objective and that the plan has been executed successfully by the MCM. Instead, a goal is marked as failed when the Mission Planner is unable to define a plan to accomplish it. Lastly, the waiting state means that either the system is trying to fulfil the objective, or it has not been attempted yet. The Mission Planner starts by processing the first objective, and every time a goal reaches a terminal state, the next one is selected. The Mission Planner finishes executing planning routines once the last objective reaches a terminal state. In this situation, the mission is considered as terminated.

The approach adopted in this thesis shares the planning strategy outlined in [15], but results in a structure and executing method similar to the algorithm introduced in [20]. The plan is formulated as a sequence of actions through a back-chaining process

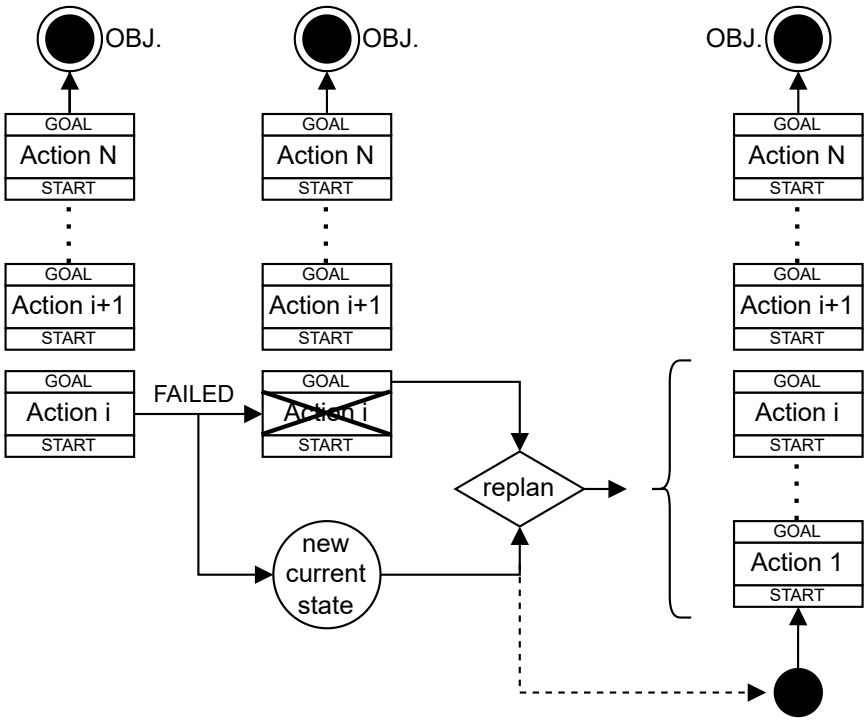


Figure 3.3: Schema presenting graphically the procedure to execute a replan after an action failed.

of skills that initiates from the objective state. The algorithm is presented in detail in section 4.4. Notably, by adopting this method, the resulting arrangement can be identified as a Last-In-First-Out (LIFO) queue, given that the first step inserted in the succession is the last to be executed. This plan is called the Action Plan and is stored in the planner and accessed by the MCM, which performs each planned step by generating a sub-tree as described in section 3.4.3. Additionally, by employing a sequence to represent the plan, the planner can effectively monitor the execution of the mission and detect any failed or interrupted steps.

This significantly facilitates the re-planning procedure, depicted in figure 3.3. Assuming that the objective does not change after a skill fails, all actions planned subsequent to the failed skill remain valid, as the plan was established through backward chaining. Thus, the planner reinitiates the planning procedure starting from the action that follows the failed step, propagating backwards until the new current state is reached. It is advisable to provide the UAV with an action to be executed during this process. This way, the drone does not stop every time a re-planning is triggered, but it executes a predefined skill to achieve a smooth transition between actions. Thus, the Mission Planner needs to define two different sets of parameters for each skill added to the Action Plan. The first identifies the "nominal action", which defines the parameters sent to the Skill Manager to execute the mission step under ordinary circumstances. The second set defines the "transition action", which is the instruction to be processed during a planning phase. Illustrative transition actions include loitering or proceeding

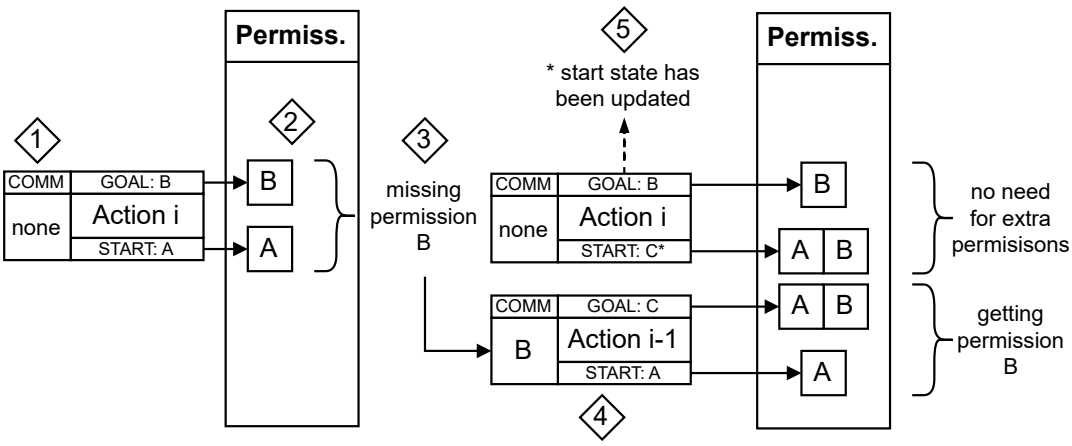


Figure 3.4: Schema presenting graphically the algorithm used to generate the Communication Plan in parallel to the Action Plan.

in the same direction as defined by the nominal action with a reduced speed.

The drawback of this approach lies in the planning algorithm being "greedy". Indeed, the planner propagates backwards by using skills that result optimal for the current step, which resembles an optimizer finding a solution by following local minima. In complex scenarios, this aspect may lead to inefficient plans or, in the worst scenario, to the planner not being able to solve the planning problem. Another limitation lies in the dependency of the planning algorithm on the system's state representation. The state variables must provide sufficiently detailed information to enable the planner to make optimal choices among the available actions.

Key steps of the delivery mission in the UDW project are the permission requests to enter the wind farm and asking for approval from the operator for the planned route or to perform particular manoeuvres. It would be optimal to perform such communications in parallel to ordinary navigation tasks to execute the mission efficiently. However, the current algorithm does not allow for the planning of concurrent actions. Thus, it is necessary to adapt it to implement this specific feature. The approach proposed in this paper is to generate different sequences related to specific sets of skills, i.e. "autopilot skills", "control skills" and "communication skills". By producing inter-dependent action sequences for more than one set of skills, concurrent tasks can run on multiple subsystems at the same time. The method described below is a proof-of-concept implementation that is tailored for the UDW-specific use case of requesting permissions while approaching specific air space volumes.

In order to add concurrent communication, the planner creates an additional plan, called Communication Plan. For every step in the Action Plan, there exists a step in the Communication Plan. This step may be empty or specify a list of permissions the UAV must ask for, concurrently with the execution of the corresponding skill in the Action Plan. The algorithm to compose the Communication Plan follows the explanation presented below. The steps are reported in the schema of figure 3.4 using numbered diamond shapes.

1. When the planner adds a new skill to the Action Plan, the corresponding communication step is always empty.

2. Consequently, the planner checks if the action goal and its starting state differ in necessary permissions. In the schema, the starting point is associated with permission of type "A", and the goal point with permission of type "B".

3. If the goal indicates permissions that are not present in the back-propagated state, it means the plan needs an auxiliary action during which the UAV performs the necessary requests. In this case, a request for permission of type B is added.

4. The auxiliary action depends on the one previously planned. In the example, the drone has planned for a fly-to-waypoint action from point A to point B that causes the UAV to cross the border of a flight area that needs specific permission in point C. Thus, the auxiliary skill is an additional fly-to-waypoint instruction from point A to point C. This skill commands the drone to reach the border while asking for permission to enter the area.

5. Finally, the starting state of the nominal action is updated to match the cross point.

Using this method, it is possible to chain multiple auxiliary actions in case the borders of several special flight zones are crossed. In fact, the back-propagated state of the first planned auxiliary action is the same as the nominal action before the plan is updated. In the schema, the nominal action in "Plan before" and the auxiliary action in "Plan after" both start from state A. Thus, the algorithm can be iterated until the permissions of the starting state and the goal state for the last planned action match. In the example, this is not necessary as the starting and goal states of the auxiliary action do not differ in necessary permissions.

The drawback of this method dwells, again, in the exploitation of local minima. Indeed, the algorithm always takes as reference the border crossing point closest to the goal point, which may result in sub-optimal solutions. For example, in case two crossing points are particularly close to each other, the planner allocates two different action steps. Nevertheless, it would be more convenient to conceive one single action associated with two permission requests. Another edge case is the presence of concave flight zones, which may lead to multiple permission requests for the same area. This issue, however, is strictly related to the management of permissions and their expiration rule, thus it is dependent on the specific use case. In general, all these flaws can be addressed by a more precise algorithm design, but they are ignored in this study for simplicity.

3.4 Mission and Contingency Manager

The primary objective of the developed design is to enable a drone to execute a nominal mission plan while also being able to adapt to unexpected changes in its environment. This is achieved through a combination of proactive and reactive behaviors. The drone's nominal mission refers to its intended course of action under normal conditions. However, the real world is dynamic and can present unforeseen challenges. In addition to the nominal mission, there are elements of uncertainty, such as changes in the environment or unexpected events. These uncertainties can impact the drone's ability to execute its plan successfully. An example scenario is provided: when the drone needs permission to enter a wind farm, it needs to move to a target point and simultaneously communicate with the control tower to request permission. This illustrates the parallel execution of multiple actions, where the success of one or more actions is crucial for the overall mission's progress. In the context of executing a complex plan involving parallel actions, the concept of a scheduling problem arises. The successful completion of certain actions is a prerequisite for the drone to move on to subsequent steps of the plan.

Nevertheless, it is a fact that not all stages of the plan might unfold according to their intended course due to possible failures or unforeseen contingencies. This is when the Mission Planner assumes a pivotal role. It is responsible for dynamically creating the mission plan from start to finish. Rather than relying on a fixed, predetermined plan, the drone autonomously generates its plan based on the current situation. If any step fails to be completed or if the drone encounters unexpected obstacles, the MCM triggers a process of re-planning. This involves generating a new sequence of actions that can help the drone recover from setbacks and continue progressing toward its mission objectives.

The structure of the behavior tree designed to govern the mission management process is depicted in figure 3.6. This choice of behavior trees as the underlying control mechanism not only aligns with their proven advantages in modularity, scalability, and concurrency management, but also sets the foundation for an efficient and effective mission management framework.

3.4.1 Behavior tree structure

The convention used to display behavior trees in this paper is reported in figure 3.5. Used control node types are:

- (par) parallel node.
- (fb) fallback node.
- (seq) sequence node.
- (reac seq) reactive sequence node.

Moreover, in all behavior tree figures, every node has a label that indicates its function inside the tree. Lastly, every parent node ticks its children from left to right,

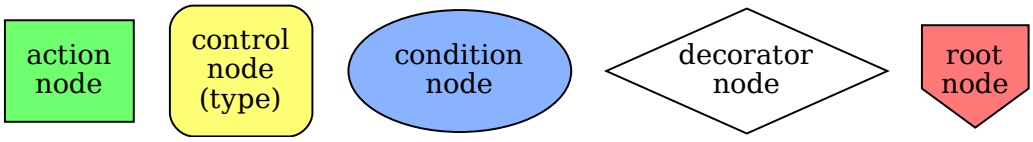


Figure 3.5: Convention used in this document to draw behavior tree schemes.

or from top to bottom. During the description, the node being examined is linked to the schema representation indicating between brackets the label used in the figure, followed by the specific type of node, if necessary.

Follows the detailed description of the behavior tree structure, shown in figure 3.6. The architecture consists of four main branches: monitoring, contingency management, executing and planning, respectively represented by the nodes named "monitor", "contingency behavior (fb)", "exec plan (seq)" and "re-plan proc (par)". The arrangement begins with the root node (root), succeeded by a parallel node (main threads, par). This parallel node oversees the two fundamental processes of the structure: monitoring and mission and contingency management. The monitoring process is symbolized by the monitor node (monitor), which consistently returns a running state. It serves as the bridge between the Mission and Contingency Manager and the data collection running on the Executive layer. This node receives a stream of data, which is loaded inside the behavior tree blackboard. The monitor is also responsible for partial data elaboration. It processes the data and raises specific flags to trigger contingency measures when necessary. The position of the monitor node in the tree structure is crucial, ensuring a high update frequency of the information saved in the blackboard and the state of the flags. Notably, the flags are reset in the contingency management branch upon the termination of the associated action to ensure proper information flow inside the tree, i.e. no interruption of the contingency manoeuvre if during its execution the states are set back to nominal values. The mission and contingency management branch starts with a decorator node functioning as the primary loop node (loop, run until failure). This loop node continues its operation until the associated child node encounters a failure. Given the returned state rule of monitor and loop nodes, the parallel node ticking them has a failure threshold set to one. This way the tree fails as soon as the MCM branch fails, interrupting the execution of the whole system. Notably, this is the only method to define an exit point of the program because the loop node returns a terminal state only when its child fails.

The mission and contingency management continues with a fallback node (MCM, fb), responsible for orchestrating the transitions between standard mission execution (nominal behavior, reac seq), contingency management (contingency behavior, fb) and mission termination (end mission). This design choice, similar to the solution proposed by Segura-Muros et al. in [20], reflects the necessity of deliberative autonomous systems to be composed of a high decisional level and a low reactive level, as underlined by several papers [27, 28]. The first represents the deliberative and dynamic behaviour where the nominal mission is generated and executed. The second is a static subtree where the actions triggered to cope with specific contingency events are manually

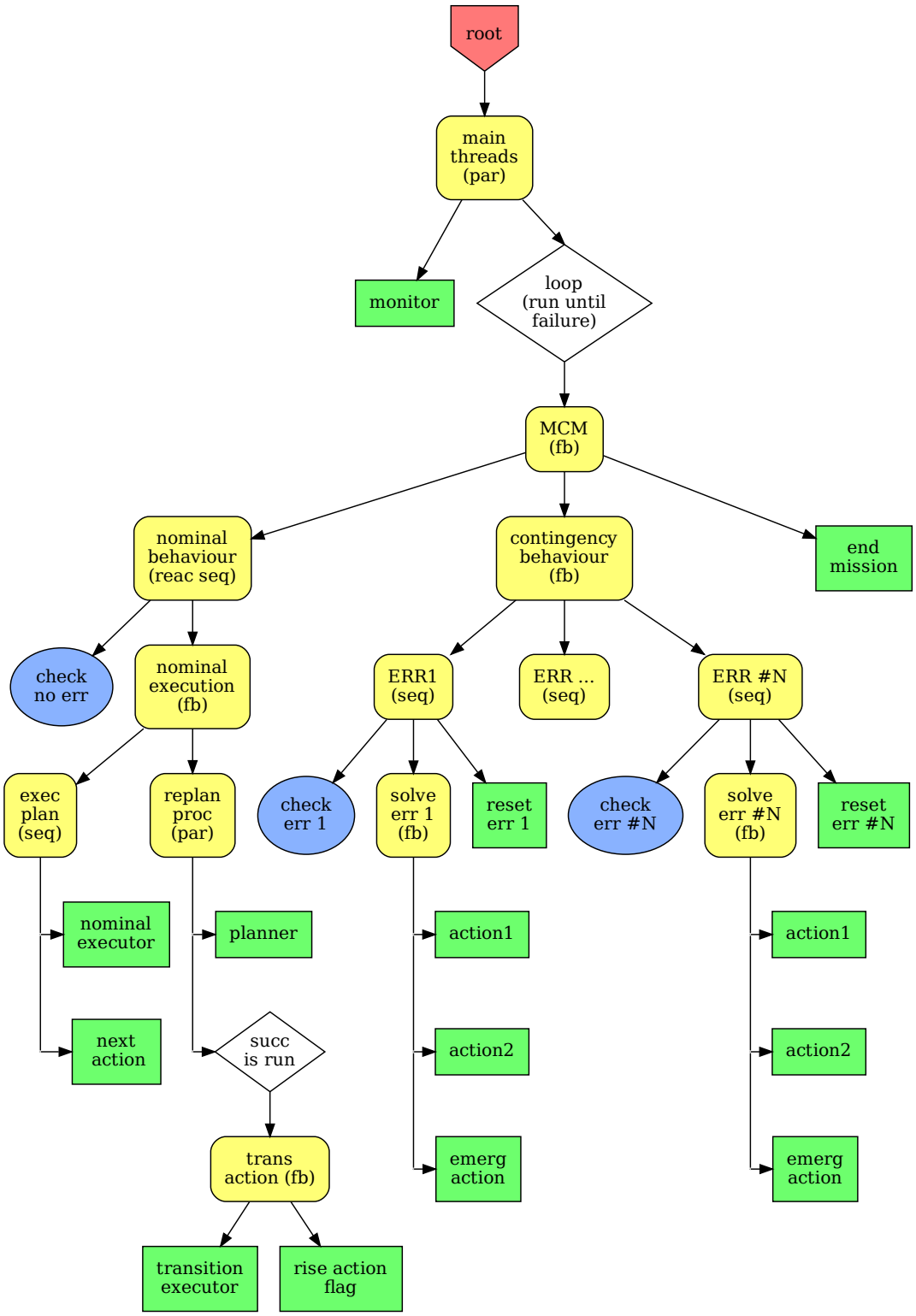


Figure 3.6: Behavior tree structure used inside the Mission and Contingency Manager.

programmed. This method avoids a planning phase when reacting to unforeseen situations, ensuring a quick response. It is important to remember that control nodes have an intrinsic priority between their children. These nodes tick the children in the same order as they are defined in the tree structure. The nominal behavior is the left-most node of the mission management fallback node, hence it is the first to be ticked. To rapidly switch to the contingency management branch, the nominal behavior node is a reactive sequence node. The associated condition node (check no errors) evaluates whether any error that could activate a contingency action has occurred, and returns a failure state if the evaluation is positive. The subsequent node is the nominal mission execution, which orchestrates the switch between planning and execution phases. The second child node within the MCM fallback node contains the contingency management subtree (contingency behavior, fb). This subtree employs condition nodes to assess flag errors and initiate necessary contingency actions, as elaborated further in section 3.4.2. This design decision prevents the unnecessary ticking of extra condition nodes during the standard mission execution. Instead, it utilizes an auxiliary flag to transition to the contingency behavior when needed. The third and last child of the MCM node (end mission) represents the mission termination state.

The core execution of the mission is overseen by a fallback node (nominal execution, fb). This node splits into two primary phases, which, when combined, define the standard behavior expected within the system. The first of these phases is the execution phase (exec plan, seq), while the second is the planning phase (re-plan proc, par). In both these phases, the executor represents a key module of the tree. It is endowed with the responsibility of managing and overseeing the execution of every step planned by the Mission Planner. This is achieved by generating a subtree with a predefined layout designed particularly to initiate the skill and monitor its progress, as further described in section 3.4.3.

Within the execution phase, there are two distinctive steps. Initially, a tree is generated to execute a single nominal action using the executor module (nominal executor). Once the action is completed, the subsequent node (next action) acknowledges its completion. This acknowledgement is forwarded to the Mission Planner and serves as the authoritative signal for the behavior tree to advance to the next pre-defined action. The planner updates the stored plan by eliminating the completed step and sends the next action to be executed to the "next action" node. This subsequently loads the new step into the behavior tree and returns a successful state. This procedure is depicted in figure 3.7. The state propagates backwards until it reaches the main loop node and, given that the loop persists until a child node reports a failure state, it proceeds to tick the execution subtree once more. This starts a new execution phase, which manages the last loaded action. However, failure during execution can arise under different circumstances: when no action is currently selected, if the plan is empty with no following action to select, or if the executor encounters a failure while attempting to execute the action. In any of these scenarios, the planning branch is triggered.

During the planning stage, the planner node collects the current system state and shares it with the Mission Planner. A timeout check is also implemented by the planner node to ensure the planning process doesn't exceed a certain time limit. Once the

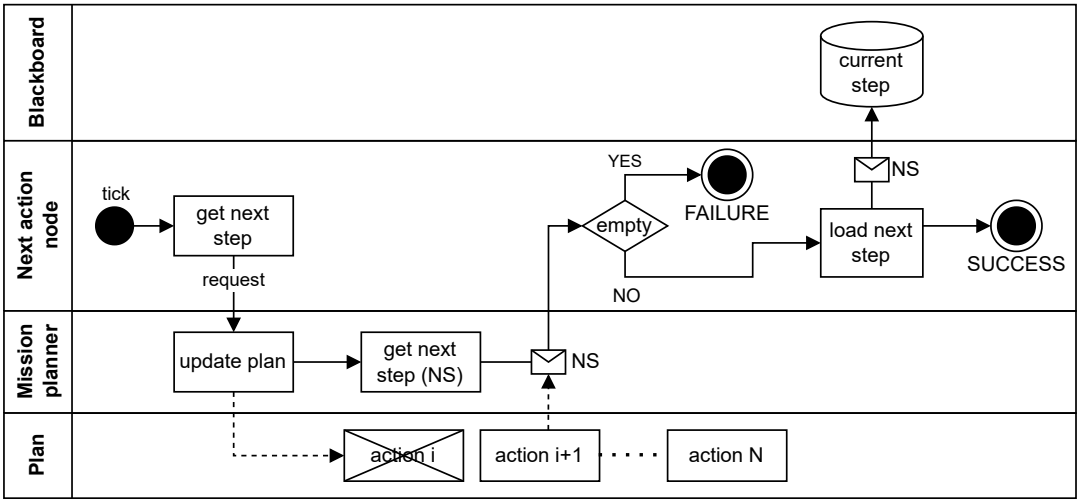


Figure 3.7: Schema presenting graphically the steps occurring when the behavior tree requests the next planned step to the Mission Planner.

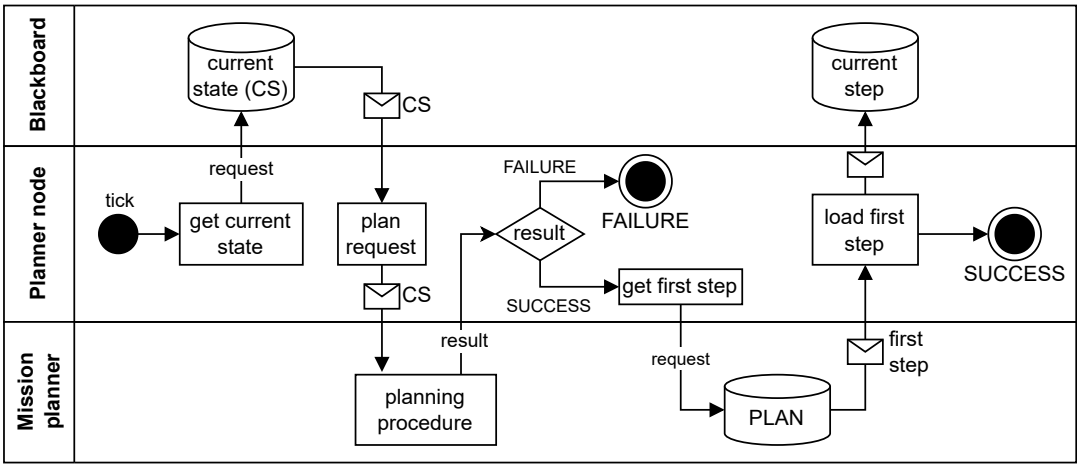


Figure 3.8: Schema presenting graphically the steps occurring when the behavior tree triggers a plan routine.

Mission Planner receives the system state, it initiates the planning procedure. The resulting plan is then returned to the planner node. Subsequently, the node loads the initial action from the formulated plan into the behavior tree and signals a successful completion, thereby initiating a new execution routine. The process is reported in figure 3.8 for clarity.

Concurrently, a decorator node (succ is run) propagates the tick signal to a fallback node (trans action, fb) whose first child is the executor (transition executor) responsible for running the transition action specified by the step currently loaded in the tree. In case no action is available, e.g. during the initial planning when the tree is ticked for the first time, a default wait action is selected. Notably, if the transition action node fails, its parent ticks the following child (rise action flag). This node raises an error

flag to trigger the contingency branch and then returns failure. The decorator node has a key role in the planning phase and applies a special ticking rule to its child. As long as the child node status is running or failure, it is propagated upward with no alteration. Instead, when the received state is success, the decorator node signals running, and continues to return this state as long as its branch is not interrupted, without ticking its child again. This assembly is adopted to execute the transition action without signalling its completion to the parallel node supervising the planning phase. In this arrangement, both the success and failure thresholds of this parallel node are set to one. Using this strategy, the planning branch can be terminated as soon the Mission Planner produces a valid plan in order to immediately start to process it, even if the transition action is not completed. On the other hand, the decorator node never returns success, ensuring that the executor does not terminate the planning branch when completing the transition action. Nevertheless, the action can still trigger a failure in the branch, allowing for proper contingency management.

A fundamental edge case occurs when the Mission Planner finishes the objectives to elaborate. In this scenario, no plan is forwarded to the planner node, which as a result returns failure. This leads the nominal execution branch to fail with no error flag currently up. Under these circumstances, the contingency subtree detects no error, skipping all children and returning with a failure state as well. Thus the MCM node proceeds with ticking its last child: the "end mission" node. This node reflects the mission termination as perceived by the Mission Planner. It defines the exit point of the tree by always returning failure after predefined functions are eventually executed, e.g. reporting the mission result by listing the fulfilled and failed objectives and disarming the UAV. The failure state of this node propagates to the loop node, which returns failure as well, thus terminating the entire tree.

A notable characteristic of the system's design is the method used to handle mission failures. The nominal mission branch, representing the primary course of action, fails only when the system is unable to generate a feasible plan within the current context. This highlights the system's commitment to maintaining a logical approach to problem-solving. This approach ensures that the mission itself is safeguarded from failure except in scenarios where the system's logical reasoning is unable to converge to a viable solution. Importantly, the system accommodates various minor setbacks during the actual execution of actions. These setbacks, which might arise due to environmental conditions, technical faults, or other transient factors, do not inherently trigger mission failure. This design philosophy acknowledges the inherently dynamic and uncertain nature of real-world operations, emphasizing the separation between the logical feasibility of a plan and the practical obstacles that might emerge during its execution. In essence, the system's robustness is built on its ability to pivot from execution to re-planning in response to failures. This approach ensures that mission success remains achievable, contingent upon the system's capability to logically address the current situation, while also accommodating the inherent complexities that can arise during the execution of actions. This dynamic capability to shift from execution to reevaluation reflects the system's adaptability and resilience in the face of unforeseen challenges.

3.4.2 Contingency management

The contingency management subtree operates as a fallback mechanism (contingency behavior, fb), as suggested by Albore et al. in [42]. It diverges into distinct subtrees, each overseeing the resolution of a single error occurrence. To achieve this, a sequence node (ERR#, seq) is utilized, serving as a mechanism for checking individual errors. Within this sequence, the first child is a condition node (check err #) responsible for evaluating whether the associated error flag has been activated. When the flag indicates an issue, the sequence advances to initiate the corresponding error response action. Conversely, if the flag remains inactive, the sequence returns a failure signal, prompting the evaluation of the subsequent error flags. To address specific errors, the design accommodates the inclusion of multiple potential actions for resolution. To streamline this, all desired actions are linked to a single fallback node (solve err #, fb), which systematically attempts each action (action; action2; emerg action) in sequence until a successful one emerges. Lastly, the sequence node associated with the solved error ticks its last child (reset err #), resetting the appropriate flag. An advisable design practice is to incorporate an "abort-mission" action (emerg action) as the final option within this fallback node. This ensures an additional layer of safety in cases where earlier actions prove inadequate in addressing a specific emergency. Notably, if the user desires to terminate the mission after executing the emergency action, the skill should terminate with a failure status. This way the entire branch associated with the error fails and the system reaches the end mission node, effectively terminating the mission.

The inherent priority structure of control nodes plays a pivotal role in this framework. It can be exploited to establish priority relationships between raised errors and the corresponding triggered actions. A visual representation of this concept is provided in Figure 3.9, where "error 1" is processed before "error 2", and the system

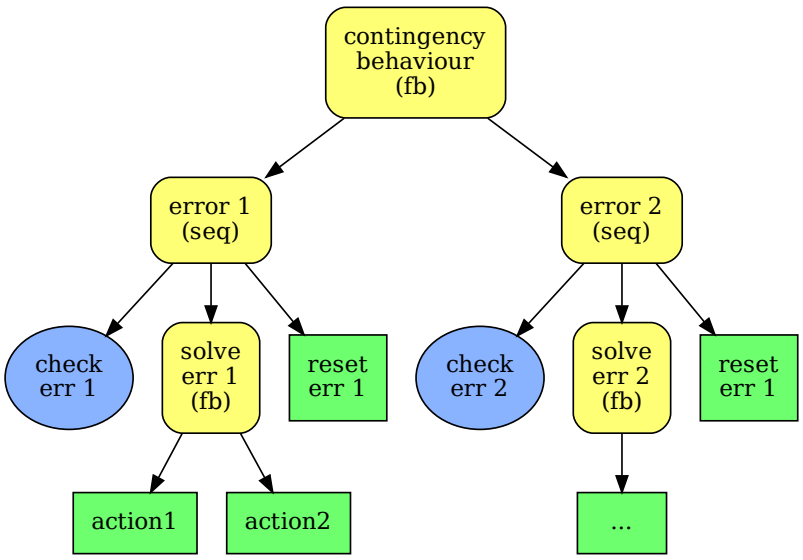


Figure 3.9: Basic behavior tree structure used for contingency management.

attempts "action 1" before resorting to "action 2" if the former proves unsuccessful. In case multiple flags are raised from the monitor, the contingency management branch triggers the action associated with the most urgent error, returning with a successful state and resetting the appropriate flag when the action is completed. Consequently, the main loop ticks again the contingency behavior node, reevaluating all error flags from scratch. This ensures that, if a flag rises during the contingency action execution, the system properly follows the priority encoded in the tree structure when checking the contingency conditions.

Additionally, by effectively segmenting the branches and introducing auxiliary error flags, a hierarchical organization of errors can be achieved within subbranches.

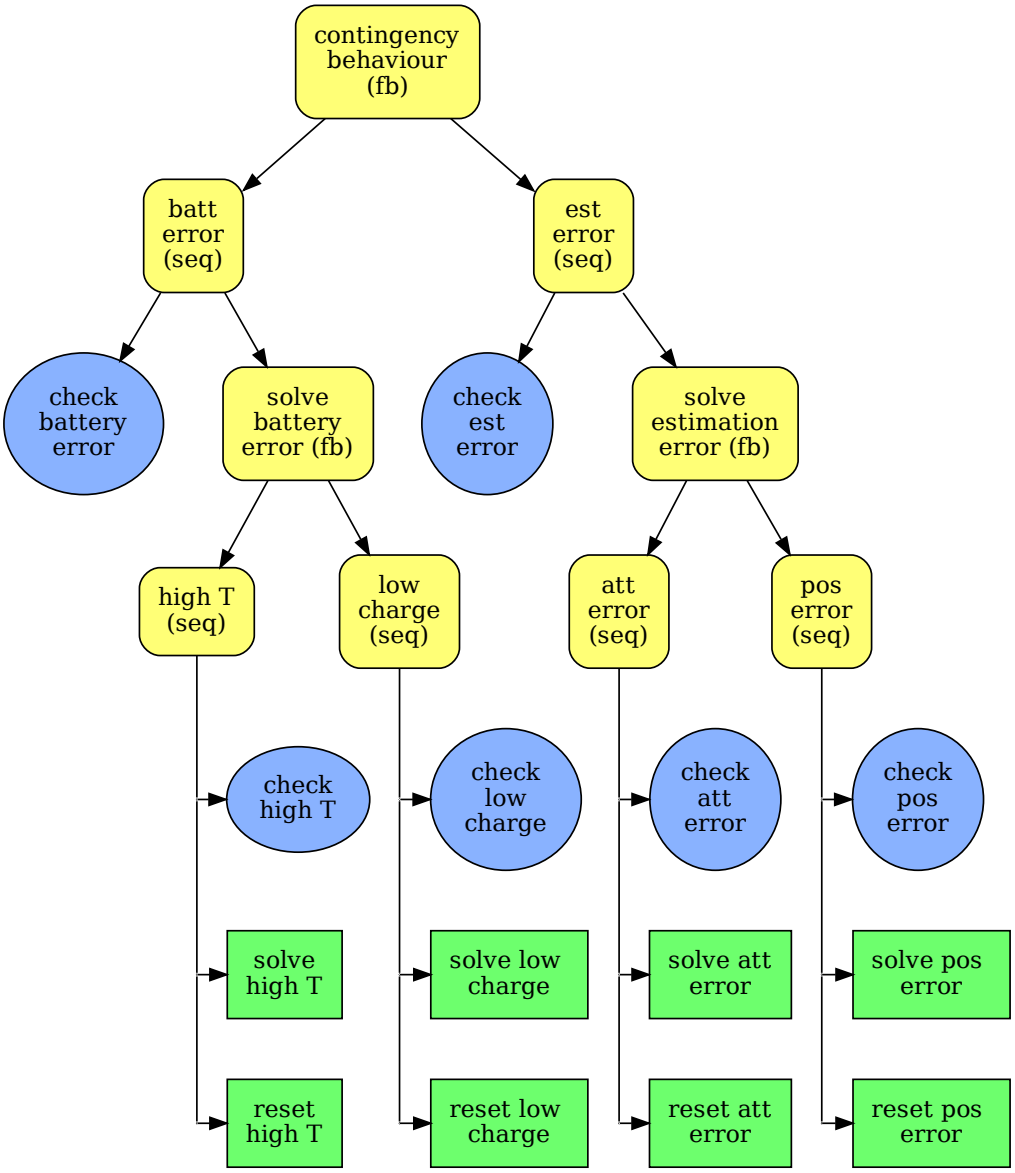


Figure 3.10: Basic behavior tree structure used for organizing contingency errors hierarchically.

An illustrative instance of this arrangement is presented in figure 3.10 for clarification. Within this structure, the primary contingency fallback node (contingency behavior, fb) partitions into subgroups: battery errors (battery error, seq) and estimation errors (estimation error, seq). An auxiliary flag is raised when any error within the given subgroup is detected. Each subgroup in itself forms a sequence node, characterized by an initial condition node (check battery error; check est error) that assesses the auxiliary flag's status. If the flag signals that an error belonging to the subgroup is active, the subsequent fallback node is ticked (solve battery error, fb; solve estimation error, fb). This node unfolds similarly to the first contingency fallback node, ticking a set of sequence nodes, each one addressing a specific error. This structure rapidly bypasses branches where no pertinent errors have transpired, thus enabling rapid access to the correct contingency case. This approach significantly improves the handling of complex systems, particularly those situations waiting for the assessment of several prior flags. Notably, the structure is repetitive and multiple nesting levels can be added by associating a subgroup of errors to every sequence node. The arrangement terminates with the fallback node attempting the desired actions for the identified error. Oppositely to the ordinary flags, the auxiliary ones are reset by the monitor node. In its routine, it always resets all auxiliary flags and then proceeds by raising them again according to the normal ones. This avoids adding a reset node for the auxiliary flags while also ensuring no interruption of the contingency actions.

The architecture built upon behavior trees offers a straightforward mechanism for contingency management through the inherent tree structure, specifically through fallback or sequence nodes. This clarifies the pathway for state transitions, a feature not transparently offered by FSMs. Indeed, in FSM architectures, the central challenge arises from the ambiguity of event origins. FSMs lack a built-in signalling mechanism to handle state transitions, requiring external events to manage them. By contrast, behavior trees provide an unambiguous framework for managing the outcomes of subtrees, with control nodes explicitly designed to handle successes or failures. Addressing the use of HFSMs, particularly those modelled after Harel Statecharts [43], one might argue that they too allow a nominal-to-contingency state transition. However, this transition is not as straightforward as it initially appears. HFSMs require the failed action to post an "error" event, triggering a transition from a nominal to a contingency state. While this can be manageable, it introduces additional complexity by adding potentially non-prioritized events to an already existing event queue. The lack of mechanisms for event prioritization in FSMs could lead to delayed critical transitions, thus reducing the advantages HFSMs might have over alternative architectures. In contrast, behavior trees provide a clear and structured framework for efficiently managing both standard and contingency scenarios, making them a compelling option for navigating complex, dynamic environments.

3.4.3 Executor

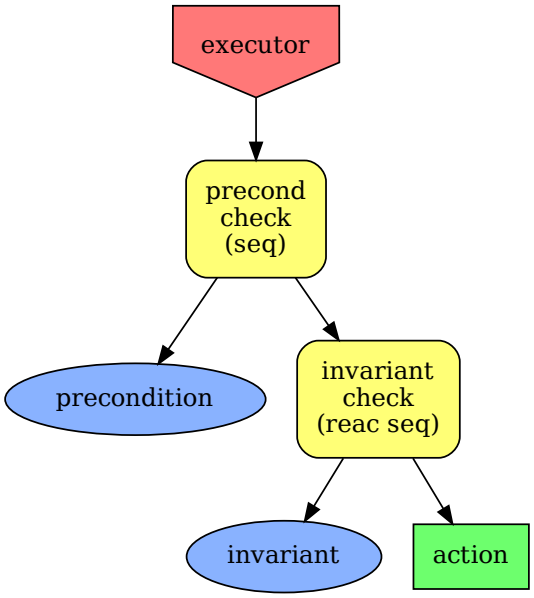


Figure 3.11: Behavior subtree structure used for executing a single action.

The executor module is responsible for carrying out the actions generated by the planner. Once the planner synthetizes a plan, the action to be executed is loaded on the tree blackboard. The information loaded on the blackboard comprehends a set of preconditions, invariants, and supplementary parameters used to define the execution of the skill. The executor makes use of preconditions and invariants to initiate and oversee the execution of the skill. When the executor is ticked for the first time, it generates an appropriate behavior tree to carry out the current action and starts it. Whenever the executor node is ticked again, it propagates downwards the tick signal and then propagates upwards the state resulting from the subtree. Thus, it behaves as a connection node that solely propagates information from the main tree to the action tree. The advantage of this solution is the remarkably simple generated sub-tree, with respect to the opposite design choice of generating a behavior tree representing the entire plan execution, adopted in [15]. This allows for a swift propagation of the tick signal through the behavior tree, being the depth of the overall tree significantly reduced. Moreover, the tree creation is also faster, a positive aspect if the system runs several replanning phases where a new subtree is generated each time. The drawback of this design choice is the need to create a subtree for each action, rather than instantiating the whole structure once at the start of the mission. Thus, in case the mission proceeds smoothly, the adopted solution is less efficient.

Illustrated in Figure 3.11 is the structure of the subtree generated by the executor. At its root stands the executor itself, serving as the port for propagating the tick signal from the main tree. Directly beneath it lies a sequence node (precond check, seq) that splits into two child nodes. The initial node in this sequence is a condition node (precondition) responsible for asserting whether the preconditions of the currently

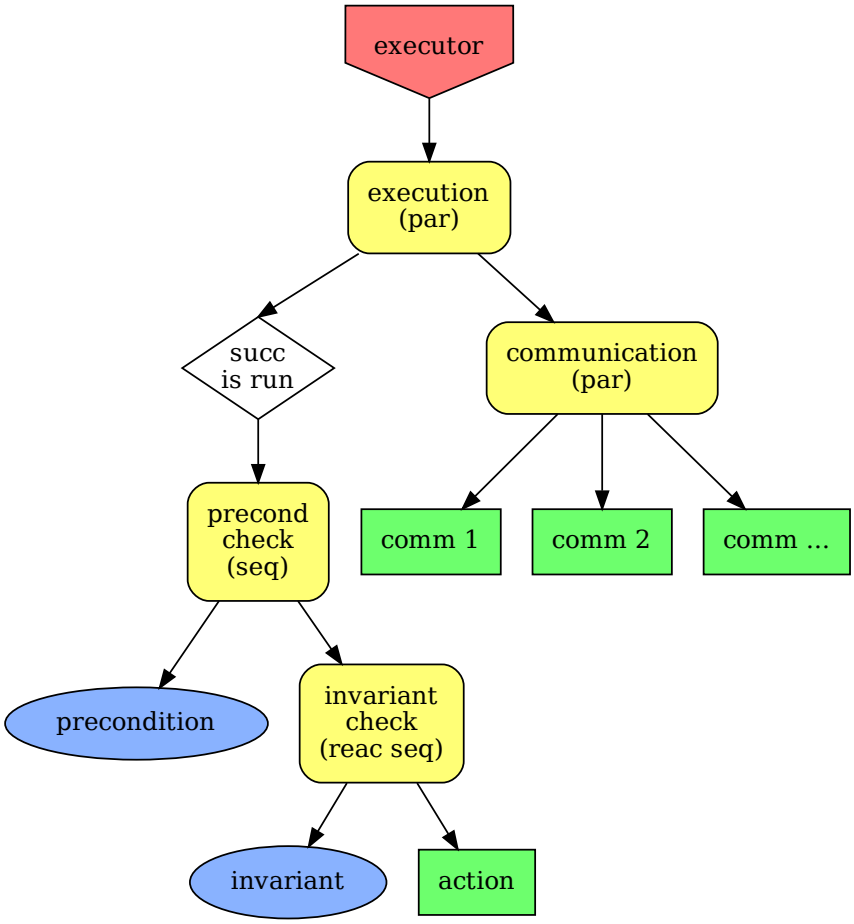


Figure 3.12: Behavior subtree structure used for executing the auxiliary action along with the necessary permission requests.

executed action are met. If this condition is validated, the subsequent node in line for evaluation is a reactive sequence node (invariant check, reac seq). This particular node incorporates the reactive behavior essential for monitoring the execution of the skill. It achieves this by regularly testing invariant conditions of the skill. Henceforth, the first child node in the reactive sequence node is a condition node (invariant) tasked with confirming whether the invariant constraints align with the current state of the system. Meanwhile, the second child node (action) is entrusted with the actual execution of the planned step.

Regarding the coordination of the communication and auxiliary action, a specific subtree structure is used, shown in figure 3.12. In this arrangement, a parallel node (execution, par) oversees the execution of the auxiliary action and the permission requests. All communication skills are grouped in a dedicated parallel node (communication, par). Notably, the auxiliary action is preceded by a success-is-running decorator node (succ is run) to avoid propagating upwards the success of the skill. The "execution" node has both the success and failure thresholds set to one. This means that the action can trigger a failure, but can't lead the parent node to return

success thanks to the interface provided by the decorator node. On the other side, the communication parallel node has the failure threshold set to one and returns success only if all its children succeed. This strategy guarantees that the system triggers a re-plan whenever either the communication skill or the auxiliary action reports a failure. Additionally, the executor refrains from advancing to the next planned step until confirmations for all the permission requests are received.

This design highlights the integration between the executor and the Mission Planner. In cases where a third planning sequence needs to be generated, for example, to incorporate an additional skill set relying on a distinct subsystem, the executor can easily be expanded. This is achieved by adding a new sub-branch beneath the "execution" node, which manages the execution of the skills defined by this third sequence. Finally, the tick system allows for a flawless propagation of the states upward in the tree nodes hierarchy, allowing for proper elaboration of errors.

Implementing a similar system using FSMs or HFSMs would be a complex task. FSMs fundamentally operate on a bottom-up event-handling system. This constrains the architecture by making it difficult to introduce higher-level logic that can override or preempt lower-level states, a functionality crucial for tasks like invariant checks that need to execute on every tick. In dynamic and unpredictable settings, the FSM model's lack of event prioritization is a drawback, as all events are treated with the same urgency unless explicitly programmed to do otherwise. Additionally, FSMs have a state-centric design that demands explicit definitions for transitions between states. This makes them less adaptable to changing conditions, as introducing a new behavior often necessitates updating multiple transitions. While HFSMs offer a hierarchical structure, the super-state usually remains unaware of the activities within its sub-states. This makes it challenging to coordinate complex, parallel activities across different hierarchical layers without resorting to custom logic. Conversely, behavior trees enable a more scalable and flexible architecture, allowing for independent nodes whose transitions are regulated by their parent nodes. This makes them particularly well-suited for managing complex behaviors in dynamic environments.

4 Implementation

4.1 Employed middleware and libraries

For the implementation of the presented design, the Robot Operating System (ROS) [44] has been used as the foundation and linking system between the architecture layers. Specifically, this project is based on ROS 2 Foxy, one of the many available versions of ROS 2, the latest ROS major version. ROS is used in several robotic studies [22, 25] to cope with the necessity of a middleware to manage real-time communications. It provides a robust communication infrastructure that allows various software components (nodes) to communicate with each other in a distributed and scalable manner. This communication is based on a publish-subscribe messaging system. ROS is highly modular, allowing developers to design and build robotic systems by combining existing software components. It abstracts the hardware layer, making it possible to write robot control software that is hardware-agnostic. This abstraction simplifies development and facilitates the use of different robot platforms.

In ROS, communication between nodes is achieved through a flexible and modular system consisting of topics, services, and actions. These communication mechanisms enable nodes to exchange data and interact with each other in a distributed robotic system. Follows an overview of topics, services, and actions in ROS 2:

- **Topics** are one of the core communication mechanisms in ROS 2. They facilitate asynchronous, publish-subscribe communication between nodes. Nodes can publish data on a topic, and any interested nodes can subscribe to that topic to receive the data.

- **Services** provide synchronous, request-response communication between nodes. They allow one node (the client) to send a request to another node (the server), which processes the request and sends back a response.

- **Actions** extend the capabilities of ROS 2 by providing asynchronous, goal-oriented communication. They allow a client node to send a goal request to a server node, which processes the goal over time and provides feedback on its progress. Once completed, the server sends a result back to the client.

A key component of the ROS 2 framework is the "executor". It refers to a component responsible for executing the tasks associated with nodes and managing the communication and coordination among these nodes. Specifically, the ROS 2 executor is a part of the middleware that facilitates the execution of callbacks for various nodes and handles message passing between them. It allows nodes to work asynchronously by managing their concurrency and synchronization, meaning that they can run their callbacks independently. Notably, all communications mechanisms described above run using callbacks. Indeed, ROS 2 nodes are designed to work in an event-driven architecture. This means that nodes can be idle most of the time, waiting for events to occur, such as receiving a message on a topic or a service request. Instead of using busy-wait loops, ROS 2 nodes use "spinning" to efficiently handle events. To ensure

that the callbacks are executed when the corresponding events occur, it is necessary to "spin" the node. This means that the node enters a mode where it actively checks for events and executes the associated callbacks.

In a typical ROS 2 application, multiple nodes run concurrently, each with its spinning loop initiated by a spin function. The executor, on the other hand, is responsible for coordinating the execution of these multiple nodes. It ensures that each node gets a chance to execute its callbacks in a non-blocking manner and that the overall application remains responsive to events. To do this, callbacks are registered in a First-In-First-Out (FIFO) queue. Then, the executor consumes the queue by properly managing the function execution and parallelism constraint.

A major concern is the interfacing between behavior tree nodes and ROS nodes. It primarily arises from the different execution paradigms and thread management: behavior trees use a tick-based execution, while ROS nodes are event-driven. ROS 2 typically manages its nodes and callbacks in a multi-threaded manner, with separate threads for spinning nodes, handling incoming messages and executing callbacks. A behavior tree, on the other hand, operates in a single-threaded manner by design, as it expects its nodes to be executed sequentially. This sequential execution can conflict with the ROS multi-threaded execution model. The solution used in this project is to assign one ROS node to a tree node, an approach also implemented in Nav2 [45], the ROS Navigation Stack which is also based on behavior trees. This way, the behavior tree oversees the execution of the ROS functionalities. Every time a tree node that embeds a ROS node is ticked, it spins the associated ROS element, thus starting the execution of the queued callbacks. This way, the execution is constrained to follow the behaviour specified by the tree and interactions with the ROS environment are enabled only when requested.

As for the design of the behavior tree orchestrating the mission manager, there exist mainly two libraries to design behavior trees: "BehaviorTree.CPP" [29] and "py_trees" [46]. Other libraries like "Behavior3" [47], "task_behavior_engine" [48], "beetree" [49] and "behavior-tree" [50] are no more maintained. "BehaviorTree.CPP" is written using C++, while "py_trees" is based on Python. Both programming languages are widely used in the robotic field and the libraries provide overall the same functionalities. However, these two programming languages present some intrinsic differences. For this project, one key aspect is concurrency. C++ provides fine-grained control over memory management and low-level system resources, making it suitable for creating high-performance multithreaded applications. It also offers standard libraries that provide thread management (thread), synchronization primitives, and memory management tools (mutex). On the other hand, Python offers a built-in threading module for creating and managing threads. However, Python's Global Interpreter Lock restricts the execution of multiple threads in a Python process, limiting true parallelism. Moreover, C++ is well known for its high efficiency and large employment in real-time applications, while Python may suffer lower performances due to its high-level scripting and the presence of an interpreter. For these reasons, and also considering that the current software stack developed by DLR for the UDW project is mainly developed using C++, the same programming language has been used for the coding of this thesis project. Thus, the employed library to manage behavior

trees is "BehaviorTree.CPP".

The "BehaviorTree.CPP" library is built with modularity as a key characteristic. It allows developers to define and reuse individual nodes and compose them into complex behavior trees. This modular approach simplifies code maintenance and promotes code reusability. "BehaviorTree.CPP" supports various types of nodes commonly found in behavior trees, including action nodes, condition nodes, control nodes and decorator nodes. Developers have the capability and responsibility to craft customized node types, tailoring behavior trees to suit their particular application requirements. This extensibility fosters adaptability and versatility across different use cases. Furthermore, programmers can delineate the structure of the behavior tree in separate Extensible Markup Language (XML) files facilitating runtime loading of the behavior tree layout, a key characteristic to implement the design proposed in this thesis for the executor. This approach also enables the reuse of subtrees defined in distinct files by assigning a unique ID to each behavior tree, making it possible to insert them as subtrees within another behavior tree. Regarding the adopted data-sharing infrastructure across different nodes, the "BehaviorTree.CPP" library implements a "port" system to access the key-value pairs saved in the blackboard. Every node defines a set of input and output ports to access and set the blackboard entries. Each port is defined by an ID, which is then linked to the key of the associated blackboard variable in the XML schema. Thus, the linking is done in run-time when the tree is built, and the definition of the nodes is completely decoupled from the tree in which they are used, allowing their full reusability in different structures.

Furthermore, it is crucial to highlight that the library prioritizes real-time applications, making it particularly suitable for robotics use cases. Efficiency in executing behavior trees is fundamental in robotics applications, and "BehaviorTree.CPP" has been purposefully crafted to meet these demands. Notably, it seamlessly integrates with ROS, solidifying its position as a valuable tool in the field of robotics. It is worth mentioning that this library is also adopted in Nav2, underlining its significance within the robotics community.

4.2 Functional layer

4.2.1 PX4 Autopilot

For the execution of simulation tests of the proposed solution, the functional layer of the control architecture is built upon the PX4 Autopilot software suite [40]. The PX4 Autopilot stack plays a pivotal role, managing the core control mechanisms of the drone and serving as the principal agent for autonomous navigation and operation. As an autopilot, it is charged with overseeing essential flight operations, including position, orientation, and velocity control, to ensure precise adherence to prescribed flight paths and responsiveness to incoming navigational commands. Within the structure of the control architecture, it operates as the functional layer, providing an interface for high-level command inputs and translating them into real-time operational commands, thereby bridging the gap between abstract control directives and practical drone behaviors.

To facilitate comprehensive testing without the prerequisite of physical hardware, PX4 provides support for multiple simulation environments. Two prominent PX4 simulation environments are Gazebo and jMAVSim (Java MAVLink Simulator). For this project, jMAVSim has been selected for its simplicity and ease of use. It is a lightweight, Java-based multirotor simulator primarily designed for PX4 development. While not as high-fidelity as Gazebo, jMAVSim provides reasonably realistic dynamics. In the context of these simulations, jMAVSim serves as the hardware/physics layer simulating the flight dynamics, while the PX4 Autopilot stack represents the functional layer operating on the UAV.

4.2.2 Communication interface

The communication protocol adopted by the PX4 Autopilot employs micro Object Request Broker (uORB) messages. These messages are designed to provide a lightweight and efficient means of inter-process communication in real-time and embedded systems, particularly for sharing information between different modules, tasks, and processes in PX4's architecture. uORB messages follow a publish-subscribe pattern, where one module publishes data, and other modules can subscribe to receive that data. This decoupled communication paradigm allows for modularity, as modules can be developed independently and exchange information seamlessly through uORB. Within PX4, uORB messages facilitate the exchange of critical information related to flight control, sensor fusion, navigation, mission planning, telemetry, and various other aspects of UAV operations. Their structured and reliable communication capabilities enable the different components of PX4 to collaborate seamlessly, enabling effective control and navigation of UAVs.

4.3 Executive layer

In the implementation, the entire executive layer is programmed as a single ROS node. It communicates with the functional layer via several topics provided by the PX4 Autopilot running in the jMAVSim simulation. On the other side, it provides several action servers as an interface for the mission manager, composing the Skill Manager. Moreover, the ROS node represents the monitor module, elaborating the data coming from the PX4 Autopilot and providing them to the MCM packed in a single introspection topic. A comprehensive representation of the executive layer ROS node, called "px4_commander", is provided in figure 4.1.

4.3.1 Functional layer interface

The uORB topics of interest for interfacing the functional layer with the executive layer are listed below. Topics marked as "in" provide input messages to the functional layer, while topics identified by "out" contain messages generated by the PX4 Autopilot.

- "/in/vehicle_command" allows external systems or modules, such as ground control stations or companion computers, to send high-level commands to the autopilot. These commands can include requests for takeoff, landing, waypoint

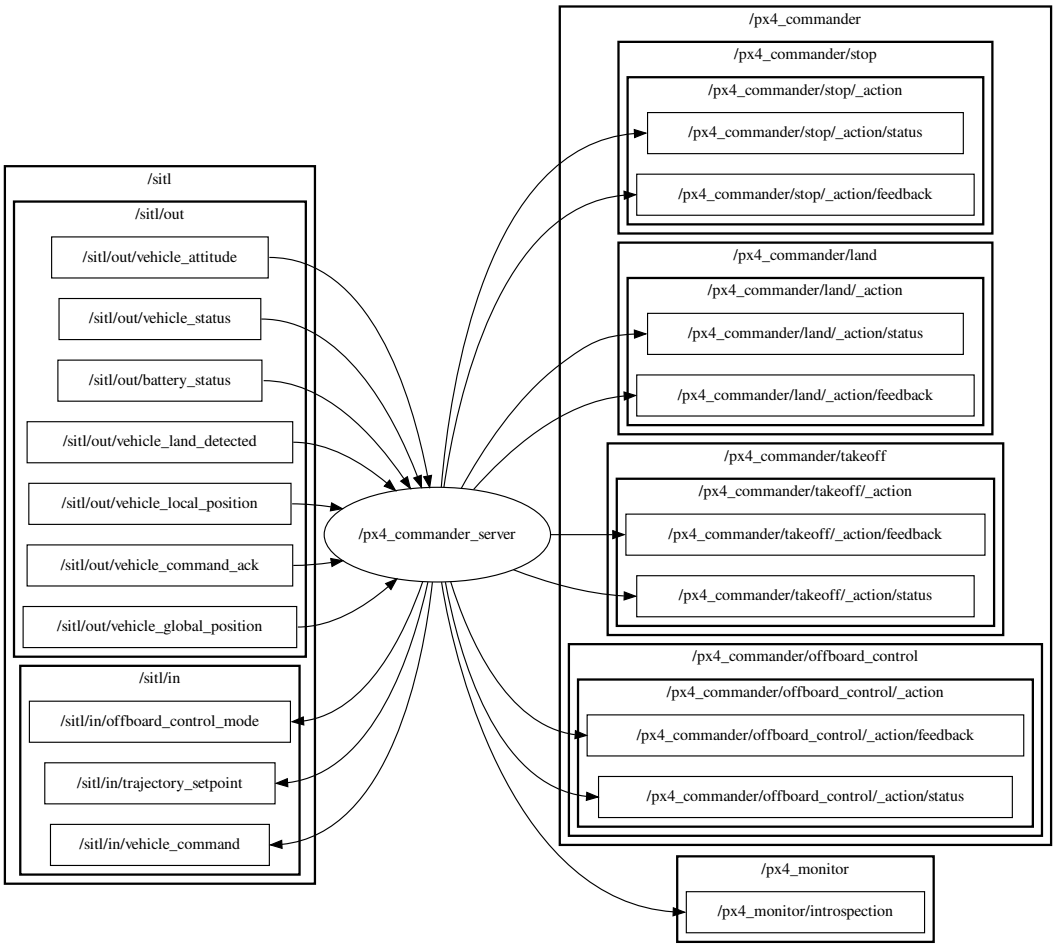


Figure 4.1: Graph showing the ROS messaging structure of the executive layer ROS node, generated using RQT Graph, a standard tool of the ROS framework.

navigation, loitering, and other mission-related instructions. It is also used to switch the vehicle between different flight modes, such as manual mode, altitude control mode or position control mode. The Skill Manager sends "vehicle_command" messages to execute commands like arm/disarm, takeoff, land and loiter.

- "/out/vehicle_command_ack" is used to provide feedback and acknowledgements regarding the execution of high-level commands and instructions sent via the "vehicle_command" topic. When a command is sent to the autopilot through the "vehicle_command" topic, the autopilot processes the request and then generates an acknowledgement message. This acknowledgement indicates whether the command was received and executed successfully or if there were any issues during execution. The returned status can include information about successful execution, errors encountered during execution, or the current state of the vehicle in response to the command.

- "/in/offboard_control_mode" allows for the control of the vehicle by an offboard system. When the autopilot receives specific commands through this topic, it transitions the vehicle into offboard mode. In the PX4 Autopilot system, this mode allows external systems, such as companion computers, to take direct control of the vehicle's movements. External systems can send commands that specify the desired vehicle behaviour, such as setting target positions, velocities, or attitudes. This control paradigm is often used in automation scenarios where external systems take over control for tasks like precision landing, package delivery, or inspection missions.

- "/in/trajectory_setpoint" is used to communicate high-level trajectory setpoints to the flight controller. It works in tandem with the offboard mode to enable autonomous flight and advanced control of the vehicle. The topic typically includes information about the desired position, velocity, acceleration, and yaw (heading) setpoints for the vehicle. These setpoints describe how the vehicle should move through space over time.

- "/out/vehicle_status" provides a comprehensive snapshot of the vehicle's operational state, including various modes, armed/disarmed status, safety flags, and more. These flags can indicate critical conditions or events, such as GPS glitches, sensor failures, or low battery levels, which may affect the vehicle's operation. The topic is essential for fault detection and management. Anomalies detected in the vehicle's systems or sensors can trigger safety measures or mode changes to ensure safe operation.

- "/out/vehicle_attitude" provides real-time data about the vehicle's orientation in three-dimensional space. The attitude information is represented using quaternions.

- "/out/vehicle_local_position" contains data related to the vehicle's position in a local reference frame. Local position data is reported in a local coordinate system, often referred to as the North-East-Down (NED) frame. In this frame, the initial takeoff position serves as the reference point (0,0,0), and the vehicle's position is measured relative to this reference. The position data is usually reported in meters.

- "/out/vehicle_global_position" contains data related to the vehicle's position on the Earth's surface in a global coordinate system. This means it provides information about the latitude, longitude, altitude, and other global position-related parameters. Latitude and longitude are specified in degrees, while altitude is usually reported in meters above a reference point.

- "/out/vehicle_land_detected" is used to determine if the vehicle has initiated or detected a landing manoeuvre. It provides information regarding the landing state of the vehicle. Various conditions can trigger landing detection, including altitude, velocity, and proximity to the ground. These conditions are typically

topic name	publish rate [hz]
vehicle_land_detected	1
vehicle_attitude	240
vehicle_local_position	123
vehicle_global_position	123
vehicle_status	2
battery_status	99

Table 4.1: Publish rate of the topics provided by the PX4 Autopilot used as the source of data for the identification of the system's current state.

defined by the flight controller's software and may vary depending on the vehicle configuration.

- "/out/battery_status" provides information about the vehicle's battery, including voltage, current, capacity, charge percentage, temperature, and health.

Table 4.1 reports the publish rate of the topic listed above. The topic "/out/vehicle_command_ack" is not reported as it does not provide a continuous stream of information. The above data must be taken into consideration for the design of the executive and deliberative layers. As different information is updated at different rates, the system might need to wait for the information to be refreshed before initiating specific functions.

To translate uORB topics into ROS topics, it is necessary to run a bridge or a middleware that supports both UORB and ROS and acts as a translator between the two. The official documentation of the PX4 Autopilot [40] suggests to use of the Micro XRCE-DDS (uXRCE-DDS) communications middleware. It is a lightweight and efficient Data Distribution Service (DDS) implementation designed for resource-constrained and embedded systems. DDS is a standardized middleware protocol used for data-centric communication in distributed systems, particularly in the field of robotics. Like other DDS implementations, uXRCE-DDS is focused on data-centric communication. It allows devices and applications to publish and subscribe to data topics, enabling efficient and flexible data exchange. uXRCE-DDS supports various Quality of Service (QoS) policies that allow developers to specify the reliability, durability, and other characteristics of data communication. This ensures that data is delivered according to the desired constraints and requirements.

The uXRCE-DDS middleware consists of a client operating on PX4 and an agent operating on the companion computer, as depicted in figure 4.2. These two components run bidirectional data communication, using serial, UDP, TCP, or a custom link for connectivity. The agent serves as a mediator for the client, facilitating the publication and subscription of topics within the global DDS environment. The client is an integral part of the PX4 Autopilot software suite and comes pre-installed. Therefore, it is necessary to only initiate the agent on the companion computer to enable the translation process. The configuration file "PX4-Autopilot/src/modules/uxrce_dds_client/dds_topics.yaml" defines which uORB topics generate a corresponding ROS topic. Not all topics are

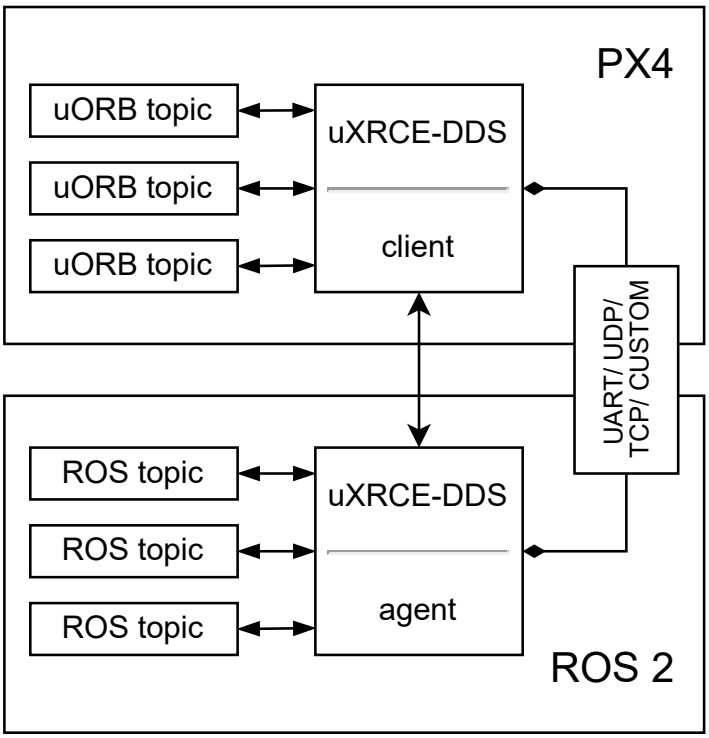


Figure 4.2: Schema showing the role of the Micro XRCE-DDS (uXRCE-DDS) communications middleware in the interfacing between the executive layer and the functional layer. The figure is inspired by the schema provided in the official documentation of the PX4 Autopilot [40] covering the PX4-ROS2 integration.

listed in this file by default, it is necessary to add them based on the needs of the project. Regarding the topics previously listed, "vehicle_command_ack", "vehicle_land_detected" and "battery_status" must be inserted manually.

Once all topics are set up, the executive layer node subscribes to all "out" topics, while it publishes on all "in" ones, as shown in figure 4.1. Notably, in order to subscribe to the PX4 Autopilot topics, it is necessary to define a compatible QoS profile.

4.3.2 Mission and contingency manager interface

The Skill Manager provides a set of actions, which the MCM can initiate in order to carry out the planned steps. Each skill is represented by a ROS action server managed by the Skill Manager. The action servers created to test the proposed design are described below.

Land The system lands in the current (X, Y) position. This is achieved by publishing a message on the "vehicle_command" topic, specifying "land" as the type of the command. The system marks the landing as successful once the functional layer reports on the "vehicle_land_detected" topic that the UAV is on the ground.

- Goal: none.
- Result: reached altitude, in meters.
- Feedback: current altitude and Z velocity, in meters and meters per second, respectively.

Takeoff The UAV takes off in the current (X, Y) position, varying the altitude by the amount specified by the goal in meters. For instance, if the current altitude is 3 meters and the goal specifies 3, the final altitude is 6 meters. To carry out a takeoff task, the Skill Manager first arms the drone, if necessary. Then, initiates the takeoff procedure by publishing a message on the "vehicle_command" topic, specifying "takeoff" as the type of the command and the target altitude. The takeoff action terminates when the target altitude is reached with a Z velocity close to zero.

Action definition:

- Goal: differential altitude for takeoff, in meters.
- Result: reached altitude, in meters.
- Feedback: current altitude and Z velocity, in meters and meters per second, respectively.

Stop The UAV stops in the current position, initiating a loiter procedure. This command is also called when an action is cancelled in order to overwrite it. When this action is requested, the system first checks if the UAV is flying. If the drone is on the ground, it will keep the current armed/disarmed state. Otherwise, if the drone is flying, a loiter mode is initiated. This is done by sending a message on the "vehicle_command" topic specifying the nature of the command (set mode), the main mode (auto), and the sub-mode (loiter). The system terminates the action and returns the result once the velocity has reached a value close to zero.

Action definition:

- Goal: none.
- Result: (X, Y, Z) position, in meters.
- Feedback: current (X, Y, Z) position and velocity, in meters and meters per second, respectively.

Offboard It commands the UAV to follow a specified position or velocity-controlled command. In case it receives a position command, the drone will reach the specified waypoint following a straight trajectory. The command also specifies a velocity constraint using the velocity goal field. The action terminates once the target position is reached with a velocity close to zero. Instead, if the velocity is controlled, the command moves the UAV with the specified velocity and terminates when the timeout specified in the goal description runs out. Upon termination, the Skill Manager initiates

a loiter procedure. It is also possible to specify the timeout as infinite. In order to switch to offboard mode, the PX4 Autopilot expects to receive an "offboard_control_mode" message with a minimum frequency of 2 Hz as "proof of health" [40] of the companion board and its connection status. This message also specifies the type of offboard mode, e.g. position or velocity control. The actual position/velocity command is specified in a "trajectory_setpoint" message, which needs to be sent only once. In the current implementation of the planner, only the position control mode is used, hence this skill is also referred to as "position-control" action.

Action definition:

- Goal: a "mode" field indicates whether the action is a position command (mode 0), or a velocity command (mode 1). Then, the desired position and velocities are specified, respectively with yaw or yaw rate to vary the drone orientation. All values are referred to the NED frame, in meters, meters per second, radiants and radiants per second, respectively. Finally, the timeout is defined in seconds.

- Result: reached (X, Y, Z) position and velocity, in meters and meters per second, respectively.

- Feedback: current (X, Y, Z) position and velocity, in meters and meters per second, respectively.

Finally, the executive layer provides a topic called "introspection" where all information provided by the functional layer about the system is elaborated and aggregated. This represents the foundation of the introspection routine run by the system being the primary source of real-time information about the current status of the UAV. The monitor node of the MCM tree subscribes to this topic to receive constant updates about the UAV status. Given the different publishing rates of the combined topics reported in table 4.1, the rate set for the introspection topic is $100Hz$. The chosen value aims to update almost all data consistently, while not overloading the system with a high number of requests. Notably, "vehicle_attitude" is the only topic whose publish rate is considerably higher than the chosen value. This is because the stability control of the drone is not addressed in the MCM, as it is considered a fundamental functionality provided by the flight controller.

The overall information flow, starting from the MCM until the simulation environment offered by jMAVSim, is summarized in figure 4.3.

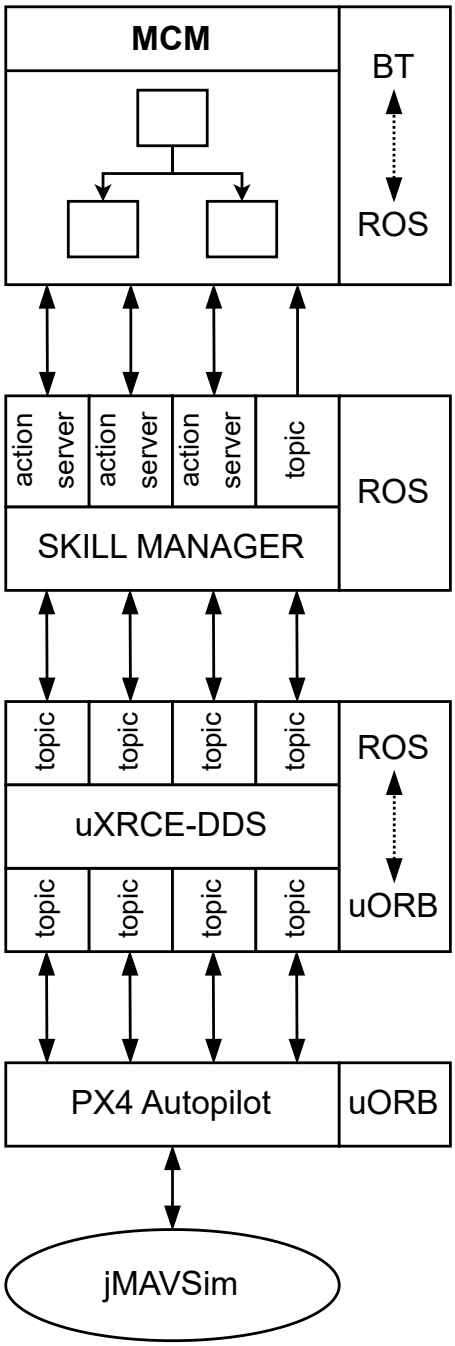


Figure 4.3: Schema summarizing the information flow and the infrastructure connections between the several blocks constituting the proposed architecture and the simulation environment.

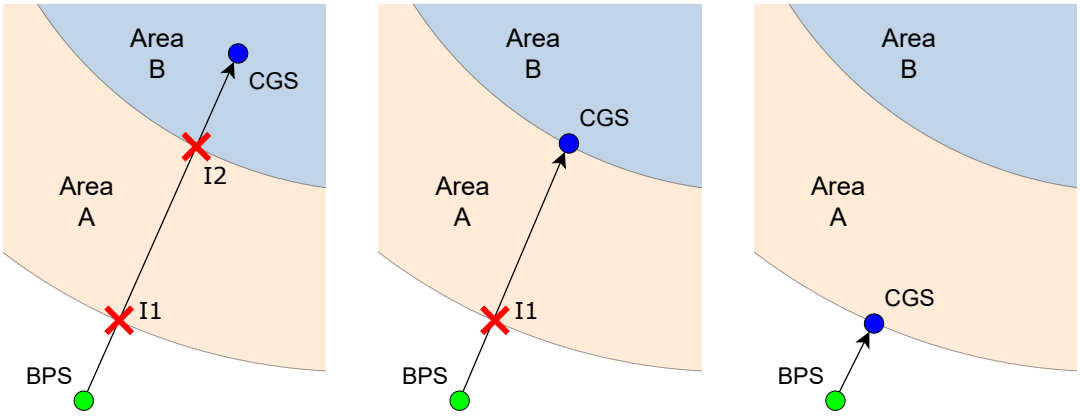
4.4 Mission Planner

In order to implement the outlined algorithm for the Mission Planner, one key aspect is a proper definition of the state. As previously stated, it is important to design the state such that it is descriptive enough to allow proper action selection from the Mission Planner. For the testing of the proposed solution, current position, flying state, and received permission are the fundamental information necessary to properly select a skill among the previously defined actions. Additionally, the current flight mode of the PX4 Autopilot is included in the state definition to monitor whether the selected skill is currently being executed. This attribute of the state is particularly used in the invariants definition. The definition of a goal consists of associating an additional flag to a state variable. While this last defines the objective, the additional attributes memorize whether the goal is successful, failed, or is still waiting to be attempted.

The next step is to define the action entity. First, it stores the state it is programmed to reach to facilitate future replanning. This state is labelled as the "goal state" of the skill. Second, the action defines two sets of parameters to memorize both the nominal action and the transition action associated with the planned step. Each set defines the information necessary to send a proper action request to the Skill Manager, along with two state definitions. These are used to define the preconditions and the invariants associated with the skill. Each set also stores an action type, selected among the ones previously described in section 4.3.2. While the type stored in the set associated with the nominal action always matches the type of the action object itself, the type saved in the transition action set may be different. Every action provides three fundamental functions. The first is called the test function, and it is used to assert whether the current action is able to reach a given state. For example, if the input state specifies that the drone is flying, such a state can't be reached using a land action, while a takeoff or a position-control skill represents a feasible action. The second method is "propagate backwards" and takes as input two states: the current skill goal and the starting state of the system. Given those inputs, the function computes the closest possible state to the starting one which respects the preconditions and execution of the skill. For instance, for a land action, the backwards-propagated state is equal to the goal state, but the flying state switches from "on ground" to "in air" and the altitude is raised by the default takeoff differential altitude. Instead, for a "position control" skill, the flying state and the altitude are the same as the goal state while the X, Y coordinates match the starting point. The last function, addressed as "set parameter", sets the parameters of the skill to reach the given goal. Internally, it first checks if the selected skill is feasible to reach the objective state. Then, given the goal, generates the correct parameters both for the nominal action and the transition action.

The last entity necessary to implement the planner is the plan. The plan object stores two sequences, one associated with the nominal plan and the other with the communication plan. The items sharing the same position in the two different sequences define a plan step. For this reason, the two queues must share the same length. Moreover, the plan object links to the goal it tries to accomplish, facilitating the objective progression monitoring. If the plan is emptied during execution, the goal is marked as successful. Instead, if the planner fails to back-propagate the goal to

the starting state, the plan is deleted and the objective is set as failed. The goals are loaded at the startup of the program from a configuration file and saved in the Mission Planner. Regarding the plan, only the one currently under execution is stored.



(a) First step. **(b)** Second step. **(c)** Third step.

Figure 4.4: Example of a possible scenario where is necessary to plan several auxiliary actions for a position-control action taking the drone from point BPS to point CGS. The red crosses denote the intersection point of the drone trajectory with the border of the areas.

When the system starts, the Mission Planner loads the configuration file where the desired goals of the mission are defined. Moreover, this file contains information about special areas accessible only after requesting and receiving specific permission. When the MCM send a request to the Mission Planner to create a plan, it communicates the starting state of the system. Consequently, the Mission Planner first asserts if the memorized plan is empty. If the check is confirmed, it fetches the first goal in the memorized queue, marking it as the current goal state (CGS). Otherwise, if the plan is not empty, the planner interrogates the next step to be executed. This represents the failed step which needs to be replanned. Thus, the Mission Planner extracts the goal state of the associated action, marks it as the CGS, and deletes the step. After the initialization of this state, the planner iterates over all available actions, checking which of them are feasible using their test function. All feasible actions are then back-propagated using the proper function and the resulting states are compared using a "matching score". This score assesses how close are the starting state of the system and the given one. The action whose back-propagated state (BPS) scores higher is the one selected as the next step of the plan and the corresponding parameters are generated using the "set parameter" function.

Afterwards, the Mission Planner assigns the permission to the two states. This is done by checking if the position specified in them lies inside a special area, adding the corresponding permission to the state. This procedure is skipped for the BPS if it matches the starting state. In case, after this step, BPS and CGS differ in assigned permissions, the routine to plan the auxiliary action starts. Notably, this routine is

action-dependent and it is planned to be integrated into the action object in future developments. For land, takeoff and stop actions, the associated auxiliary action is a stop action during which the missing permissions are requested. Instead, the algorithm used for position-control skills takes into consideration the single intersections with the border of the areas, as shown in figure 4.4. After all crossing points are computed, the one closest to CGS is selected; in the example is the point "I2". Then, the planner adds a position-control action with I2 as the goal, and a communication request for the area whose border is crossed; in the example area "B". Lastly, the Mission Planner marks point I2 as the CGS and iterates the procedure. The algorithm ends when the path going from BPS to CGS does not cross any area's border. Notably, in the implementation of this algorithm, it is not necessary to update the preconditions of the already planned actions. Indeed, the precondition of the position-control skill does not evaluate the position attribute of the state. This is because the position-control action is agnostic of the starting point and needs only the goal point to be executed.

After the necessary communication steps are planned, the Mission Planner marks the BPS as the new CGS and proceeds to plan the next step for the current goal. The planning phase is completed successfully when the BPS matches the starting state. Instead, the goal is marked as failed if, when asserting which action can reach the CGS, none is found. In this case, the Mission Planner proceeds to select the next memorized goal. The mission terminates once all goals have been processed.

4.5 Mission and Contingency Manager

The mission manager is composed of several building blocks: the skill nodes, responsible for sending requests to the Skill Manager, the monitor node, the nodes checking and resetting the flags, the executor node and the planner node.

The skill nodes all access the same blackboard entry to retrieve the parameters used to build the request for the Skill Manager. This blackboard variable is sufficient to define any of the previously defined skills. It contains one integer to represent the mode of the skill, two sets of three floating-point values to define general parameters, and one more floating-point variable to indicate a timeout. Every different skill node is associated with the corresponding action server provided by the Skill Manager. When first ticked, the node fetches the parameters from the blackboard, composes the request for the server, and sends it. On subsequent ticks, it receives the acknowledgement from the server if the request has been accepted and later the feedback from the action execution. Once the server signals the termination of the action, the positive or negative outcome is returned to the tree using the corresponding node status success or failure.

The monitor node subscribes to the introspection topic provided by the Skill Manager. When the node is ticked, it consumes all available callbacks, but it only retains the last received message. This method ensures the lowest possible computational load on the node while providing the system with updated data. However, only the most recent message is considered and processed by the tree node, while any previous messages are overwritten and effectively discarded. The system mitigates this drawback by setting a tick period of the behavior tree low enough to match the publishing rate of the introspection topic. Thus, being the publish rate of the topic

$100Hz$, the upper bound for the tick period of the tree is $10ms$. Precisely, the tick period is used as a "sleeping" period between two different ticks of the tree. Consequently, the resulting tick frequency is diminished, but the provided value serves as a suitable first approximation. After the data has been received, the monitor processes it and raises the appropriate flags. Flags are saved in the blackboard as arrays of boolean variables. Check and reset flag nodes are created specifying which flag they can access. This way, it is possible to modify and expand the raised flags without modifying the ports of the nodes.

The role of the executor nodes is to generate a subtree to run and monitor skills. This process occurs when the node is ticked for the first time, switching to the running state. The executor first reads from the blackboard the specified step to be executed, containing both the skill and the communication definition. From the skill, it extracts the skill identifier, the parameters necessary to compose the request, the preconditions and the invariants. Parameters, preconditions and invariants are separately loaded in specific blackboard entries and the skill name is used to define the subtree. For this last step, the executor uses templates defined in separate XML files, shown in listing 1 and 3. The first is used to define a standalone action, the second outlines the sub-branch used to execute an auxiliary action along with the necessary communications. Notably, the name of each tag is the node name, the attributes define the ports of the node and the assigned value is the corresponding blackboard key between curl brackets. The executor parses the file and substitutes the "ACTION" placeholder with the skill name. If communications are planned, it further modifies the text by adding the necessary requests. To facilitate this process, the placeholder "COMMUNICATIONS" is used. To define single request nodes, the executor parses the layout shown in listing 2 substituting the placeholder "AREA_ID" with the identifier of the permission. In this case, the port reads a default value rather than accessing a blackboard entry, which otherwise should be specified between curl brackets in the XML schema. This allows for the concurrent request of several permissions without creating multiple blackboard entries. The "permissions" attribute is used to save on the blackboard the list of received permissions. Notably, for the testing of the system, there is no expiration rule for the permissions. Thus, once the UAV receives the authorization to enter an area, such a permit is never lost during the mission.

Lastly, the planner node serves as the interface with the Mission Planner. When a planning is triggered by the tree, this node sends a request, containing the current system state, to the Mission Planner. Then, the planner routine is started in a different thread and the planner node monitors its advancement. This method aligns with the general design rule for behavior tree nodes to run computationally expensive algorithms in separate processes in order to preserve the tick frequency of the tree. If a plan is successfully created, the planner node reads the first planned step, loads it on the blackboard and returns with a success state. Instead, it returns failure if no plan has been defined, or running if the planning process is still under execution.

```
<BehaviorTree ID="executor">
  <Sequence>
    <PrecondCheck
      state="{state}"
      precond="{precond}"
    />
    <ReactiveSequence>
      <InvCheck
        state="{state}"
        inv="{inv}"
      />
      <ACTION
        params="{params}"
        state="{state}"
      />
    </ReactiveSequence>
  </Sequence>
</BehaviorTree>
```

Listing 1: Template used by the executor to create the subtree to run and monitor a skill.

```
<RequestPermission
  area_id="AREA_ID"
  permissions="{permissions}"
/>
```

Listing 2: Template used by the executor to create a "request permission" node.

```
<BehaviorTree ID="executor">
  <Parallel name="execution"
    success_count="1"
    failure_count="1"
  >

    <SuccessIsRunning>
      <Sequence>
        <PrecondCheck
          state="{state}"
          precond="{precond}"
        />
        <ReactiveSequence>
          <InvCheck
            state="{state}"
            inv="{inv}"
          />
          <ACTION
            params="{params}"
            state="{state}"
          />
        </ReactiveSequence>
      </Sequence>
    </SuccessIsRunning>

    <Parallel>
      COMMUNICATION
    </Parallel>

  </Parallel>
</BehaviorTree>
```

Listing 3: Template used by the executor to create the subtree to run and monitor an auxiliary skill and perform the necessary communication requests

5 Results

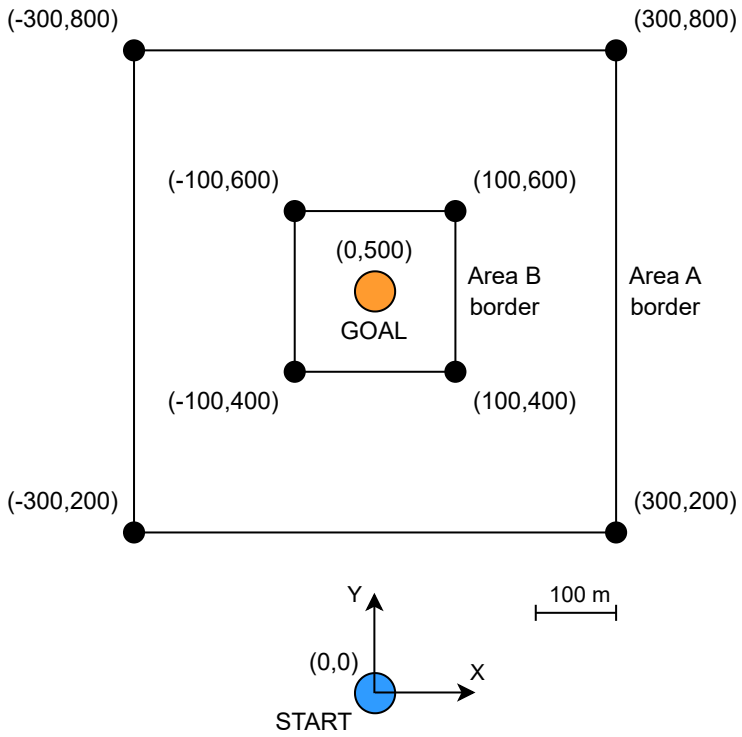


Figure 5.1: Representation of the 2D map used to run the tests.

In this section, the results from the simulated tests are discussed. The simulation environment provided by jMAVSim is extremely simple, consisting of an infinite flat space. To test the basic functionalities of the MCM, the system defines two square areas with restrictive permissions, as shown in figure 5.1. The drone is positioned in coordinates (0,0,0) with respect to the NED reference frame, in meters, by default. The position is marked in the figure with a blue circle named "START". Also, the UAV starts in a disarmed state, i.e. it is on the ground, not flying. The Mission Planner is provided with two goal states. The first is to reach position (0,500,0) in a no-flying state. This corresponds to the "GOAL" point marked in the figure in orange. This means that the UAV needs to take off from the starting position, cross the border of both areas and finally land in the target position. This step is used to test the functionality of the MP to instruct the necessary communications and the executor's ability to assemble the subtree correctly to ask for the necessary permissions. The execution of this goal is also used as a testbed for the replanning capabilities of the MCM, evaluating if the planned steps are correctly generated, loaded in the tree and executed, and to assert the performances of the designed execution-replanning structure. The second goal is the starting state itself. In this setup, the mission represents a delivery to the goal point

and the consequent return-to-home instruction. This step is used to test the execution of multiple objectives and the procedure to correctly terminate the mission.

For the data collection, the ROS log system is used. This is present by default in all skill nodes, which signal the start and the termination, either successful or not, of the action. If necessary, key nodes, e.g. the executor node and the planner node, are equipped with a dummy ROS node to access the logging function it provides. This allows all nodes to use the same interface, providing a common clock to register the timestamp of the messages. Thus, log messages are used as the main source of information to elaborate the performance of the system.

The system on which the simulation is running is a desktop computer equipped with 16 GB of RAM and an Intel i7-6700 CPU working with a clock frequency of 3.40 GHz. The installed OS is Windows 10 Enterprise, and the simulation runs on a Windows Subsystem for Linux (WSL), version 2. The installed Linux distribution is Ubuntu 20.04, the recommended version to run ROS2 Foxy. Every test is executed with three different values for the behavior tree's tick period: 10, 5 and 1 milliseconds. This asserts that the switching mechanic between branches is not influenced by this parameter.

The tests aim to inspect two different aspects. The first is the mission execution speed of the tree, analysed by measuring the interval between the end of the execution of a skill, or its interruption, and the start of the next action. For this purpose, the timestamps published by the executor logs are used. The second aspect is the reactivity of the system in response to contingency situations. It is measured as the delay between the signalling of the contingency case and the start of the associated response action.

5.1 Mission execution performances

In order to test the switch between the execution and planning branches, the request from the UAV to enter a specific area is either denied or not answered. This way, the "request permission" node fails and triggers the replanning routine. Notably, during the simulation events are triggered randomly, thus they differ between different runs. This experiment wants to examine if the proposed structure works as expected and what is its impact on the performances of the tree. Samples register the time delay between the success or halt of one executor and the start of the next one.

Data is sorted into three different classes:

- Nominal-to-Nominal transitions (N2N): This represents the transition interval between the completion of a step and the start of the next one. It measures the performance of the adopted design choice of executing single actions instead of creating a unique tree for the whole plan.

- Nominal-to-Trans transitions (N2T): This value measures the delay between the nominal action failure and the start of the transition action. The interval represents the reactivity of the system to switch from the execution branch to the planning branch.

Transition type	reported value	tick period		
		10 ms	5 ms	1 ms
N2N	max [s]	0,0823	0,0878	0,0824
	min [s]	0,0493	0,0472	0,0435
	avg [s]	0,0648	0,0693	0,0625
	std dev [s]	0,0114	0,0121	0,0119
N2T	max [s]	0,0557	0,0680	0,0682
	min [s]	0,0204	0,0302	0,0249
	avg [s]	0,0422	0,0506	0,0523
	std dev [s]	0,0080	0,0079	0,0108
T2N	max [s]	0,1208	0,1103	0,1043
	min [s]	0,0317	0,0359	0,0316
	avg [s]	0,0714	0,0791	0,0797
	std dev [s]	0,0258	0,0185	0,0232

Table 5.1: Elaborated data reporting the time duration of the transitions between execution and planning branch, or between two consecutive steps of the plan.

- Trans-to-Nominal (T2N): The final case occurs when a plan has been successfully generated and its execution starts. This value is also related to the same performance inspected in the first case. Indeed, the generation of a subtree for single planned steps is expected to result faster than the creation time of a whole tree. Even if the latter case is not tested, the computed interval is considered a key performance indicator (KPI) of the system.

For each class of data, the number of collected samples is between fifteen and thirty, depending on the frequency of occurrence of each transition. For example, the N2N transition is the one occurring less frequently, given that the generated plan consists of a limited number of steps. Table 5.1 reports the maximum value (max), minimum value (min), average (avg) and standard deviation (std dev) for each class of samples. All values are reported in seconds.

Firstly, the system behaves as expected. It correctly switches between planning and execution behaviors, it is capable of processing multiple goals and terminating the mission properly. As expected, the average values across all types of transitions reveal that a higher tick rate does not lead to reduced transition times. Indeed, transitions from the execution branch to the planning branch and vice versa occur in a single tick loop, without any state being completely propagated upward to the root node.

However, in order to properly analyze these data, an important aspect to consider is

Transition type	reported value	tick period		
		10 ms	5 ms	1 ms
P2P	max [s]	0,1208	0,1103	0,1036
	min [s]	0,0395	0,0399	0,0481
	avg [s]	0,0658	0,0697	0,0724
	std dev [s]	0,0219	0,0195	0,0187
P2O	max [s]	0,0812	0,0878	0,0824
	min [s]	0,0317	0,0472	0,0435
	avg [s]	0,0574	0,0735	0,0611
	std dev [s]	0,0175	0,0165	0,0133
O2P	max [s]	0,0545	0,0699	0,0650
	min [s]	0,0493	0,0489	0,0498
	avg [s]	0,0529	0,0555	0,0560
	std dev [s]	0,0021	0,0085	0,0050
O2O	max [s]	0,0504	0,0560	0,0630
	min [s]	0,0204	0,0302	0,0249
	avg [s]	0,0359	0,0430	0,0377
	std dev [s]	0,0083	0,0090	0,0109

Table 5.2: Elaborated data reporting the time duration of the transitions considering the type of actions.

the type of actions involved in the transition. Indeed, in a second analysis, the data has been parsed differently by focusing on the type of actions composing the transition. The intuition is that the skill type was also impacting the transition delay. Particularly, given the mission setup, a "position-control" action results more complex than the others, because it is the only skill during which permission requests are sent. For this reason, data are clustered as follows: transition from a "position-control" action to another "position-control" action (P2P), transition from a "position-control" skill to a different one (P2O), identified as "other" action, transition from an "other" action to a "position-control" action (O2P) and transition from an "other" action to a another action from the same family (O2O). Results are reported similarly as before in table 5.2.

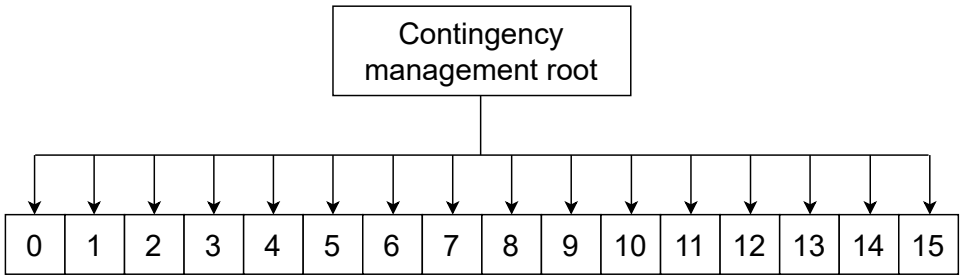
The data indicates that transitions involving "position-control" skills generally exhibit longer duration compared to other types. This extended duration can be

attributed to two main factors: the complexity of the subtree associated with steps requiring communication, and the time-consuming nature of the "halting" procedure. The latter involves recursively calling a function to interrupt all nodes in a branch and is particularly impactful when a transition occurs, interrupting the existing branch. Additionally, the creation of a more complex subtree itself takes longer, contributing to the overall duration. A review of table 5.1 supports this observation, revealing that "T2N" transitions have longer durations than "N2T" transitions. This is likely because the planning branch, consisting of five nodes, is more complex than the execution branch, which has only three nodes. Given that these branches are static, their inherent complexity becomes more significant when a transition triggers the "halting" procedure. Moreover, transitions from planning to execution are more time-consuming than the reverse because only during the execution phase communication nodes are instantiated. This process includes the construction of the associated ROS nodes, which by default assert if the associated ROS server is available. This process results in extra waiting time for the response from the server, increasing the duration of the transition.

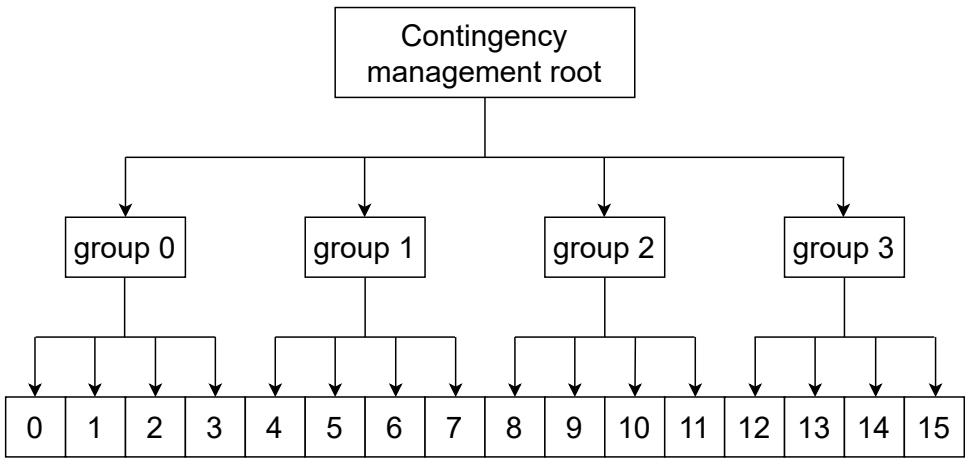
In conclusion, this analysis offers a detailed examination of transition performance metrics within the UAV system, categorizing them into distinct types and scrutinizing the influence of action complexity. The empirical data suggests that the nature of the action involved in the transition, specifically "position-control" actions, exerts a notable impact on transition duration. This is ascribed to the complexity of the corresponding subtree and the requisite "halting" procedures. These insights contribute to a more refined understanding of system performance, underscoring the necessity for targeted optimization strategies, particularly in contexts necessitating swift transitions between planning and execution branches. From these findings, it can also be predicted the performances achieved by adopting the opposite design choice, hence building a behaviour tree representing the entire plan. In this case, the resulting complexity would be several orders higher than the subtree representing a single action. The positive result would be improving considerably the execution of the plan under nominal conditions. Indeed, the system would not need to retrieve the next action and generate a new subtree to execute it. The negative outcome resides in the impact on the transition time between planning and execution, where the halting and creation processes would need to manage a considerably bigger tree. When selecting the design choice for a specific case, two main aspects must be taken into consideration. First is how often could a replanning phase be triggered, a situation that depends on the type of mission and the environment in which the operations are conducted. Second is in which case higher performances are preferred, and the desired trade-off between the two cases. Addressing the problem of system safety, it is advisable to focus on a higher reactivity in case an action fails, rather than obtaining a faster execution of the mission. An example is the case where a path-following skill deviates from the given path over a desired threshold. In this case, the failure of the action may also result in a hazardous state of the system, and the switch to the transition action should be prioritized. An important aspect to consider is that the halting procedure occurs also when the system needs to switch to the contingency branch, hence adopting a complex structure for the

plan execution would also deteriorate the reactivity performances of the contingency management system. Under these circumstances, it is preferable to adopt the design proposed in this study.

5.2 Contingency management performances



(a) Flag structure used in the first test case.



(b) Flag structure used in the second test case.

Figure 5.2: Flag structures used in the two different test cases used to analyse the performances of the contingency management branch.

This test evaluates the reactivity performances of the proposed behaviour tree design. In particular, it serves as a KPI to assert how the behaviour tree performs with respect to FSMs. Even if data regarding the performances of the counterpart are not available, this test wants to analyze the reactive time concerning other system characteristics, such as the number of possible single contingency cases. Moreover, this test wants to explore possible strategies to achieve a higher reactivity of the system, e.g. by segmenting the contingency actions into subgroups.

To address these questions, two cases are explored. In both scenarios, sixteen different flags are used, numbered from zero to fifteen to represent their position in the array where they are stored in the blackboard. In the first case, all flags are positioned at the same level below the main contingency management node. In the second case, the sixteen flags are partitioned into four groups, numbered from zero to three representing the array position of the corresponding auxiliary flag memorized in the blackboard. The two hierarchies are reported in figure 5.2.

To inject errors inside the tree, a dedicated ROS node has been created. It provides a service to prompt the flags to rise and publish them on a topic to which the monitor node subscribes. The flags are up only during one publication, after which they are immediately reset. This is sufficient given that the monitor processes the message from the topic and updates the flag array stored in the blackboard, but never reset the flags afterwards. The objectives of the mission are not changed, but the presence of the areas is ignored for easier testing. The action associated with each flag is a stop skill. The time interval used to represent the reactivity of the tree starts when the monitor processes the message containing the flags up and finishes when the contingency action is triggered. The results of the tests are reported in figures 5.3, 5.4 and 5.5.

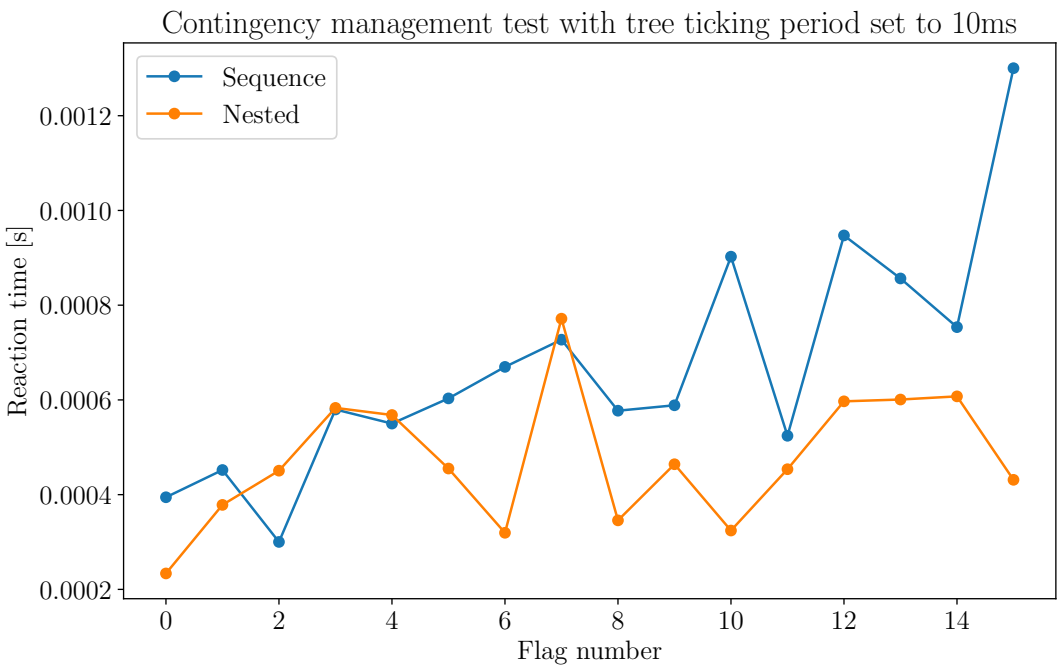


Figure 5.3: Test results of the reactivity of the system to contingent events, using a tick period of the tree equal to $10ms$.

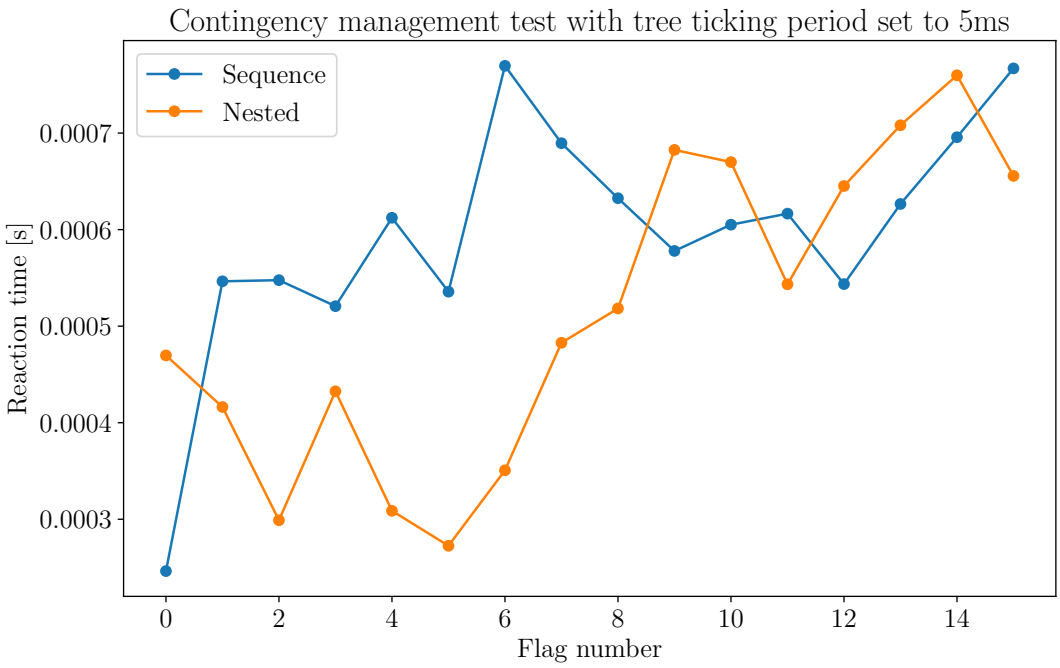


Figure 5.4: Test results of the reactivity of the system to contingent events, using a tick period of the tree equal to $5ms$.

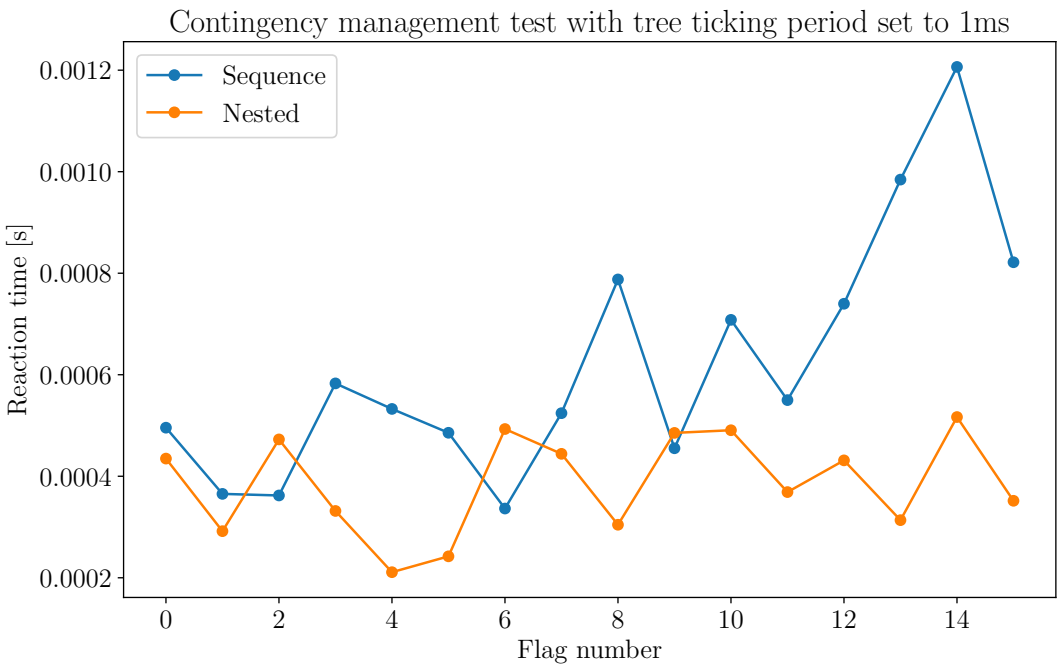


Figure 5.5: Test results of the reactivity of the system to contingent events, using a tick period of the tree equal to $1ms$.

Tick period [ms]	Average reactivity [s]	
	sequence	nested
1	0.00062	0.00039
5	0.00060	0.00051
10	0.00067	0.00051

Table 5.3: Average time of reaction of the system to contingent events using sixteen different triggers.

During these tests, the architecture correctly executed the contingency actions, regenerated a plan starting from the new resulting state, and proceeded with fulfilling the given goals by carrying out the planned steps. As expected, the reactivity of the system is not impacted by the tick period of the tree, as the average reactivity time, reported in table 5.3, does not present any relation with the used tick period. As expected, fallback and sequence nodes, governing the switching between different branches, do not back-propagate the returned state of the children at each tick if it results in a terminal state. Instead, they proceed with the execution of their tick rule. This means that the tick period applied to the tree influences only the waiting time triggered when the root node is fed back with a running state. Thus, the tick period influences the frequency at which the monitor node is executed, hence the refresh rate of the flags triggering the contingency action. However, the "BehaviorTree.CPP" library provides "wakeup" methods to initiate the ticking process on the occurrence of specific events, representing an efficient solution to mitigate this problem. Regarding the two different proposed structures to manage the triggering of contingency actions, it can be observed from table 5.3 that the nested architecture shows always an average response time lower than having all flag checks at the same level. Hence, proper structuring of the contingency management branch can lead to better performance of the system.

In conclusion, the observed delay in the response of the system is due to the execution of the internal mechanism of the control nodes. The average response time is always lower than one millisecond, hence the system presents high reactivity to hazardous situations. The maximum registered delay in the response is $0.0013s$. The value is probably not comparable to the capabilities of FSMs, where the interval between the switching from one state to another is expected to be a low constant, comparable to a function call operation. However, if we consider the performances of other systems operating onboard the UAV, one milliseconds represent an acceptable value. For example, flight controllers provided by PX4 have a nominal operating frequency of 50 Hz, which can be raised to a maximum of 200 Hz. Finally, it must be considered that presented data are collected from a simulation environment, thus reaction intervals may vary depending on the machine running the simulation and may differ during an actual flight test.

6 Conclusions

This thesis is centred on the development and evaluation of a novel onboard mission and contingency management system for Unmanned Aerial Vehicles, with a particular emphasis on the use of behavior trees. The objective is twofold. First, the thesis aims to test the feasibility of behavior trees for UAV control by creating a robust, modular, and autonomous system capable of effectively navigating complex and dynamic environments, such as those encountered in off-shore wind farm maintenance. Second, it seeks to assess the performance of behavior trees in this specific context, offering a quantitative analysis of their efficiency through key performance indicators.

The research addresses several critical questions, including how to design an autonomous system that is both safe and efficient, how to manage contingencies effectively, and how to plan and execute concurrent actions. An additional contribution of this work is its quantitative analysis of the performance of behavior trees, filling a gap in existing academic literature. The evaluation of these KPIs also allows for a comparison of the proposed design, based on behavior trees, with applications adopting a different modelling formalism, e.g. FSMs. This comparative analysis is crucial for substantiating the advantages and potential limitations of using behavior trees for UAV mission and contingency management.

The design choices made in the thesis are carefully aligned with the addressed problems underlined before. The architecture of the proposed system is divided into several components, including a Skill Manager, a Mission planner, and a Mission and Contingency Manager. Each of these components serves a specific purpose and is designed to contribute to the overall robustness and flexibility of the system. The Skill Manager is designed for modularity, allowing for the easy addition or modification of skills. The Mission Planner endows deliberation by planning missions based on real-time data and user inputs. The Mission and Contingency Manager, which is the core of the system, employs behavior trees to manage both regular mission tasks and contingencies. This choice leverages the proven efficiency of behavior trees in managing complex behaviours in a modular and robust manner.

One limitation of the proposed approach resides in the static design of the contingency management system. Even if underlined as an important constraint to ensure high reactivity, it also constitutes a limitation in the design phase. Indeed, it is necessary to cover all faulty situations the UAV can encounter, otherwise the system will face immediate mission failure. This concept goes under the term "brittleness of autonomy", which refers to the limitations or vulnerabilities in an autonomous system that can cause it to fail or behave unpredictably when it encounters situations that it was not specifically designed to handle.

The results of the study offer valuable insights into the efficiency and effectiveness of the proposed system. A quantitative analysis of key performance indicators shows that behavior trees are an efficient design choice for this application. The study regarding mission execution performances led to the conclusion that the complexity of the behavior tree is one of the major factors negatively affecting its effectiveness. Furthermore, by evaluating the responsiveness of the tree in reacting to unforeseen events, this thesis demonstrates that the system can operate safely in complex environments

and manage contingencies effectively. Moreover, the quantitative analysis validates the design choices made, confirming that behavior trees are effective in this context.

In summary, the thesis contributes to the current academic research by not only introducing a novel tree design tailored for UAV mission and contingency management, but also by offering a quantitative analysis of the performance of behavior trees in this specific application. This research holds the potential to impact not just the specialized domain of offshore wind farm maintenance, but also a diverse range of other fields where the deployment of UAVs could offer substantial benefits.

The current work provides a proof-of-concept of the design, thus further testing on a more complex system is necessary. Moreover, it would be interesting to test the modular capabilities of the system, embedding more complex monitoring and planning modules and extending the existing architecture.

References

[1] G. Drozeski, B. Saha, and G. Vachtsevanos, "A Fault Detection and Reconfigurable Control Architecture for Unmanned Aerial Vehicles", in *2005 IEEE Aerospace Conference*, Big Sky, MT, USA: IEEE, 2005, pp. 1–9, ISBN: 978-0-7803-8870-3. DOI: 10.1109/AERO.2005.1559597. (Visited on 04/06/2023).

[2] C. Pippin, "Integrated Hardware/Software Architectures to Enable UAVs for Autonomous Flight", in *Handbook of Unmanned Aerial Vehicles*, K. P. Valavanis and G. J. Vachtsevanos, Eds., Dordrecht: Springer Netherlands, 2015, pp. 1725–1747, ISBN: 978-90-481-9707-1. DOI: 10.1007/978-90-481-9707-1_58. (Visited on 02/16/2021).

[3] "Drones for Forest Fires: In-depth Case Study | FlytNow". Available: https://www.flytbase.com/blog/drones-in-forest-fire-response (visited on 08/13/2023).

[4] Zipline, "Zipline Instant Delivery & Logistics". Available: https://www.flyzipline.com/ (visited on 08/12/2023).

[5] M. A. Skoog, L. R. Hook, and W. Ryan, "Leveraging ASTM Industry Standard F3269-17 for Providing Safe Operations of a Highly Autonomous Aircraft", in *2020 IEEE Aerospace Conference*, Big Sky, MT, USA: IEEE, Mar. 2020, pp. 1–7, ISBN: 978-1-72812-734-7. DOI: 10.1109/AERO47225.2020.9172434. (Visited on 07/10/2023).

[6] "New wind turbine installations in Germany grow almost 50% in early 2023", May 2023. Available: https://www.cleanenergywire.org/news/new-wind-turbine-installations-germany-grow-almost-50-early-2023 (visited on 08/15/2023).

[7] Alexander Donkels, S. Cain, J. Wilhelm, J. Dipperman, J. Janke, and T. Bruns, "An Approach for Integration of Transport Drones into Offshore Wind Farms", in *Deutscher Luft- Und Raumfahrt Kongress (DLRK)*, Stuttgart, Germany, Sep. 2023.

[8] L. Castano and H. Xu, "Safe decision making for risk mitigation of UAS", in *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, Atlanta, GA, USA: IEEE, Jun. 2019, pp. 1326–1335, ISBN: 978-1-72810-333-4. DOI: 10.1109/ICUAS.2019.8797774. (Visited on 04/06/2023).

[9] G. Vachtsevanos, B. Lee, S. Oh, and M. Balchanos, "Resilient Design and Operation of Cyber Physical Systems with Emphasis on Unmanned Autonomous Systems", *Journal of Intelligent & Robotic Systems*, vol. 91, no. 1, pp. 59–83, Jul. 2018, ISSN: 0921-0296, 1573-0409. DOI: 10.1007/s10846-018-0881-x. (Visited on 03/30/2023).

[10] F. Ingrand and M. Ghallab, "Deliberation for autonomous robots: A survey", *Artificial Intelligence*, vol. 247, pp. 10–44, Jun. 2017, ISSN: 00043702. DOI: 10.1016/j.artint.2014.11.003. (Visited on 05/24/2023).

[11] B. Durand, K. Godary-Dejean, L. Lapierre, R. Passama, and D. Crestani, "Fault tolerance enhancement using autonomy adaptation for autonomous mobile robots", in *2010 Conference on Control and Fault-Tolerant Systems (SysTol)*, Nice, France: IEEE, Oct. 2010, pp. 24–29, ISBN: 978-1-4244-8153-8. DOI: 10.1109/SYSTOL.2010.5676030. (Visited on 04/06/2023).

[12] F. Steinmetz and R. Weitschat, "Skill parametrization approaches and skill architecture for human-robot interaction", in *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, Fort Worth, TX, USA: IEEE, Aug. 2016, pp. 280–285, ISBN: 978-1-5090-2409-4. DOI: 10.1109/COASE.2016.7743419. (Visited on 05/24/2023).

[13] J. Guiochet, M. Machin, and H. Waeselynck, "Safety-critical advanced robots: A survey", *Robotics and Autonomous Systems*, vol. 94, pp. 43–52, Aug. 2017, ISSN: 09218890. DOI: 10.1016/j.robot.2017.04.004. (Visited on 04/06/2023).

[14] A. Leite, A. Pinto, and A. Matos, "A Safety Monitoring Model for a Faulty Mobile Robot", *Robotics*, vol. 7, no. 3, p. 32, Jun. 2018, ISSN: 2218-6581. DOI: 10.3390/robotics7030032. (Visited on 05/24/2023).

[15] C. Gao, Y. Zhai, B. Wang, and B. M. Chen, "Synthesis and Online Re-planning Framework for Time-Constrained Behavior Tree", in *2021 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, Sanya, China: IEEE, Dec. 2021, pp. 1896–1901, ISBN: 978-1-66540-535-5. DOI: 10.1109/ROBIO54168.2021.9739542. (Visited on 04/14/2023).

[16] M. Colledanchise and P. Ögren, *Behavior Trees in Robotics and AI: An Introduction*. Jul. 2018. DOI: 10.1201/9780429489105. arXiv: 1709.00084 [cs] (visited on 04/06/2023).

[17] R. d. P. Pereira and P. M. Engel, "A Framework for Constrained and Adaptive Behavior-Based Agents", 2015. DOI: 10.48550/ARXIV.1506.02312. (Visited on 04/14/2023).

[18] M. Colledanchise and P. Ogren, "How Behavior Trees modularize robustness and safety in hybrid systems", in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Chicago, IL, USA: IEEE, Sep. 2014, pp. 1482–1488, ISBN: 978-1-4799-6934-0. DOI: 10.1109/IROS.2014.6942752. (Visited on 04/05/2023).

[19] R. Ghzouli, T. Berger, E. B. Johnsen, S. Dragule, and A. Wąsowski, "Behavior Trees in Action: A Study of Robotics Applications", 2020. DOI: 10.48550/ARXIV.2010.06256. (Visited on 07/11/2023).

[20] J. A. Segura-Muros and J. Fernandez-Olivares, "Integration of an Automated Hierarchical Task Planner in ROS Using Behaviour Trees", in *2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, Alcala de Henares: IEEE, Sep. 2017, pp. 20–25, ISBN: 978-1-5386-3462-2. DOI: 10.1109/SMC-IT.2017.11. (Visited on 05/24/2023).

[21] M. Iovino, E. Scukins, J. Styrud, P. Ögren, and C. Smith, *A Survey of Behavior Trees in Robotics and AI*, May 2020. arXiv: 2005.05842 [cs] (visited on 03/30/2023).

[22] D. Stojcsics, D. Boursinos, N. Mahadevan, X. Koutsoukos, and G. Karsai, "Fault-Adaptive Autonomy in Systems with Learning-Enabled Components", *Sensors*, vol. 21, no. 18, p. 6089, Sep. 2021, ISSN: 1424-8220. DOI: 10.3390/s21186089. (Visited on 04/11/2023).

[23] C. Archibald and E. Petriu, "Model for skills-oriented robot programming (SKORP)", in *Optical Engineering and Photonics in Aerospace Sensing*, K. L. Boyer and L. Stark, Eds., Orlando, FL, Mar. 1993, pp. 392–402. DOI: 10.1117/12.141787. (Visited on 05/24/2023).

[24] S. Bøgh, O. S. Nielsen, M. R. Pedersen, V. Krüger, and O. Madsen, "Does your robot have skills?", in *Proceedings of the 43rd International Symposium on Robotics*, VDE Verlag GMBH, 2012.

[25] C. Lesire, D. Doose, and C. Grand, "Formalization of Robot Skills with Descriptive and Operational Models", in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Las Vegas, NV, USA: IEEE, Oct. 2020, pp. 7227–7232, ISBN: 978-1-72816-212-6. DOI: 10.1109/IROS45743.2020.9340698. (Visited on 07/11/2023).

[26] A. Albore, D. Doose, C. Grand, J. Guiochet, C. Lesire, and A. Manecy, "Skill-based design of dependable robotic architectures", *Robotics and Autonomous Systems*, vol. 160, p. 104 318, Feb. 2023, ISSN: 09218890. DOI: 10.1016/j.robot.2022.104318. (Visited on 05/24/2023).

[27] M. Crosby, F. Rovida, M. Pedersen, R. Petrick, and V. Krüger, "Planning for robots with skills", in *Planning and Robotics (PlanRob) Workshop at the International Conference on Automated Planning and Scheduling (ICAPS)*, 2016.

[28] Y. Pane, V. Mokhtari, E. Aertbelien, J. De Schutter, and W. Decre, "Autonomous Runtime Composition of Sensor-Based Skills Using Concurrent Task Planning", *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 6481–6488, Oct. 2021, ISSN: 2377-3766, 2377-3774. DOI: 10.1109/LRA.2021.3094498. (Visited on 07/11/2023).

[29] "BehaviorTree/BehaviorTree.CPP: Behavior Trees Library in C++. Batteries included." Available: https://github.com/BehaviorTree/BehaviorTree.CPP (visited on 08/27/2023).

[30] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ogren, "Towards a unified behavior trees framework for robot control", in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, Hong Kong, China: IEEE, May 2014, pp. 5420–5427, ISBN: 978-1-4799-3685-4. DOI: 10.1109/ICRA.2014.6907656. (Visited on 04/11/2023).

[31] L. Grunske, P. Lindsay, N. Yatapanage, and K. Winter, "An Automated Failure Mode and Effect Analysis Based on High-Level Design Specification with Behavior Trees", in *Integrated Formal Methods*, D. Hutchison *et al.*, Eds., vol. 3771, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 129–149, ISBN: 978-3-540-32240-5. DOI: 10.1007/11589976_9. (Visited on 04/06/2023).

[32] P. A. Lindsay, K. Winter, and N. Yatapanage, "Safety Assessment Using Behavior Trees and Model Checking", in *2010 8th IEEE International Conference on Software Engineering and Formal Methods*, Pisa, Italy: IEEE, Sep. 2010, pp. 181–190, ISBN: 978-1-4244-8289-4. DOI: 10.1109/SEFM.2010.23. (Visited on 04/06/2023).

[33] M. Colledanchise and L. Natale, "Handling Concurrency in Behavior Trees", *IEEE Transactions on Robotics*, vol. 38, no. 4, pp. 2557–2576, Aug. 2022, ISSN: 1552-3098, 1941-0468. DOI: 10.1109/TRO.2021.3125863. (Visited on 04/11/2023).

[34] R. Colvin, L. Grunske, and K. Winter, "Timed Behavior Trees for Failure Mode and Effects Analysis of time-critical systems", *Journal of Systems and Software*, vol. 81, no. 12, pp. 2163–2182, Dec. 2008, ISSN: 01641212. DOI: 10.1016/j.jss.2008.04.035. (Visited on 04/11/2023).

[35] G. Behrmann, A. David, and K. G. Larsen, *A Tutorial on Uppaal*, M. Bernardo and F. Corradini, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 200–236, ISBN: 978-3-540-30080-9. DOI: 10.1007/978-3-540-30080-9_7. Available: https://doi.org/10.1007/978-3-540-30080-9_7.

[36] H. Zhou, H. Min, and Y. Lin, "An Autonomous Task Algorithm Based on Behavior Trees for Robot", in *2019 2nd China Symposium on Cognitive Computing and Hybrid Intelligence (CCHI)*, Xi'an, China: IEEE, Sep. 2019, pp. 64–70, ISBN: 978-1-72814-091-9. DOI: 10.1109/CCHI.2019.8901959. (Visited on 05/24/2023).

[37] A. Klöckner, "Interfacing Behavior Trees with the World Using Description Logic", in *AIAA Guidance, Navigation, and Control (GNC) Conference*, Boston, MA: American Institute of Aeronautics and Astronautics, Aug. 2013, ISBN: 978-1-62410-224-0. DOI: 10.2514/6.2013-4636. (Visited on 05/24/2023).

[38] S. Zaman, G. Steinbauer, J. Maurer, P. Lepej, and S. Uran, "An integrated model-based diagnosis and repair architecture for ROS-based robot systems", in *2013 IEEE International Conference on Robotics and Automation*, Karlsruhe, Germany: IEEE, May 2013, pp. 482–489, ISBN: 978-1-4673-5643-5. DOI: 10.1109/ICRA.2013.6630618. (Visited on 04/06/2023).

[39] S. S. Osder, "Reconfigurable helicopter flight control system", 5 678 786, Oct. 1997.

[40] "Open Source Autopilot for Drones". Available: https://px4.io/ (visited on 08/27/2023).

[41] F. Junger, S. Schopferer, S. Benders, and J. C. Dauer, "Talking to Autonomous Drones: Command and Control Based on Hierarchical Task Decomposition", in *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, Athens, Greece: IEEE, Jun. 2021, pp. 968–977, ISBN: 978-1-66541-535-4. DOI: 10.1109/ICUAS51884.2021.9476753. (Visited on 08/31/2023).

[42] A. Albore, D. Doose, C. Grand, C. Lesire, and A. Manecy, "Skill-Based Architecture Development for Online Mission Reconfiguration and Failure Management", in *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*, Madrid, Spain: IEEE, Jun. 2021, pp. 47–54, ISBN: 978-1-66544-474-3. DOI: 10.1109/RoSE52553.2021.00015. (Visited on 05/24/2023).

[43] D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987, ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(87)90035-9. Available: https://www.sciencedirect.com/science/article/pii/0167642387900359.

[44] ROS, "ROS: Home". Available: https://www.ros.org/ (visited on 08/27/2023).

[45] "Nav2 — Navigation 2 1.0.0 documentation". Available: https://navigation.ros.org/ (visited on 08/27/2023).

[46] "Splintered-reality/py_trees: Python implementation of behaviour trees." Available: https://github.com/splintered-reality/py_trees (visited on 08/27/2023).

[47] "Behavior3". Available: https://github.com/behavior3 (visited on 08/27/2023).

[48] "ToyotaResearchInstitute/task_behavior_engine: A behavior tree based task engine written in Python". Available: https://github.com/ToyotaResearchInstitute/task_behavior_engine (visited on 08/27/2023).

[49] "Futureneer/beetree: A lightweight implementation of Behavior Trees in Python that can interface with the ROS framework." Available: https://github.com/futureneer/beetree (visited on 08/27/2023).

[50] "Behavior-tree - crates.io: Rust Package Registry", Oct. 2021. Available: https://crates.io/crates/behavior-tree (visited on 08/27/2023).

[51] H. Lane, "IEEE REFERENCE GUIDE", Available: https://ieeeauthorcenter.ieee.org/wp-content/uploads/IEEE-Reference-Guide.pdf (visited on 08/12/2023).

[52] L. R.J. Costa, *Aalto Thesis Template*, Feb. 2023. Available: https://wiki.aalto.fi/display/Aaltothesis/Aalto+Thesis+LaTeX+Template (visited on 08/12/2023).