TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fakultät für Informatik

CSR-26-07

Implementation and Enhancement of a Mission Scheduling Component for Autonomous UAV

Monica Chambial · Batbayar Battseren · Wolfram Hardt

Mai 2026

Chemnitzer Informatik-Berichte

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Implementation and Enhancement of a Mission Scheduling Component for Autonomous UAV

Master Thesis

Submitted in Fulfilment of the
Requirements for the Academic Degree
M.Sc.

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Monica Chambial
Student ID: 481106
Date: 12.02.2026

Supervising tutor: Prof. Dr. Dr. h. c. Wolfram Hardt
Dr. Batbayar Battseren

Acknowledgement

I would like to express my sincere gratitude to everyone who supported and guided me throughout the journey of completing this thesis. This work would not have been possible without the contributions, encouragement, and patience of many individuals.

First and foremost, I would like to thank my thesis supervisor, Dr. Batbayar Battseren, for his continuous guidance, constructive feedback, and unwavering support throughout the course of this work. His technical expertise, insightful suggestions, and calm encouragement were invaluable in shaping both the direction and quality of this thesis. His patience and willingness to discuss ideas in depth greatly contributed to overcoming challenges encountered during the research and implementation phases.

I am also deeply indebted to my academic supervisor, Prof. Dr. Dr. h. c. Wolfram Hardt, for his guidance and academic oversight. His broad expertise, critical perspective, and emphasis on systematic research provided essential direction and ensured the academic rigor of this work. His support created a solid framework within which this thesis could be developed successfully.

I would further like to acknowledge the academic environment at Technische Universität Chemnitz, which provided the necessary resources and an intellectually stimulating atmosphere for conducting this research. The discussions, feedback, and exposure to advanced topics in autonomous systems and embedded software architectures significantly enriched my learning experience.

Finally, I would like to express my heartfelt gratitude to my parents, brother, and my partner for their constant belief in me and their unwavering encouragement. Their emotional support, patience, and understanding during times of uncertainty and intense workload gave me the strength and motivation to persevere. This achievement would not have been possible without their continuous support.

Abstract

Autonomous Unmanned Aerial Vehicles (UAVs) increasingly use onboard companion computers to handle mission logic, perception tasks, and system management. These systems are often designed as microservice-oriented architectures made up of several interacting components. While this design improves modularity, it also creates challenges in system startup and supervision on resource limited embedded platforms.

This thesis presents the design and implementation of a mission based initialization system for autonomous UAVs. The proposed Mission Scheduling Component (SCH) is developed for micro-services based architectures. It interprets mission descriptions written in Behavior Tree based XML format and maps mission tasks to the required software components through a static mapping process. The scheduler controls component startup, monitors runtime health using heartbeat signals. And performs component restart or safe shutdowns when needed. The system is implemented as a lightweight C++ daemon running on a Linux based embedded platform (Jetson Nano). Experimental results show that the scheduler performs correct mission-dependent component initialization and reliable fault supervision with minimal runtime cost. These findings indicate that mission driven system initialization can improve autonomy and reliability in embedded UAV operations.
Keywords: Software component configuration, Behaviour Tree, Mission representation languages, System initialization

Contents

List of Figures

List of Tables

List of Abbreviations

UAVs Unmanned Aerial Vehicles

AI Artificial Intelligence

RPC Remote Procedure Calls

KSAR Knowledge Source Activation Record

VM Virtuel Machine

FSMs Finite State Machines

FSA Finite State Automaton

BTs Behavior Trees

PDDL Planning Domain Definition Language

HTN Hierarchical Task Networks

IPC Inter Process Communication

TML Task-based Mission specification Language

LTL Linear Temporal Logic

BDL Block Definition Language

DSL Domain Specific Languages

DMDE Dynamic Mission Definition and Execution framework

TDL Task Definition Language

SCH Mission Scheduling Component

BLB Blackboard

XML Extensible Markup Language

JSON JavaScript Object Notation

SWCs Software Components

ABI Application Binary Interface

UI User Interface

1 Introduction

1.1 Background

Today unmanned aerial vehicles (UAVs) are not single machines but complex systems composed of airframes, propulsion units, sensors, communication links, power supplies, control software and human operators. Early practitioners learned that developing these elements in isolation leads to systems that do not work well together. This insight launched a systems engineering approach in which designers think of the UAV as a "system of systems," integrating mission requirements with subsystems such as flight controls, communications and payloads [1, 2]. From the viewpoint of the UAV system itself, understanding its evolution means tracing how its hardware and software components have grown more sophisticated and integrated over time. The sections that follow describe these stages and highlight how they shape today's UAV architectures.

1.1.1 Historical UAV system components

Early unmanned aerial vehicles, known as drones, were created mainly to replace balloons, which were the primary method of aerial observation over the trenches during the war. UAVs conducted surveillance tasks without putting soldiers at risk on the battlefield [3]. These early UAVs were mainly used for reconnaissance, jamming and decoy operations. These early UAV systems had a few components, as shown in Fig. 1.1. Initially, radio signals were used from the line of sight to communicate with the UAVs. Later, it became possible to control the unmanned aircraft with wireless telemetry components after the data link and sensors were introduced in UAV systems. However, the UAVs were still limited to short range communication. This led to satellite based UAV communication systems. With the increasing demand of missions more electronic units and sensors were deployed into the UAVs payloads. Development in the UAV sector stalled due to rising costs associated with sensors and electronic units used in designing UAVs [4, 3].

Planning and Control: Software Side As discussed in the section 2.5 of [1], defining missions for UAVs can be even trickier than classifying the vehicles themselves. Early literature grouped missions into reconnaissance, target acquisition and designation, electronic warfare, communications relay and weapon delivery. But later practitioners realized that missions overlap and evolve. For example, a reconnaissance platform may also carry a laser designator or communications relay. The

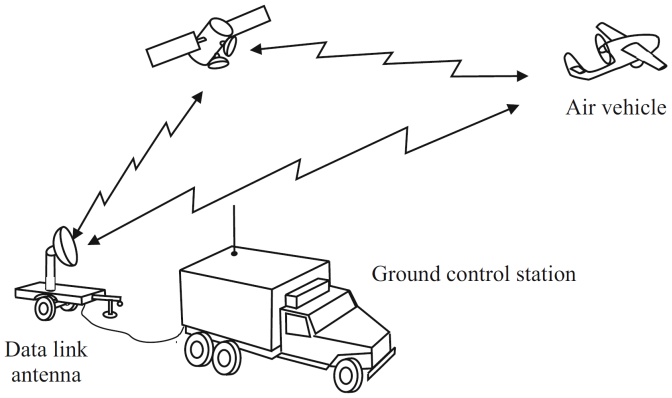


Figure 1.1: Early UAV Basic System Components [1]

difficulty of mission classification underscores that UAVs are ultimately tools serving broader operational goals. And the system must be flexible enough to adapt payload and software to the mission at hand.

Mission planning and operational control involve defining flight paths, waypoints, sensor tasks and contingency procedures. Modern UAVs can execute preprogrammed routes using GPS and inertial navigation systems, but human operators remain in the loop to handle unexpected events. And this control ranges from manual using joysticks or trackballs, to fully autonomous with intermediate modes that allow operators to intervene while autopilots maintain stability, as explained in the section 9.2 of [1]. Ground station software architectures, operator training workload, and human machine interfaces play critical roles. A UAV's effectiveness depends on how its software and interface support the operator's tasks.

1.1.2 Affects of AI in UAVs

Early UAV systems relied almost exclusively on tightly coupled flight control loops implemented on micro-controller based flight controllers. While sufficient for basic navigation and control, such architectures limited computational headroom and were not designed to support data intensive or adaptive decision making tasks. As UAV missions rely on perception driven intelligence, including object or anomaly detection, tracking, mapping, and adaptive path planning. These tasks typically involve processing high bandwidth sensor data and machine learning algorithms. Such workloads require heterogeneous computing resources (CPU, GPU, accelerators) and significantly higher memory bandwidth than those available on conventional flight controllers. These are explained elaborately in sections II–V of [6, 7].

Multiple studies emphasize that onboard AI processing is essential for autonomy under degraded or disconnected conditions. Intrusion detection systems, for example, must continue to function even when communication links are jammed or unavailable, necessitating local inference and decision making [8]. Similarly, real time perception and tracking applications in remote sensing demand immediate on-

board processing to avoid delays inherent in cloud-based architectures [9]. This shift toward edge AI, i.e. executing AI models directly on the UAV. A need for computing platforms that are both powerful and flexible, yet decoupled from the real time flight control loop is created.

Companion Computers as UAV Intelligence Nodes The companion computer emerged as a response to these requirements. Typically implemented using embedded Linux platforms equipped with CPUs, GPUs, or specialized accelerators, the companion computer serves as a dedicated execution environment for AI, perception, mission logic, and high level decision making. Such platforms enable the deployment of computation intensive payloads while preserving the reliability and determinism of the flight controller [10].

Faster onboard computation enables quicker decision cycles, reduced hover times, and more efficient trajectory execution effects. This outweigh the additional power consumed by the compute subsystem itself, as explained in sections I–II of [11]. These findings reinforce the role of companion computer not merely as an auxiliary processor, but as a central enabler of intelligent autonomy.

Increasing Software Complexity on the Companion Computer While the introduction of the companion computer solved the raw computational bottleneck, it also introduced a new challenge: software complexity. AI enabled UAV systems now consist of multiple interacting software components with different lifecycles, resource demands, and reliability requirements. Modern UAV software increasingly resembles a distributed cyber physical system rather than a monolithic embedded application, as explained in section II of [6].

Furthermore, edge AI surveys highlight that onboard systems must simultaneously handle several functions often under dynamic conditions [12]. As mission profiles diversify, UAVs can no longer be limited to a single fixed task. They now need to accommodate multiple mission types while efficiently sharing the same hardware resources.

1.1.3 Microservices based UAV software architecture

Early companion computer software stacks often followed monolithic or layered architectures, where several task logic were compiled into a single executable or a tightly coupled process set. While this approach simplifies deployment, it becomes difficult as the code base grows and many developers work on the same application. The short-comings of monolithic architecture is discussed in section 2.1 of [13].

As, a solution microservices emerged along with its key advantages such as modularity, independent scaling and deployment [14]. By decomposing functionality into small well-defined services, developers can reason about system behavior at the component level rather than at the level of a monolithic code base. Matlekovic and Schneider Kamp has explained in sections 4-5 of [13], how they migrated from

monolithic to microservices based architecture. Microservices also support heterogeneity in implementation technologies. Companion computer software often integrates components written in different languages, For e.g., C++ for performance critical modules and Python for AI pipelines. An advantage highlighted in both precision agriculture UAV systems [15] and drone as a service platforms [16].

From a system perspective, microservices improve fault isolation. A failure in one service does not necessarily propagate to the entire system, which is particularly relevant for long duration or autonomous missions. Battseren in [17] demonstrates that loosely coupled components combined with a shared knowledge base can significantly improve robustness in safety critical onboard processing systems. Chapter 6 in [17] explains microservices. As system grows with respect to several components, the complexity grows exponentially.

1.2 Problem Statement

Recent years acknowledged increased interest in microservices based UAV architectures consisting of multiple independent software components that must be initialized, executed, supervised, and terminated without human intervention. The challenge is in automating the system initialization and remove the human dependency from deciding which components, when to start and how to start for a UAV mission.

The core problem addressed in this thesis is therefore: Creating an automated, mission driven system initialization mechanism. That is used for launching, supervising, and shutting down software components in blackboard based, micro service oriented UAV architectures operating on resource constrained onboard hardware. This enables fully autonomous, repeatable mission execution without reliance on manual setup procedures.

1.3 Motivation

The motivation for this thesis is rooted in the growing dependence of autonomous UAV systems on microservice based onboard software architectures, which introduce new challenges in system coordination, and runtime reliability.

Autonomy requirement: Autonomous UAV missions operate with human dependency on scenarios to start components manually on startup and pre flight scripting. This dependency is limiting to the true autonomous UAV system. Mission execution therefore requires an onboard mechanism that can initialize, coordinate, and supervise software components solely based on a mission description, without human involvement.

Resource efficiency constraint: UAV companion computers, such as the Jetson Nano, operate under strict CPU, memory, and power limitations. Launching all

available services irrespective of mission needs leads to unnecessary resource consumption and contention. A mission driven scheduler that activates only required components is essential for efficient use of onboard resources and for preserving computational headroom for time critical tasks.

Reliability and fault management: Autonomous missions must tolerate partial software failures without immediate human intervention. Create a centralized mission scheduling component that supervises runtime health of components via shared memory. It enables early fault detection and controlled recovery, reducing the risk of silent failures and mission loss.

In summary, the problem addressed by this work is not merely an implementation detail but a system level necessity. Reliable, mission driven initialization of software components is a prerequisite for achieving a dependable UAV operation on constrained onboard hardware.

1.4 Research Aim and Objectives

1.4.1 Research Aim

The proposed mission scheduling component enables automated initialization, supervision, and shutdown of software components from a mission definition, with explicit focus on deployment on resource constrained companion computers such as the Jetson Nano.

1.4.2 Research Objectives

To achieve this aim, the following objectives are defined:

Objective 1: To design a modular and deterministic mission scheduling architecture that interprets mission files, maps tasks to software components, and manages component life cycles via a shared memory blackboard.

Objective 2: To implement the proposed architecture as a C++ daemon capable of automated component startup, runtime supervision using heartbeat monitoring, and controlled shutdown behavior.

Objective 3: To integrate the scheduler into a realistic UAV software stack composed of loosely coupled microservices operating in a Linux based environment.

Objective 4: To evaluate the scheduler under representative mission scenarios with respect to correctness of component selection, and reliability of runtime supervision.

Together, these objectives aim to validate a reusable mission scheduling component as a practical architectural contribution toward reliable autonomous UAV operation under real world resource constraints.

1.5 Scope and Limitations

The scope is limited to automated initialization, runtime supervision, and controlled shutdown of software components using predefined mission descriptions on resource constrained companion computers.

Several aspects are intentionally excluded. Mission planning and decision making logic such as, path planning or behavior tree execution are assumed to be handled externally. Likewise, low level flight control algorithms and stabilization mechanisms are treated as independent components and are not modified or redesigned as part of this work.

The scheduler does not intervene in internal real time scheduling, threading, or execution policies within individual components. Its responsibility ends at inter component initialization. Fault handling is also limited in scope: while heartbeat loss detection and controlled stopping are implemented, advanced recovery mechanisms such as dynamic replanning, component hot swapping, or redundancy management are beyond the boundaries of this thesis. Finally, the work does not include hardware in the loop simulation or aerodynamic modeling, as the focus remains strictly on software architecture and runtime behavior.

By narrowing the scope in this way, the thesis allows for a focused and in depth investigation of mission orchestration under realistic onboard constraints such as, shared memory IPC, modular software components, and limited computational resources without conflating the orchestration concerns with unrelated control or planning problems.

1.6 Thesis Structure

This thesis progresses logically from background of UAVs. Chapter 2 outlines the theoretical and technical concepts, covering component based design, IPC, blackboard architectures, containerization, and the AREIOM Platform. Chapter 3 surveys related research on mission representation, initialization, orchestration, and monitoring for embedded and IoT systems. Chapter 4 introduces the conceptual design of the SCH, including system overview, use cases, and design rationale. Chapter 5 details its implementation, components, mission files, and scheduling logic. Chapter 6 evaluates SCH through functional performance, and fault recovery tests focusing on heartbeat based monitoring and restart behavior. Finally, Chapter 7 summarizes the main contributions and outlines future research directions.

2 Fundamentals

2.1 Component Based Software Design

Component based software architecture views a software system primarily as a composition of independently developed units, rather than as a single program constructed through incremental coding. These components already contain the necessary functionality, but their internal implementations are hidden. Developers interact with them only through well defined interfaces.

This architectural stance emerges from the need to support open systems that must operate across heterogeneous platforms, and continuously evolving requirements as a normal condition of system life as explained in section 1.1 of [18]

2.1.1 What constitutes a component

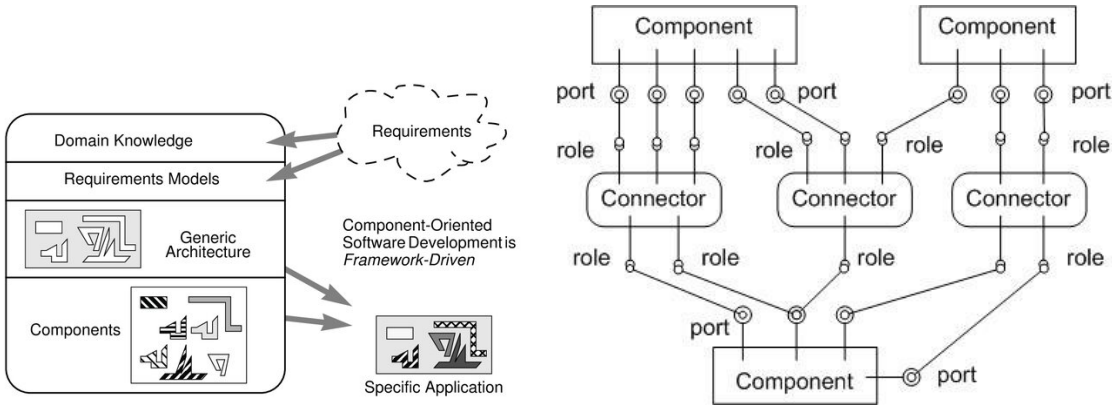
A component is designed from the outset to participate in composition with other components. Unlike objects, which encapsulate state and behavior at a relatively fine granularity, components may encapsulate any useful software abstraction, including modules, services, behaviors, or subsystems. A component is described as a static abstraction with plugs:

- Static, because it is a long lived entity stored independently of any particular application.
- Abstract, because its internal realization is hidden.
- With plugs, because it exposes explicit connection points used for composition .

Individually, a component does not necessarily perform a complete computation. Meaningful system behavior arises only when components are connected and configured together, similar to mechanical parts whose function emerges only when assembled [19].

2.1.2 Architecture as composition structure

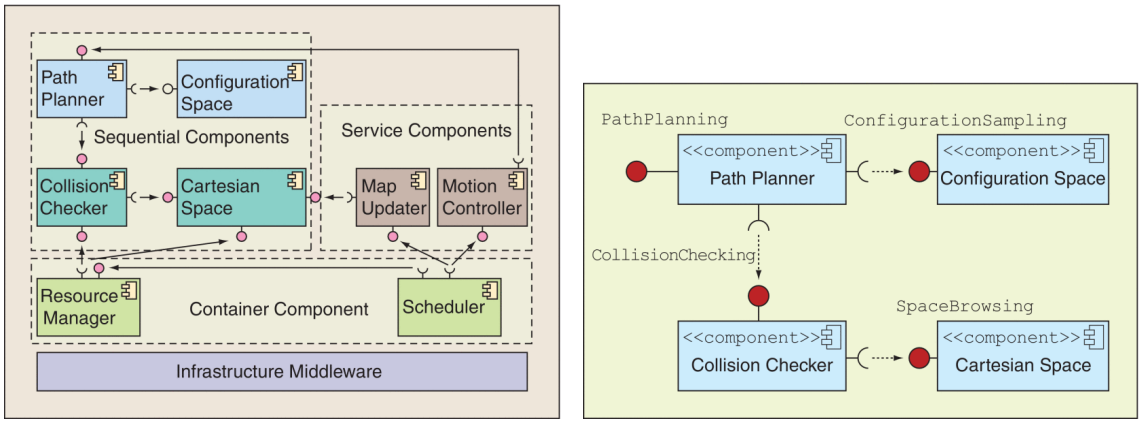
The architecture specifies interaction patterns, communication mechanisms, configuration parameters, and coordination rules that allow independently developed components to function as a coherent system [22]. Figure 2.2(b) motion planning



(a) [20] (b) [21]

Figure 2.1: Conceptual separation between computational behavior and compositional structure

component interconnected by means of provided and required interfaces. Component based systems explicitly treat components as architectural units. The flexibility, reusability, and evolvability of the system depend on how component interactions are structured and how much variability is isolated behind interfaces [23]. Figure 2.2(a) illustrates the architecture of a component assembly.



(a) [23] (b) [22]

Figure 2.2: Architectural decomposition into computation, communication, and configuration

2.1.3 Component lifecycle

Authors in [24] extends the traditional waterfall model for software product lifecycle to develop component based systems. In the component-based variant of the Wa-

terfall model, the traditional phases are preserved but their content changes significantly. The requirements, design, and implementation phases all become constrained and shaped by the existing assortment of components and by decisions about reuse. Further in component based system development, Component assessment involves finding candidates, selecting the best fit, verifying properties, and storing them in repositories with metadata like test results. Component development mirrors system development but prioritizes reusability, generality, and adaptability across multiple future systems. See Figure 2.3, how components are reused to realize a system.

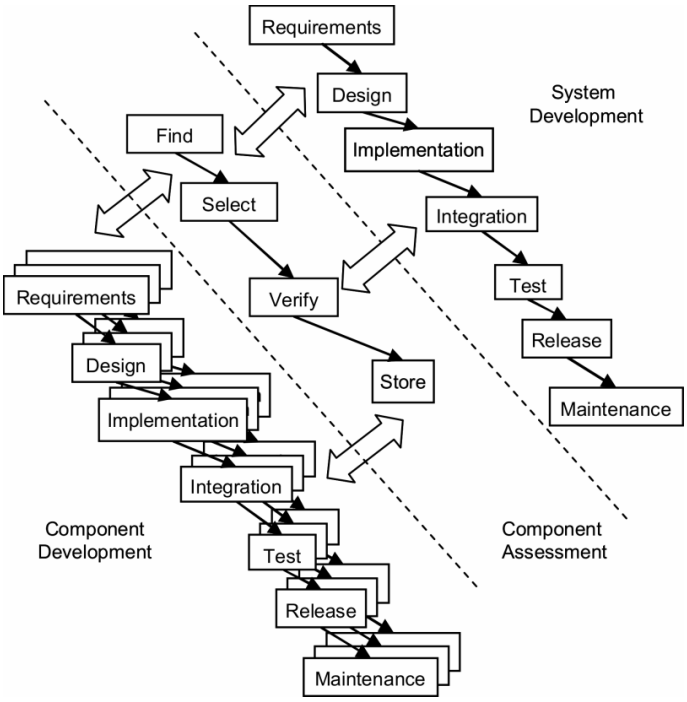


Figure 2.3: Parallel processes of Component-based development [24]

Jackson explains in [25] the components based on CPS cyber physical systems and proposes that behaviours themselves should be the primary components of design, and that they are always the result of interaction between the machine and multiple problem domains such as devices, people, and engineered artifacts. In section 2.3 of [25] Jackson explains the insistence on formal models of both the machine and relevant parts of the problem world, so that the concrete system behaviour can in principle be calculated as their parallel composition, denoted CSB = $M \parallel W$, while acknowledging that such models are contingent on an explicitly defined operating envelope.

Crnkovic explain in [26] how an approach can support both development and maintenance of embedded systems. He does this by explaining core CBD principles, outlining the defining issues, properties of embedded software, and suggesting ways to adapt component thinking to these contexts.

2.2 Inter Process Communication (IPC)

IPC refers to mechanisms that allow separate processes to exchange data and synchronize their actions while maintaining isolation. In Linux based systems common IPC facilities include pipes, message queues, shared memory, signals, and socket based communications. These mechanisms enable processes to coordinate, share information, and respond to events. Which is a critical capability in multitasking embedded environments where components must work in concert. Refer to section 6.4 of [27] for understanding in detail.

2.2.1 Mordern IPC Mechanisms

Dhaske's survey examines a range of IPC methods in modern operating systems. From classic options like shared memory and pipes to newer approaches such as UNIX domain sockets and RPC. Shared memory provides excellent speed but demands careful synchronization to avoid race conditions, making it best for high volume, local data exchange. Pipes and FIFOs are simpler and suitable for parent child communication but offer only moderate performance. Sockets and RPCs, though more complex, support both local and networked communication with decent efficiency [28].

2.2.2 IPC Performance in Linux

Krishnaveni and Ruby's paper [29] did performace testing on Linux kernel 2.2.5-15 and FreeBSD 4.1/4.2 for comparing pipe, messages queue, streaming and datagram sockets. They tested throughput across varied message sizes to mimic embedded networking scenarios, finding Linux outperformed the others overall.

Key Performance insights they found were that message queues edged out pipes for small payloads due to their record based design, reliability, flow control, and preserved boundaries, unlike byte stream pipes. For larger transfers, sockets outperformed the others: UDP datagrams delivered the highest speed by avoiding stream setup overhead and kernel queue copying, while TCP provided consistent, optimized performance across all sizes, refer figure 2.4.

Use message queues for small, structured messages within the same system that require reliable delivery and ordering. Switch to UDP sockets for high-volume data transfers or networked tasks where maximum speed matters more than guaranteed delivery. Their analysis of memory usage, transfer speeds, and buffer sizes helps engineers pick the right IPC method for each scenario.

2.2.3 Signals and Semaphores in IPC

All the above referred papers [27, 30, 29, 31, 28] on IPC, explains the deadlock and race condition problem. While the previous papers emphasize data transfer methods. Linux IPC also relies on signals and semaphores for control and synchronization.

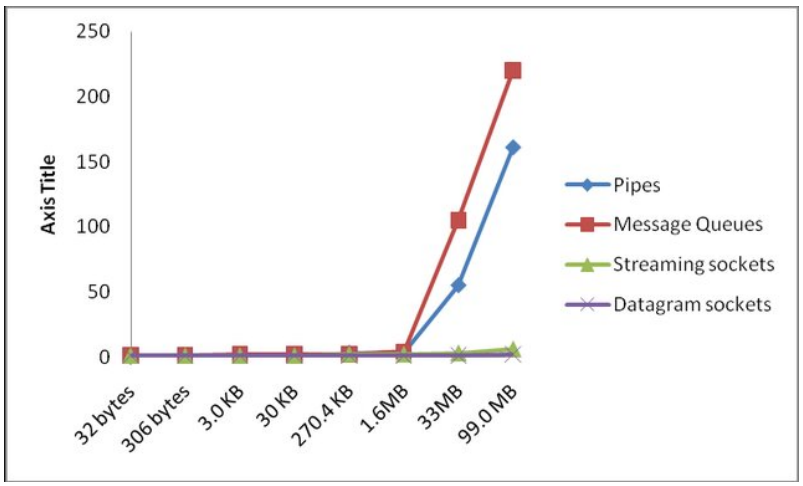


Figure 2.4: Comparison of response time for pipe, queue and sockets [29]

Signals act as lightweight software interrupts from the OS, notifying processes of events like child process termination or timer expiry. They carry minimal data, just a signal number and optional small value therefore they are ideal for triggering actions like sending SIGTERM or SIGUSR1 rather than transferring bulk information.

Semaphores serve as synchronization tools within the shared memory setups. They function as protected counters that processes wait on or signal. This ensure mutual exclusion to avoid race conditions. Semaphores don't move data themselves but coordinate access, guaranteeing that producer fills a shared buffer before a consumer reads it. In Linux IPC particularly for real time or embedded systems, signals handle event notifications while semaphores enable safe coordination, complementing core data channels like queues, pipes, and sockets.

2.3 Blackboard Architecture

A blackboard architecture is a distributed problem solving approach in which multiple agents contribute incrementally to a shared solution state. It is commonly characterized by three core elements: (i) the blackboard, (ii) knowledge sources, and (iii) a control mechanism It is stated in the abstract of [32].

2.3.1 Shared solution representation

In Hayes Roth's formulation, blackboard assumes that all solution elements generated during problem solving are recorded in a structured, global database called the blackboard. Blackboard's structure organizes elements along two axes: solution intervals and levels of abstraction. Solution intervals correspond to regions of a solution on problem specific dimensions (e.g., in multiple task planning, plan execution time intervals). Levels of abstraction represent the solution in different degrees of

detail (e.g., a higher level may represent task sequencing, while a lower level includes task details and travel routes) assumptions are explained in the section 3.1 of [33].

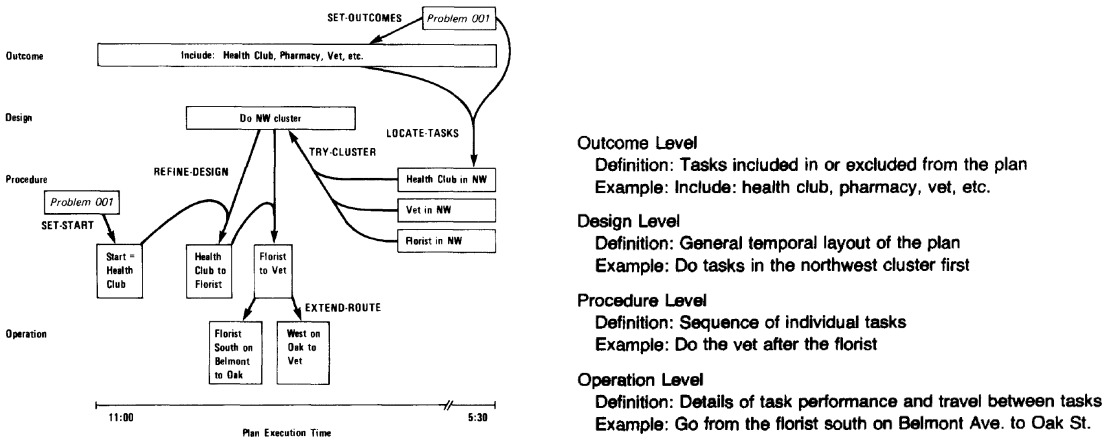


Figure 2.5: (a) illustrate how a partial plan is represented across time (solution intervals) on the blackboard. (b) Levels of abstraction for the multiple-task planning blackboard [33]

Craig describes a closely related principle: a blackboard system should include a hierarchical, global database, a collection of knowledge sources, and a scheduler. Craig also notes that an early HEARSAY-II organization used a single blackboard divided into multiple abstraction levels, and its organization is shown in Chapter 3 Fig. 1 of [34].

2.3.2 Knowledge sources

In the blackboard model, solution elements are generated and recorded by independent processes called knowledge sources. Knowledge sources have a condition–action format, where the condition specifies situations (often requiring a configuration of blackboard elements) in which the knowledge source can contribute, and the action specifies behavior that typically creates or modifies solution elements on the blackboard [33]. Craig likewise states that knowledge sources are structured as condition action pairs. Where the condition monitors blackboard events and the action makes changes to the blackboard by modifying entries or solution islands explained in section 2.3 of [34]. In Hayes-Roth's account, knowledge sources are independent (they do not invoke one another) yet cooperative, because they contribute solution elements to a shared problem by responding only indirectly via information on the blackboard, explained in assumption2 section 3.1 of [33].

A blackboard architecture includes a control scheduler responsible for selecting which eligible actions to execute. Craig states that the scheduler acts upon triggered knowledge sources and uses problem specific parameters to determine which

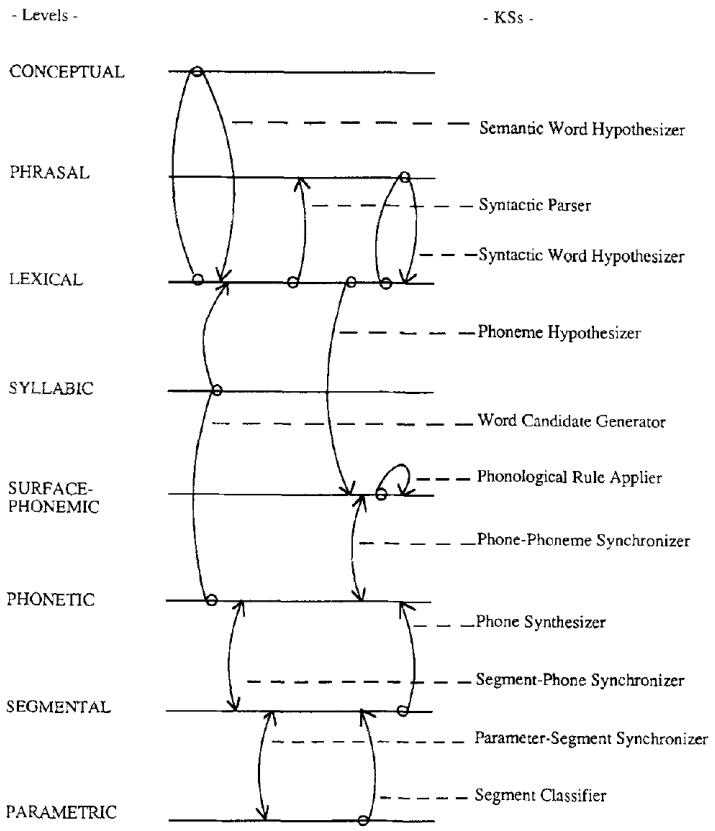


Figure 2.6: HEARSAY-II Knowledge Sources 1975 [34]

knowledge source action to execute. And that multiple knowledge sources may execute concurrently in a parallel environment. Craig also defines opportunism as a control strategy that makes the best use of the current solution state, where the scheduler may temporarily ignore a top down strategy to act on a promising solution island [34, 32].

In Hayes-Roth's formulation, blackboard activity is often event-driven, where each change to the blackboard constitutes an event that may trigger one or more knowledge sources in the presence of additional information. Each unique triggering produces a KSAR that represents the particular knowledge source by a blackboard event. On each problem solving cycle, a scheduling mechanism chooses a single KSAR to execute its action, resolving competition among pending activations using scheduling criteria assumption2 and assumption three in section 3.1 of [33].

Draper et al. describe a distributed blackboard system for image understanding in which schema instances (object experts) execute independent, potentially concurrent recognition strategies and communicate asynchronously through a global blackboard. They state that the control component of each schema schedules the application of knowledge sources to gather appropriate support for or against a hypothesis. They also define system components including the blackboard, knowledge

Table 2.1: Attributes of knowledge sources [33].

Attribute	Definition
Name	Identifying label
Problem-Domain	Domain(s) of application
Description	Characteristic behavior
Condition	Situation of interest: Trigger and Pre-Condition
Trigger	Event-based predicates for triggering
Pre-Condition	State-based predicates required for execution
Condition-Vars	Specification of variables used in Condition
Scheduling-Vars	Specification of variables to be used in scheduling
Action	Program of blackboard changes

sources, and interpretation, noting that strategies fulfill the role of a scheduler, in chapter 2 of [35].

2.3.3 Blackboard as an architectural pattern

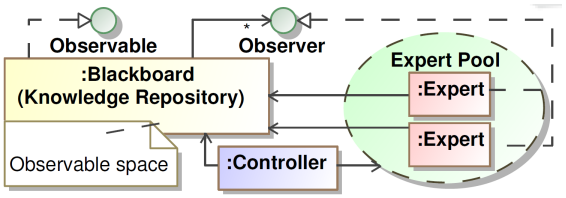


Figure 2.7: A UML object diagram depicting the structure of the blackboard pattern [36]

Solms describes the blackboard pattern as infrastructure for specialized processing units (experts) to collaborate in developing a possibly partial or approximate solution to a difficult problem where no deterministic solution strategy is known. Solms states that an observable blackboard hosts both the problem specification and the current state of the solution. Solms further specifies that a controller manages a pool of experts and feeds problems into a blackboard observed by them. Experts decide when to process published information or requests. Outputs are fed back into the blackboard and may trigger further activities. The controller decides when a solution or sufficient approximation has been obtained, refer section 3.2 of [36].

Table 2.2: Impact of architectural patterns on organizational qualities. An up/down-arrow indicates that the pattern makes it easier/more difficult to realize a quality attribute, whilst the tilde symbol indicates that it has no significant effect on a quality attribute [36].

Pattern	**Quality Attribute**								
Quality	Innovatability	Learnability	Flexibility	Reliability	Integrability	Reusability	Scalability	Performance	Auditability
Microkernel	∼	∼	↑	∼	↑	↑	∼	↓	↑
Blackboard	↑	↑	↑	↓	↑	↑	↑	↓	↑
Hierarchcal	↓	↓	↓	↑	↓	↓	↑	∼	∼
Pipes & Filt	↓	↓	↑	∼	∼	↑	∼	∼	∼

2.4 Containers

Containers are a form of operating system level virtualization that allow multiple isolated user-space instances to run on a single host kernel. They package an application along with its dependencies and environment, but unlike traditional VMs, containers do not bundle a separate operating system for each instance. This means all containers on a host share the host's OS kernel, making them much lighter and faster to start compared to VMs. Because containers avoid the overhead of emulating hardware and booting full OS images, they are significantly smaller in size and can launch almost instantly. This lightweight nature allows hundreds of containers to reside on a host and enables efficient scaling and resource utilization in cloud environments [37].

2.4.1 Linux Namespaces: Isolation of Resources

Linux namespaces are fundamental to container isolation, partitioning global system resources so that each container sees a private, sandboxed view of the operating system. When a process is placed in a new set of namespaces, it has the illusion that it is the only process on the machine for those resource domains [37]. The Linux kernel provides several types of namespaces, each isolating a particular resource or abstraction:

- PID Namespace: Gives the container its own process ID space, so processes inside a container have PID 1 as their init and are not visible to processes in other containers. This ensures containers have isolated process trees.
- Mount (Filesystem) Namespace: Provides each container with what appears

to be its own distinct filesystem hierarchy. A container can have its own root filesystem and mount points, preventing it from seeing or affecting the host or other containers' files.

- Network Namespace: Gives each container its own network stack, including interfaces, IP addresses, routing tables, and port ranges. To the processes inside, it looks like they have unique network devices and addresses, enabling containers to have overlapping IPs or even mimic separate machines on a network.

- UTS (Unix Time-sharing System) Namespace: Allows a container to have its own hostname and domain name, isolating system identifiers so that a container can rename itself without affecting the host.

- IPC Namespace: Provides isolation of inter-process communication resources (System V IPC and POSIX message queues), so that IPC mechanisms (shared memory segments, semaphores, etc.) in one container are distinct from those in another.

- User Namespace: Isolates user and group ID numbers, letting containers run as a non-root user on the host even if they have a "root" user inside the container. In a new user namespace, processes can have elevated privileges (UID 0) inside the container while the kernel maps that to an unprivileged UID on the host, greatly enhancing security.

Using these namespaces, a container runtime can polyinstantiate the global environment [38, 37].

2.4.2 Control Groups (cgroups): Resource Management

While namespaces isolate what each container can see, Linux control groups regulate what each container can use. Cgroups are a kernel feature that limits and monitors the resource usage of processes. By organizing processes into hierarchical cgroups and applying resource controllers, the kernel can enforce quotas and priorities on CPU, memory, I/O, and more. In practice, each container's processes are placed in specific cgroup subsystems that govern resources [39, 38].

Cgroup policies can be configured via the virtual filesystem interface (usually under /sys/fs/cgroup/), setting limits like memory max usage, CPU shares, block I/O priorities, etc. For example, when Docker runs a container with memory limits (docker run –memory=X), it creates a corresponding cgroup in the memory controller and assigns the container's process to it; the kernel will enforce that the container cannot exceed X bytes of RAM. Similarly, CPU shares or quotas can ensure one container doesn't starve others of CPU time. In summary, cgroups provide a fine-grained mechanism to partition and distribute system resources among containers. By binding containers to their cgroups, the runtime achieves controlled

performance isolation in addition to the security isolation provided by namespaces [38, 40].

2.4.3 Union File Systems and Layered Images

Containers also rely on efficient storage mechanisms to manage the file system contents of images and containers. The concept of a layered image is central to container portability and efficiency. A container image is a packaged file system that includes the application and its dependencies (libraries, run times, etc.), but it is typically built in layers rather than as one monolithic archive. Modern container systems use union file systems or similar copy on write file systems to achieve this layering. Each command in the process of building an image creates a new layer on top of the previous ones. These layers are stacked to form the final image. Conceptually, the image is composed of a base layer (for example, a minimal OS) and additional layers capturing incremental changes (application files, configurations, etc. added on top) [37].

This layered design has several advantages. First, layers can be shared and reused between images. For instance, if two images both use the same base OS layer (say, Ubuntu 22.04), that layer needs to be stored on disk and in memory only once, even if dozens of containers use it. Each container image just adds its specific differences on top. Second, launching a new container is fast because the runtime can combine the pre-existing layers without duplicating data; it uses a union mount so that the container sees a single coherent file system assembled from the layers. Any writes a container makes at runtime (to its file system) can be directed to a thin writable layer on top (often called the container layer), leaving the underlying image layers untouched (copy-on-write). This means multiple containers can safely share the same read-only image layers simultaneously. It also means that distributing images is efficient: only new layers need to be downloaded when pulling an image, if the base layers are already present on the host [37]. In summary, the layered image approach and union file systems make container images lightweight, shareable, and portable, which is crucial for scaling containers in practice [38].

2.4.4 Containerization in Embedded Systems

Container technology was first used in cloud systems, but it is now also common in embedded and IoT devices. Embedded systems benefit from containers because the application and its environment are packaged together, which separates software from hardware and makes deployment easier. As a result, control software does not need to be tied to one specific device, and studies show that containers can decouple control logic from physical systems even in real time settings [41]. This is especially useful in IoT scenarios where the same software must run on many different devices, and platforms such as AWS Greengrass and Azure IoT Edge demonstrate that containers are well suited for portable and lightweight edge deployments [42].

Embedded devices often have limited CPU power, memory, and storage, so the extra cost of containers must be considered carefully. Containers are lighter than virtual machines, but they still introduce overhead in the form of startup delays, increased memory use, and slower input or output on low power hardware [42]. These effects are important in real time applications, so lightweight container runtimes are used to reduce the impact. One example is BalenaEngine, which is smaller than a standard Docker installation while still being compatible with it [39]. Containers can also be used together with real time Linux systems, and with the right configuration, studies show that containerized applications can still meet strict timing requirements on shared hardware [41].

Overall, containerization improves portability, modularity, and the ability to update software in embedded systems. It allows software to change without requiring new hardware and helps to extend device lifetimes. With careful management of limited resources, containers remain a practical option for many modern embedded and edge platforms [41, 42].

2.5 AREIOM: Adaptive Research Multicopter Platform

2.5.1 Modular Software Components: Mission, Vision, and Navigation Subsystems

The proposed software architecture is organized into three primary subsystems: Mission, Vision, and Navigation. Each responsible for a distinct set of system functions. This separation enables clear responsibility boundaries and supports scalable and maintainable system design.

The Mission subsystem coordinates high level system behavior and execution through a collection of cooperating software components. Within this subsystem, the Scheduler (SCH) is responsible for organizing and sequencing mission related operations, while the Control Handler (CTH) manages the processing and execution of control commands. Decision making logic is encapsulated within the Expert System (EXS), which contributes intelligent reasoning capabilities to guide autonomous behavior. In parallel, the Ground Control Server (GCS) serves as the communication interface between the onboard system and ground based control units, enabling mission supervision and operator interaction [17].

The Vision subsystem provides perception and visual feedback necessary for navigation and inspection tasks. This subsystem consists of multiple specialized components. The Camera Gimbal Control (CGC) manages camera orientation to support target tracking and stabilization. Visual data acquisition is handled by the Acquisition (ACQ) component, while the Object Detection (DET) module processes incoming data to identify relevant objects in the environment. The Streamer (STR) component enables real-time transmission of video data, and the Recorder Streamer

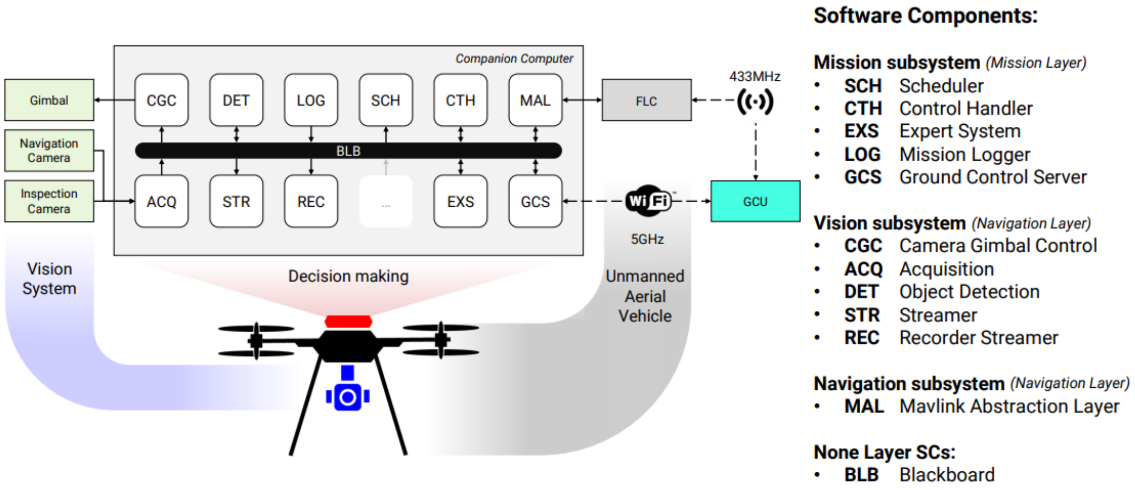


Figure 2.8: AREIOM Software Architecture [17]

(REC) supplements this functionality by storing selected data streams for offline analysis.

The Navigation subsystem is responsible for vehicle motion control and positioning. Its functionality is primarily realized through the MAVLink Abstraction Layer (MAL), which provides a simplified interface for interacting with the MAV's flight controller. By abstracting low-level communication details, this subsystem ensures reliable command execution and state feedback necessary for stable and accurate navigation.

2.5.2 Shared Memory Based IPC: Enabling Inter Component Communication

Efficient communication between software components is achieved through a shared memory based IPC mechanism, which forms a central element of the system architecture. This approach allows multiple components to exchange data through a common memory region, ensuring timely and coordinated information flow across the system. By enabling direct access to shared data, IPC minimizes communication overhead and supports fast data exchange, which is essential for real time system operation.

The shared memory mechanism provides a structured and efficient means of inter component communication by allowing multiple processes to read from and write to a designated memory space concurrently. From the perspective of each component, this shared region behaves similarly to local memory, enabling straight forward data access. To maintain data consistency and avoid concurrency related issues such as race conditions or data corruption, synchronization mechanisms mutexes and

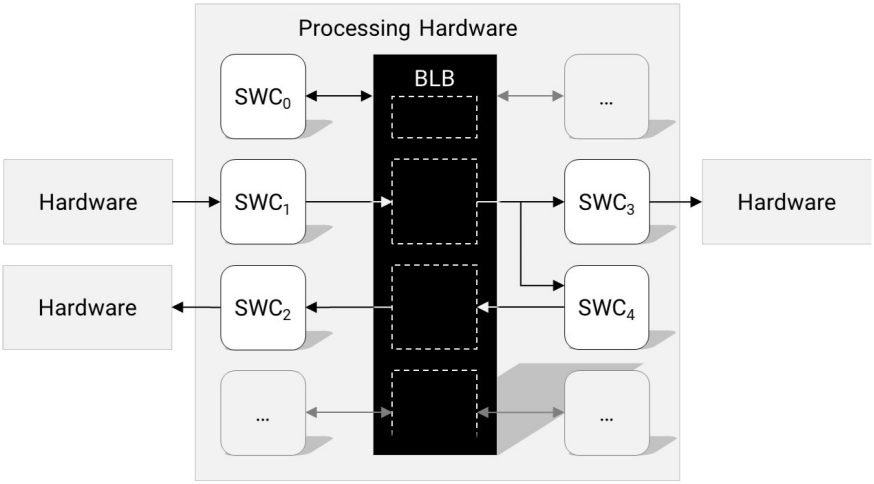


Figure 2.9: Shared memory based IPC [17]

semaphores are employed. These mechanisms regulate access to shared memory regions, ensuring controlled and safe data interaction among processes.

Effective coordination between software components is critical for timely and reliable mission execution. For instance, when the Mission Scheduler (SCH) assigns a new task to the Control Handler (CTH), the task information can be written directly into shared memory. The Control Handler can then retrieve this data immediately, without the delays associated with more complex communication protocols. Such low latency data access is especially important for applications that require rapid responses and tight integration between perception, decision making, and control components [17, 43, 44, 45, 46, 47].

3 Literature Review

3.1 Mission Representation in Autonomous Robotic Systems

Mission representation is a critical aspect of autonomous robotic and UAV systems. It encompasses how high level tasks and objectives, i.e. mission are specified, modeled, and ultimately executed by robots. In general, a mission is described at design time in some formal representation language, which is then interpreted or executed at run time by an underlying control model or architecture. At a high level, mission representation approaches in the literature can be grouped into two categories:

- **Mission Control Models**: These define how missions are executed and controlled. this include run time frameworks that dictate the robot's behavior and decision logic during mission execution.
- **Mission Specification and Configuration Languages**: These are the design time representation. Designers encode mission plans, objectives, and constraints in a structured way that can later be interpreted by the control system.

3.1.1 Mission Representation as Control Models

FSMs and FSA

Finite state machines or Finite State Automaton have long been a staple for robot control. In an FSM based controller, the mission is represented as a set of states. Each state often corresponding to a behavior or an action, and transitions triggered by events or conditions. A classic example is the MissionLab system's finite state acceptor representation. Where actions are nodes, and perceptual triggers, which define the conditions for transitioning to the next action, serve as the directed edges connecting them. An entire mission plan can thus be seen as a graph of states connected by sensor driven transitions, defining a workflow of activities [48]. FSMs are conceptually simple and offer clear execution semantics, at any time the robot is "in" a particular state and will switch to another state when the corresponding condition is met. This simplicity has made FSMs widely adopted in practice. In recent years, FSM based approaches remain among the most widespread solutions for encoding robot and UAV behaviors during mission execution [49]. They excel in well defined scenarios where the sequence of actions and conditions is mostly predetermined.

Strengths: Finite state machines are easy to understand and visualize. They allow designers to explicitly map out the mission logic and are supported by many tools. FSMs enumerate states and transitions explicitly. This makes the mission execution logic transparent. For smaller missions, the logic is also relatively straight forward to debug or verify. They have been effectively used in single robot scenarios and even in coordinated multi robot missions by deploying an FSM on each robot, refer section 4 of [48]. Many commercial or standard autopilot systems also implicitly use an FSM for missions, where a waypoint following mission can be viewed as an FSM cycling through waypoints.

Limitations: The simplicity of FSMs comes at the cost of scalability and flexibility. As missions grow in complexity with more states and concurrent tasks, the state graph can quickly expand and become difficult to manage. This problem is often referred to as state explosion. Adding new behaviors or handling numerous contingencies increases the number of states and transitions, reducing readability and maintainability. FSMs also handle concurrency poorly. A basic FSM is single threaded and allows only one active state at a time. Representing concurrent sub tasks therefore requires complex extensions or several state machines running in parallel. FSMs are not inherently modular either. Changes in one part of the mission logic can affect other parts, and reusing sub graphs of behavior is difficult. Recent studies show that while both FSMs and newer models can accomplish the same tasks, maintaining mission behavior as a Behavior Tree becomes easier as task complexity increases [50]. In summary, FSMs tend to become brittle in complex or highly dynamic missions where many alternative flows or interrupts must be managed.

Use Cases: FSMs are well suited for missions divided into clear sequences of modes or stages with defined event triggers. Typical examples include structured inspection routines, scripted surveillance patrols, and competition scenarios where the sequence of steps is predetermined. Early demonstrations of multi robot missions also used FSMs. For example, an indoor and outdoor reconnaissance mission was implemented with both UAVs and UGVs running its own state machine to represent its specific role in the mission [48]. Even today, many robotics middlewares provide state machine libraries for creating mission executives, such as SMACH in ROS. FSMs remain common in industrial settings like manufacturing robotics. Because of this wide use, many engineers are already comfortable with the model, even though newer paradigms are available.

Behavior Trees

Behavior Trees have gained significant popularity in robotics over roughly the last decade as an alternative to FSMs for modeling robot decision logic and mission execution. Originally coming from the video game industry, BTs offer an extensible tree based representation of missions [51]. A behavior tree organizes behaviors such as actions or sub tasks in a hierarchical tree structure. It uses control flow nodes like Sequence, Selector, and Parallel to determine the execution order and decision logic.

The tree is usually executed from top to bottom on each tick or event. Nodes return statuses such as success, failure, or running, these control how the traversal proceeds. This structure decouples sequencing, fallback decision making, and concurrency in a modular way.

Conceptual Role: In mission representation, a Behavior Tree acts as a runtime policy or control program for the robot. It is a formalism that lies between purely reactive behaviors and high level planning. This provides a execution model that can orchestrate complex tasks while being responsive to changes. Each branch of the tree can encapsulate a sub mission or behavior. Using it designers can compose complex missions by nesting simpler behaviors. This hierarchical structuring is touted as more modular and reusable than monolithic FSMs [51]. BTs are proved useful for facilitating modular design and code reuse, because subtrees can be swapped or reused across different missions. They also have an explicit notion of fallback via selector nodes, which facilitates the encoding of robust behavior, such as trying an alternative when the preferred action fails [52].

Strengths: BTs include readability and modularity. Many robotics developers find BTs more intuitive to comprehend and modify than equivalent state machines [53]. The tree structure often shown as a graphical diagram, aligns well with the way humans naturally break down tasks into hierarchical steps. Because behaviors and conditions are encapsulated in nodes and subtrees, BTs lend themselves to incremental development: new capabilities can be added as branches without rewriting the entire structure. Another strength is their inherent support for reactivity and fallback behaviors. A BT continuously reevaluates conditions and can switch behaviors if a higher priority condition becomes true. This is very useful in dynamic environments. They handle concurrent behaviors through parallel nodes more naturally than a basic finite state machine. This structure allows actions to be interleaved, but careful design is needed to prevent conflicts. BTs have formal semantics and can generalize several other control architectures, including certain FSM patterns. They provide a unifying framework for sequencing and selection tasks [51].

Limitations: While behavior trees improve on FSMs in many respects, they also have their own limitations. One limitation is that, in their standard form, behavior trees do not explicitly model state beyond what is represented in the tree structure and the running statuses. This means that if a mission requires remembering complex state or coordinating many parallel conditions, a behavior tree can become large or may need extensions such as memory variables or decorators to handle it. Additionally, the execution of a behavior tree is typically managed by a tick loop. This approach may not be as straightforward for a beginner to understand as a simple FSM diagram, and therefore introduces a learning curve for effective use. Some critics point out that because behavior trees allow many ways to compose behaviors, ensuring correctness and avoiding conflicts can be challenging without careful design. Formal verification of behavior trees remains an active research area. Despite these challenges, ongoing studies and direct comparisons suggest that behavior trees scale better with increasing complexity. For example, an experimental comparison

found that while a robot's external behavior could be equivalent under an FSM or a BT, maintaining the BT was easier as the mission complexity increased. This improvement was largely due to better modularity and separation of concerns [50].

Use Cases: BTs have been successfully applied in a wide range of autonomous robot missions, from service robots to drones. They are also common in artificial intelligence for game non player characters, and this practice has translated to robotics. Examples include mobile manipulators performing sequences of tasks with contingencies, or unmanned aerial vehicles conducting surveillance while reacting to unexpected events, such as detecting a target and switching to a tracking behavior. A concrete use case appears in mobile manipulation and long duration robot missions. BTs enable the combination of deliberative actions with reactive checks. For instance, a BT can represent a high level plan while continuously monitoring battery levels or safety conditions as higher priority branches. In multi robot teams, BTs are sometimes used to define each robot's role in a loosely coupled manner. They can simplify coordination by structuring behaviors, although explicit synchronization often requires additional mechanisms. The robotics community has developed standard BT libraries and graphical editors, which are integrated into Robot Operating System (ROS) ecosystems as a popular choice for the executive layer. The combination of human readability and machine interoperability has made BTs a key component of modern robotic software frameworks. They serve as an intermediate mission representation that functions both as a modeling language and an efficient runtime control policy [51].

Petri Net Based Mission Controllers

Another line of mission execution models in robotics is based on Petri nets, a well known formalism for concurrent systems. Petri nets provide both a graphical and mathematical means to model states and events with explicit representation of concurrency, synchronization, and resource sharing. In robotics, Petri Net Plans, or PNPs, have been introduced as a formal model for representing and executing multi robot plans [54]. A Petri Net Plan extends basic Petri net theory to encode robot actions, sensing, and control structures. According to Ziparo et al. in section 3.1 of [55], a single robot Petri Net Plan consists of places and transitions arranged to represent various stages of action. Places correspond to stages such as the start of an action, action execution, and completion, whereas transitions represent events such as starting, stopping, or receiving external triggers. Tokens capture the robot's progression through these stages. The Petri net model can be enhanced with control places and transitions corresponding to high level programming constructs such as conditional branches, loops, or parallel sections. This enables Petri Net Plans to encode complex missions involving conditionals, iteration, concurrency, and interrupts within a unified formal model [55].

Strengths: Petri net based control models are highly expressive, especially for multi robot or multi agent missions requiring concurrency and synchronization. Unlike a linear FSM, a Petri net naturally supports multiple tokens, allowing a robot

or several robots to occupy multiple states or perform subtasks concurrently. This property aligns well with missions where multiple objectives or parallel processes must occur simultaneously. The formal foundation of Petri nets also supports analysis and verification. Properties such as reachability, deadlock freedom, and liveness can be examined using well established Petri net theory. Petri Net Plan frameworks often include execution engines that ensure correct firing of transitions based on the net's state. For instance, a transition will only occur when all prerequisite tokens are present, guaranteeing logical consistency.

Petri nets manage synchronization between robots through shared transitions. In a multi robot Petri Net Plan, one robot's transition can produce a token that enables another robot's transition, effectively coordinating behaviors between agents. This capability to model coordinated team plans provides a significant advantage in multi robot scenarios, as discussed in sections 1 and 2 of [55].

Limitations: The expressiveness of Petri nets comes with increased design complexity. Building a Petri net model for robotic missions requires specialized knowledge and may not be intuitive for users unfamiliar with formal modeling. The graphical representation can also become complicated for large missions, although hierarchical approaches such as colored or object oriented Petri nets help group related components. While Petri nets excel in representing concurrency and synchronization, they may be less intuitive for purely sequential logic compared to FSMs or BTs. Another limitation is tool support and integration. FSMs and BTs are widely integrated into common robot software frameworks, whereas Petri net execution engines are more specialized and less standardized. Despite these challenges, interest in Petri net approaches remains strong. Recent research has explored the use of colored Petri nets, a compact and data aware variant, for UAV mission control to manage complex coordination and resource constraints more effectively. The literature suggests that Petri Net Plans are valuable when high assurance of correct coordination is required, though they demand a more formal and disciplined design process [49, 55].

Use Cases: Petri net models are particularly suitable for multi robot missions that involve tight coordination or synchronized actions. Examples include cooperative surveillance or search tasks where multiple robots must periodically rendezvous or exchange information. These can be naturally represented as places for states and transitions for communication events. One documented example involves a cooperative UAV–UGV task in which a UAV guides a ground robot toward a target. Using a Petri Net Plan, the UAV's detection event can be modeled as a transition that produces a token enabling the UGV's approach action. This ensures that the ground robot proceeds only after the aerial robot has located the target [56]. Petri nets have also been applied in robot soccer and swarm robotics, where multiple robots must coordinate roles and actions under strict time constraints [54]. While Petri net based frameworks are primarily seen in academic and research environments such as multi robot programming prototypes, they continue to influence modern mission specification languages. Some of these languages incorporate Petri net semantics at their core [57].

Deliberative Planning and Hybrid Control Models

In contrast to fixed state or scripted execution models, some mission control approaches rely on online planning and decision making. These deliberative or AI planning based models treat mission execution as a continuous planning problem, where the system generates action sequences to achieve goals and may replan as the environment changes. A prominent example is the use of automated planners with languages such as PDDL to represent mission domains and goals. In robotics, a classical planner can create a sequence of actions to accomplish a high level mission objective, which is then executed and monitored by a runtime system, often referred to as an executive.

For instance, a mission like "map these five waypoints and take photos of any detected objects" can be given to a planner that computes an optimal route or task sequence. Execution semantics in such systems are typically embodied in a plan execution engine or a hybrid architecture. In these architectures, the planner generates tasks, and a lower reactive layer executes them while responding to immediate sensor feedback.

In the literature, purely classical planning approaches, such as STRIPS based planners, are recognized as powerful for mission plan generation but must be adapted or combined with reactive mechanisms to be effective in robotics [58]. Robots operate in dynamic environments where plans may fail or conditions change unexpectedly; thus, modern frameworks integrate planners with reactive control loops. Some hybrid architectures combine HTNs or policy learning at higher levels with state machines or behavior trees at lower levels for task execution. The execution semantics of such systems often revolve around continually checking plan progress, validating preconditions, and triggering replanning when necessary. For example, if an action fails or a new obstacle is observed, a replanning process may be initiated. A runtime planner can therefore be viewed as a control model that selects the next action based on the current world state and goal, rather than following a predefined sequence.

Strengths: The main strength of deliberative planning models lies in their flexibility and autonomy. A robot is not restricted to a hard coded sequence of actions but can derive its own solution in real time, which is crucial for complex or open world missions. This capability enables adaptation to unpredictable environments. Here, the mission specification may be as abstract as a goal condition or a reward function, allowing the robot to make decisions to achieve that goal. Another major advantage is optimality: planners can optimize paths or schedules according to a cost function, something fixed scripts cannot achieve. Additionally, formal planning languages such as PDDL or temporal logic specifications allow missions to be expressed at a very high level. For example, "visit these locations and gather data" without specifying how. This high level abstraction reduces the burden on human designers. In multi robot missions, deliberative approaches can handle dynamic task allocation and cooperative behavior at runtime by using algorithms to assign roles or sub goals as conditions evolve.

Limitations: Pure planning systems face several challenges in real time robotic control. Planning algorithms can be computationally expensive and may fail to meet timing requirements when mission spaces are large. Consequently, many systems restrict planning to the high level while relying on faster reactive controllers for execution; otherwise, a robot might spend too much time thinking when it should be acting. Another limitation concerns predictability and safety. Automatically generated plans must be validated to ensure they do not violate constraints such as power limits or safety distances. This requires encoding numerous domain specific rules within the planner, which can add considerable complexity. Planners also typically assume an accurate environmental model; if the model is incorrect or becomes outdated, the resulting plan can be sub optimal or fail entirely. For these reasons, integration with reactive layers and continuous monitoring is emphasized in the literature [58]. A deliberative planner alone does not constitute a complete execution model, it requires an executive to monitor execution, trigger replanning, and often a low level controller to manage real time actions. Designing such integrated systems is non trivial. As a result, many researchers have proposed hybrid control architectures, consisting of deliberative, executive, and reactive layers. These frameworks are powerful but complex to implement and tune.

Use Cases: Planning based control is commonly used in high level task planning for service robots, such as a domestic robot planning how to serve drinks to several guests in a specific order. It also appears in logistics robots that schedule deliveries, and planetary rovers that plan daily paths and science activities. In multi UAV systems, researchers have developed temporal planners that dynamically assign UAVs to sub tasks such as area coverage or target tracking. This adaptability is vital for missions like disaster response, where goals can shift based on new information [58].

A notable formal approach in the literature involves LTL specifications. In such frameworks, users describe missions as temporal logic formulas for example, "Eventually survey area A and then B, and always avoid nofly zone C." The system then automatically synthesizes a correct by construction controller or plan that satisfies the specified mission [59]. This approach merges planning with formal methods, ensuring that if a plan fulfilling the specification exists, the controller will find it. Such methods have been successfully demonstrated for high level reactive mission planning, where the outcome is an automaton or set of policies that guarantee fulfillment of the LTL specified requirements [59]. These use cases illustrate that planning based control is most advantageous when missions require reasoning over alternatives or when the environment can significantly influence the best course of action.

Reactive and Behavior Based Architectures

A final category worth noting includes purely reactive or behavior based control architectures, such as subsumption architectures, rule based systems, and BDI (Belief–Desire–Intention) agent architectures. In these models, there may not be an explicit graph or tree representing the mission. Instead, the system consists of a set

of behaviors or rules that concurrently evaluate conditions and produce actions.

For example, a rule based reactive planner might include rules of the form, "IF condition X holds, THEN execute behavior Y," with mission logic emerging from the interaction of these rules. The subsumption architecture a classic behavior based approach layers simple behaviors (such as obstacle avoidance, corridor following, or goal seeking) with priorities. At runtime, the highest priority applicable behavior takes control. These systems were historically popular for robust low level control. In terms of mission representation, they often lack a clear design time script; their "representation" is implicit in the collection of programmed behaviors. Nevertheless, they are sometimes combined with higher level mission scripting for instance, a top level finite state machine may switch between behavior based control modes.

Strengths: Purely reactive control excels in dynamic, uncertain environments because the robot can respond immediately to stimuli without relying on preplanned sequences. Reactive systems are generally robust to variations in the environment; for example, a robot will always avoid an obstacle if one suddenly appears, regardless of any higher level goal. The absence of rigid structure can also contribute to fault tolerance. If one behavior fails, the mission does not necessarily halt other behaviors continue to function. This can be compared to an insect whose other legs continue moving even if one leg becomes stuck, analogous to independent reactive modules continuing to operate.

Limitations: The main drawback of purely reactive systems is their difficulty in guaranteeing the completion of complex missions that require sequential logic or memory of past actions. They may fall into local loops or fail to reach their objectives if not carefully designed. Because there is no explicit high level representation therefore it is also challenging for humans to specify complex missions in this paradigm. As a result, purely behavior based architectures are typically used as complementary mechanisms within structured mission representations, rather than as stand alone mission specification methods.

Use Cases: Low level navigation and real time obstacle avoidance almost always rely on reactive control, such as PID controllers, potential fields, or rule based avoidance algorithms. In terms of mission execution, purely reactive architectures were used in early mobile robots for behaviors such as wandering, following, and simple goal seeking. In modern practice, they are rarely used alone for mission level tasks but are common within hybrid systems. For example, a higher level planner might invoke a reactive behavior such as "wander until the target is found," which is internally implemented as a rule based behavior.

The BDI model is primarily an AI paradigm, but its execution mechanism effectively functions as event driven reactive planning supported by a library of plan templates. Although less common in mainstream robotics literature over the past decade, it remains influential in multi agent system research [60].

Summary of Control Models

In summary, mission control models define the runtime execution semantics for robotic missions. Finite state machines provide deterministic sequencing with simple semantics but can become unwieldy for complex tasks. Behavior trees offer a more

modular and reactive structure, increasingly adopted for complex missions requiring flexibility. Petri net based controllers introduce formal concurrency handling, well suited for synchronized multi robot missions, albeit at the cost of greater design complexity.

Deliberative planners and hybrid architectures enable online decision making and adaptivity, which are essential for open ended missions, but must be paired with execution mechanisms to meet real time and reliability constraints. Reactive, behavior based schemes underpin all of these models by ensuring that robots can respond immediately to environmental events, though they do not in themselves provide a convenient mission level programming interface.

Finally, As noted by Molina et al., purely sequential waypoint scripts are rigid and cannot adapt to dynamic changes or branching logic. This limitation has driven ongoing efforts toward richer and more flexible execution models, including combinations such as planners integrated with reactive control, or state machines augmented with concurrent threads. These hybrid models better address the needs of modern autonomous missions.

3.1.2 Mission Representation as Specification Languages

Mission specification languages are the notations, formats, or tools that enable human designers or higher level software to describe the missions a robot or team of robots should perform. These can take various forms, including textual domain specific languages, markup files such as XML, JSON, or YAML, graphical interfaces where missions are visually composed, and formal logic specifications. Unlike control models, which focus on execution behavior, specification languages concern the representation and encoding of mission requirements, which are later interpreted or compiled into executable form. Over the past decade, the literature has introduced a wide range of mission specification approaches, often tailored to different abstraction levels and user expertise. The main categories and representative examples are reviewed below.

Domain Specific Mission Programming Languages

A number of DSLs have been proposed specifically for describing robotic missions. These languages provide constructs such as keywords, syntax, and data structures that align with the conceptual elements of missions (tasks, actions, events, robots, and so on). As a result, they make it easier to encode complex missions without dealing with low level programming details.

Early efforts included TDL and PRS like agent languages, but more recent examples are particularly noteworthy. TML (Task Mission Language) is a DSL designed for UAV missions within the Aerostack framework [61]. TML uses an XML based syntax that is both machine and human readable, allowing users to define missions in terms of a hierarchical task tree with associated actions and an event handling section for reactive behaviors. The conceptual aim of TML is to provide a logical,

Table 3.1: Comparative View of Mission Representation as Control Models

Model	**Execution Semantics**	**Strengths**	**Limitations**	**Typical Use Cases**
Finite State Machines (FSMs)	Deterministic state transitions based on events or conditions	Simple, intuitive, widely used; good for linear or well structured tasks	State explosion with complex missions, poor modularity, weak concurrency support	Structured missions, industry applications, sequential UAV/UGV tasks
Behavior Trees (BTs)	Hierarchical, reactive traversal with control flow nodes (e.g., Sequence, Selector)	Modular, readable, supports fallback and reactivity, scalable for complex tasks	Needs careful design for memory/state, harder to formally verify, some learning curve	Mobile manipulation, patrol, adaptive missions, robotic games/NPC logic
Petri Nets (PNPs)	Token based concurrency and synchronization through places and transitions	Formal concurrency support, verification friendly, suitable for multi robot coordination	High complexity, harder to design, less mainstream tool support	Cooperative robotics, synchronized team behaviors, robotics research demos
Deliberative Planners	Plan generation and dynamic replanning using symbolic reasoning (e.g., PDDL, HTN)	Goal driven autonomy, flexibility, supports runtime adaptation, high level task planning	Computational cost, needs accurate models, requires integration with execution monitors	Autonomous logistics, planetary rovers, task scheduling, dynamic goal missions
Reactive Architectures	Event driven or rule based reactive behavior arbitration (e.g., subsumption, BDI)	Robust to dynamic changes, fast response, no need for preplanned missions	Lack of memory, hard to ensure goal completion, difficult to express high level logic	Obstacle avoidance, emergency response, behavior layer of hybrid architectures

high level representation of missions that abstracts away low level implementation details the user specifies what tasks to perform and when or under what conditions certain events trigger actions, rather than how to implement them.

As noted earlier, a TML mission is executed by an interpreter that follows the task tree in a depth first manner while reacting to events to deviate from the sequence when necessary [58]. Thus, TML serves as a prime example of a mission specification language tightly integrated with a runtime control model in this case, a custom reactive planner or executive within the Aerostack framework.

Another example is BDL, a DSL for defining missions of teams of robots with reconfigurable hardware. BDL was created to facilitate coordination among multiple UAVs and allow flexibility across different sensor configurations. Its syntax and semantics combine elements of UML activity diagrams and Petri nets. A BDL mission is represented as a network of "blocks" (nodes) connected by flow relations, supporting parallel branches and synchronization, much like an activity diagram, while inheriting concurrency semantics from Petri nets.

A BDL mission file models the mission at a high level including patterns for multi robot collaboration and is deployed to a runtime system called DMDE. DMDE contains a mission player that executes the mission by instantiating the specified blocks at runtime. The language is designed to be hardware agnostic and extensible; for instance, BDL supports placeholders for specific robot capabilities that are bound to concrete implementations at runtime, enabling mission reusability across different robot platforms [57].

Beyond TML and BDL, several other DSLs have been proposed. MDL (Mission Description Language) is cited as a declarative, XML based language for defining missions, likely originating from a research project focused on multi robot coordination. There are also POMDP based or policy specification languages, although these belong more to the planning domain. Another noteworthy family is PPR (Plans, Procedures, and Rules) languages used in space robotics for example, NASA's PLEXIL, developed in the mid2000s.

More recently, the PROMISE framework aimed to let users specify high level goals for multiple robots collaboratively, rather than prescribing low level steps. PROMISE introduced a language emphasizing multi robot collaboration and abstract goal descriptions [62]. Experimental studies have shown that such DSLs lower the entry barrier for domain experts who are not traditional programmers. For example, the FLYAQ system enabled non experts such as firefighters to specify drone missions using a high level modeling language, which was then automatically compiled into low level flight plans [63].

Strengths: DSLs offer expressiveness tailored to robotic missions without the verbosity of general purpose programming languages. They often include abstractions for common mission patterns such as patrol, survey, or object delivery and can enforce structural correctness that helps prevent design errors. For instance, TML's task tree ensures that every mission has a well defined beginning and end, and that event handlers are explicitly defined, improving verifiability [58].

Another strength is their readability and usability for end users. Many DSLs em-

ploy intuitive syntax or pseudo natural language. For instance, the PROMIS framework allows missions to be specified directly in terms of high level tasks and teams, aligning with how non programmers conceptualize multi robot missions. Some DSLs, such as BDL, are designed to be domain independent by enabling the definition of custom mission tasks as plug in extensions. Importantly, because DSLs have formal semantics, they support verification and automatic translation. A mission DSL script, for example, can be compiled into a finite state machine or directly into executable robot code, ensuring correctness and consistency between specification and implementation.

Limitations: The effectiveness of a DSL depends heavily on its supporting tools and on the range of missions it can naturally express. Some DSLs are too low level, essentially mirroring general purpose programming, while others are overly abstract and limit user control. Adopting a DSL also requires learning its syntax and semantics, which can be a significant investment especially when the language lacks broad adoption or mature tooling beyond research prototypes.

DSLs can also restrict flexibility: by constraining how missions are represented, they may make it difficult to specify unconventional behaviors that fall outside their abstractions. For example, if a DSL assumes a strictly hierarchical task decomposition, it might be ill suited for dynamic, non hierarchical missions. Furthermore, every DSL must interface with real robot software; mismatches between the language abstractions and the actual hardware capabilities can lead to either oversimplified or repeatedly adjusted mission definitions.

In practice, many mission DSLs have not achieved widespread adoption due to these limitations. The literature often observes that most existing mission specification approaches "focus only on a subset" of desirable features such as usability, domain independence, or extensibility [62]. No single DSL fully addresses all needs, which is why multiple coexist today and why integration with underlying runtime control remains essential an interaction explored in the next section.

Use Cases: DSLs are widely used in advanced research projects and frameworks where missions are sufficiently complex to warrant a dedicated specification language. For example, TML has been used in indoor drone competitions involving reactive mission tasks, such as search and rescue scenarios with moving targets. Teams could quickly adjust missions by editing XML definitions rather than modifying code [58].

BDL has been applied in scenarios involving modular robotic teams for instance, drones with interchangeable sensors or components. Because BDL missions reference abstract capabilities rather than specific hardware, they remain valid even when the configuration changes, making them ideal for re-configurable deployments [57]. Another DSL, Buzz, was designed as a programming language for drone swarms and enables concise expression of swarm behaviors and interactions [62].

Buzz has been used in academic experiments involving large numbers of drones performing cooperative tasks, such as collective surveillance or ad hoc network formation. Using a high level swarm DSL greatly simplifies specification compared to programming each drone individually in C++.

In summary, DSLs excel in describing multi robot or complex single robot missions where higher level abstractions significantly reduce programming effort and potential for error.

Graphical Mission Configuration Tools

In parallel with textual languages, many mission representation approaches use graphical formalisms. These include state charts, flowcharts, behavior tree editors, and custom GUI based mission planners. Graphical tools are a form of mission specification mechanism they represent missions visually and often export a machine readable format such as XML that corresponds to the designed mission. For example, Mission Lab's GUI, developed at Georgia Tech, allowed users to draw the mission's finite state machine graphically by placing behavior icons and connecting them with transitions. This approach is essentially graphical mission programming, where Mission Lab compiles the visual design into executable behavior code [48]. Similarly, Robot ML is a graphical modeling language based on UML/MARTE that enables users to model missions and robot architectures within a unified environment [62]. Robot ML provides a modeling palette for creating state machines and activity diagrams representing robot behavior, and it supports automatic code generation for execution.

Another example is the use of flowchart style mission editors in high level tools. Schwartz et al. As cited in Gutmann et al. presented a mission representation using a flowchart like diagram that is intuitive and easy to construct, essentially allowing users to depict tasks and decision branches visually. The advantage of these diagrams is that domain experts can quickly express mission logic without writing code. However, as noted in the literature, such flowcharts may "lack proper support for concurrency," as they typically enforce single threaded logic unless specifically extended [62].

Graphical tools include standard robotics GUI mission planners such as ArduPilot Mission Planner, DJI Ground Control Station, and QGround Control. Users mark waypoints on a map and define basic commands such as takeoff, go to, or land. The result is a mission script, usually a list of waypoints and associated actions, that the UAV executes. Although not academic innovations in themselves, these interfaces are the defacto mission specification tools in the UAV domain. Their functionality is limited mainly supporting sequential waypoint missions [58]but research tools are increasingly building on them to introduce more advanced capabilities. For instance, FlyAQ provided a graphical interface in which users such as firefighters could draw no fly zones, specify survey regions, and select mission templates e.g., "perimeter sweep", with the system automatically generating a safe flight plan that met those constraints [63]. Another framework, the Mission Definition System for UAV based infrastructure inspection developed within an EU project, uses a GUI for defining missions with points of interest and measurement actions, translating them automatically into executable flight plans [64].

Strengths: Graphical mission representations are highly accessible because they

use visual metaphor ssuch as blocks, arrows, and maps that align with how humans plan tasks. They significantly reduce the learning curve and enable non programmers to design missions intuitively. For complex systems, graphical models, particularly UML based ones, help manage complexity through hierarchical diagrams or multiple abstraction layers (for example, a top level state chart can encapsulate sub state charts). Many graphical tools also integrate simulation and verification features. For instance, developers can step through a state chart to simulate mission logic or apply model checking to validate UML based state machines if formal semantics are provided. When based on standardized formalisms such as UML or SysML, graphical models also enhance communication and coordination among development teams in large engineering projects.

Limitations: A common drawback is that graphical models become unwieldy for large missions a single diagram may not fit on one screen or page, making it harder to manage than text based specifications. In some cases, graphical notations can also be ambiguous or informal; a simple flowchart might omit details required for execution unless the tool enforces completeness. Another limitation is vendor lock in: proprietary GUI tools may store missions in closed formats, hindering export and integration with other autonomy frameworks. In research contexts, some graphical approaches are criticized for being ad hoc and lacking formal semantics, which makes it difficult to reason about mission correctness. Frameworks such as Robot ML attempt to address this by grounding their models in formal UML semantics [62].

Expressing concurrency and complex event logic is another challenge. Many GUI based tools simplify their interfaces by omitting those features, limiting the expressive power of the resulting missions. For example, a standard waypoint interface cannot easily capture conditional logic such as “perform X until condition Y becomes true, then switch to Z.” To address such limitations, advanced graphical tools often integrate scripting components or adopt state chart based modeling instead of simple flow diagrams.

Use Cases: Graphical mission specification is widely used in field robotics where domain experts are directly involved in mission planning or supervision. For example, in disaster response exercises, a commander might graphically assign survey areas to different UAVs on a map effectively defining missions that are translated into executable commands by an underlying system. Graphical representations are also valuable for education and rapid prototyping; students or researchers can create mission state machines or behavior trees visually and immediately test them in simulation or on physical robots.

In industry, model driven development tools such as Math Works State flow or IBM Rhapsody are commonly used to design robot task logic through diagrams that are automatically converted into embedded code effectively serving as graphical mission specifications. Graphical interfaces also play an important role in human robot interaction, where users may define missions by drawing routes or constructing flowcharts on a tablet. The accessibility and immediacy of visual mission design make graphical tools especially suited to rapid iteration in early development stages. With the ongoing improvement of modeling tools and integration with automation

frameworks, increasingly complex missions can now be designed entirely through graphical interfaces.

Markup and Data Formats

Many mission specifications are stored in structured data formats such as XML or JSON. These formats do not constitute mission models themselves but serve as containers for data that describe models defined elsewhere. In practice, mission files typically adhere to a predefined schema, as seen in XML based task languages or waypoint definitions used by UAV ground control systems[61]. Their main strengths lie in machine readability and the ease with which they can be generated, transmitted, and modified. Importantly, these formats provide a means of representation rather than semantics. The mission logic is interpreted and reconstructed at runtime. This separation between format and model promotes modularity and interoperability, enabling missions to be designed visually, stored in standardized files, and executed by general purpose engines.

Strengths: Using standardized data formats makes mission representations easier to parse, validate, and even auto generate. XML schemas, for example, can be used to validate mission files, ensuring that required elements and attributes are present. Engineers are generally familiar with these formats, and they integrate well with version control systems, allowing mission file differences to be tracked and merged like code. Another advantage is scalability: even large and complex mission specifications remain manageable when supported by XML or JSON tooling, such as XSLT transformations or JSON query utilities. For multi robot systems, standardized mission formats in JSON can be shared among heterogeneous robots, as long as each platform knows how to interpret the shared schema.

Limitations: Human readability can decline as mission complexity increases. Deeply nested XML or JSON structures can become as difficult to follow as raw source code if not well documented. These formats also introduce redundancy and can be error prone to edit manually: a missing bracket or closing tag can invalidate an entire file. As a result, such representations are typically used behind the scenes, with users interacting instead through higher level DSLs or graphical interfaces. Another limitation concerns performance: parsing large XML files or executing transformations may add slight overhead, though this is usually negligible compared to mission execution timescales.

The main caution is not to confuse the format with the model. A mission file in XML has no inherent meaning without knowledge of the schema's semantics and its link to an execution model. For example, a tag such as ¡state name="Survey"/¿ has no operational significance unless it is known to represent a state within a finite state machine where Survey triggers specific behaviors or actions.

Use Cases: Structured data formats are ubiquitous in modern robotic mission systems. In ROS based systems, missions are often defined in configuration files for instance, a sequence of waypoints encoded in a YAML list. Web based robot dashboards use JSON to transmit mission plans via REST APIs. In research contexts,

XML is frequently employed to encode complex plans, allowing different planning and execution tools to exchange data efficiently. The Task Description Language proposed by Simmons et al. was text based but can be viewed as an early structured task representation [65].

A notable standardization effort is BT.XML, in which behavior tree libraries define a shared XML notation for representing trees. This has enabled a degree of cross compatibility: a behavior tree created in one tool can be executed in another library if both conform to the XML specification similar to how UML diagrams can be exchanged via XMI.

In summary, markup and structured data formats serve as ubiquitous implementation mechanisms, ensuring that robotic missions can be stored, exchanged, validated, and processed in a uniform and interoperable manner.

Formal Specification Languages

At the highest level of abstraction, mission requirements can be specified using formal languages such as temporal logics or other mathematical formalisms. One widely studied approach uses Linear Temporal Logic, or LTL, and related logics such as cosafe LTL and Signal Temporal Logic to describe missions [66]. For example, an LTL formula might describe a surveillance mission requirement such as Target Detected, meaning "repeatedly detect the target over time," which ensures persistent search. Another example could be not Alarm until Battery Low, then eventually go to Charge Station, meaning "if no alarm occurs until the battery is low, the robot must eventually go to the charging station." These logic based specifications are declarative. The user specifies what conditions the mission must satisfy over time rather than how to achieve them. Formal methods such as model checking or reactive synthesis can then generate an automaton or control strategy that guarantees the specification is met, if possible.

This process effectively automates mission programming and provides correctness guarantees. KressGazit et al. pioneered this approach in 2009 by converting high level LTL mission specifications into hybrid controllers for robotics [59, 67]. Research has continued along this path, improving scalability and user accessibility. Recent developments include structured English like patterns that compile into LTL formulas [68].

Another formalism used for mission specification is Temporal Action Logic, or TAL, developed by Doherty et al. [62]. TAL describes missions in a high level logic that specifies actions and their effects over time. It can generate task plans that account for complex temporal constraints. Researchers have also developed property specification patterns tailored for robotics. These serve as reusable templates such as "Eventually do A" or "Do B until C." They help users create formal mission specifications without requiring deep expertise in temporal logic [68].

Strengths: The key advantage of these approaches is their clarity and mathematical precision. Formal languages provide unambiguous specifications and allow formal verification or automatic synthesis of correct controllers. This capability is

vital for safety critical domains such as autonomous driving or space robotics, where the system must never enter unsafe states and must achieve its objectives when possible. Formal specifications are platform independent. They describe missions using abstract conditions and events that apply across different systems. In multi robot scenarios, temporal logic can capture collaboration and sequencing requirements, ensuring coordination between robots by design.

Limitations: Writing formal specifications requires specialized knowledge. Even with pattern libraries and structured English tools, users must still think in terms of temporal logic, which many find unintuitive. This can lead to incorrect specifications that produce valid but unintended behaviors.

Computational complexity is another concern. Synthesis and verification can become expensive for large specifications or complex environment models. Advances in realizability analysis and abstraction techniques have improved scalability, but challenges remain.

Integrating synthesized controllers with robot systems can also be difficult. Controllers often take the form of automata that must interface with low level robot actions and sensors. This requires explicit mapping between logical propositions and real world signals. Bridging discrete logic with continuous motion planning adds another layer of complexity.

Overall, formal languages are extremely powerful for describing high level mission logic, but they usually form part of larger hybrid systems rather than acting as standalone solutions.

Use Cases: Formal mission specification techniques are used in various domains. In autonomous driving, temporal logic can encode traffic rules and mission objectives, enabling automated synthesis of decision making modules. In robotics competitions such as DARPA challenges, teams have used temporal logic to define complex behaviors and synthesize compliant controllers.

Temporal logic is also applied in adaptive or safety critical missions. For example, a UAV required to maintain continuous communication can have this property defined in logic and automatically enforced by the synthesized controller.

A practical example is multi robot persistent surveillance. In such missions, one can specify that “each area A, B, and C must be visited repeatedly by some robot, and no area may remain unvisited for more than T minutes.” The corresponding temporal logic formula can then generate a patrol strategy that satisfies those conditions.

Recent research integrates formal specification with higher level mission description languages. For instance, a DSL may allow users to attach LTL constraints to mission plans so that monitors or enforcement behaviors are automatically generated. Other approaches compile structured mission descriptions such as “go to region 1 or region 2, avoid region 3, then return home” into optimization or automata based models. This bridges declarative mission logic with motion planning and execution in an automated, verifiable way.

Strengths and Weaknesses: Comparative View of Languages

Mission specification languages vary primarily in their design objectives, intended users, and level of maturity. Studies highlight an inherent trade off between usability and expressiveness. Graphical and markup based languages are more accessible to non programmers but tend to be restricted to particular mission types or platforms. In contrast, programming centric solutions, such as Java or script based languages for swarm missions (e.g., KARMA and Buzz), offer greater expressive power at the cost of requiring advanced development skills. XML based languages like TML and MDL simplify mission definition but limit flexibility. To balance these factors, intermediate frameworks such as PROMISE seek to offer better mission specification while minimizing user effort. In terms of strengths [62]:

- DSLs and markup languages excel in readability and rapid development. They enable domain experts, such as field operators, to specify missions through simple forms or scripts without writing software code.
- Graphical tools make mission logic immediately understandable and help detect design errors early. For example, missing connections or unreachable states can be identified at a glance.
- Formal specification languages provide precision and eliminate ambiguity, offering strong guarantees of correctness.

In terms of limitations [62]:

- Many DSLs and tools are tied to specific robotic platforms or middleware. Missions designed in those environments may not transfer easily to other systems, although general frameworks such as RobotML attempt to address this issue.
- Graphical models that lack concurrency support cannot effectively represent multi threaded mission behavior.
- Formal methods offer theoretical rigor but face practical challenges. Integrating formal specifications with real time control and unpredictable physical environments remains difficult.

Typical Use Cases: The choice of mission specification language often depends on the target application [62].

- For teams of heterogeneous robots performing complex collaborative tasks, languages oriented toward multi robot coordination such as BDL or Buzzare typically used. These may be complemented by temporal logic specifications that define global mission goals.
- For single UAV missions that follow structured workflows, state charts or behavior trees are more common. These can appear in graphical tools or XML based mission planners such as typical ground control station (GCS) software.

- For safety critical missions, such as those in space exploration or medical robotics, formal specification and analysis approaches are often mandatory. In such contexts, languages supporting model checking or the synthesis of provably correct controllers are preferred.

Table 3.2: Comparative View of Mission Representation as Specification Languages

Model	**Execution Semantics**	**Strengths**	**Limitations**	**Typical Use Cases**
Domain Specific Languages (DSLs)	Translate to or interpreted by underlying control models (e.g., FSM, BT, interpreter)	Readable, task oriented, enables rapid mission definition by non programmers	Often tied to specific frameworks, limited flexibility for unforeseen behaviors	UAV inspection missions, indoor drone tasks, robotic team coordination
Graphical Tools (e.g., State charts, Flowcharts, BT editors)	Visual representations compiled into runtime executable models	User friendly, accessible for domain experts, supports debugging and simulation	Poor scalability for large missions, limited formal semantics (unless standardized)	Prototyping, educational robotics, mission sketching for UAV GCSs
Markup / Structured Formats (XML, JSON, YAML)	Serialization of mission models interpreted by control engines	Machine readable, supports validation, tool integration, widely used	Poor human readability for complex logic, semantics depend on schema or control model	ROS based mission configuration, BT/FSM serialization, GCS mission files
Formal Specification Languages (e.g., LTL, TAL)	Automatically synthesized into controllers or verified against models	Provides correctness guarantees, precise mission description, platform independent	Requires expertise, computational overhead, harder integration with low level systems	Safety critical systems, adaptive missions, verification of temporal goals

3.2 Component Orchestration and System Initialization

3.2.1 Boot Time and Process Level Initialization

Embedded Linux boot up in UAVs and robots follows a fixed sequence: a boot loader hands control to the Linux kernel, which then launches an init process to start user space components. Early approaches use simple static scripts or run levels to start mission critical processes at boot, often with hard coded parameters or environment variables for configuration. For example, a UAV's autopilot software and support modules may be invoked via initialization scripts (e.g., /etc/rc.local or System V init) with commandline arguments or environment variables specifying modes and sensor interfaces. Such command based startup is straightforward but inflexible each process starts with predefined settings, and any change requires a reboot or manual intervention. In practice, this static model yields a system architecture that rarely changes after initialization. Configuration parameters (e.g., network addresses, sensor IDs) are typically passed in at launch either via environment variables or a global parameter server to customize each component's behavior [69]. This approach ensures a predictable boot sequence and initial state but lacks dynamic adaptability.

Modern embedded Linux distributions for robotics increasingly employ dependency aware init systems (e.g., systemd) to manage boot time services. These enable parallel startup and explicit dependency ordering, which can reduce boot delays and ensure prerequisites (drivers, network services) are ready before higher level processes start. However, the fundamental paradigm remains static: a fixed set of processes is defined to launch at boot. Academic literature notes that such static startup mechanisms assume all required components are known in advance and remain available. For instance, ROS 1 (Robot Operating System) uses static launch files (XML) that enumerate which node processes to start, along with their parameters and inter process remapping settings. These launch descriptions can include conditional logic and nested includes but ultimately describe a predetermined configuration of the system at startup. Timperley et al. (2022) observe that ROS launch files result in architectures that are largely fixed after boot components and connections do not typically change at runtime unless manually relaunched [69]. This static initialization paradigm offers simplicity and transparency: system engineers can carefully script the exact sequence of initializations, ensuring that sensors, actuators, and communications are brought up in the correct order with known parameters. The approach is also lightweight, with minimal orchestration overhead, which suits resource constrained single board computers onboard UAVs.

Limitations: The static boot time model struggles as system complexity grows. Because processes are launched in a predefined manner, dependency management is manual or relies on basic ordering rules. Missing or late starting dependencies (e.g., a sensor driver) can cause failures that the init process cannot intelligently

resolve. There is little builtin resilience if a critical process crashes, traditional init scripts will not restart it unless a watchdog or ad hoc looping script is used. Even frameworks like ROS 1 launch offer only rudimentary supervision (a respawn flag to restart a node if it dies). There is no global oversight of system health in static init: the system "fires and forgets" processes at boot. As Portugal et al. (2025) note, ROS 1 (a prototypical static launch system) lacks support for real time operation and fully distributed multi robot deployments [70]. In essence, early stage initialization mechanisms optimize for a predictable startup but not for adaptivity or fault tolerance. They assume a stable runtime environment, which in safety critical autonomous systems is a significant drawback. If a component hangs or a new module needs to be launched dynamically, the static framework has no general solution. Scalability is also limited adding more processes or inter process dependencies increases boot configuration complexity and boot time, with no runtime load balancing. These limitations set the stage for architecture level orchestration approaches that introduce modularity and runtime dynamism.

3.2.2 Architecture Level Orchestration Approaches

As autonomous systems grew more complex, researchers moved toward service oriented and middleware centric architectures. The goal was to organize software components beyond simple boot time scripts. In a service oriented UAV software system, functionality is decomposed into loosely coupled services with well defined interfaces. These services often run as independent processes or even on separate nodes, communicating through middleware. This mirrors the Service Oriented Architecture principles used in enterprise systems, adapted for robotics. NASA's ICAROUS framework for UAS is explicitly service oriented. It separates autonomy functions into services for conflict detection, resolution, and mission management. These services interact through a common interface using monitor–resolver patterns to enhance reliability and modularity [71].

Each capability such as path planning, vision processing, or flight control is treated as an independent service. This design makes UAV software more extensible and fault tolerant. A failing service can be restarted or replaced without crashing the entire system. Magyar et al. (2015) note that robotic software platforms were introduced to handle common tasks such as state management, communication, and synchronization, allowing developers to focus on innovation [72]. Middleware and service oriented frameworks thus take over the "glue" logic once managed by static boot scripts. Middleware centric architectures now dominate UAV and robotics software. Middleware provides an abstraction layer between hardware or operating systems and higher level software components. It manages message passing, data marshalling, and component life cycles.

The Robot Operating System (ROS) is the best known example. Described as a robust middleware for integrating robotic modules [70], ROS popularized the publish/subscribe pattern in robotics. Each software module, or node, publishes data or offers services. Other modules subscribe or call them, all mediated by the ROS mid-

dleware layer involving the master node and transport system [73]. This approach decouples components and enables distributed operation across multiple processes or machines while maintaining a unified architecture. ROS handles name registration, message typing, and data transport, freeing developers from writing custom networking code [70].

Architecturally, this is a distributed asynchronous system rather than a monolithic program. A UAV's perception, planning, and control modules can run as separate nodes connected by topics for example, the camera node publishes images that a vision node subscribes to. According to Portugal et al. (2025), such middleware creates a critical abstraction layer that integrates diverse hardware and software components. Over the past two decades, many middleware systems such as OROCOS, MOOS, and YARP have been developed, but ROS remains the most widely used [70]. Its success illustrates the dominance of middleware centric orchestration: instead of a single initialization script, architecture now defines interacting components managed by middleware for communication and synchronization.

Architecture level orchestration adds mechanisms for configuration, dependency management, and limited runtime supervision. These improve on static boot. Configuration has shifted from shell scripts to structured metadata files for example, ROS launch files or YAML configurations that list parameters and connections. Timperley et al. (2022) describe that ROS 1 launch files let developers declare which nodes to start, their parameters, and any communication remapping [73]. This formalization helps manage dependencies. The launch system ensures required services such as the ROS master or a database node start before dependent components. It can also verify that critical environment variables or parameters are set, reducing mis configurations. ROS Discover by Timperley et al. analyzes ROS launch configurations, checking for missing connections or mismatched parameter types [73]. This highlights how dependency wiring is essential in such systems.

Compared to raw boot scripts, middleware based configurations are more declarative and scalable. Complex UAV software stacks with many nodes can be managed through hierarchical launch files and plugin descriptions. In larger systems, a ROS launch can even include another robot's configuration to form a multi robot system. Despite these advances, middleware based systems still have limitations. ROS 1 primarily relies on static launch time composition. Once running, the configuration cannot easily change adding a new node or rerouting data must be done manually. Most frameworks also lack autonomic supervision. The middleware routes data but does not monitor node health or replace failed components unless the developer adds a supervisor process. ROS 1 has additional weaknesses, including no real time guarantees and limited peer to peer coordination without a central master [70].

As the number of services increases. For example, in UAV fleets or sensor networks, service oriented design alone is insufficient. Systems need runtime orchestration to decide which services run where and how they recover from faults. These challenges have driven interest in dynamic, cloud, and container based approaches. Even so, the service and middleware paradigm was a major milestone. It introduced modularity, abstraction, and reuse. Components could be developed and tested in

isolation. Then integrated through middleware, its a clear improvement over static boot. Magyar et al. (2015) compare different robotic middleware systems, showing how they simplify development by managing communication and enabling software reuse [72]. The ROS ecosystem in particular demonstrates this through its library of interoperable modules for mapping, control, and more.

A related evolution is the adoption of containerization and microservices for UAV software deployment. While ROS nodes already provide modularity, researchers have explored packaging components as containers to isolate dependencies and ensure consistency. A container based deployment treats each major function as a microservice that can run on a UAV, at the edge, or in the cloud. This approach addresses portability and dependency conflicts in complex robotics software. For example, a UAV may run a vision module inside a Docker container onboard, while a heavy AI or database module runs remotely. Containers seamlessly interconnect in operation. They also simplify integration of heterogeneous environments, allowing different library or OS versions to coexist. Lumpp et al. (2023) highlight containerization as a best practice in robotics because of its portability and isolation benefits [74]. It allows packaging ROS nodes or other programs with all required libraries crucial for reproducibility when deploying on multiple drones. Containerization aligns with service oriented design since it provides a practical implementation method for services. Its true value appears in the next stage: using container orchestration to manage and scale these services dynamically at runtime.

3.2.3 Runtime Orchestration and HealthAware Supervision

The current state of the art is moving toward dynamic runtime orchestration. In this paradigm, the system can supervise, start, stop, and relocate components during operation. It can adapt to changing conditions or recover from faults automatically. This marks a major shift from static, architecture centric paradigms. Instead of a fixed set of processes or services, systems now rely on a managed set of containers or tasks controlled by an orchestration platform in real time. In resource constrained edge environments such as UAV onboard computers or remote robots, adopting cloud derived orchestration technologies like Kubernetes presents both challenges and opportunities. Current research is focused on bringing container orchestration to the edge for autonomous systems. Lightweight frameworks such as K3s and KubeEdge have been demonstrated on UAV platforms to coordinate multiple software components and even clusters of drones [75]. The goal is to apply key features of cloud orchestration automated deployment, scheduling, monitoring, and self healing to robotics. A clear example is presented by Awada et al. (2022), who integrated Kubernetes into a federated aerial edge computing system. Earlier multi drone systems could not efficiently share resources. Awada and colleagues proposed AirEdge, a system that treats a fleet of drones and edge servers as a unified resource pool managed from a federated control plane. Using a lightweight Kubernetes tool called Kubermatic, the system dynamically deploys tasks across UAVs and ground edge nodes. It determines where each software component should run to meet per-

formance goals [76]. If one drone becomes overloaded, tasks are offloaded to another drone or an edge server in real time.

Kubernetes orchestration provides elasticity by scaling services up or down. It also offers builtin self healing, when a containerized component fails. The orchestrator restarts or redeploys it on another node. Zhang et al. (2023) demonstrated this approach by integrating K3s, a minimal Kubernetes distribution, with ROS 2 in a multi robot system. Their orchestrator monitored node health and recovered failed nodes with minimal downtime, improving robustness [75]. Compared with static systems that require human intervention or a full reboot after a crash, an orchestrated system reacts immediately to failures. Another important aspect of runtime orchestration is health aware scheduling and supervision. Orchestrators not only recover from failures but also proactively manage the system based on health metrics and performance constraints. In safety critical autonomous systems, this includes monitoring real time deadlines, CPU or memory usage, and sensor or actuator health. The system can reconfigure itself when conditions become abnormal. Lumpp et al. (2023) introduced RTKube, an extension of Kubernetes that adds real time awareness for autonomous robots. Standard orchestrators cannot manage real time tasks such as flight control loops, which is a major limitation. Robotic systems often combine both hard real time control and best effort computation [74]. RT Kube adds a monitoring layer that detects missed deadlines at runtime. It can migrate or throttle lower priority containers to maintain timing guarantees. In experiments on a mobile robot, when a high priority control task lagged due to CPU contention, the orchestrator automatically relocated perception tasks to another node, freeing resources [74]. This health aware approach goes beyond fault tolerance. It continually optimizes system performance and reliability. The system ensures that each component not only runs but also meets its quality of service requirements such as timing or accuracy. If not, it reconfigures by restarting, migrating, or scaling to restore acceptable performance. For UAV and robotic platforms, runtime supervision can also include physical health metrics such as battery level, sensor status, or communication quality. Although most research has focused on computational orchestration, some studies combine it with system health management. A supervisory controller, for example, may shut down or restart noncritical services when battery power is low or temperatures are high. While UAV specific implementations remain limited, interest in this direction is increasing.

Robotics middleware is also converging with orchestration frameworks. ROS 2 has introduced managed life cycles for nodes, allowing each node to transition between configured, activated, and deactivated states under external control [70]. This enables higher level orchestrators for either ROS processes or container managers to safely bring down or restart system components. ROS2 also supports launching nodes inside containers and integrating with external orchestration tools. Portugal et al. (2025) note that ROS2 was designed to overcome ROS1's limitations in distributed deployments. Its life cycle management and managed launch features enable better supervision in multi robot systems [70]. Runtime orchestration provides clear benefits. It enhances fault tolerance through automatic recovery, improves

scalability by enabling dynamic component management, and optimizes resource use by scheduling tasks where capacity is available. These advantages have been proven in experimental platforms. For instance, orchestrating a vision algorithm across a drone swarm via edge computing reduces latency and maintains operation even when drones drop out [76]. Orchestrators such as Kubernetes include valuable mechanisms like health checks, replication controllers, and load balancers. Adapted for edge deployment, they address many limitations of earlier static or middleware approaches. However, challenges remain. Full Kubernetes orchestration is too resource intensive for most UAVs with limited CPU and memory. This has led to lightweight variants such as K3s, MicroK8s, and KubeEdge. In KubeEdge, the control plane runs in the cloud while a lightweight agent operates on the robot. Even with these optimizations, orchestration introduces nontrivial overhead. Careful tuning is required to avoid wasting computational resources. Network unreliability adds another challenge. Orchestration frameworks generally assume stable communication, but UAV swarms often experience intermittent connectivity. Awada et al. (2022) addressed this by implementing edge level federation so that orchestration decisions can be made locally. Their scheduling algorithm also considered drone mobility and availability [76]. Real time performance is another concern , As Lumpp et al. (2023) point out, standard orchestrators cannot ensure strict timing guarantees, making extensions such as RTKube essential [74]. Finally, dynamic orchestration increases system complexity. The orchestrator itself acts as an additional OS like layer. Ensuring that this layer does not become a single point of failure or a source of software errors remains an open research issue. Dependable robotics literature identifies this as a key challenge for safety certification. In summary, runtime orchestration and supervision represent the latest stage in the evolution of initialization mechanisms for autonomous systems. The field is moving from static, predefined startup sequences toward adaptive systems that can reinitialize or reconfigure components on the fly.

This progression is reflected in recent research. Early 2010s studies focused on boot time optimization and static scheduling. The mid 2010s introduced middleware and service frameworks, still largely static after startup. From the late 2010s onward, containerization and cloud native techniques began dominating robotics [74, 75]. The long term vision is a self managing UAV or robotic system, one that not only initializes at boot but continuously orchestrates its components during operation. Such systems aim to maintain optimal performance, reliability, and safety without human intervention.

Table 3.3 Evolution of system initialization paradigms in embedded/autonomous systems, from static boot up to dynamic orchestration. Each paradigm builds on the previous, increasing flexibility and autonomy at the cost of greater complexity. Sources: boot time characteristics from Timperley et al. 2022; middleware patterns from Magyar et al. 2015 and Portugal et al. 2025; dynamic orchestration features from Lumpp et al. 2023 and Awada et al. 2022.

Over the past decade, research in autonomous robotics and UAV systems has followed a clear trajectory in how initialization and orchestration are handled. Early

Table 3.3: Comparison of System Initialization and Orchestration Paradigms

Paradigm	**Mechanism and Scale**	**Config and Dependencies**	**Runtime Supervision**	**Strengths**	**Limitations**
Boot-Time Static Initialization	Single-host OS startup. Fixed processes launched at boot on one device.	Static scripts or launch files. Dependencies defined manually or via init order.	Minimal supervision. Optional process restart via scripts.	Simple and predictable. Low overhead. Suitable for small systems.	Inflexible after boot. No dynamic reconfiguration. Poor scalability.
Architecture Level Orchestration	Middleware-based coordination of components. Supports single or multiple hosts.	Declarative launch files (XML, YAML). Middleware manages discovery and naming.	Basic monitoring of connections. Node restart often external or manual.	Modular design. Improved reuse and integration. Handles moderate complexity.	Largely static at runtime. Limited fault handling. Integration complexity increases.
Runtime Dynamic Orchestration	Container-based orchestration with a control plane. Supports distributed nodes.	Configuration-as-code. Dependencies declared and resolved dynamically.	Active health checks. Automatic restart and rescheduling on failures.	High resilience and adaptability. Supports complex missions and updates.	Higher overhead and complexity. Real-time guarantees are challenging.

systems relied on static, boot time initialization, a single shot setup that was simple and reliable but not scalable. As system complexity increased, middleware centric and service oriented architectures emerged. These allowed multiple components to launch and integrate more systematically, though the runtime configuration often remained static. More recently, inspired by cloud computing, researchers have begun exploring dynamic orchestration techniques on robotic platforms. These enable runtime supervision, self healing, and resource aware adaptation. This evolution reflects the growing demand for autonomy and robustness.

Literature from 2015 to 2025 highlights this trend showing a rise in containerization, edge orchestration frameworks, and health monitoring supervisors. Despite progress, open challenges remain. Issues such as real time assurance, network reliability, and computational overhead still require further investigation. Experimental solutions like RT Kube address some of these challenges but remain in early stages of adoption [74]. In summary, system initialization in autonomous platforms has evolved from a one time boot procedure to a continuous orchestration process. Each paradigm retains relevance. Static frameworks remain suitable for small or deeply embedded systems. Middleware based launch systems such as ROS continue to dominate industrial and research applications. Dynamic orchestration is beginning to shape deployed systems, especially in resilient multiUAV networks and autonomous driving fleets. Each stage has contributed essential building blocks from simple initialization and configuration passing to structured launch descriptions and finally to full orchestration with runtime feedback. Together, these advances form the foundation for future autonomous platforms that can both start themselves and maintain operation under adverse conditions. The transition toward dynamic health aware orchestration is still underway. Current studies offer promising prototypes and case studies, but further work is needed in longterm field validation and standardization. The robotics community is moving beyond static initialization toward systems capable of orchestrating and supervising themselves in real time autonomous robots [74, 76].

3.3 Monitoring Techniques for Embedded and IoT Systems

Embedded and Internet-of-Things (IoT) systems operate in environments where crashes or hangs can have serious consequences. Section 3.2 of the thesis System Health Monitoring for Autonomous MAV System reviews research on diverse monitoring techniques designed to keep such systems reliable. The approaches span from reflective frameworks that observe software at run-time to hardware-supported mechanisms. Each technique brings unique strategies for early fault detection, recovery and performance optimisation. The following sections summarise these techniques without reproducing the original wording.

Run-time Reflection Framework

Run-time reflection frameworks provide instruments for logging, monitoring and diagnosing software behaviour. They offer real-time visibility into system activities, enabling quick identification of failures using dedicated monitoring components. Engineers specify system behaviours and timing requirements with structured assertion languages such as SALT, which support the generation of monitors that verify properties at run-time. Model-based diagnosis complements this by using descriptions of a system and observed outputs to infer possible causes of failures. Together, reflective frameworks and diagnostic techniques foster rapid fault detection and facilitate strategies for recovery or detailed logging when anomalies occur [77].

Component-Based Model

The component-based model treats software as a collection of self-contained modules with well-defined interfaces. Components communicate via these interfaces and can emit events when certain conditions arise. A monitoring framework integrates with the system to listen for these events, processing them according to predefined rules. This processing may involve logging events, taking corrective actions or notifying users and other components. By capturing events in real time, the framework gathers information about component health and behaviour; this data supports run-time adjustments, error identification and performance analysis. Event-based monitoring in component-based systems therefore enables the detection of failures and timely corrective measures while preserving modularity [78].

Model-Driven Architecture (MDA)

Model-Driven Architecture blends rule-based and model-based reasoning to classify system behaviour. In this approach, a set of rules defines normal behaviour; deviations from these rules signal abnormalities. Concurrently, abstract component models capture expected interactions and dependencies; comparing observed behaviour to these models reveals discrepancies. Combining both techniques yields a more comprehensive monitoring strategy that enhances anomaly detection. The integration of rules and models allows systems to detect and classify unusual behaviour with greater accuracy, providing a powerful tool for embedded-system reliability [79].

OFMspert Architecture

The OFMspert architecture assists operators in managing complex systems such as satellite control centres. It interprets operator actions, infers intentions and responds in real time using a blackboard modela shared data space where various program parts write information that others can read. For error detection, OFMspert monitors telemetry data (raw state data). When abnormal telemetry signals appear, the architecture automatically launches a troubleshooting process to determine causes

and suggest solutions. By acting as a vigilant assistant that continuously monitors system performance, OFMspert reduces the need for manual intervention and shortens response times [80].

Distributed Component-Based Software Architecture

Distributed component-based architectures aim to build real-time applications that span multiple platforms and languages. Implementations often rely on middleware such as the Common Object Request Broker Architecture (CORBA) to enable communication. Each distributed component periodically sends status messages to a central monitoring entity; if messages are absent beyond a threshold, corrective actions handle failures or unavailability. An integration tool helps compose components visually, generate adapters, adjust properties and embed real-time monitoring and scheduling features. This tool thereby facilitates fault tolerance and real-time features in distributed applications [81].

Software Watchdog Service

The Software Watchdog architecture is designed for automotive safety systems to detect component failures. It integrates three main units: a heartbeat monitoring unit, program flow checking unit and task state indication unit. The heartbeat unit verifies that software components (runnables) execute on schedule by tracking their aliveness and arrival rates. The program flow checking unit ensures the correct sequence of execution, detecting deviations indicative of program errors. The task state indication unit logs errors and produces reports to derive error states for tasks and applications. Results from these units are integrated to enable global fault detection and to trigger response actions. The watchdog architecture has been incorporated into the EASIS platform for integrated safety systems in vehicles, underscoring its relevance to automotive reliability[82].

Heartbeats Framework

Self-aware computing systems adjust their behaviour to meet goals despite changing conditions. The Heartbeats framework provides a way for applications to monitor and communicate their performance and objectives to external observers. Systems employing this framework can dynamically allocate resources and adapt to power, performance and metering challenges across embedded, mobile and high-performance environments. Complementary tools such as Smartlocks use reinforcement learning to optimise synchronisation in multicore systems. Experimental results show that these mechanisms enable applications to articulate goals, while system services evaluate goal fulfillment and apply adaptive strategies to adjust the architecture, operating system and algorithms in real time. Such adaptability enhances performance, reliability and resource efficiency [83].

Integrated Event-Driven Monitoring Approach

An integrated event-driven monitoring approach embeds specialised code within the application to generate events, handle errors and monitor security and performance. These hooks track execution in real time, enabling immediate detection and handling of operational issues. The method employs a hybrid strategy that couples software instrumentation with hardware probes, combining detailed software monitoring with minimal hardware intrusion. When the monitoring system detects a failed component, it initiates an immediate restart or recovery, reducing downtime and improving system reliability. By blending software and hardware insights, this approach delivers comprehensive oversight without significantly impacting performance [84].

Cross-Layer Framework

The cross-layer framework targets monitoring in semiconductor products. It combines embedded instrumentation with the IEEE 1687 IJTAG standard to provide organised data transport and an asynchronous emergency signalling system for rapid fault detection. Software components such as a Fault Manager and an Instrument Manager respond to fault events, manage diagnostics and handle fault-related information. Status registers like FCX flags aid in fault detection and classification, while a Health Map offers a real-time view of system health to assist the operating system in scheduling. The framework goes beyond traditional fault-tolerance techniques by localising, diagnosing and classifying faults promptly, ensuring reliable operation even as fault rates increase [85].

Reliability Microkernel Framework (RMK)

The Reliability Microkernel Framework enhances the dependability of Linux applications by inserting a special kernel module. This framework comprises pins interfaces to the system and hardware. And modules tuned to detect and fix errors in specific applications. RMK uses processor features and operating system interfaces to create pins that underpin application specific reliability strategies. It can monitor unresponsive programs, save application state periodically and restore it quickly after a crash. Its adaptability allows configuration on the fly and compatibility across different platforms; self-checking features help identify and resolve system and application issues, thereby increasing overall system reliability [86].

Monitoring techniques for embedded and IoT systems range from reflective software frameworks and event driven architectures to cross layer hardware software hybrids. Each method reviewed in Section 3.3 provides tools to detect faults promptly and maintain system health whether by analysing logs, capturing events, blending rules with models, or inserting hardware instrumentation. The diversity of these approaches illustrates that reliable system design often requires combining multiple strategies ensuring that failures are detected early, diagnosed accurately and mitigated effectively refer Table 3.4.

Table 3.4: Overview of Discussed Monitoring Techniques

Architecture	**Technique**	**Details**
The runtime reflection framework [77]	Logging, Monitoring, Diagnosis, Mitigation, Runtime Verification	Structured Assertion Language (SALT) is used for specifying the expected behaviour of the underlying system
Component based model [78]	Event based monitoring, polling, RPC	Components provide events that can be captured and processed by the monitoring framework
Model Driven Architecture (MDA) [79]	The System State Monitor (SSM) collects runtime data from sensors, actuators, and software components to continuously observe and monitor the system's behaviour	Combination of rule based reasoning and model based reasoning techniques is used to identify the normal or abnormal behaviour
OFMspert architecture [80]	Intent Inferencing, Opportunistic Reasoning, Blackboard Assessment	The blackboard in OFMspert architecture represents the operator's current state and system status, storing goals, plans, tasks, and actions
Distributed component based software architecture [81]	Each component periodically sends status message to a central monitoring entity	If a signal is not received within the expected time frame, appropriate actions are taken to address failures or unavailability
Software Watchdog for automotive safety systems [82]	Heartbeat Monitoring Unit, Program Flow Checking (PFC) Unit, Task State Indication (TSI) Unit	Different monitoring techniques are integrated with the main platform for enhancing software components' safety and reliability
Heartbeats Framework [83]	Application Heartbeats Framework and Smart locks which is a self aware synchronization library using reinforcement learning for multicore system performance optimization	Heartbeats technique and a synchronization library are utilized to achieve self aware systems
Event Driven Monitoring Approach [84]	Uses event emitters, profiling code, error handlers, logging, tracing mechanisms, assertions, performance/security monitoring hooks, and custom probes	Emphasizes real time tracking and dynamic system recovery by detecting faults promptly, boosting system robustness and reliability
Cross layer framework [85]	Embedded Instrumentation, Asynchronous Emergency Signaling, Fault Manager (FM)	Through the techniques described, the framework aims to achieve proactive fault identification, efficient management, and uninterrupted functionality
Reliability Microkernel Framework (RMK) [86]	RMK Pins, RMK Modules, Application Hang Detection, OS Hang Detection, Transparent Application Check pointing	RMK emphasizes on demand configuration and portability, allowing for tailored reliability techniques for specific applications.

4 Conceptualization of Mission Scheduling Component

4.1 System Overview

4.1.1 Use Case

Within the context of this thesis, the primary use case focuses on the mission scenarios, as illustrated in Figure 4.1. Table 4.1 categorizes the defined missions based on their decision making complexity. Missions 1 to 4 represent scenarios with elementary decision logic and do not rely on image-based perception. Missions 5 to 8 introduce higher behavioral complexity by incorporating motion decisions triggered after reaching specific waypoints, while still excluding visual detection tasks. Missions 9 to 11 extend the mission model by integrating image detection capabilities. However, their control flow remains condition based rather than adaptive. Mission 12 advances this paradigm by exhibiting adaptive decision-making behavior, whereas Mission 13 further extends the decision logic through iterative processes. For this thesis a single mission is selected from each decision making class. Proposed SCH software component is tested on mission 3, 6, 11, 12, and 13.

Table 4.1: Defined Autonomous Missions [17]

Mission	**Waypoint**	**Decision Making**	**Image Processing**
1–4	4–6	Basic	None
5–8	8–34	Complex	None
9–11	6–12	Conditional	Insulator detection
12	6	Adaptive	Insulator and damage detection
13	6	Iterative	Insulator and damage detection

Collectively, these scenarios provide a foundation for the design, development, testing, and evaluation of the proposed architecture. They also demonstrate its adaptability across a wide range of mission applications with minimal or no required modifications.

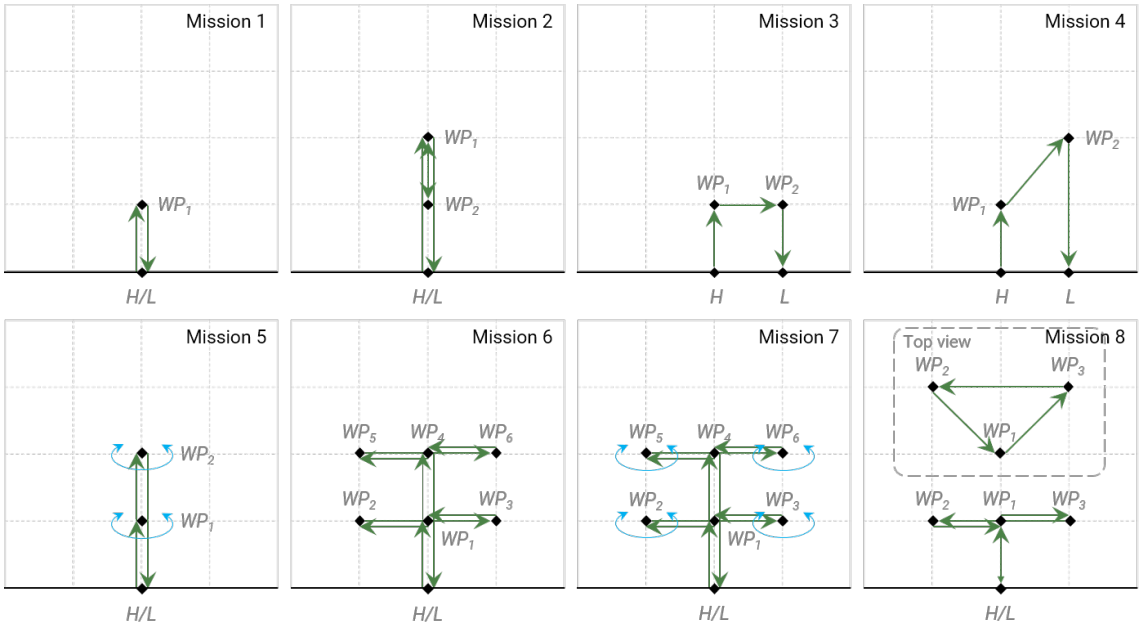


Figure 4.1: Simple autonomous missions [17]

4.1.2 Use Case Description

Mission3

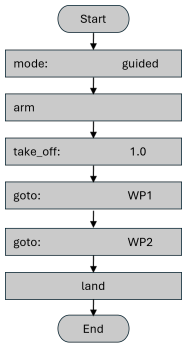


Figure 4.2: Mission 03 flow chart

Mission3 4.2 represents a straightforward, sequential navigation scenario designed to validate basic system initialization and waypoint execution behavior. Upon mission start, the UAV transitions into guided mode, enabling external command control. Once the mode is set, the vehicle is armed, followed by an automated take-off to a predefined altitude of 1.0 m, ensuring a controlled and stable ascent. After reaching the target altitude, the UAV navigates autonomously to the first predefined waypoint (WP1) and subsequently proceeds to the second waypoint (WP2).

These navigation steps are executed in a strictly linear manner without any conditional branching. Upon successfully reaching the final waypoint, the mission concludes with a controlled landing sequence and after which the mission terminates. Mission3 serves as a baseline scenario for validating fundamental system behaviors, including component initialization, command sequencing, and waypoint execution. Its deterministic structure makes it particularly suitable for testing the reliability of SCH and control interfaces under nominal operating conditions, without introducing adaptive decision making mechanisms.

Mission6

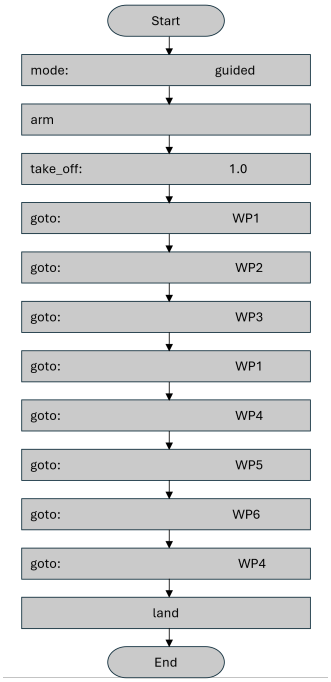


Figure 4.3: Mission 06 flow chart

Mission6 4.3 represents a structured waypoint based navigation scenario that introduces increased operational complexity through extended traversal paths and waypoint revisits. The mission begins by transitioning the UAV into guided mode, allowing externally controlled navigation commands. After successful mode configuration, the vehicle is armed and performs an automated take off to a predefined altitude of 1.0 m, ensuring a stable and controlled ascent. Following take off, the UAV executes a predefined sequence of waypoint transitions. It initially navigates to WP1 and subsequently proceeds through WP2 and WP3, forming the first navigation segment. The mission then directs the UAV to return to WP1, introducing deliberate waypoint re-entry to validate navigation consistency and path repeatability. After revisiting WP1, the UAV continues towards WP4, followed by sequential navigation to WP5 and WP6, extending the mission's spatial coverage. In the final

navigation phase, the UAV returns to WP4 before initiating the landing procedure. The mission concludes with a controlled descent and system shutdown, marking successful completion. This mission is designed to evaluate the scheduler's ability to manage longer, deterministic execution chains involving multiple waypoints and intentional revisits. Unlike adaptive or sensor driven missions, Mission 6 relies on predefined control flow without conditional branching, making it well suited for validating command sequencing, waypoint handling, and mission endurance under nominal operating conditions.

Mission11

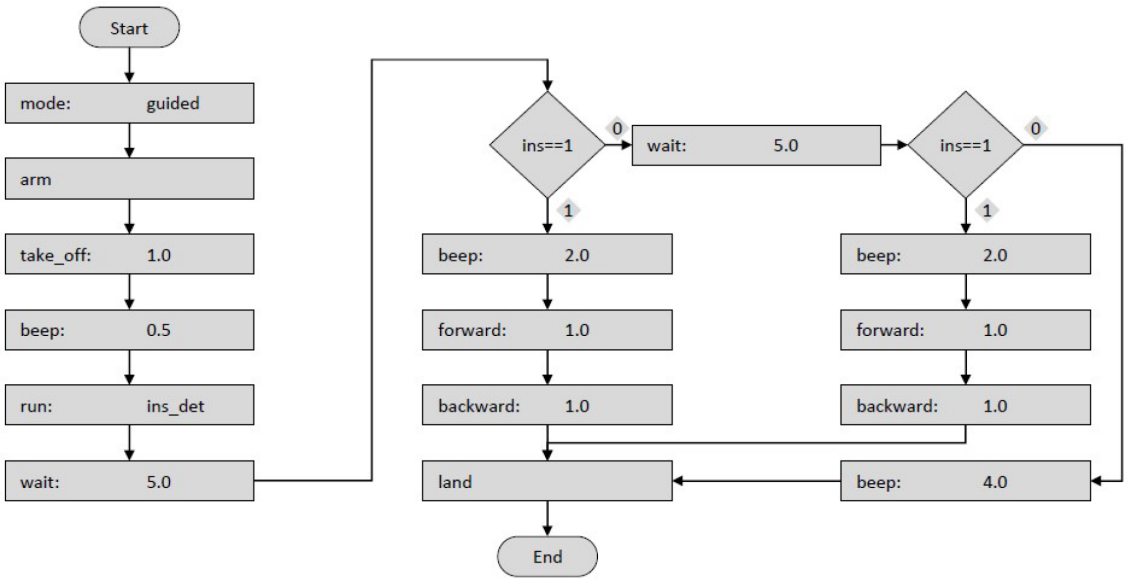


Figure 4.4: Mission 11 flow chart

Mission11 4.4 focuses on dynamic navigation and real time object detection, serving as a practical example of how drones can actively respond to their surroundings. At startup, the drone emits a short beep to indicate readiness, followed by a stabilization period allowing its onboard systems to initialize properly. After this preparation, the Object Detection Block is triggered to locate a specified target within the environment. This block plays a crucial role by enabling conditional operations that depend on sensor feedback. When the target object is detected, the drone executes a set of predefined maneuvers such as emitting a confirmation tone, moving forward by one meter, and then returning to its initial position. These components collectively enable conditional and responsive workflows, demonstrating the drone's ability to dynamically interact with its environment.

Mission 11 also implements an alternative logic path for cases where the object is not found, enabling continuous mission progress without halting operations. In

such cases, the drone emits another signal and pauses for a configured interval before attempting detection again. This process is governed by an iterative control structure using Break Loop and Continue Loop blocks, allowing repetitive scanning until either the object is identified or a maximum iteration count is reached. This adaptive approach makes Mission 11 particularly well suited for applications like search and rescue, where drones must navigate uncertain environments and detect specific objects or individuals.

Mission12

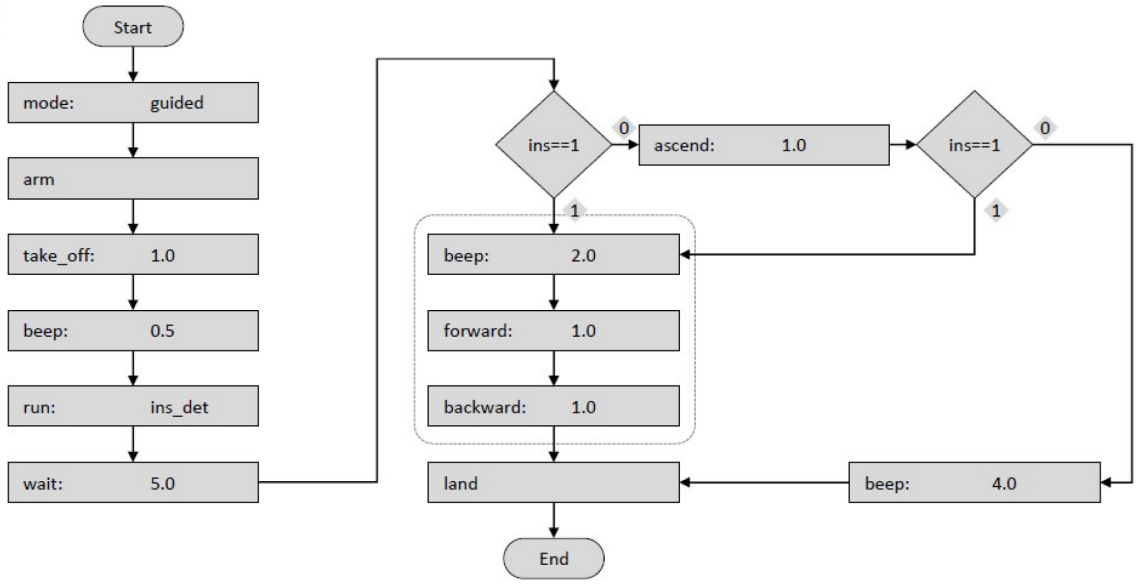


Figure 4.5: Mission 12 flow chart

Building on Mission 11, Mission 12 4.5 introduces altitude control as an additional workflow component. The mission begins similarly with system readiness checks and stabilization but now adds vertical mobility, increasing both complexity and operational versatility. When an object is detected, the drone triggers a function block that calls a predefined sequence of actions: emitting a beep, performing forward and backward motion, and adjusting altitude according to mission parameters. The modular design of the Function Block supports reusability across missions, reducing redundancy and simplifying future mission development.

If the object is not detected, the drone follows an alternate procedure: it ascends by one meter and restarts the detection process. This capacity for dynamic altitude adjustment adds significant value for tasks such as inspecting multi level structures or navigating uneven terrain. Through integration of Ascend and Descend blocks, Mission 12 demonstrates that vertical navigation can be achieved without compromising environmental adaptability. It highlights how modularity and responsiveness

allow drones to complete complex tasks efficiently under changing conditions.

Mission13

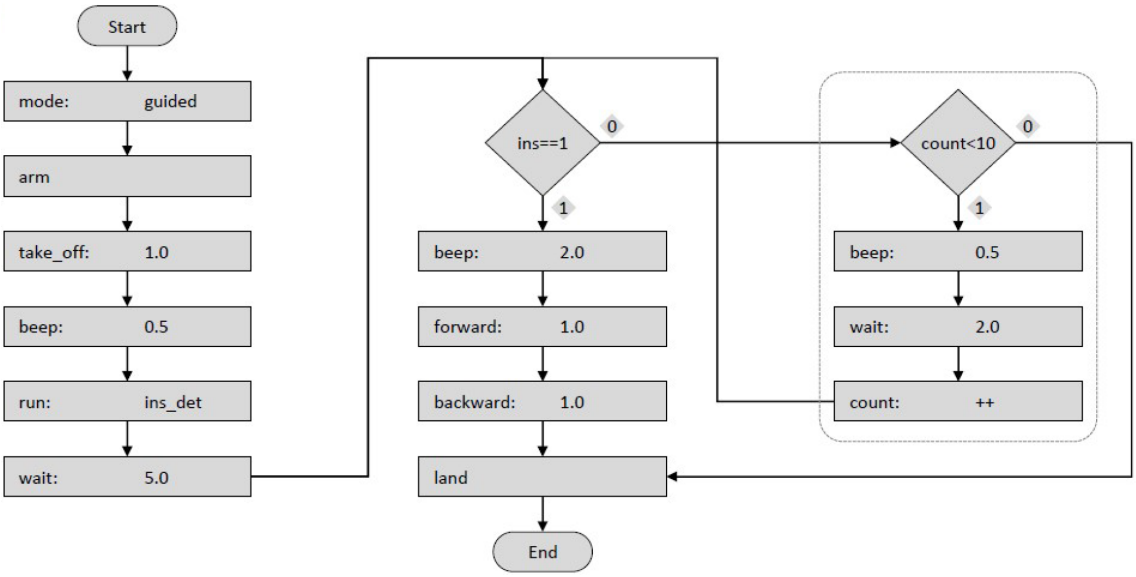


Figure 4.6: Mission 13 flow chart

Mission13 4.6 introduces an additional layer of control through iterative workflows governed by a counter variable. After initialization, the drone defines a modular function within a Function Block that encapsulates repeated actions such as emitting a sound signal, performing movement routines, and returning control to the main flow. The central feature of this mission is the use of the Counter Block, which orchestrates repeated object detection and associated functions in a loop until a specified iteration limit is reached. For example, the drone might sequentially ascend, perform a scan, and descend multiple times. The Break Loop block provides precise control, terminating the cycle once the counter condition is satisfied and ensuring efficient use of resources.

An alternative route is defined for situations when loop conditions are not met, prompting the drone to emit a warning signal and pause before retrying. This mission showcases the integration of iterative logic, conditional branching, and modular function design within a single, cohesive workflow. Such nested logic enables advanced behaviors suitable for use cases like environmental monitoring, agricultural surveys, or industrial inspection tasks.

4.2 Proposed Design

4.2.1 SCH Component Overview

SCH serves as the central orchestration layer within the system. It is responsible for coordinating mission execution and managing component life cycles. It maintains a global understanding of the system state and ensures that all configuration and activation decisions align with the defined mission objectives. At system startup, SCH initializes as a background service, and brings up the system by running the GCS component. Missions are selected via a ground control interface. These files encapsulate mission goals, behavioral logic, and required subsystems in a formalized schema, allowing automated interpretation by the onboard system.

Upon reception of a mission file, the scheduler parses its contents to identify the corresponding operational tasks and dependencies. This interpretation phase abstracts high-level mission intent from implementation specifics, supporting flexibility and modularity in system design. Based on predefined mappings between mission tasks and software capabilities, the scheduler determines which system components must be activated to fulfill the mission requirements. Component activation follows a controlled sequence that ensures dependency resolution and resource efficiency, while nonessential modules remain inactive to preserve computational capacity and power, as shown in Fig. 4.7.

During execution, the scheduler continuously supervises system health and operational consistency. It monitors the activity and availability of relevant components, detecting anomalies or failures, and applying recovery procedures when possible to maintain mission continuity. This capability enables adaptive fault management and contributes to the overall robustness of the system. When a mission concludes or is aborted, the scheduler oversees a clean termination phase, ensuring that all active components are properly shut down, as shown in Fig. 4.8. Through this cycle of interpretation, coordination, and supervision, the SCH Component provides a structured and autonomous mechanism for dynamic mission management, forming a critical link between high level planning and low level system execution.

Functional Requirements

Functional requirements that guide the conceptual design:

- Modularity: Components must be independently deployable and replaceable.
- Scalability: The system must support increasing numbers of missions and components.
- Robustness: Failures of individual components must not collapse the entire system.
- Maintainability: System behavior should be modifiable through configuration rather than code changes.

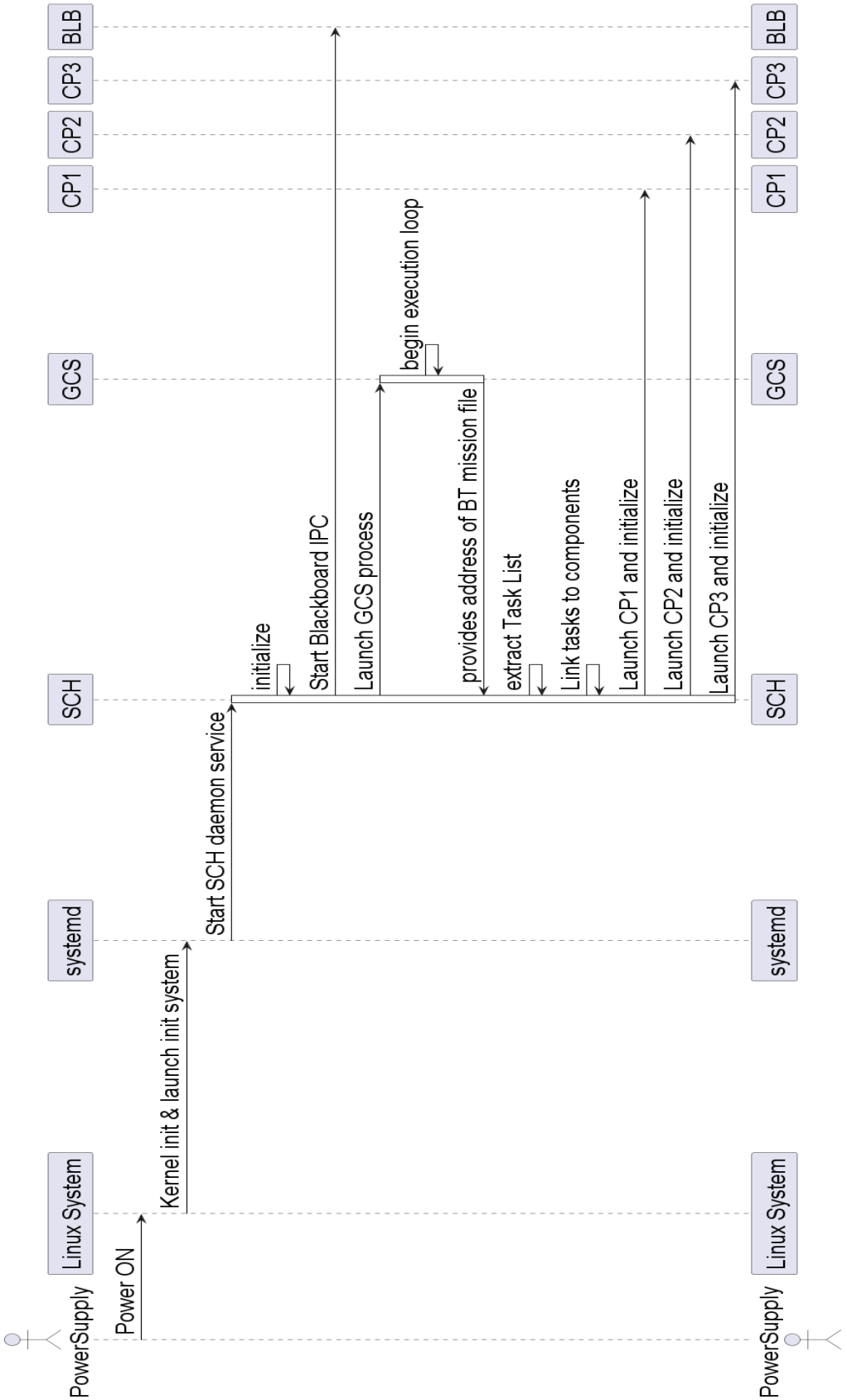


Figure 4.7: SCH behavior before flight

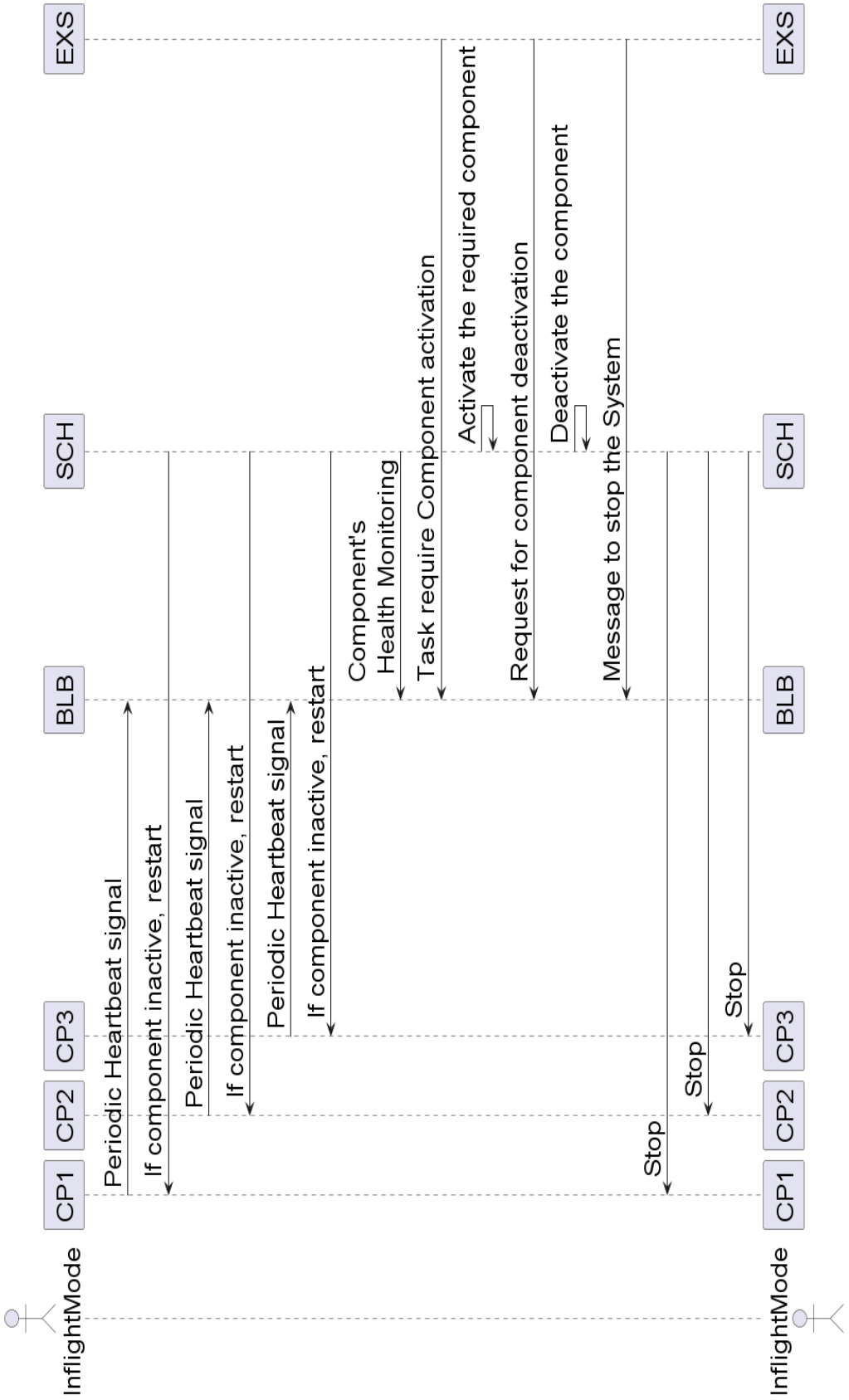


Figure 4.8: SCH behavior in flight

- Deterministic Behavior: Mission execution must follow predictable and repeatable logic.

Mission File Interpretation Model

At the core of the proposed architecture lies a mission file interpretation model that decouples mission definition from runtime execution. A mission is represented as a structured, machine readable description that expresses the intended operational behavior of the UAV without embedding explicit execution details. Several mission representation paradigms are commonly employed in autonomous systems, including state machines, behavior trees, task graphs, and procedural scripts, as discussed in Section 3.1.1. Although these paradigms differ in their execution semantics, they share a common requirement. That is mission descriptions must be interpretable by a supervisory entity capable of transforming high level intent into executable actions.

In the proposed architecture, the scheduler does not execute the mission representation directly. Instead, SCH performs a static interpretation step in which the mission file is parsed to extract a unique set of task identifiers. These identifiers define abstract mission capabilities. Such as navigation, streaming, or detection rather than specific executable processes. This interpretation based approach offers several advantages compared to tightly coupled execution models.

- Mission files remain platform independent and reusable
- Subsystem components are not required to understand mission structure
- And mission analysis can be conducted before runtime.

By restricting the scheduler's role to mission intent extraction, the architecture prevents mission-specific logic from being hardcoded into SCH, avoiding architectural bloating and preserving modularity.

Static Mapping Mechanism (Mission Task → Component)

After extracting mission task identifiers, the scheduler translates them into executable components through a static mission to component mapping process. This mapping defines which software subsystems are needed to realize each abstract mission task, converting mission intent into a concrete system configuration. In autonomous systems, task to component resolution can follow either a dynamic or a static approach. Dynamic resolution relies on runtime mechanisms such as capability discovery, service registration, or semantic matching between task descriptions and available components. Although these methods provide flexibility in heterogeneous or evolving systems, they require runtime reasoning, introduce non deterministic behavior, and increase computational as well as integration complexity. Such properties are undesirable on embedded UAV platforms, where resources are limited and predictable system behavior is essential. The proposed architecture strategy ensures.

- Task to component associations are defined offline and evaluated during mission initialization
- Resolving all dependencies before execution, SCH creates a complete and deterministic component set prior to startup
- Enables precise control of initialization behavior and removes the need for runtime overhead

In this process, SCH performs a single resolution pass that identifies all required components, merges shared dependencies, and ensures that each component is instantiated only once. This prevents redundant launches, keeps resource usage predictable, and maintains clear life cycle boundaries. The static mapping method aligns with common practices in safety critical and resource constrained systems, where the exact system composition must be verified before execution.

Component Health Monitoring Model

The proposed SCH architecture implements a well defined health monitoring mechanism for each software component. In component based and service oriented robotic systems, reliable operation depends not only on correct functional behavior but also on the system's ability to observe and manage the runtime health of its constituent elements. Since components are deployed as independent processes or services, failures may occur locally without immediately propagating throughout the system. Without centralized supervision, such failures can result in inconsistent system states in which individual subsystems become unresponsive, degraded, or stalled, while the overall system continues operating under false assumptions of correctness. This lack of global situational awareness poses significant risk in autonomous operations, particularly during long-duration or safety critical missions.

Within the proposed architecture, SCH functions as the supervisory authority responsible for

- Preserving system level health awareness.
- Monitors the operational status of each component via periodic heartbeat signals
- Detects anomalies such as missed heartbeats or irregular responses
- And triggers predefined recovery actions, such restarts of affected components

By explicitly separating health monitoring and recovery control from the functional logic of individual components, the architecture establishes clear boundaries of responsibility. This design relieves components from complex fault handling tasks, while SCH enforces a unified framework for system wide fault management. The result is improved fault containment, enhanced maintainability, and compatibility with well established fault management practices commonly applied in autonomous and safety critical systems, as discussed in Section 3.3.

4.3 Technology and Concept Selection

4.3.1 BT for Mission Files

Mission execution in autonomous systems requires a representation capable of expressing sequencing, conditional branching, parallel execution, and failure handling in a structured and analyzable form. Common mission control models include FSMs, state charts, BTs, and hybrid task networks. FSMs are widely used for their simplicity. However, as mission complexity grows, FSMs often suffer from state explosion, decreased readability, and limited modular reuse. State charts address some of these limitations by introducing hierarchy and concurrency. Despite this improvement, they remain closely tied to execution semantics and are difficult to modify incrementally.

Behavior Trees have become a dominant mission representation model in robotics and autonomous systems. They offer modular composition, clear execution semantics, and transparent failure propagation. BTs also separate control flow from task implementation, allowing mission logic to evolve without altering the underlying subsystem code. In the proposed architecture, Behavior Trees serve as the mission level abstraction, while execution is carried out by independently managed software components. This approach aligns with current research and industry practice, where BTs are preferred for modeling complex autonomous behaviors due to their scalability and maintainability. Furthermore, BTs enable static analysis of mission structure, which directly supports the scheduler's mission interpretation and component resolution processes described in Section 3.1.1.

4.3.2 XML Mission Representation Language

Once BTs are selected as the mission control model, a suitable serialization and storage format must be defined. Common mission representation formats include binary encodings, DSLs, JSON, YAML, and XML. XML has been widely adopted in robotics and embedded systems because of its self descriptive structure, schema validation support, and comprehensive tooling ecosystem. Compared with JSON and YAML, XML provides more explicit hierarchical semantics and supports validation through mechanisms such as DTD and XSD. These features are useful for verifying mission file correctness before execution.

In the proposed scheduler, XML based mission files offer several advantages. They provide a clear hierarchical representation of Behavior Tree nodes and enable robust parsing with mature, lightweight libraries. They also combine human readability with machine interpretability, simplifying both development and debugging. Furthermore, XML mission specifications are well established in existing robotic frameworks and planning systems, enhancing interoperability and long term maintainability. The choice of XML therefore represents a conservative yet well supported design decision suitable for safety oriented autonomous UAV systems, as discussed in Section 3.1.2.

4.3.3 XML Based Static Mapping

The mission scheduler requires a mechanism to map abstract mission tasks to concrete software components. This mapping can be implemented through embedded code logic, database backed configuration, semantic ontologies, or external configuration files. Embedding mapping logic directly into the scheduler code would increase coupling and reduce flexibility. Semantic reasoning methods, although powerful, introduce computational overhead and additional complexity that are not ideal for embedded UAV platforms.

The proposed architecture uses a XML based static configuration to define task to component relationships explicitly. XML is widely preferred for configuration because it is simple to read, lightweight to parse, and supported across most programming environments. Compared with JSON, XML provides a more compact syntax for key value structures and avoids the verbosity often associated with hierarchical representations. The static nature of this mapping ensures deterministic component resolution, which is essential for predictable system startup and reliable supervision. This approach is consistent with configuration driven composition strategies commonly applied in component-based and microservice oriented architectures.

4.3.4 Microservices Architecture for SCH

Architectural decomposition of UAV software systems generally follows one of three approaches: monolithic, modular, or microservices based. Monolithic designs combine all functionality into a single executable, which simplifies deployment and configuration. However, this design tightly couples system components and limits fault containment as system complexity increases. Modular architectures introduce clearer separation between functional units, improving maintainability and readability. Despite this benefit, the modules often share the same runtime environment, which restricts independent lifecycle management and weakens fault isolation.

Microservices based architectures extend this separation further. They organize the system as a collection of independently deployable services, each implementing a specific function and communicating through explicit interfaces. This model enforces distinct responsibility boundaries, simplifies failure handling, and allows for more precise resource control. In robotic and UAV systems, microservice inspired designs have attracted growing interest as platforms integrate more sensors, perception pipelines, and mission level autonomy. These capabilities demand scalable and resilient software structures, as discussed in Section 1.1.3.

The SCH component architecture follows this microservices oriented philosophy. SCH functions as an independent subsystem that operates as a separate service, with all communication and coordination handled through an IPC mechanism.

4.3.5 Containerization

Deploying microservices in embedded environments introduces challenges related to dependency management, build reproducibility, and runtime isolation. Embedded platforms often combine heterogeneous software stacks, hardware specific libraries, and strict resource constraints. These factors make it difficult to maintain consistent deployments across development, testing, and operational environments. Without proper isolation, changes in one component's environment can unintentionally affect others, resulting in fragile system behavior.

Containerization technologies address these challenges. By providing lightweight execution environments where applications are packaged together with their libraries, configuration files, and dependencies. Unlike virtual machines, containers share the host operating system kernel, which significantly reduces memory and processing overhead. This efficiency makes containers well suited for resource constrained embedded and edge platforms, including UAV companion computers, where predictable performance and efficient resource use are critical. Beyond isolation, containerization supports a structured deployment workflow through versioned images and reproducible builds. Each software component can be built, tested, and deployed as an immutable unit, minimizing configuration drift between development and operational contexts. Container run times also allow controlled allocation of CPU time, memory, and device access, which promotes predictable performance under varying workloads.

In the proposed system, containerization encapsulates individual software components to ensure consistent and repeatable operation across execution environments. Treating each component as a self contained deployment unit simplifies integration, allows independent component evolution, and strengthens the system's microservices oriented design. This approach reflects current best practices in robotic systems engineering and edge computing, as outlined in Section 2.4.

4.3.6 Daemon-Based SCH Execution

The mission scheduler operates as a daemon process that is initialized during system boot and remains active throughout the UAV's operational lifetime. Alternative execution models include on demand startup or mission scoped controller processes. Running the scheduler as a daemon ensures continuous availability for system initialization, mission selection, and health supervision without manual intervention. This model is common in embedded Linux systems and autonomous platforms, where persistent supervisory processes are necessary. By functioning as a daemon, the scheduler maintains responsibility for system life cycle management and upholds the watchdog and supervisory control principles established in safety critical embedded systems.

4.4 summary

This chapter presented the conceptual design of the proposed SCH architecture and explained rationale behind the corresponding architectural and technological decisions. It began with a system level overview that introduced the mission use cases guiding the design process and supporting the evaluation of SCH under different levels of mission complexity. The discussion then detailed the SCH component, outlining its responsibilities and the functional requirements that define its role within the overall UAV software system.

The chapter also examined the core elements of the proposed approach, including the mission file interpretation model, the static task to component mapping mechanism, and the centralized component health monitoring framework. Together, these design elements enable deterministic system initialization, modular component activation, and reliable runtime supervision. Furthermore, this chapter addressed the selection of key enabling technologies, including BT based mission representation, XML based configuration artifacts, microservices oriented architectural organization, containerized deployment, and daemon based scheduler operation.

5 Implementation

5.1 System Preparation

5.1.1 Hardware Setup (Jetson Nano, Sensors, Network)

NVIDIA Jetson Nano

The conceptual system architecture assumes a dual computer UAV setup, consisting of a real time flight controller and a companion computer. The SCH component executes on the companion computer and it interacts with multiple onboard software components. [87, 88]. The NVIDIA Jetson Nano is used as a companion computer for AREIOM. It is an embedded computing platform designed to support modern autonomous and intelligent systems. It combines heterogeneous processing capabilities with a full featured Linux software environment. Positioned as an entry level device within the Jetson family, it enables the deployment of complex, multi component software architectures on resource constrained edge platforms. Its design emphasizes computational flexibility, energy efficiency, and system level integration rather than single purpose acceleration. This makes it well suited for autonomous systems that coordinate perception, decision making, and control subsystems. At the hardware level, Jetson Nano integrates a quad core ARM CortexA57 CPU with a 128 core Maxwell GPU, creating a heterogeneous execution environment. This architecture allows compute intensive workloads and control logic to coexist on a single embedded device. The platform offers standard interfaces such as Ethernet, USB, camera inputs, GPIO, and serial communication buses, which simplify integration into larger robotic or unmanned systems. Its power efficient design, supported by configurable operating modes, enables long duration operation in embedded and mobile scenarios [89].

Equally important is the software ecosystem provided for Jetson Nano. The platform runs a Linux based operating system distributed through NVIDIA JetPack, which includes kernel support, device drivers, and system libraries for embedded development. This environment supports standard Linux abstractions such as processes, daemons, file systems, shared memory, and IPC. As a result, Jetson Nano accommodates architectural patterns common in distributed and component based systems, including background services, supervised execution, and modular deployment [90].

In mission oriented autonomous systems, these characteristics enable a clear separation between system orchestration and functional components. Instead of integrating mission logic directly into individual modules, high level coordination can

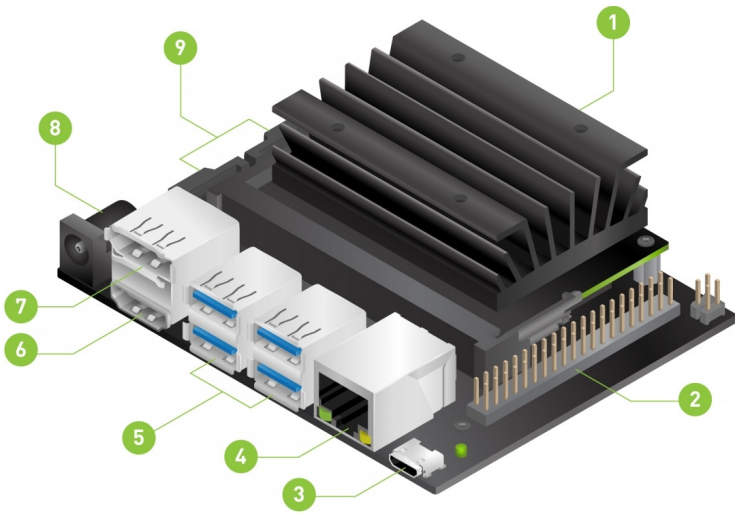


Figure 5.1: 1. microSD card slot for main storage, 2. 40pin expansion header, 3. MicroUSB port for 5V power input, or for Device Mode, 4. Gigabit Ethernet port, 5. USB 3.0 ports (x4), 6. HDMI output port, 7. Display Port connector, 8. DC Barrel jack for 5V power input, 9. MIPI CSI2 camera connectors [90]

be managed by a dedicated scheduling or orchestration service. This service can run as a persistent background process, active from system startup, and responsible for managing the life cycle and interactions of subordinate components. The stability of the Linux runtime and the predictability of process management on Jetson Nano make this separation practical. Within this context, the Scheduler (SCH) component is implemented as a long running daemon responsible for mission level coordination rather than direct task execution. Its operation does not depend on specific Jetson Nano hardware features but benefits from the platform's ability to support continuous background execution, reliable IPC, and robust process supervision. The blackboard based IPC mechanism used by the system aligns with the shared memory capabilities of the Linux kernel, enabling efficient state exchange between components while preserving loose coupling [91].

5.1.2 Programming Language C++ for development

In this thesis project, the Scheduler (SCH) component is implemented in C++. It must function as a long running system level orchestration service that performs deterministic mission interpretation, component life cycle control, and runtime supervision. This choice is based on C++'s suitability for IPC driven, multi process embedded Linux systems, where reliable execution and low overhead are essential. C++ provides the technical foundations required for SCH in several important ways.

C++ offers strong systems programming features and robust IPC integration. SCH continuously interacts with other processes through IPC, particularly the black-

board (BLB) component. The language provides efficient access to Linux IPC primitives such as shared memory, sockets, pipes, and message queues, as well as OS level APIs for process creation and supervision. This enables SCH to integrate tightly with the execution environment while preserving clean, well defined interfaces to other components. It also ensures deterministic performance and precise resource control. SCH performs time sensitive supervision tasks, such as periodic health monitoring through heartbeat signals, and must respond quickly if a component becomes unresponsive. C++ allows predictable performance and explicit control of memory and CPU usage, which is crucial for embedded systems with limited resources and strict timing constraints.

SCH also requires maintainable integration with mission and configuration artifacts. It relies on structured inputs, such as mission files and a static mapping file that link mission node identifiers with executable components. C++ supports efficient parsing, validation, and deterministic handling of these inputs, allowing mission coordination to remain configuration driven rather than hard coded. C++ further contributes to portability and long term maintainability. Although SCH targets Embedded Linux deployment, it can be built and tested across multiple platforms without modification. This flexibility streamlines development workflows and supports architectural consistency across diverse hardware environments.

The development process additionally benefits from C++'s mature tooling ecosystem, including debuggers, profilers, sanitizers, and unit testing frameworks. These tools facilitate fault isolation, shorten development cycles, and increase reliability. Such capabilities are particularly important for SCH, since failures in orchestration logic can compromise system availability. By implementing SCH in C++, this project achieves a robust foundation for mission based orchestration and health aware supervision while maintaining low runtime overhead and precise integration with IPC and Linux process control [92, 93].

5.1.3 Software Ecosystem for Development

The development environment for this work was structured around Visual Studio Code (VS Code). It was selected for its lightweight design, extensibility, and strong support for remote and container based development workflows. Instead of relying on a traditional monolithic IDE, VS Code functioned as a flexible front end that integrates seamlessly with embedded Linux targets and modern software tool chains.

A central aspect of the setup was the use of VS Code Dev Containers. This feature allowed the entire build environment including compiler tool chains, dependencies, and runtime libraries to be encapsulated within containerized environments. The approach ensured consistency between development and deployment systems and prevented issues caused by host specific library versions or configuration mismatches. By defining the development environment declaratively, the software stack remained reproducible and portable across different machines.

To interface with the target platform, VS Code Remote Explorer over Secure Shell (SSH) was used to establish a persistent connection to the Jetson Nano. This con-

figuration enabled access to the embedded device as a remote development target while retaining the full editing, debugging, and version control capabilities of VS Code. Source code editing, compilation, and execution were performed directly on the Jetson Nano without requiring physical access to the hardware. Unlike traditional cross compilation workflows, this setup supported native compilation within the target environment. Builds were executed inside development containers running on the Jetson Nano. Although the platform is resource constrained, this method ensured binary compatibility with system libraries and removed discrepancies between host and target architectures. Containerization minimized performance overhead by providing a controlled and streamlined runtime environment, keeping development both efficient and predictable.

SSH based remote access further simplified testing and debugging tasks. Runtime logs, process states, and IPC could be monitored in real time, which was particularly valuable when validating long running daemon components such as the scheduler SCH. Rapid development cycles were achieved through direct deployment and immediate execution on the target platform, eliminating repetitive manual setup.

Overall, the configured development environment formed a cohesive, efficient, and reproducible software ecosystem. By combining VS Code, Dev Containers, and SSH based remote development, the workflow supported safe and efficient development of embedded, IPC driven software while maintaining close alignment with the actual runtime conditions of the deployed system.

5.1.4 Folder Structure

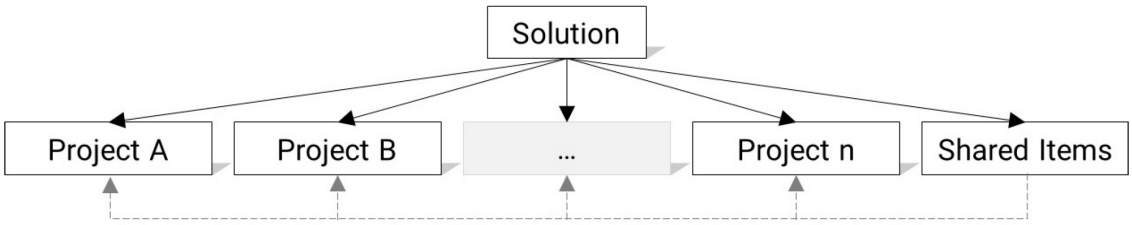


Figure 5.2: Folder Structure [17]

Each software component is developed as an independent project. One of the principles of the development process is usage of "Shared Items". This shared item folder further contains configs, missions, and scripts folders.

configs folder has .env.dev file which contains runtime configuration for all the components. This way we have one shared configuration source for the whole system. Another reason when we make changes inside the container, to apply them, the image must be rebuilt. Therefore, it is better to prevent rebuilds when settings change, and keeps code separate from runtime configuration, which changes very often. Also,

when the code is deployed onto an another Jetson. It may have different camera index, IP addresses, log path, ports, connected sensors. Each Jetson would only require to edit this single file to match its camera paths, IPs, or mission settings. This makes the setup consistent, clean, and easy for everyone. missions folder stores xml representation of all designed missions. scripts folder contains start and stop scripts to every component.

5.2 Prerequisite For SCH Implementation

5.2.1 Mission Files

To realize, validate, and systematically test the SCH component, the concept of mission files is introduced. Mission files serve as the primary input artefacts to the SCH and describe the operational intent of the UAV in a structured and machine readable form. From the perspective of the SCH, a mission file represents a complete specification of *what* needs to be executed, while the SCH itself is responsible for deciding *how* and *when* the corresponding software components are initialized, coordinated, and supervised.

In the developed system, missions are externalized into dedicated mission files rather than being hard coded within the scheduler logic. This separation is intentional and aligns with principles of modularity and configurability. By decoupling mission definition from the SCH implementation, new missions can be introduced, modified, or tested without recompiling or altering the scheduler component. This design choice is particularly important for experimental UAV platforms, where mission logic evolves frequently during development and testing phases.

For the purpose of this work, a set of representative missions: Mission3, Mission6, Mission11, Mission12, and Mission13 are defined. These missions cover different operational scenarios and complexity levels, these missions are discussed in 4.1.2. All selected missions are designed using a unified representation model and are exported as XML files, which are then consumed by the SCH at runtime.

Missions as Behavior Trees

BTs provide a structured and modular framework for describing the decision making logic of autonomous agents. They map environmental conditions to corresponding actions, enabling clear representation of complex behaviors. Within a BT, conditions evaluate the state of environment, while actions modify it. The hierarchical organization of these elements allows designers to express intricate control policies with clarity and scalability. The design of Behavior Trees follows a few key principles. Behaviors are arranged hierarchically, allowing complex tasks to be broken down into simpler components. Task sequencing ensures that actions execute in a defined order, where each step depends on the success of the previous one. Task selection introduces alternatives for achieving the same objective, creating flexibility in decision making. Reactivity is achieved by allowing high priority tasks to interrupt

lower priority ones when environmental conditions change. Modularity is supported through a consistent node interface, where every node receives an execution signal and returns a standard status indicating its outcome.

A BT is composed of two categories of nodes: control flow nodes and execution nodes, arranged in a directed hierarchical graph. Execution begins at the root, which periodically emits tick signals at a fixed rate. These ticks propagate through the control flow nodes, activating execution nodes when required. Each node runs only when it receives a tick and then returns one of three possible states: Running: active execution, Success: goal achieved, or Failure: execution failed.

The classical BT model includes three types of control flow nodes: Sequence, Fallback, and Parallel. Sequence nodes activate their children in order and succeed only when all of them succeed, stopping if any child fails or remains running. Fallback nodes also evaluate their children sequentially but succeed as soon as one child returns Success, providing alternative strategies for reaching a goal. Parallel nodes activate multiple children simultaneously and determine their status based on defined success or failure thresholds. Execution nodes consist of two main types: Action nodes, which perform operations over time, and Condition nodes, which evaluate logical propositions about the environment and return either Success or Failure.

Behavior Trees support reactivity and preemption by design. Because ticks are continuously propagated from the root, high priority branches are reevaluated at every tick and can interrupt lower priority tasks when specific conditions become true. This mechanism enables rapid responses to critical events such as safety violations or resource constraints. Once the triggering condition no longer holds, the interrupted behavior can resume seamlessly. Originally developed to control non player characters in video games, Behavior Trees proved effective for managing complex, adaptive behaviors driven by continuous feedback. Their clarity and modularity later led to widespread adoption in robotics, where similar challenges arise in achieving robust and reactive control. Further research expanded the BT framework with memory based nodes, utility driven behavior selection, probabilistic execution models, and concurrency extensions for handling parallel tasks.

In conclusion, BT represent a mature and expressive method for describing hierarchical and reactive behaviors in autonomous agents. Their balance of modularity, reactivity, and theoretical rigor makes them an effective foundation for mission representation and execution in modern robotic systems. A detailed explanation of BTs and their execution semantics is provided in [94].

Groot2 Tool

To design and manage Behavior Tree based missions, this project uses the Groot2 graphical editor. Groot2 offers a visual interface for assembling BTs from predefined control flow nodes along with user defined action and condition nodes. The graphical approach helps reduce the cognitive effort involved in creating complex mission logic compared with writing it directly in code or editing raw XML files.

A key capability of Groot2 is its ability to export BTs as XML files that are compatible with the BehaviorTree.CPP framework. These XML files serve as the mission input for The SCH component. From a system integration viewpoint, using Groot2 helps maintain consistency and correctness in mission definition. The tool's visual structure and validation features limit errors in tree composition that might otherwise cause runtime faults or undefined behavior. The XML export format also promotes interoperability and reproducibility by providing mission files that are platform independent and easy to inspect or modify. Overall, Groot2 allows both developers and domain specialists to create missions at a conceptual level while generating standardized XML mission files ready for direct processing. This workflow accelerates mission development and ensures a clean separation between mission design and execution.

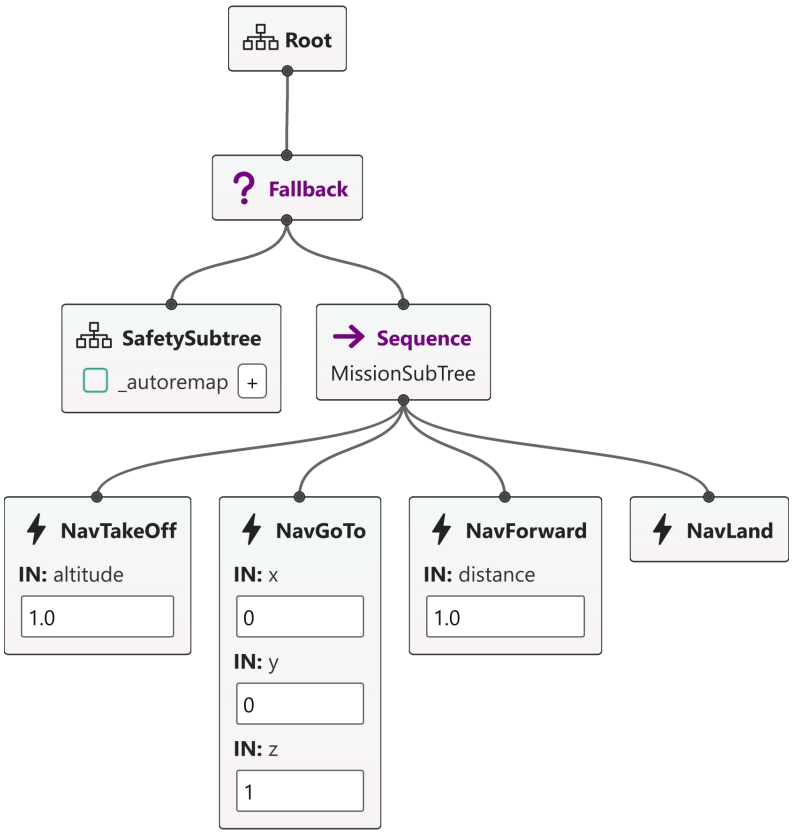


Figure 5.3: BT representation of mission 3

Mission 3 as BT Mission 3 uses a Behavior Tree that separates safety supervision from mission execution while staying responsive to environmental changes. Execution starts at the root node, which sends tick signals to its child nodes at regular intervals. The root connects to a Fallback node that prioritizes a safety subtree over mission execution. This ensures safety checks occur before any mission action. The

safety subtree uses nested Fallback nodes to perform checks and trigger recovery behaviors. First, the battery level is evaluated using the BattOK condition. If it fails, NavLand triggers an emergency landing. If successful, communication health is verified through CommOK. A communication failure activates NavRTL, commanding the vehicle to return to its launch point. Only when both safety conditions pass does the mission execution branch activate. The mission follows a Sequence node that enforces strict action ordering. It begins with NavTakeOff to reach a set altitude, followed by NavGoTo to move to a waypoint. Next, NavForward commands forward motion, and the mission concludes with NavLand. The continuous propagation of tick signals keeps the system reactive, allowing safety behaviors to interrupt mission actions when conditions change. This structure demonstrates how Behavior Trees enable modular and safety aware mission control in autonomous UAV systems.

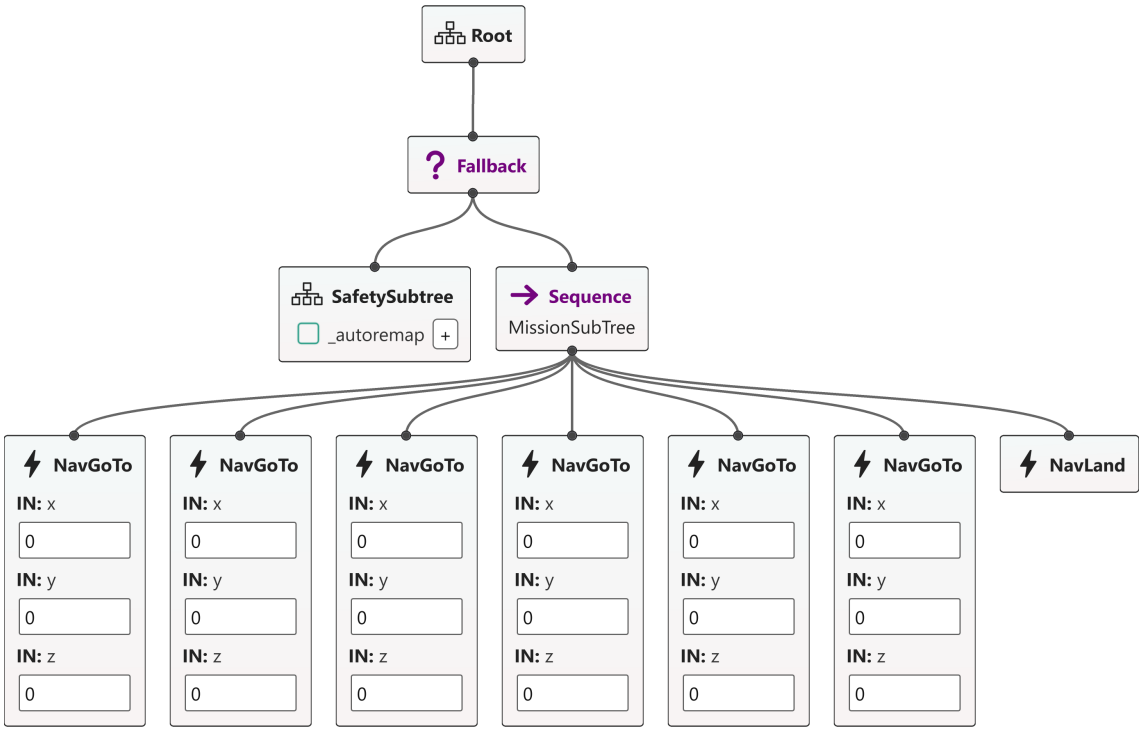


Figure 5.4: BT representation of mission 6

Mission 6 as BT Mission 6 uses a Behavior Tree that combines safety supervision with a waypoint based mission strategy. Execution begins at the root node, which sends tick signals at regular intervals to update the tree. Below the root, a Fallback node defines a priority structure where safety checks always run before mission actions. Its first branch is the Safety Subtree, which monitors system conditions during the entire mission. Inside this branch, nested Fallback nodes check the battery and communication status. The BattOK condition is checked first. If

it fails, the NavLand action starts an emergency landing. If the battery is normal, the CommOK condition is evaluated. A communication failure triggers the NavRTL action, returning the vehicle to its launch point. Only when both conditions succeed does the Safety Subtree return failure, allowing the main mission branch to execute. The mission logic is defined by a Sequence node that enforces strict action order. It contains several NavGoTo actions that move the vehicle through a set of waypoints, followed by a final NavLand action for a safe landing. Each NavGoTo command must finish before the next one starts, ensuring a predictable mission flow. During the entire process, tick signals continue to flow from the root. This allows the system to constantly check safety conditions and interrupt mission actions if needed. The resulting Behavior Tree shows how waypoint missions can be implemented in a modular and reactive way while maintaining safety awareness, making it suitable for structured navigation in autonomous UAVs.

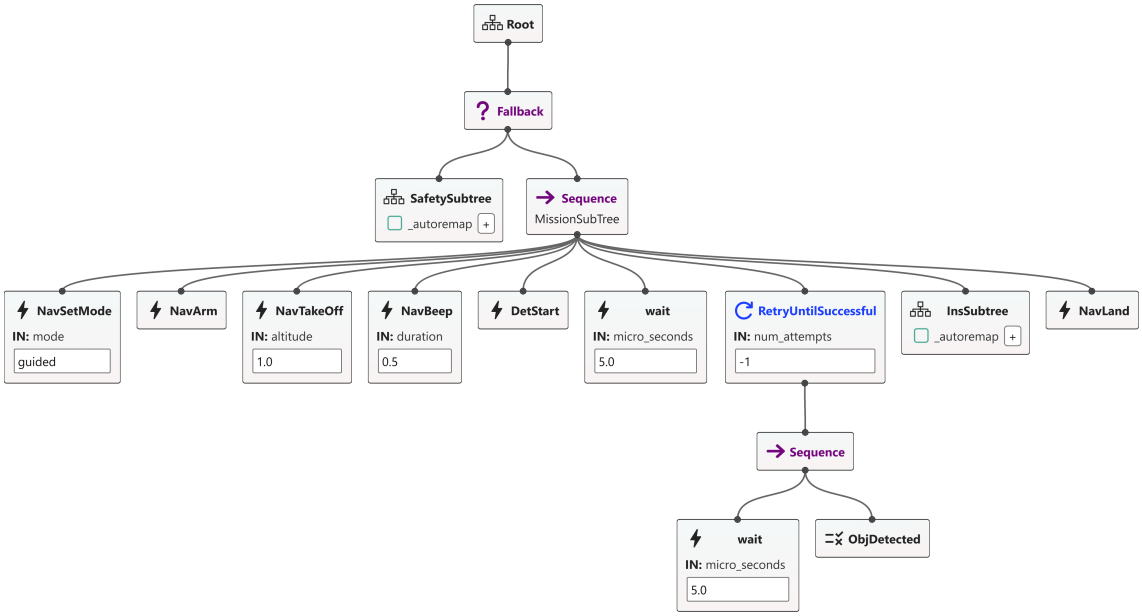


Figure 5.5: BT representation of mission 11

Mission 11 as BT Mission 11 uses a Behavior Tree that combines safety monitoring, vehicle setup, perception based decision making, and structured mission completion within a single control model. Execution starts at the root node, which sends tick signals through the tree to maintain reactive behavior. Below the root, a Fallback node defines the priority order, making sure that safety checks always run before mission actions. The Safety Subtree evaluates system conditions using nested Fallback nodes. The BattOK condition is checked first. If it fails, the NavLand action performs an emergency landing. If the battery is normal, the CommOK condition is evaluated. A failure here triggers NavRTL, which returns the vehicle to

its launch point. Only after all safety checks pass does control move to the main mission branch. The mission sequence is represented by a Sequence node that enforces a fixed order of tasks. It begins with NavSetMode to set the vehicle into guided mode, followed by NavArm and NavTakeOff, which prepare and lift the UAV to target altitude. After takeoff, NavBeep provides feedback, and the DetStart action begins the perception phase. A timed wait action stabilizes the system before a RetryUntilSuccessful decorator repeatedly checks for the ObjDetected condition. This loop continues until the target is detected. Once an object is found, the sequence proceeds to an inspection subtree. This part includes a short NavBeep, followed by forward and backward movements that simulate a local inspection. When the inspection is complete, the mission ends with a final NavLand for a controlled landing.

Throughout execution, tick signals continue to update the tree. This allows safety conditions to interrupt any mission step when necessary. The BT structure in Mission 11 shows how complex, perception driven missions can be organized into modular, reactive, and safety aware elements suitable for autonomous UAVs.

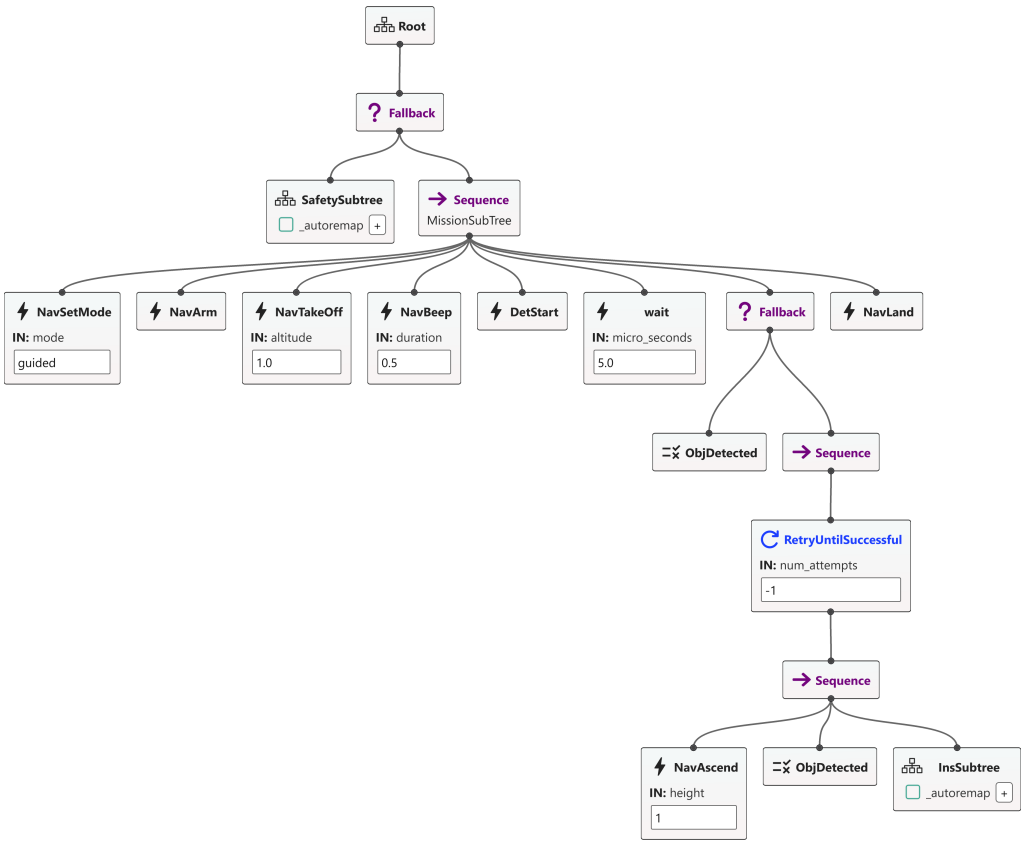


Figure 5.6: BT representation of mission 12

Mission 12 as BT Mission 12 uses a Behavior Tree that extends the perception based structure of Mission 11 by adding an adaptive altitude adjustment before in-

spection. Execution starts at the root node, which sends tick signals through the tree to maintain continuous reactivity. A Fallback node below the root manages priorities, ensuring that safety checks run before mission actions. The Safety Subtree includes nested Fallback nodes that evaluate system conditions. The BattOK condition is checked first, and if it fails, NavLand initiates an immediate landing. If the battery is normal, the CommOK condition is evaluated. A communication failure triggers NavRTL, returning the vehicle to its launch point. Only when all checks pass does control move to the mission sequence. The MissionSubTree, defined as a Sequence node, enforces a fixed order of tasks. It begins with NavSetMode to configure guided mode, followed by NavArm and NavTakeOff, which prepare and lift the UAV to a preset altitude. After takeoff, NavBeep provides feedback, and DetStart activates the perception system. A wait period then allows the system to stabilize. A Fallback node manages the perception results. If ObjDetected returns success, the mission proceeds to its final phase. If detection fails, a RetryUntilSuccessful decorator starts a retry sequence. This sequence uses NavAscend to increase altitude before checking ObjDetected again, allowing adaptive repositioning when initial detection fails. When detection succeeds, the mission proceeds to the inspection sequence. It includes a short NavBeep, followed by forward and backward motion to inspect the target area. The mission ends with a final NavLand for a controlled landing. Throughout Mission 12, tick propagation keeps the system reactive. Safety checks continuously monitor conditions and can interrupt the mission if needed, ensuring robust and adaptive UAV behavior.

Mission 13 as BT Mission 13 uses a Behavior Tree that builds on the perception based missions of 11 and 12 by adding limited retries and explicit failure handling for object detection. Execution starts at the root node, which sends tick signals through the tree to keep behavior reactive. A Fallback node below the root manages priorities, ensuring that safety checks run before mission actions. The SafetySubtree includes nested Fallback nodes that monitor system conditions. The BattOK condition is checked first; if it fails, NavLand performs an emergency landing. If the battery is normal, the CommOK condition is evaluated. Communication loss triggers NavRTL, returning the UAV to its launch point. Only after all safety checks pass does the mission sequence start. The MissionSubTree, structured as a Sequence node, enforces a fixed order of actions. The UAV is first set to guided mode using NavSetMode, armed with NavArm, and then takes off to a predefined altitude with NavTakeOff. After takeoff, NavBeep provides feedback, and DetStart starts the perception process. A wait period allows stabilization before object detection begins. Detection logic is handled by a Fallback node. It first attempts to detect the object using a RetryUntilSuccessful decorator that limits the number of retries. Inside each retry, the UAV beeps, waits, and checks the ObjDetected condition. If detection succeeds, the mission continues to the inspection sequence. This sequence performs a short beep and executes forward and backward movements for local inspection. If all retries fail, the Fallback node activates the failure branch, causing a controlled

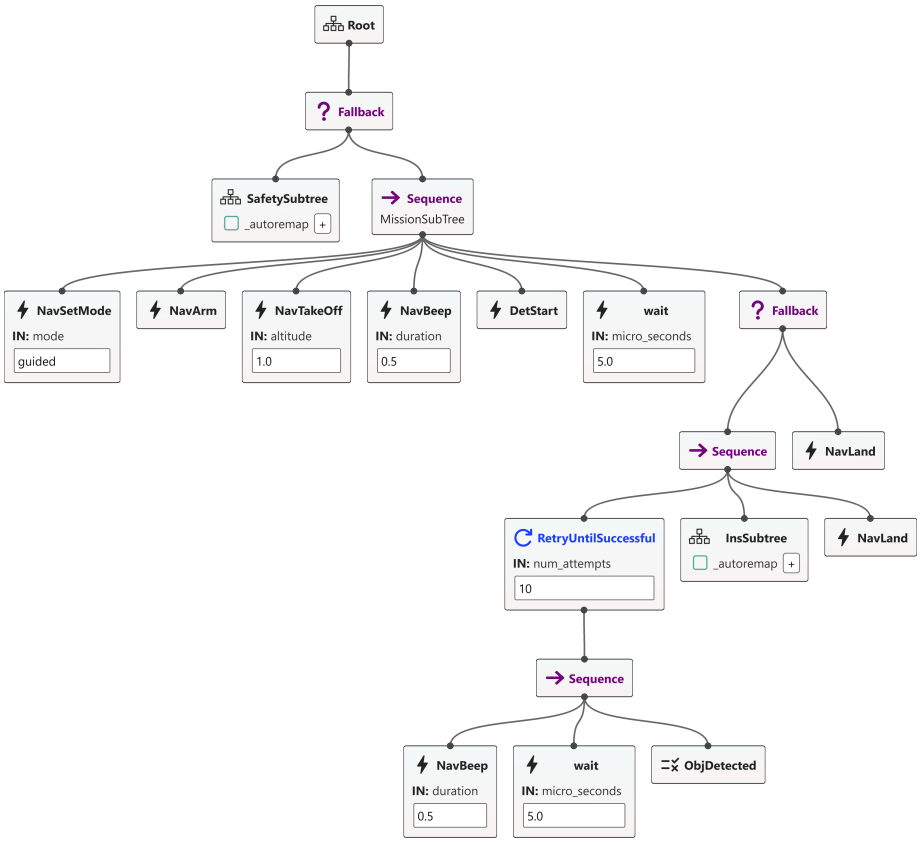


Figure 5.7: BT representation of mission 13

landing through NavLand. Throughout Mission 13, continuous tick propagation keeps the system responsive, allowing safety conditions to interrupt execution at any time. The limited retries prevent endless loops, creating a reliable and safety aware mission workflow.

5.2.2 Deploying Other Software Components

Containerizing BLB and DET Components to Target Device

The BLB and DET components are part of the system. They were previously developed to test various functionalities of the AREIOM project. To deploy them on the target Jetson Nano device, they were containerized. For testing the implementation of the SCH component, BLB was required to enable IPC communication, along with at least one additional component to support component initialization based on the mission.

Procedure of Creating Dockerfiles for PreExisting Software Components

When containerizing software components developed prior to this work, the objective was to create a runtime only Docker image that accurately reproduces the execution environment on the target device. Same procedure was applied to the BLB and DET components. It is designed to be reusable for other software components, even when the source code is not available.

Step 1: Identify the runtime entry binary The executable that represents the entry point of the software component must be identified. In the AREIOM development environment, binaries typically follow a structured layout, for example:

```
dev1/<component>/bin/ARM64/Release/<binary>.out
```

This binary is treated as the main process of the container.

Step 2: Determine runtime shared library dependencies To identify all shared libraries required at runtime, the `ldd` tool is executed on the host device:

```
ldd <binary>.out
```

This step reveals all dynamically linked libraries needed for execution. Kernel provided components and the dynamic loader are excluded from further consideration.

Step 3: Map shared libraries to operating system packages Each required `.so` library identified by `ldd` is mapped to its corresponding Debian/Ubuntu package. This mapping determines which runtime packages must be installed in the container image using the system package manager. Only runtime dependencies are considered; build time tools are intentionally excluded.

Step 4: Identify required runtime assets In addition to the executable, component may rely on configuration files, resource directories, models, or scripts at runtime. These assets must be explicitly identified to ensure that the containerized environment mirrors the expected file system structure of the native deployment.

Step 5: Select a compatible base image A base image compatible with the target device and operating system is selected. For the Jetson Nano platform used in this work, an Ubuntu 20.04 base image was chosen to match the host system and ensure ABI compatibility.

Step 6: Define a minimal runtime environment The Dockerfile is constructed to install only the runtime dependencies identified in previous steps. This results in a lightweight image that reduces attack surface and avoids unnecessary tooling.

Step 7: Configure execution context A non root user is created within the container, and a fixed working directory is defined. The executable and runtime assets are copied into this directory, with appropriate ownership and execution permissions applied.

Step 8: Specify shutdown and execution behavior To terminate running components, container is configured to forward `SIGINT` signals to the application. The executable is defined as the container entry point using an absolute path for deterministic startup behavior.

Generic Dockerfile Template for Prebuilt Software Components

For the procedure described above 5.2.2, below is a generic Dockerfile template

```
FROM ubuntu:20.04

ENV DEBIAN_FRONTEND=noninteractive \
LANG=C.UTF8

# Install minimal runtime dependencies
RUN apt-get update && apt-get install -y --no-install-
   recommends \
<runtime-packages> \
&& rm -rf /var/lib/apt/lists/*

# Create non-root runtime user
RUN useradd --create-home --shell /bin/bash appuser
WORKDIR /home/appuser

# Copy binary and runtime assets
COPY ./bin/ARM64/Release/<binary>.out /home/appuser/<binary>.
   out
# COPY ./configs   /home/appuser/configs
# COPY ./Resources /home/appuser/Resources

RUN chmod +x /home/appuser/<binary>.out && \
chown -R appuser:appuser /home/appuser

USER appuser

# Ensure graceful shutdown
STOPSIGNAL SIGINT

# Absolute entry point definition
ENTRYPOINT ["/home/appuser/<binary>.out"]
CMD []
```

The template focuses on reproducibility, simplicity, and correct runtime behavior. By following the defined steps and adjusting the template as needed, other software components can be containerized consistently and efficiently, even without access to their source code.

5.2.3 Component Start/Stop Management Scripts

System components are started and stopped using dedicated shell scripts that are triggered by the scheduler. These scripts handle environment variable setup, container execution parameters, and logging configuration. This method separates deployment logic from the scheduler's code, making the system easier to maintain and allowing components to be updated independently. Script based life cycle management is widely used in embedded Linux systems because it simplifies coordination and control. The system also supports stop commands to safely end missions or shut down components. When a stop request is received, the scheduler manages an orderly shutdown of all active components. A controlled shutdown is important to maintain system stability and prevent data loss or corruption.

Generic Start Script Template for Containerized Software Components

```
#!/usr/bin/env bash
set e

# Component configuration
COMPONENT_NAME="<componentname>"
CONTAINER_NAME="${COMPONENT_NAME}"
IMAGE_NAME="<imagename:tag>"

echo "[start${COMPONENT_NAME}] Starting from image ${
   IMAGE_NAME} ..."

# Remove any previously existing container instance
echo "[start${COMPONENT_NAME}] Removing old container if
   present ..."
docker rm f "${CONTAINER_NAME}" >/dev/null 2>&1 || true

# Detect interactive terminal and select run mode
if tty s; then
echo "[start${COMPONENT_NAME}] TTY detected > running in
   foreground"
DOCKER_RUN_MODE="it"
else
echo "[start${COMPONENT_NAME}] No TTY detected > running
   detached"
DOCKER_RUN_MODE="d"
```

```
fi

# Launch container
docker run rm ${DOCKER_RUN_MODE} \
name "${CONTAINER_NAME}" \
network host \
ipc host \
v /dev/shm:/dev/shm \
e TERM=xterm256color \
"${IMAGE_NAME}"

echo "[start${COMPONENT_NAME}] Container exited with status $
   ?"
```

The start script enforces a clean startup by removing any stale container instance before execution. Host networking and shared memory access are enabled to support low latency IPC, while terminal detection allows the same script to be reused in both development and automated runtime scenarios.

Generic Stop Script Template for Containerized Software Components

To complement the start procedure, a generic stop script template is provided below. This script ensures controlled shutdown of a running container and optionally performs component specific cleanup actions such as shared memory cleanup or log removal.

```
#!/usr/bin/env bash
set e

# Component configuration
COMPONENT_NAME="<componentname>"
CONTAINER_NAME="${COMPONENT_NAME}"
IMAGE_NAME="<imagename:tag>"

# Optional cleanup configuration
CLEANUP_ENTRYPOINT="<cleanupbinary>"
CLEANUP_ARGS="<cleanupargs>"

LOG_DIR="<hostlogdir>"
CLEAR_LOGS=false

echo "[stop${COMPONENT_NAME}] Stopping '${CONTAINER_NAME}'
   ..."

# Stop container only if running
if docker ps format '{{.Names}}' | grep Fxq "${CONTAINER_NAME
   }"; then
```

```
docker stop "${CONTAINER_NAME}" >/dev/null
echo "[stop${COMPONENT_NAME}] Container stopped."
else
echo "[stop${COMPONENT_NAME}] Container not running."
fi

# Execute componentspecific cleanup logic if required
if [ n "${CLEANUP_ENTRYPOINT}" ]; then
echo "[stop${COMPONENT_NAME}] Performing runtime cleanup ..."
docker run rm ipc host \
entrypoint "${CLEANUP_ENTRYPOINT}" \
"${IMAGE_NAME}" ${CLEANUP_ARGS}
fi

# Optional hostside log cleanup
if [ "${CLEAR_LOGS}" = true ]; then
rm f "${LOG_DIR}"/*.log || true
echo "[stop${COMPONENT_NAME}] Logs cleared."
fi

echo "[stop${COMPONENT_NAME}] Shutdown completed."
```

The stop script supports graceful termination by explicitly checking the container state before issuing a stop command. Optional cleanup steps allow components to release shared resources or persistent artifacts, ensuring a consistent system state before subsequent restarts or mission reconfiguration.

5.3 SCH Component Implementation

The scheduler runs as a daemon process that starts automatically when the system boots. It remains active for the entire operational period and supervises mission execution without interruption. Running the scheduler as a daemon is a common practice in autonomous and safety critical embedded systems. Component activation follows a defined order: blackboard initialization, mission selector startup, mission parsing and mapping, and subsystem activation. This controlled sequence ensures that all components are ready before mission execution begins and prevents race conditions during startup. To remove the dependency from GCS to have the UI for selecting mission file. A dummy component, mission-seclector is developed. The mission scheduler is implemented as a dedicated C++ application that serves as the main control unit for mission execution. Its main tasks include setting up IPC, launching auxiliary components, interpreting mission configurations, and supervising runtime execution.

Reading the Mission File Path from the Blackboard

Explanation. The Scheduler (SCH) reads the mission file path from the Blackboard field `mission.message`. Since this field is stored as a Cstyle character array, SCH first copies it into a string and removes trailing white space (e.g., `\n`, `\r`) to avoid invalid paths. If the mission path is still empty, SCH keeps polling the Blackboard for a bounded number of attempts until a valid path becomes available.

Algorithm 1 Reading the Selected Mission Path from the Blackboard

```
1: function ReadMissionPath(bb : BlackboardHandle)
2:     msgPtr ← bb.data.mission.message
3:     if msgPtr is NULL then
4:         return ""
5:     end if
6:     path ← String(msgPtr)
7:     while path is not empty and LastChar(path) is whitespace do
8:         RemoveLastChar(path)
9:     end while
10:    return path
11: end function
```

Extracting Node IDs from Behavior Trees

Explanation. After loading the mission XML, SCH searches for the `<BehaviorTree>` element and performs a depth first traversal over all XML elements inside it. Whenever an element contains an `ID` attribute, SCH inserts it into a set to eliminate repeated node IDs. The resulting set represents the node types required by the mission.

Mapping BT Nodes to Components using `StaticMapping.xml`

Explanation. SCH loads `StaticMapping.xml` and builds a lookup table mapping each BT node ID (`id`) to a component name (`component`). After extracting the mission node IDs, SCH resolves each ID through this table. Missing mappings are reported as warnings. To avoid starting a component multiple times, SCH deduplicates by component name when building the final set of required components.

Heartbeat Monitoring and Restart of a Component

Explanation. SCH continuously monitors the mission selector component through the Blackboard process slot `PROCESS_GUEST`. The heartbeat is represented by a timestamp field (`process[idx].time`). If the timestamp does not update within a fixed timeout window, SCH considers the component unresponsive and restarts it. To prevent repeated restarts in every cycle, SCH triggers only one restart and waits until a new heartbeat is observed.

Algorithm 2 Extraction of Unique Behavior Tree Node IDs from Mission XML

```
1: function ExtractNodeIDs(root : XmlElement)
2:     ids ← ∅
3:     if root is NULL then
4:         return ids
5:     end if
6:     bt ← FirstChildElement(root, "BehaviorTree")
7:     if bt is NULL then
8:         return ids
9:     end if
10:    Scan(bt, ids)
11:    return ids
12: end function
13: function Scan(node : XmlElement, ids : Set¡String¿)
14:    if node is NULL then
15:        return
16:    end if
17:    if node has attribute "ID" then
18:        ids ← ids ∪ { node.Attribute("ID") }
19:    end if
20:    for all child in Children(node) do
21:        Scan(child, ids)
22:    end for
23: end function
```

5.4 Summary

This chapter explained the implementation of the proposed mission scheduling system. It covered system setup, software architecture, mission parsing and mapping, and runtime supervision. The implementation shows how mission abstractions are converted into executable software components through a controlled and predictable process. The integration of shared memory coordination, static mapping, heartbeat monitoring, and daemon execution creates a strong base for reliable autonomous UAV missions. The next chapter presents system testing and experimental validation.

Algorithm 3 Building the Required Component Set from Node IDs and Static Mapping

```
1: function BuildComponentSet(nodeIDs : Set¡String¿, mappingXmlPath : String)
2:     nodeIdToComp ← EmptyMap()
3:     components ← EmptyMap()
4:     if LoadXml(mappingXmlPath) fails then
5:         Display error “Failed to load mapping XML.”
6:         return components
7:     end if
8:     mapRoot ← RootElement(mappingXmlPath)
9:     for all nodeEl in ChildrenNamed(mapRoot, ”Node”) do
10:        idAttr ← nodeEl.Attribute(”id”)
11:        compAttr ← nodeEl.Attribute(”component”)
12:        if idAttr is NULL or compAttr is NULL then
13:            Display warning “Mapping entry missing id/component; skipping.”
14:        else
15:            nodeIdToComp[idAttr] ← compAttr
16:        end if
17:    end for
18:    for all id in nodeIDs do
19:        if id not in nodeIdToComp then
20:            Display warning “No mapping for node ID [id].”
21:        else
22:            compName ← nodeIdToComp[id]
23:            if compName is empty then
24:                Display warning “Empty component name for node ID [id].”
25:            else
26:                components[compName].name ← compName
27:            end if
28:        end if
29:    end for
30:    return components
31: end function
```

Algorithm 4 Heartbeat Monitoring and Automatic Restart of MissionSelector

```
1: function MonitorHeartbeat(bb : BlackboardHandle)
2:     TIMEOUT_SEC ← 2
3:     haveSeenHeartbeat ← False
4:     lastHeartbeat ← 0
5:     restartIssued ← False
6:     while True do
7:         Sleep(1 second)
8:         if bb.data is NULL then
9:             Display error "Blackboard pointer is NULL in monitor loop."
10:            continue
11:        end if
12:        idx ← PROCESS_GUEST
13:        if idx < 0 or idx ≥ MAX_PROCESS_NUMBER then
14:            Display error "PROCESS_GUEST index out of range."
15:            continue
16:        end if
17:        proc ← bb.data.process[idx]
18:        if proc.id == PROCESS_NOTDEFINED then
19:            Display warning "MissionSelector not registered yet in BLB."
20:            continue
21:        end if
22:        hb ← proc.time
23:        if hb == 0 then
24:            Display warning "No heartbeat timestamp for MissionSelector yet."
25:            continue
26:        end if
27:        if haveSeenHeartbeat == False or hb ≠ lastHeartbeat then
28:            haveSeenHeartbeat ← True
29:            restartIssued ← False
30:            lastHeartbeat ← hb
31:        end if
32:        now ← CurrentTime()
33:        delta ← now - hb
34:        if haveSeenHeartbeat and delta > TIMEOUT_SEC and restartIssued
   == False then
35:            Display error "Heartbeat timeout; restarting MissionSelector."
36:            RunCommand("pkill -f mission_selector")
37:            RunInTerminal("missionselector", MISSION_SELECTOR)
38:            restartIssued ← True
39:        end if
40:    end while
41: end function
```

6 Test and Evaluation

6.1 Ordered Execution of Background Components

Goal: Show SCH is able to bring required BLB and GCS (mission-selector for testing) components to execution. Test: Is execution order retained.

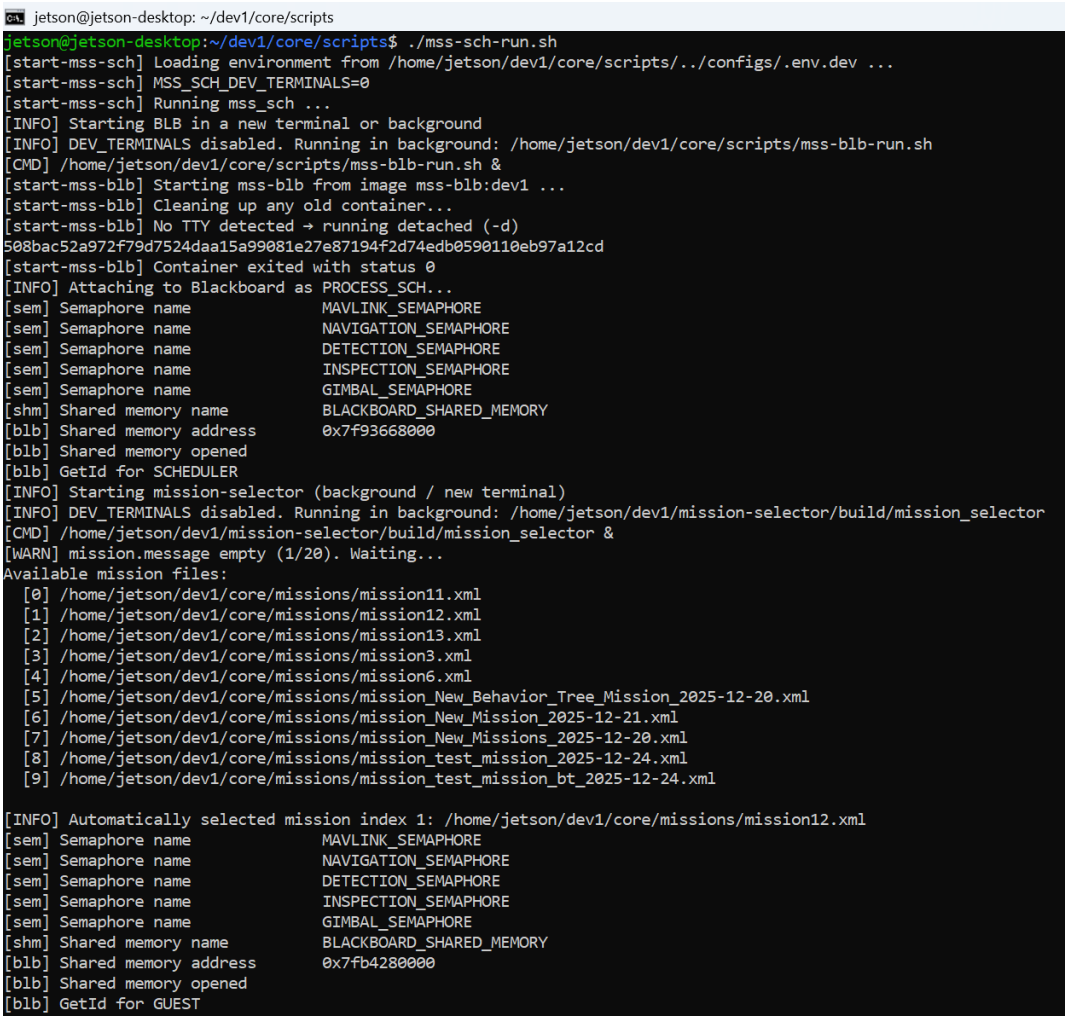


Figure 6.1: Background Components Order Test

Evidence: As shown in Figure 6.1, SCH follows a clearly defined sequential startup sequence implemented directly in sch.cpp. When launched, SCH first starts the BLB

using its configured startup script. This step intentionally occurs before any other process to ensure that the shared memory infrastructure required for IPC is available. After triggering BLB, SCH waits briefly to allow the Blackboard to finish initializing and create the necessary shared resources. Once initialization is complete SCH attaches to the Blackboard operating in slave mode. It then resets the mission.message field by clearing any previous content, preventing residual mission data from influencing the current startup. This guarantees that SCH can accurately detect whether a new mission has been provided. After successfully attaching and clearing the mission field, SCH starts the mission-selector component. It then connects to the Blackboard and writes the absolute path of the chosen mission file into the mission.message field. Finally, SCH enters a controlled polling phase and periodically checking the Blackboard for a valid mission path. This polling loop is bounded by a fixed timeout upto 20 seconds and SCH proceeds only after the mission path becomes available to ensure deterministic behavior. The resulting log sequence confirms that BLB initialization, SCH attachment, and mission-selector execution occur in a strict well enforced order while preserving the intended runtime hierarchy of background components.

6.2 Mission selection and mission file handling

Goal: Verify that the mission-selector component correctly publishes a mission file path to the Blackboard (BLB) and that the SCH reliably reads, and interprets this path without introducing file access errors.

Test: Valid mission path read from BLB and successfully processed by SCH.

```
[SUCCESS] Written mission path into Blackboard:
  mission.message = /home/jetson/dev1/core/missions/mission12.xml
[MSL] MissionSelector is now running and sending heartbeats as PROCESS_GUEST.
[INFO] Mission path: /home/jetson/dev1/core/missions/mission12.xml
[INFO] Node → Component mapping:
```

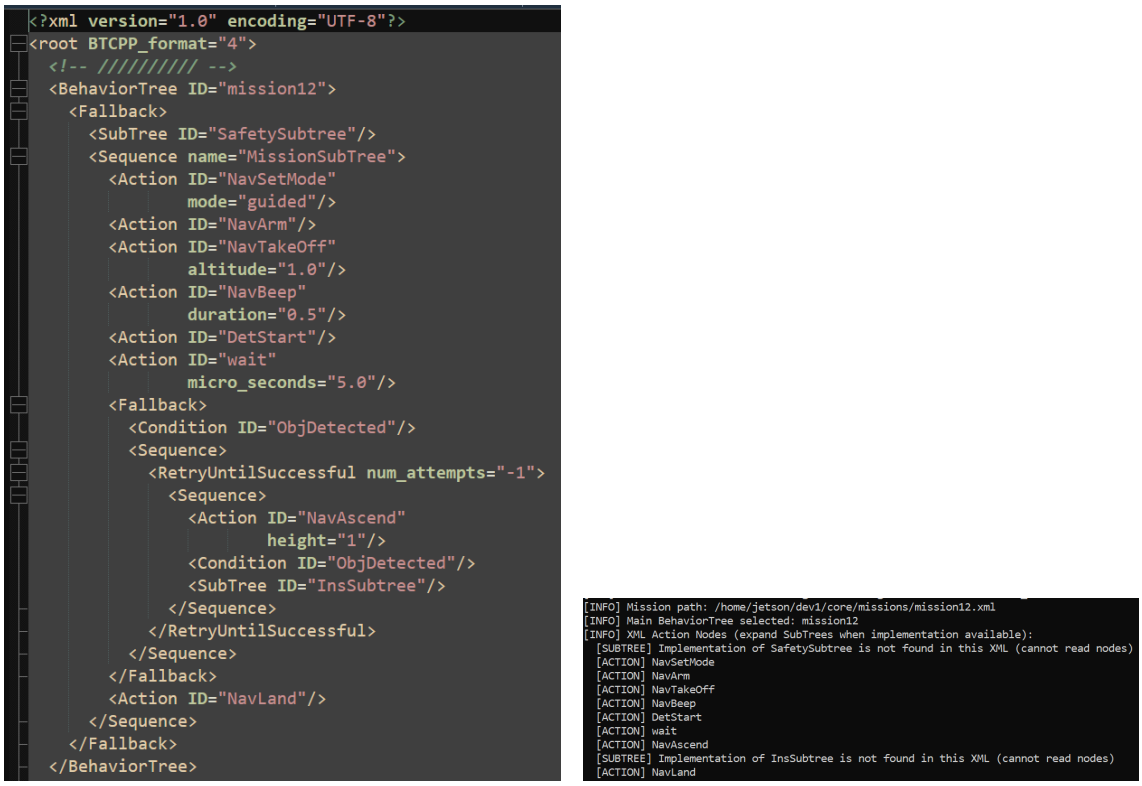
Figure 6.2: Mission file selection and Mission Path writing

Evidence: The mission-selector component accesses the configured missions directory, enumerates all available mission definition files, and randomly selects one mission file for testing purposes. It then writes the absolute path of this selected mission file into the mission.message field of the Blackboard. SCH continuously checks this field and proceeds only once the message buffer is no longer empty, indicating that a mission path has been successfully published. Before using the received mission path, SCH performs explicit sanitation by trimming leading and trailing white space characters. This step is crucial, as paths written via shared memory or generated by external components may include newline characters or trailing spaces, which would otherwise lead to file access failures when opening or

parsing the mission XML. By removing these extraneous characters, SCH ensures that the resulting string represents a valid file system path. As shown in Figure 6.2, SCH successfully extracts a clean and valid mission file path from BLB and logs it for verification. The absence of file not found or parsing errors following this step confirms that SCH not only reads the mission path correctly but also robustly handles formatting irregularities. This behavior demonstrates that the interaction between mission-selector and SCH via the Blackboard is functionally correct and resilient against common string handling issues in IPC.

6.3 Node extraction correctness

Goal: Demonstrate that SCH extracts the correct Action Node IDs from a selected mission XML and that the extracted identifiers match the action nodes defined in the original mission design exported from Groot2. Test set: Correctness of node extraction in SCH using mission 12 as a deterministic test set.



(a) mission12.xml File

(b) Execution Result

Figure 6.3: Node Extraction Test

Evidence: Figure 6.3(a) illustrates the mission12.xml file generated by the Groot2 mission design tool which serves as the input for this test. The XML file specifies the full behavioral structure of mission 12 in the BehaviorTree.CPP format version 4.

Within the BehaviorTree ID="mission12" element, executable operations are explicitly defined as Action nodes with unique identifiers such as NavSetMode, NavArm, NavTakeOff, NavBeep, DetStart, wait, NavAscend, and NavLand. The mission also includes references to SubTree elements such as SafetySubtree and InsSubtree as well as control flow constructs like Sequence, Fallback, and RetryUntilSuccessful, which serve structural purposes rather than being executable themselves.

Once mission 12 is selected for execution SCH loads the XML file and performs node extraction as implemented in sch.cpp. During this process SCH traverses the mission definition to identify executable action nodes. Structural and control flow nodes are deliberately excluded, as they do not represent executable runtime actions.

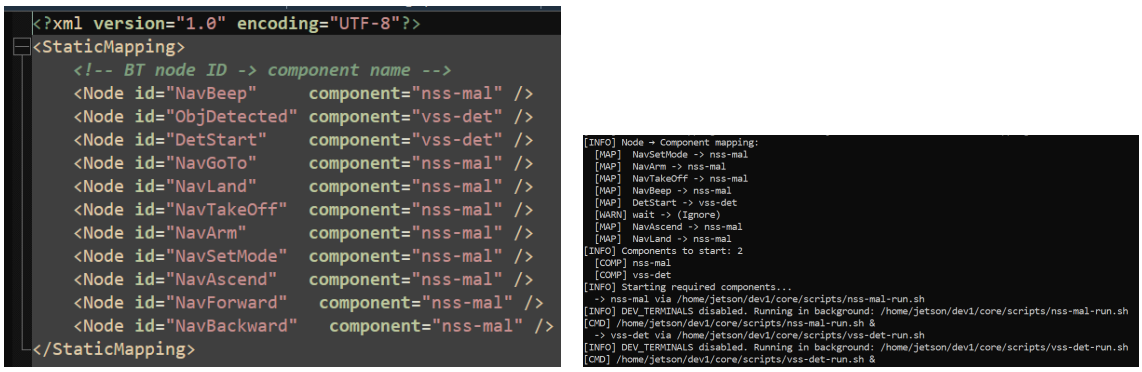
The runtime output in Figure 6.3(b) displays the result of this extraction phase. SCH first confirms the resolved mission file path and the selected main behavior tree then lists the identified action node IDs under the "XML Action Nodes" section. The output enumerates NavSetMode, NavArm, NavTakeOff, NavBeep, DetStart, wait, NavAscend, and NavLand exactly matching the Action elements defined in mission12.xml. A direct comparison of Figures 6.3(a) and 6.3(b) reveals a one to one correspondence between the ¡Action¿ nodes in the XML and action identifiers reported by SCH. No additional nodes appear and no valid actions are missing. This confirms that SCH correctly parses the mission definition and distinguishes between executable actions as designed. SCH is consistently producing an accurate set of action node identifiers. It establishes a reliable basis for the assignment of mission actions to software components and their orchestration during runtime.

6.4 Component Mapping

Goal: Correct components are launched for a given mission.

Evidence: Figure 6.4(a) presents the static mapping XML file used by SCH to translate mission action node identifiers into the corresponding software components to be executed. Each Node entry defines a deterministic link between a BehaviorTree action ID and the component responsible for that action. For instance, action nodes such as NavSetMode, NavArm, NavTakeOff, NavBeep, NavAscend, and NavLand are assigned to the navigation component nss-mal. Whereas DetStart is assigned to the detection component vss-det. This mapping file therefore serves as the scheduler's authoritative configuration for determining which components are required for a given mission.

A notable feature of the sch.cpp implementation is that SCH does not launch components directly from the raw list of action nodes. Instead, it aggregates the resolved component names into a required component set and removes duplicates before initiating any processes. This behavior is visible in Figure 6.4(b), where several mission actions map to the same component nss-mal, yet SCH reports "Components to start: 2" and lists only nss-mal and vss-det. This demonstrates that SCH correctly deduplicates the component list. Thereby avoiding redundant launches and minimizing unnecessary resource consumption on the companion computer.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<StaticMapping>
    <!-- BT node ID -> component name -->
    <Node id="NavBeep"      component="nss-mal" />
    <Node id="ObjDetected" component="vss-det" />
    <Node id="DetStart"     component="vss-det" />
    <Node id="NavGoTo"      component="nss-mal" />
    <Node id="NavLand"      component="nss-mal" />
    <Node id="NavTakeOff"   component="nss-mal" />
    <Node id="NavArm"       component="nss-mal" />
    <Node id="NavSetMode"   component="nss-mal" />
    <Node id="NavAscend"    component="nss-mal" />
    <Node id="NavForward"    component="nss-mal" />
    <Node id="NavBackward"    component="nss-mal" />
</StaticMapping>
```

(a) Static Mapping File

```
[INFO] Node → Component mapping:
  [MAP]  NavSetMode -> nss-mal
  [MAP]  NavArm -> nss-mal
  [MAP]  NavTakeOff -> nss-mal
  [MAP]  NavBeep -> nss-mal
  [MAP]  DetStart -> vss-det
  [WARN] wait -> (Ignore)
  [MAP]  NavAscend -> nss-mal
  [MAP]  NavLand -> nss-mal
[INFO] Components to start: 2
  [COMP] nss-mal
  [COMP] vss-det
[INFO] Starting required components...
  -> nss-mal via /home/jetson/dev1/core/scripts/nss-mal-run.sh
[INFO] DEV_TERMINALS disabled. Running in background: /home/jetson/dev1/core/scripts/nss-mal-run.sh
[CMD] /home/jetson/dev1/core/scripts/nss-mal-run.sh &
  -> vss-det via /home/jetson/dev1/core/scripts/vss-det-run.sh
[INFO] DEV_TERMINALS disabled. Running in background: /home/jetson/dev1/core/scripts/vss-det-run.sh
[CMD] /home/jetson/dev1/core/scripts/vss-det-run.sh &
```

(b) Execution Result

Figure 6.4: Component Mapping and Starting Test

The execution output further shows that SCH safely handles unmapped or intentionally unsupported nodes. In this scenario, the wait node appears in the extracted node list but does not correspond to a software component defined in the static mapping file. Instead of treating this as a configuration error, SCH reports wait -> (Ignore). This behavior confirms that the mapping logic in sch.cpp distinguishes between executable nodes that require external components and internal utility nodes that do not. Consequently, SCH avoids launching irrelevant components and prevents mission startup failures caused by nodes intended for internal handling.

Figure 6.4(b) demonstrates that after reporting the required component set, SCH transitions into the startup phase and executes the corresponding component launch scripts. The output shows that nss-mal is started via nss-mal-run.sh and vss-det via vss-det-run.sh. Additionally, SCH notes that developer terminals are disabled and therefore runs these scripts in the background. This also proves the process launch logic in sch.cpp which supports both interactive and background execution modes depending on configuration.

Together, the static mapping file in Figure 6.4(a) and the runtime output in Figure 6.4(b) confirm the correct behavior of SCH. SCH accurately determines the minimal set of components required for the mission. It correctly ignores nodes that do not require a component. It also eliminates redundant mappings. Finally, SCH launches each necessary component exactly once by using the configured startup scripts.

6.5 Boot and availability

Goal: Show SCH is alive immediately after power-on and reaches an operational state.

Evidence: Figure 6.5 shows the reboot sequence of the Jetson Nano, marking the beginning of the test. After system becomes available, SCH starts automatically through its configured systemd user service. This confirms that SCH operates as a

```
Last login: Wed Jan 28 13:44:34 2026 from 10.57.6.178
jetson@jetson-desktop:~$ uptime
 13:45:36 up 0 min,  2 users,  load average: 4,92, 1,14, 0,38
jetson@jetson-desktop:~$ systemctl --user status mss-sch --no-pager -l
● mss-sch.service - Mission Scheduling Service (mss-sch) - user session
   Loaded: loaded (/home/jetson/.config/systemd/user/mss-sch.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2026-01-28 13:45:23 CET; 21s ago
 Main PID: 7252 (mss_sch)
   CGroup: /user.slice/user-1000.slice/user@1000.service/mss-sch.service
           └─7252 /home/jetson/dev1/mss-sch/build/mss_sch

Jan 28 13:45:28 jetson-desktop mss_sch[7252]: [blb] Destroy the Shared memory
Jan 28 13:45:28 jetson-desktop mss_sch[7252]: [sem] Semaphore name              MAVLINK_SEMAPHORE
Jan 28 13:45:28 jetson-desktop mss_sch[7252]: [sem] Semaphore name              NAVIGATION_SEMAPHORE
Jan 28 13:45:28 jetson-desktop mss_sch[7252]: [sem] Semaphore name              DETECTION_SEMAPHORE
Jan 28 13:45:28 jetson-desktop mss_sch[7252]: [sem] Semaphore name              INSPECTION_SEMAPHORE
Jan 28 13:45:28 jetson-desktop mss_sch[7252]: [sem] Semaphore name              GIMBAL_SEMAPHORE
Jan 28 13:45:28 jetson-desktop mss_sch[7252]: [shm] Shared memory name          BLACKBOARD_SHARED_MEMORY
Jan 28 13:45:28 jetson-desktop mss_sch[7252]: [blb] Shared memory address       0x7f89d74000
Jan 28 13:45:28 jetson-desktop mss_sch[7252]: [blb] Shared memory opened
Jan 28 13:45:28 jetson-desktop mss_sch[7252]: [blb] GetId for SCHEDULER
```

Figure 6.5: Rebooting Jetson Nano

persistent background service and not as a manually executed application. Such a setup is essential for fully autonomous system startup.

Figure 6.6 verifies that the mss-sch.service is loaded and in an active (running) state shortly after boot. The presence of a valid main process ID confirms that SCH has launched successfully and has not terminated due to configuration or dependency errors. The service status output also displays early Blackboard related log entries, including shared memory and semaphore initialization. These correspond to the Blackboard setup phase implemented in sch.cpp.

Figure 6.7 provides detailed runtime output from boot until component activation. The logs show that SCH initiates the BLB startup and attempts to attach to the Blackboard. An initial attempt to start BLB in a graphical terminal fails because no display is available. SCH then continues execution and attaches to the Blackboard successfully. This sequence demonstrates the robustness of the process launch logic in sch.cpp, which allows SCH to operate reliably in headless environments. Once the Blackboard connection is established, SCH launches the mission selector component. It then enters a waiting phase until the mission path appears in shared memory. This behavior is visible in repeated log messages reporting that the mission message is empty. When a valid mission path is received, SCH loads the mission XML, extracts the action nodes, applies the node to component mapping, and determines the minimal required component set. The logs confirm that these components start automatically using their configured run scripts.

The final part of the logs shows SCH entering its monitoring loop. During this phase SCH periodically evaluates heartbeat information via the Blackboard. The recurring heartbeat messages indicate that SCH has completed its initialization sequence and is operating in a stable supervision mode. Together, these results confirm that SCH starts automatically after power-on, completes its initialization procedure, launches the required mission components, and transitions into steady-state opera-

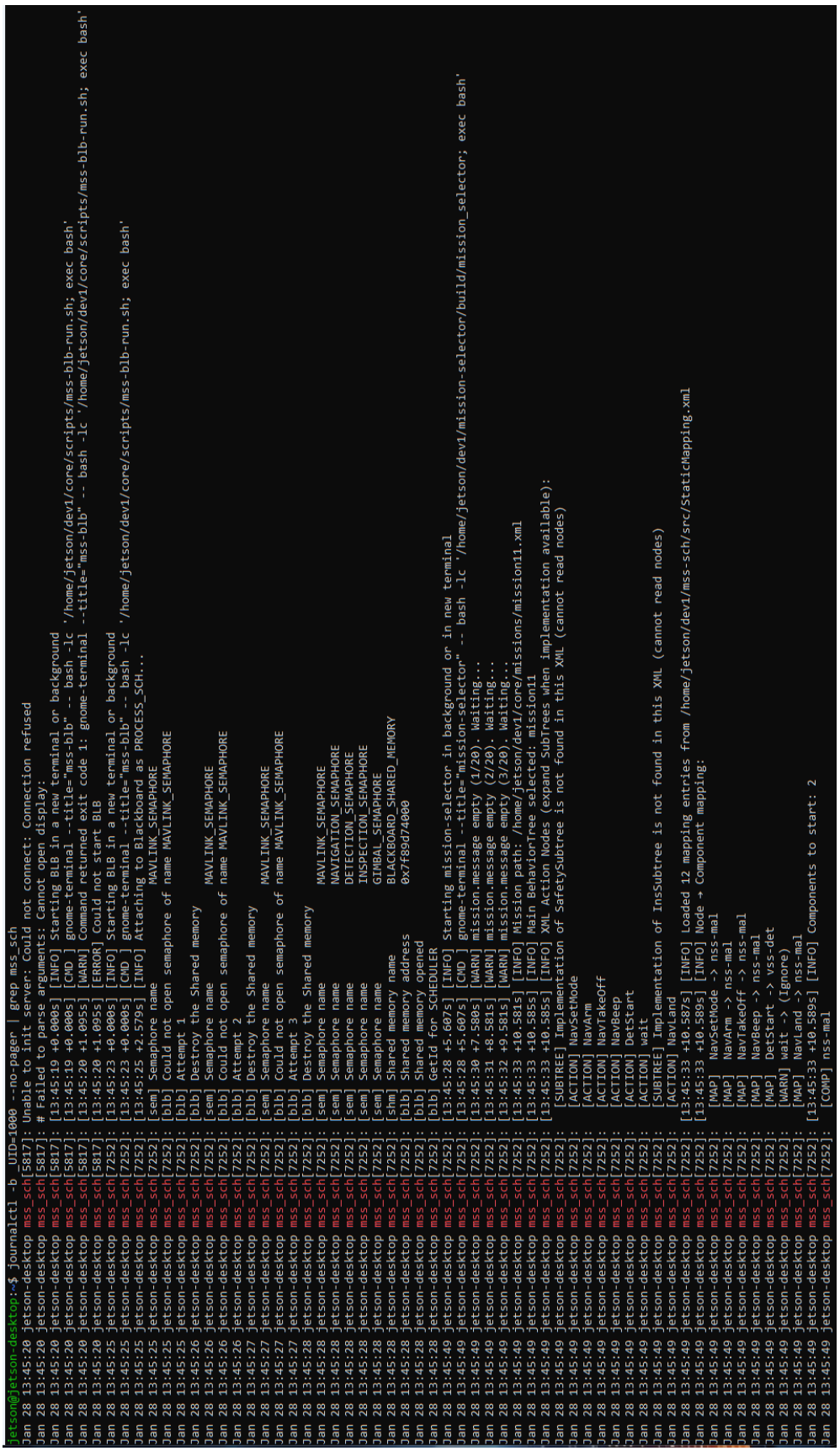


Figure 6.6: Checking the status of SCH

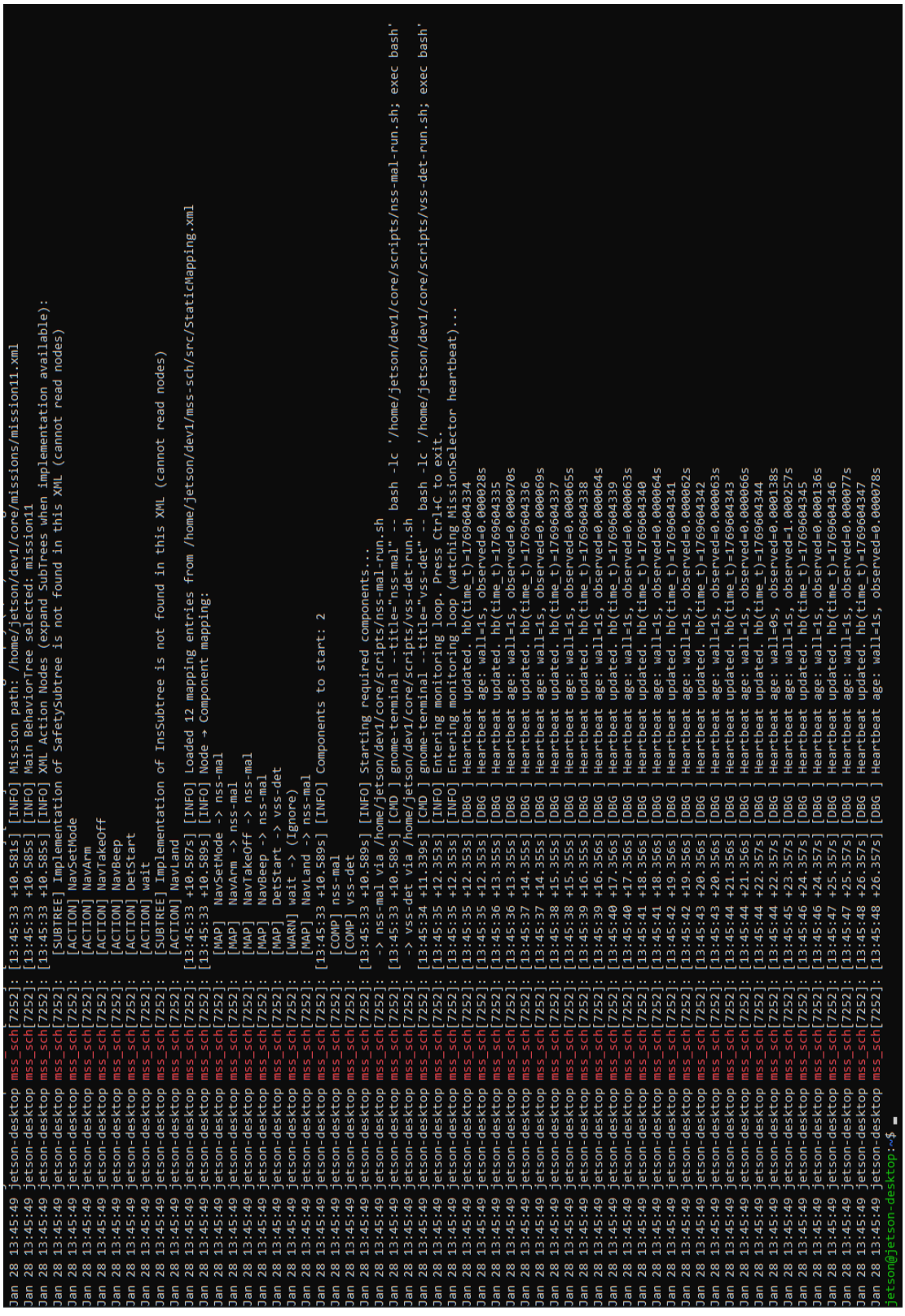


Figure 6.7: Displaying logs since the boot till component activation

tion without manual input.

6.6 Heartbeat supervision

Goal: System detects the delay in components heartbeat and restarts the component again. Test: Intentionally inducing delay in heartbeat to test system.

```
[16:13:18 +2.028s] [INFO] Entering monitoring loop (watching MissionSelector heartbeat)...
[16:13:18 +2.028s] [DBG ] Heartbeat updated. hb(time_t)=1769440398
[16:13:18 +2.028s] [DBG ] Heartbeat age: wall=0s, observed=0.000043s
[start-vss-det] Starting container 'vss-det'
[16:13:19 +3.029s] [DBG ] Heartbeat updated. hb(time_t)=1769440399
[16:13:19 +3.029s] [DBG ] Heartbeat age: wall=0s, observed=0.000081s
[16:13:20 +4.029s] [DBG ] Heartbeat updated. hb(time_t)=1769440400
[16:13:20 +4.029s] [DBG ] Heartbeat age: wall=0s, observed=0.000082s
[16:13:21 +5.029s] [DBG ] Heartbeat updated. hb(time_t)=1769440401
[16:13:21 +5.029s] [DBG ] Heartbeat age: wall=0s, observed=0.000097s
[MSL] TEST_HEALTHCHECK: stopping heartbeats intentionally
[16:13:22 +6.029s] [DBG ] Heartbeat age: wall=1s, observed=1.000204s
[16:13:23 +7.030s] [DBG ] Heartbeat age: wall=2s, observed=2.000591s
[16:13:24 +8.030s] [DBG ] Heartbeat age: wall=3s, observed=3.000788s
[16:13:24 +8.030s] [ERROR] MissionSelector heartbeat TIMEOUT. wall_age=3s (> 2s). Killing and restarting...
[16:13:24 +8.030s] [CMD ] pgrep -af mission_selector
4557 /home/jetson/dev1/mission-selector/build/mission_selector
13215 /home/jetson/dev1/mission-selector/build/mission_selector
16160 sh -c pgrep -af mission_selector
[16:13:24 +8.098s] [CMD ] pkill -TERM -f mission_selector
[16:13:24 +8.188s] [WARN] Command returned exit code 143: pkill -TERM -f mission_selector
[16:13:24 +8.188s] [WARN] pkill returned exit=143 (interpreting as non-fatal).
[16:13:24 +8.488s] [CMD ] pgrep -af mission_selector
16321 sh -c pgrep -af mission_selector
[16:13:24 +8.529s] [INFO] DEV_TERMINALS disabled. Running in background: /home/jetson/dev1/mission-selector/build/mission_selector
[16:13:24 +8.529s] [CMD ] /home/jetson/dev1/mission-selector/build/mission_selector &
[16:13:24 +8.531s] [INFO] Restart timings: kill=498ms, start_total=501ms
```

Figure 6.8: Test for delay detection in component's heartbeat

Evidence: During runtime supervision in Figure 6.5, the SCH component continuously monitored the MissionSelector heartbeat through the shared blackboard. Heartbeats were logged at a steady frequency of about 1 Hz, and the reported heartbeat age stayed below 0.1 ms immediately after each update, indicating timely and consistent signal transmission. To verify the health monitoring behavior, heartbeat emission in the MissionSelector was deliberately halted. After the final valid heartbeat, the observed heartbeat age increased linearly until it reached 3.0 s, at which point SCH registered a timeout, exceeding the configured 2 s threshold. This corresponds to an effective failure detection latency of roughly 3 s, arising from the 1 Hz monitoring rate and the discrete sampling nature of the mechanism. Once the timeout was detected, SCH terminated all running MissionSelector instances and restarted the component. The measured recovery behavior shows a kill latency of about 498 ms and a total restart latency of about 501 ms from timeout detection to process relaunch, demonstrating fast and deterministic recovery of the supervision and restart mechanism.

7 Conclusion and Future Scope

7.1 Conclusion

This thesis explored the problem of mission based software component initialization in autonomous UAV systems that use resource limited companion computers. As UAV software moves toward microservice and component based architectures, managing the startup, coordination, and monitoring of many software components has become a key challenge. Manual startup, static scripts, and operator dependent setup are limiting for autonomous missions system design.

To solve this problem, the thesis proposed and implemented a SCH for automated mission based orchestration within a blackboard based UAV architecture. The SCH reads mission descriptions defined in BT based XML files, links mission tasks to specific software components using a static mapping process, and manages the entire component life cycle, including startup, monitoring, and controlled shutdown.

The work began by reviewing the foundations of UAV system design, component based architectures, IPC, blackboard systems, and containerized deployment on embedded Linux. A literature review examined existing work on mission representation, system startup, and runtime orchestration. The review identified a gap between high level mission specification tools and practical orchestration systems suitable for embedded onboard platforms.

Based on these findings, the thesis presented the SCH design, focusing on loose coupling, predictable behavior, and low runtime overhead. The architecture combines blackboard based shared memory communication with microservice principles such as modularity, fault isolation, and clear system boundaries. Shared memory IPC was used to implement both the blackboard data exchange and component supervision efficiently.

The SCH was developed as a C++ daemon running on an embedded Linux platform, Jetson Nano. It integrates mission selection, XML parsing, component mapping, process control, and heartbeat based health monitoring. Unlike cloud based orchestration frameworks, SCH remains lightweight and suitable for onboard use, avoiding complexity and excess overhead. The results showed that mission based orchestration improves reliability and repeatability compared with manual or static initialization.

The key contribution of this thesis is the design and implementation of a reusable mission scheduling component that links mission representation with runtime orchestration. SCH enables autonomous, reliable, and fault aware mission execution. It provides a practical architectural solution for UAVs and other robotic systems

that rely on distributed software running on limited onboard resources.

In conclusion, the research successfully achieved its goals and demonstrated that mission based software orchestration is an essential step toward dependable autonomy in future UAV systems.

7.2 Future Scope

With the current implementation, the paper fulfills its stated objectives, and there are numerous directions in which future work could extend the capabilities of the proposed solution.

Custom Mission Nodes: The current system relies on predefined nodes from the BehaviorTree.cpp framework, which provide basic execution and control capabilities but are restricted to generic logic suitable only for standard behaviors. Defining custom nodes enables adaptation of both behavior and execution semantics to meet system specific requirements.

Common configuration file for all the components: Currently, runtime parameters are hard-coded within individual component source files, which makes deployment and maintenance more complex. Centralizing configuration in the .env.dev file for all components would allow each component to load only the parameters it needs, such as camera indices, IP addresses, ports, log destinations, and sensor identifiers. This centralization enables system specific settings to be adjusted in one place rather than across multiple repositories. Consequently, deploying the system to different Jetson devices would only require updating a single configuration file to reflect hardware connections and mission parameters, leading to a consistent setup, fewer configuration errors, and easier collaboration within the development team.

Integration with Mission Feedback: Currently, the SCH runs independently from the internal state of the BT. Linking orchestration decisions with mission feedback could enable finer control, such as restarting components only when specific missions fail, or adjusting scheduling dynamically as missions progress.

Scalability for Multiple UAVs: This work focuses on a single UAV. Future research could expand SCH to coordinate multiple UAVs, managing distributed system initialization and shared objectives across different vehicles.

Developing these directions would make the Mission Scheduling Component more adaptive, reliable, and scalable, helping advance autonomous UAV systems toward greater resilience and operational flexibility.

8 Bibliography

8.1 TUC Bibliography

[17] Batbayar Battseren. "Software Architecture for Real-Time Image Analysis in Autonomous MAV Missions". In: (2024).

[43] Batbayar Battseren, Uranchimeg Tudevdagva, and Wolfram Hardt. "A finite state machine based adaptive mission control of mini aerial vehicle". In: *Embedded Selforganising Systems* 5.1 (2018), pp. 6–10.

[44] R Harradi et al. "AREIOM: adaptive research multicopter platform". In: *IOP Conference Series: Materials Science and Engineering*. Vol. 1019. 1. IOP Publishing. 2021, p. 012022.

[45] Uranchimeg Tudevdagva et al. "System Design of Insulator Inspection Software for HV Distribution Lines". In: *2023 International Conference on Technology and Policy in Energy and Electric Power (ICT-PEP)*. 2023, pp. 47–51. DOI: 10.1109/ICT-PEP60152.2023.10351173.

[46] Batbayar Battseren et al. "Development of an Adaptive MAV Platform for Autonomous Inspection of High Voltage Power Lines". In: *NEIS 2023; Conference on Sustainable Energy Supply and Energy Storage Systems*. 2023, pp. 186–191.

[47] Marco Stephan, Batbayar Battseren, and Uranchimeg Tudevdagva. "Autonomous unmanned aerial vehicle development: Mavlink abstraction layer". In: *International Symposium on Computer Science, Computer Engineering and Educational Technology (ISCSET-2020), Lauta Germany*. 2020, pp. 45–49.

8.2 Bibliography

[1] Paul G Fahlstrom, Thomas J Gleason, and Mohammad H Sadraey. *Introduction to UAV systems*. John Wiley & Sons, 2022.

[2] RANDAL W. BEARD and TIMOTHY W. McLAIN. *Small Unmanned Aircraft: Theory and Practice*. 2nd ed. Princeton University Press, 2012. ISBN: 9780691149219. URL: http://www.jstor.org/stable/j.ctt7sbc4 (visited on 01/01/2026).

[3] John David Blom. *Unmanned Aerial Systems: A Historical Perspective*. Fort Leavenworth, KS: U.S. Army Command and General Staff College, 2010.

[4] Alan Hobbs. "Unmanned aircraft systems". In: *Human factors in aviation*. Elsevier, 2010, pp. 505–531.

[5] Oscar C. Valderrama-Riveros and Fernando G. Tinetti. "UV System Architecture: Challenges and Evolution". In: *Applied Sciences* 15.8 (2025). ISSN: 2076-3417. DOI: 10.3390/app15084080. URL: https://www.mdpi.com/2076-3417/15/8/4080.

[6] Gelin Wang et al. "Heterogeneous Flight Management System (FMS) Design for Unmanned Aerial Vehicles (UAVs): Current Stages, Challenges, and Opportunities". In: *Drones* 7.6 (2023). ISSN: 2504-446X. DOI: 10.3390/drones7060380. URL: https://www.mdpi.com/2504-446X/7/6/380.

[7] Emad Ebeid et al. "A survey of Open-Source UAV flight controllers and flight simulators". In: *Microprocessors and Microsystems* 61 (2018), pp. 11–20. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2018.05.002. URL: https://www.sciencedirect.com/science/article/pii/S0141933118300930.

[8] Jason Whelan, Abdulaziz Almehmadi, and Khalil El-Khatib. "Artificial intelligence for intrusion detection systems in Unmanned Aerial Vehicles". In: *Computers and Electrical Engineering* 99 (2022), p. 107784. ISSN: 0045-7906. DOI: https://doi.org/10.1016/j.compeleceng.2022.107784. URL: https://www.sciencedirect.com/science/article/pii/S0045790622000842.

[9] Anis Koubaa et al. "AERO: AI-Enabled Remote Sensing Observation with Onboard Edge Computing in UAVs". In: *Remote Sensing* 15.7 (2023). ISSN: 2072-4292. DOI: 10.3390/rs15071873. URL: https://www.mdpi.com/2072-4292/15/7/1873.

[10] Foisal Ahmed and Maksim Jenihhin. "A Survey on UAV Computing Platforms: A Hardware Reliability Perspective". In: *Sensors* 22.16 (2022). ISSN: 1424-8220. URL: https://www.mdpi.com/1424-8220/22/16/6286.

[11] Behzad Boroujerdian et al. "MAVBench: Micro Aerial Vehicle Benchmarking". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 894–907. DOI: 10.1109/MICRO.2018.00077.

[12] Patrick McEnroe, Shen Wang, and Madhusanka Liyanage. "A Survey on the Convergence of Edge Computing and AI for UAVs: Opportunities and Challenges". In: *IEEE Internet of Things Journal* 9.17 (2022), pp. 15435–15459. DOI: 10.1109/JIOT.2022.3176400.

[13] Lea Matlekovic and Peter Schneider-Kamp. "From Monolith to Microservices: Software Architecture for Autonomous UAV Infrastructure Inspection". In: *Embedded Systems and Applications*. EMSA 2022. Academy and Industry Research Collaboration Center (AIRCC), Mar. 2022. DOI: 10.5121/csit.2022.120622. URL: http://dx.doi.org/10.5121/csit.2022.120622.

[14] Lea Matlekovic, Filip Juric, and Peter Schneider-Kamp. "Microservices for autonomous UAV inspection with UAV simulation as a service". In: *Simulation Modelling Practice and Theory* 119 (2022), p. 102548. ISSN: 1569-190X. DOI: https://doi.org/10.1016/j.simpat.2022.102548. URL: https://www.sciencedirect.com/science/article/pii/S1569190X22000466.

[15] Carlos Solon Soares Guimarães et al. "Architecture Based on Services and Microservices for the Development of Unmanned Aerial Systems in Precision Agriculture". In: *2024 IEEE International Conference on Agrosystem Engineering, Technology & Applications (AGRETA)*. 2024, pp. 226–231. DOI: 10.1109/AGRETA61912.2024.10949017.

[16] Juan A. Besada et al. "Drones-as-a-service: A management architecture to provide mission planning, resource brokerage and operation support for fleets of drones". In: *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 2019, pp. 931–936. DOI: 10.1109/PERCOMW.2019.8730838.

[17] Batbayar Battseren. "Software Architecture for Real-Time Image Analysis in Autonomous MAV Missions". In: (2024).

[18] Oscar Marius Nierstrasz et al. *Object-oriented software composition*. Vol. 1. Prentice Hall Upper Saddle River, NJ, USA: 1995.

[19] Oscar Nierstrasz and Laurent Dami. "Component-oriented software technology". In: *Object-oriented software composition* 1 (1995), pp. 3–28.

[20] Oscar Nierstrasz. "Research topics in software composition." In: *LMO*. Vol. 95. 1995, pp. 193–204.

[21] Stephen Wong et al. "A scalable approach to multi-style architectural modeling and verification". In: *13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008)*. IEEE. 2008, pp. 25–34.

[22] Davide Brugali and Patrizia Scandurra. "Component-based robotic engineering (Part I) [Tutorial]". In: *IEEE Robotics & Automation Magazine* 16.4 (2009), pp. 84–96. DOI: 10.1109/MRA.2009.934837.

[23] Davide Brugali and Azamat Shakhimardanov. "Component-Based Robotic Engineering (Part II)". In: *IEEE Robotics & Automation Magazine* 17.1 (2010), pp. 100–112. DOI: 10.1109/MRA.2010.935798.

[24] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. "Component-Based Development Process and Component Lifecycle". In: *2006 International Conference on Software Engineering Advances (ICSEA'06)*. 2006, pp. 44–44. DOI: 10.1109/ICSEA.2006.261300.

[25] Michael Jackson. "Engineering by Software: System Behaviours as Components". In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Cham: Springer International Publishing, 2017, pp. 1–17. ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_1. URL: https://doi.org/10.1007/978-3-319-67425-4_1.

[26] Ivica Crnkovic. "Component-based software engineering for embedded systems". In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: Association for Computing Machinery, 2005, pp. 712–713. ISBN: 1581139632. DOI: 10.1145/1062455.1062631. URL: https://doi.org/10.1145/1062455.1062631.

[27] Marilyn Wolf. *Computers as components: principles of embedded computing system design*. Elsevier, 2012.

[28] Aniruddha Dhaske. "A Survey on Modern Inter-process Communication (IPC) Mechanisms". In: *International Journal for Research in Applied Science and Engineering Technology* 12 (Nov. 2024), pp. 1430–1436. DOI: 10.22214/ijraset.2024.65382.

[29] Ms Krishnaveni and Ruby Durairaj. "Comparing and Evaluating the Performance of Inter Process Communication Models in Linux Environment". In: (Oct. 2016), pp. 51–55.

[30] V. Kavitha, Sushmita Ravi, and Muruganandham Anand. "Comparative analysis of IPC mechanisms for Linux and UC/OS-II". In: 9 (Jan. 2016), pp. 1843–1853.

[31] Patricia K. Immich, Ravi S. Bhagavatula, and Dr. Ravi Pendse. "Performance analysis of five interprocess communication mechanisms across UNIX operating systems". In: *Journal of Systems and Software* 68.1 (2003), pp. 27–43. ISSN: 0164-1212. DOI: https://doi.org/10.1016/S0164-1212(02)00134-6. URL: https://www.sciencedirect.com/science/article/pii/S0164121202001346.

[32] Spyros Tzafestas and Elpida Tzafestas. "The Blackboard Architecture in Knowledge-Based Robotic Systems". In: *Expert Systems and Robotics*. Ed. by Timothy Jordanides and Bruce Torby. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 285–317. ISBN: 978-3-642-76465-3.

[33] Barbara Hayes-Roth. "A blackboard architecture for control". In: *Artificial Intelligence* 26.3 (1985), pp. 251–321. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(85)90063-3. URL: https://www.sciencedirect.com/science/article/pii/0004370285900633.

[34] Ian D. Craig. "Blackboard Systems". In: *Artificial Intelligence Review* 2.2 (1988), pp. 103–118. DOI: 10.1007/BF00140399.

[35] Bruce Draper, Allen Hanson, and Edward Riseman. "Learning Blackboard-based Scheduling Algorithms for Computer Vision". In: *IJPRAI* 7 (Mar. 1995). DOI: 10.1142/S0218001493000169.

[36] Fritz Solms. "Supporting Organizational Qualities Through Architectural Patterns". In: Jan. 2016, pp. 594–599. DOI: 10.5220/0005836105940599.

[37] David Bernstein. "Containers and Cloud: From LXC to Docker to Kubernetes". In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84. DOI: 10.1109/MCC.2014.51.

[38] "Container Technology". In: *Cloud Computing Technology*. Singapore: Springer Nature Singapore, 2023, pp. 295–342. ISBN: 978-981-19-3026-3. DOI: 10.1007/978-981-19-3026-3_7. URL: https://doi.org/10.1007/978-981-19-3026-3_7.

[39] Harri Manninen. "Evaluation of Container Technologies for an Embedded Linux Device". In: (2020).

[40] Srikanth Nimmagadda. "Linux Namespaces and cgroups as OS Primitives for Lightweight Virtualization: Architecture, Isolation Mechanisms, and Performance Evaluation". In: *Turkish Journal of Computer and Mathematics Education (TURCOMAT)* 9 (Aug. 2018), pp. 811–822. DOI: 10.61841/turcomat.v9i2.15258.

[41] Kilian Telschig, Andreas Schonberger, and Alexander Knapp. "A Real-Time Container Architecture for Dependable Distributed Embedded Applications". In: Aug. 2018, pp. 1367–1374. DOI: 10.1109/COASE.2018.8560546.

[42] Ragini Gupta and Klara Nahrstedt. *Performance Characterization of Containers in Edge Computing*. May 2025. DOI: 10.48550/arXiv.2505.02082.

[43] Batbayar Battseren, Uranchimeg Tudevdagva, and Wolfram Hardt. "A finite state machine based adaptive mission control of mini aerial vehicle". In: *Embedded Selforganising Systems* 5.1 (2018), pp. 6–10.

[44] R Harradi et al. "AREIOM: adaptive research multicopter platform". In: *IOP Conference Series: Materials Science and Engineering*. Vol. 1019. 1. IOP Publishing. 2021, p. 012022.

[45] Uranchimeg Tudevdagva et al. "System Design of Insulator Inspection Software for HV Distribution Lines". In: *2023 International Conference on Technology and Policy in Energy and Electric Power (ICT-PEP)*. 2023, pp. 47–51. DOI: 10.1109/ICT-PEP60152.2023.10351173.

[46] Batbayar Battseren et al. “Development of an Adaptive MAV Platform for Autonomous Inspection of High Voltage Power Lines”. In: *NEIS 2023; Conference on Sustainable Energy Supply and Energy Storage Systems*. 2023, pp. 186–191.

[47] Marco Stephan, Batbayar Battseren, and Uranchimeg Tudevdagva. “Autonomous unmanned aerial vehicle development: Mavlink abstraction layer”. In: *International Symposium on Computer Science, Computer Engineering and Educational Technology (ISCSET-2020), Lauta Germany*. 2020, pp. 45–49.

[48] Patrick Ulam, Zsolt Kira, and Thomas Collins. “Mission Specification and Control for Unmanned Aerial and Ground Vehicles for Indoor Target Discovery and Tracking”. In: *Proc SPIE* 7694 (Apr. 2010). DOI: 10.1117/12.849842.

[49] Fran Real et al. “Experimental Evaluation of a Team of Multiple Unmanned Aerial Vehicles for Cooperative Construction”. In: *IEEE Access* PP (Jan. 2021), pp. 1–1. DOI: 10.1109/ACCESS.2021.3049433.

[50] Matteo Iovino et al. *Comparison between Behavior Trees and Finite State Machines*. 2024. arXiv: 2405.16137 [cs.RO]. URL: https://arxiv.org/abs/2405.16137.

[51] Razan Ghzouli et al. “Behavior trees in action: a study of robotics applications”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. SPLASH ’20. ACM, Nov. 2020, pp. 196–209. DOI: 10.1145/3426425.3426942. URL: http://dx.doi.org/10.1145/3426425.3426942.

[52] Razan Ghzouli et al. *Behavior Trees and State Machines in Robotics Applications*. Aug. 2022. DOI: 10.48550/arXiv.2208.04211.

[53] Swaib Dragule et al. “Effects of specifying robotic missions in behavior trees and state machines”. In: *Journal of Computer Languages* 85 (2025), p. 101330. ISSN: 2590-1184. DOI: https://doi.org/10.1016/j.cola.2025.101330. URL: https://www.sciencedirect.com/science/article/pii/S2590118425000164.

[54] Vittorio A Ziparo et al. “Petri net plans: A framework for collaboration and coordination in multi-robot systems”. In: *Autonomous Agents and Multi-Agent Systems* 23.3 (2011), pp. 344–383.

[55] Vittorio Ziparo et al. “Petri Net Plans A Formal Model for Representation and Execution of Multi-Robot Plans”. In: vol. 1. Jan. 2008, pp. 79–86. DOI: 10.1145/1402383.1402399.

[56] Bruno Lacerda and Pedro U. Lima. “Petri net based multi-robot task coordination from temporal logic specifications”. In: *Robotics and Autonomous Systems* 122 (2019), p. 103289. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2019.103289. URL: https://www.sciencedirect.com/science/article/pii/S0921889019302441.

[57] Martin Schörner et al. “Modeling and Execution of Coordinated Missions in Reconfigurable Robot Ensembles”. In: *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*. 2020, pp. 290–293. DOI: 10.1109/IRC.2020.00053.

[58] Martin Molina et al. “Specifying Complex Missions for Aerial Robotics in Dynamic Environments”. In: Oct. 2016.

[59] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. “Temporal-Logic-Based Reactive Mission and Motion Planning”. In: *IEEE Transactions on Robotics* 25.6 (2009), pp. 1370–1381. DOI: 10.1109/TRO.2009.2030225.

[60] M Georgeff and A Rao. “Modeling rational agents within a BDI-architecture”. In: *Proc. 2nd Int. Conf. on Knowledge Representation and Reasoning (KR'91). Morgan Kaufmann*. of. 1991, pp. 473–484.

[61] Eric Gil. “Mutrose: A goal-oriented framework for mission specification and decomposition of multi-robot systems”. PhD thesis. Master's thesis, UnB, 2021. ix, 3, 13, 14, 20, 26, 2021.

[62] Markus Gutmann and Bernhard Rinner. “Mission Specification and Execution of Multidrone Systems”. In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 451–456. DOI: 10.23919/DATE51398.2021.9474207.

[63] Darko Bozhinoski et al. “FLYAQ: enabling non-expert users to specify and generate missions of autonomous multicopters”. In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. ASE '15. Lincoln, Nebraska: IEEE Press, 2015, pp. 801–806. ISBN: 9781509000241. DOI: 10.1109/ASE.2015.104. URL: https://doi.org/10.1109/ASE.2015.104.

[64] Juan A. Besada et al. “Drone Mission Definition and Implementation for Automated Infrastructure Inspection Using Airborne Sensors”. In: *Sensors* 18.4 (2018). ISSN: 1424-8220. DOI: 10.3390/s18041170. URL: https://www.mdpi.com/1424-8220/18/4/1170.

[65] R. Simmons and D. Apfelbaum. “A task description language for robot control”. In: *Proceedings. 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in Theory, Practice and Applications (Cat. No.98CH36190)*. Vol. 3. 1998, 1931–1937 vol.3. DOI: 10.1109/IROS.1998.724883.

[66] Ashkan Zehfroosh and Herbert G. Tanner. “Reactive motion planning for temporal logic tasks without workspace discretization”. In: *2019 American Control Conference (ACC)*. 2019, pp. 4872–4877. DOI: 10.23919/ACC.2019.8814420.

[67] Georgios Kontes and Michail Lagoudakis. “Coordinated Team Play in the Four-Legged RoboCup League”. In: *19th IEEE International Conference on Tools with Artificial Intelligence(ICTAI 2007)*. Vol. 1. 2007, pp. 109–116. DOI: 10.1109/ICTAI.2007.164.

[68] Claudio Menghi et al. *Specification Patterns for Robotic Missions*. Jan. 2019. DOI: 10.48550/arXiv.1901.02077.

[69] Christopher S. Timperley et al. "ROSDiscover: Statically Detecting Run-Time Architecture Misconfigurations in Robotics Systems". In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. 2022, pp. 112–123. DOI: 10.1109/ICSA53651.2022.00019.

[70] David Portugal, Rui P Rocha, and João P Castilho. "Inquiring the robot operating system community on the state of adoption of the ROS 2 robotics middleware". In: *International Journal of Intelligent Robotics and Applications* (2024), pp. 1–26.

[71] Uchechukwu Awada et al. ": A Dependency-Aware Multi-Task Orchestration in Federated Aerial Computing". In: *IEEE Transactions on Vehicular Technology* 71.1 (2022), pp. 805–819. DOI: 10.1109/TVT.2021.3127011.

[72] Gergely Magyar, Peter Sinčák, and Zoltán Krizsán. "Comparison Study of Robotic Middleware for Robotic Applications". In: *Emergent Trends in Robotics and Intelligent Systems*. Ed. by Peter Sinčák et al. Cham: Springer International Publishing, 2015, pp. 121–128.

[73] Christopher S Timperley et al. "Rosdiscover: Statically detecting run-time architecture misconfigurations in robotics systems". In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. IEEE. 2022, pp. 112–123.

[74] Francesco Lumpp et al. "Enabling Kubernetes Orchestration of Mixed-Criticality Software for Autonomous Mobile Robots". In: *IEEE Transactions on Robotics* PP (Jan. 2023), pp. 1–12. DOI: 10.1109/TRO.2023.3334642.

[75] Jiaqiang Zhang, Xianjia Yu, and Tomi Westerlund. "Enhancing the Resilience of ROS 2-Based Multi-Robot Systems with Kubernetes: A Case Study on UWB-Based Relative Positioning". In: *Sensors* 25.16 (2025), p. 5067.

[76] Uchechukwu Awada et al. "AirEdge: A dependency-aware multi-task orchestration in federated aerial computing". In: *IEEE Transactions on Vehicular Technology* 71.1 (2021), pp. 805–819.

[77] Andreas Bauer, Martin Leucker, and Christian Schallhart. "Runtime Reflection: Dynamic model-based analyis of component-based distributed embedded systems". In: *Modellierung von Automotive Systems* (2006).

[78] Tobias Schwalb and Klaus D. Müller-Glaser. "Extension of component-based models for control and monitoring of embedded systems at runtime". In: *2011 22nd IEEE International Symposium on Rapid System Prototyping*. 2011, pp. 142–148. DOI: 10.1109/RSP.2011.5929988.

[79] Tobias Schwalb et al. “Component-based models for runtime control and monitoring of embedded systems”. In: *Proceedings of the 5th International Workshop on Model Based Architecting and Construction of Embedded Systems*. ACES-MB ’12. Innsbruck, Austria: Association for Computing Machinery, 2012, pp. 31–36. ISBN: 9781450318006. DOI: 10.1145/2432631.2432637. URL: https://doi.org/10.1145/2432631.2432637.

[80] K.S. Rubin, P.M. Jones, and C.M. Mitchell. “OFMspert: inference of operator intentions in supervisory control using a blackboard architecture”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 18.4 (1988), pp. 618–637. DOI: 10.1109/21.17380.

[81] S.S. Yau and Bing Xia. “An approach to distributed component-based real-time application software development”. In: *Proceedings First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC ’98)*. 1998, pp. 275–283. DOI: 10.1109/ISORC.1998.666798.

[82] Xi Chen et al. “Application of Software Watchdog as a Dependability Software Service for Automotive Safety Relevant Systems”. In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN’07)*. 2007, pp. 618–624. DOI: 10.1109/DSN.2007.14.

[83] Marco D. Santambrogio et al. “Enabling technologies for self-aware adaptive systems”. In: *2010 NASA/ESA Conference on Adaptive Hardware and Systems*. 2010, pp. 149–156. DOI: 10.1109/AHS.2010.5546266.

[84] C. Watterson and D. Heffernan. “Runtime verification and monitoring of embedded systems”. In: *IET Software* 1 (5 2007), pp. 172–179. DOI: 10.1049/iet-sen:20060076. eprint: https://digital-library.theiet.org/doi/pdf/10.1049/iet-sen%3A20060076. URL: https://digital-library.theiet.org/doi/abs/10.1049/iet-sen%3A20060076.

[85] Artur Jutman, Konstantin Shibin, and Sergei Devadze. “Reliable health monitoring and fault management infrastructure based on embedded instrumentation and IEEE 1687”. In: *2016 IEEE AUTOTESTCON*. 2016, pp. 1–10. DOI: 10.1109/AUTEST.2016.7589605.

[86] Long Wang et al. “An OS-level Framework for Providing Application-Aware Reliability”. In: *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC’06)*. 2006, pp. 55–62. DOI: 10.1109/PRDC.2006.19.

[87] Xuanyao Qu et al. “Design of Automatic Search and Rescue UAV Based on Jetson Nano Combined with PX4 Pixhawk Flight Controller and Color Recognition Technology”. In: *2024 International Conference on Electrical Drives, Power Electronics & Engineering (EDPEE)*. 2024, pp. 460–466. DOI: 10.1109/EDPEE61724.2024.00092.

[88] Haris Ijaz et al. “A UAV-Assisted Edge Framework for Real-Time Disaster Management”. In: *IEEE Transactions on Geoscience and Remote Sensing* 61 (2023), pp. 1–13. DOI: 10.1109/TGRS.2023.3306151.

[89] NVIDIA Corporation. *NVIDIA Jetson Nano*. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development/. Accessed: 2026-01-17. NVIDIA, n.d.

[90] NVIDIA Corporation. *Get Started With Jetson Nano Developer Kit*. https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit.

[91] Agus Kurniawan. "Introduction to NVIDIA Jetson Nano". In: *IoT Projects with NVIDIA Jetson Nano: AI-Enabled Internet of Things Projects for Beginners*. Berkeley, CA: Apress, 2021, pp. 1–6. ISBN: 978-1-4842-6452-2. DOI: 10.1007/978-1-4842-6452-2_1. URL: https://doi.org/10.1007/978-1-4842-6452-2_1.

[92] D. Erricolo. "C++, a better language for engineering applications". In: *IEEE Antennas and Propagation Magazine* 42.4 (2000), pp. 95–101. DOI: 10.1109/74.868057.

[93] Markus Giftthaler et al. "The control toolbox — An open-source C++ library for robotics, optimal and model predictive control". In: *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*. 2018, pp. 123–129. DOI: 10.1109/SIMPAR.2018.8376281.

[94] Matteo Iovino. "Learning Behavior Trees for Collaborative Robotics". PhD thesis. KTH Royal Institute of Technology, 2023.

Chemnitzer Informatik-Berichte

In der Reihe der Chemnitzer Informatik-Berichte sind folgende Berichte erschienen:

CSR-25-01 Md. Ali Awlad, Hasan Saadi Jaber Aljzaere, Wolfram Hardt, AUTOSAR Software Component for Atomic Straight Driving Patterns, März 2025, Chemnitz

CSR-25-02 Billava Vasantha Monisha, Hasan Saadi Jaber Aljzaere, Wolfram Hardt, Automotive Software Component for QT Based Car Status Visualization, März 2025, Chemnitz

CSR-25-03 Zahra Khadivi, Batbayar Battseren, Wolfram Hardt, Acoustic-Based MAV Propeller Inspection, Mai 2025, Chemnitz

CSR-25-04 Tripti Kumari Shukla, Ummay Ubaida Shegupta, Wolfram Hardt, Time Management Tool Development to Support Self-regulated Learning, August 2025, Chemnitz

CSR-25-05 Ambu Babu, Ummay Ubaida Shegupta, Wolfram Hardt, Development of a Retrieval Model based Backend of a Tutoring Agent, August 2025, Chemnitz

CSR-25-06 Shahid Ismail, Ummay Ubaida Shegupta, Wolfram Hardt, Development of a Generative Model based Backend of Tutoring Agent, August 2025, Chemnitz

CSR-25-07 Chaitanya Sravanthi Akula, Ummay Ubaida Shegupta, Wolfram Hardt, Integration of Learning Analytics into the ARC-Tutoring Workbench, August 2025, Chemnitz

CSR-25-08 Jörn Roth, Reda Harradi, Wolfram Hardt, Implementation of a Path Planning Algorithm for UAV Navigation, Dezember 2025, Chemnitz

CSR-25-09 Alhassan Khalil, Reda Harradi, Stephan Rupf, Wolfram Hardt, Development of an Automation Framework for 1D Measurement, Dezember 2025, Chemnitz

CSR-26-01 Vismay Gunda, Shadi Saleh, Wolfram Hardt, Cloud-Based AI Solutions for Ensuring Data Quality in Predictive Models, Februar 2026, Chemnitz

CSR-26-02 Sami Mansoor Alavi, Shadi Saleh, Wolfram Hardt, Continuous Integration for Cloud-Based Swarm Farming Applications, Februar 2026, Chemnitz

Chemnitzer Informatik-Berichte

CSR-26-03 Sarah Onyinyechi Obasi, Shadi Saleh, Wolfram Hardt, Cloud-Based AI for Data Completeness Analysis and Improvement in Predictive Modeling, März 2026, Chemnitz

CSR-26-04 Atul Chandra Nath, Reda Harradi, Wolfram Hardt, GPS-based UAV Precision Landing, April 2026, Chemnitz

CSR-26-05 Josey Mol Sibi, Batbayar Battseren, Wolfram Hardt, Real Time Multi Stream Video Transmission in Autonomous UAV, Mai 2026, Chemnitz

CSR-26-06 Silpa Geetha Harshakumar, Batbayar Battseren, Wolfram Hardt, Simultaneous Video Streaming and Recording with Geotagging in Autonomous UAV, Mai 2026, Chemnitz

CSR-26-07 Monica Chambial, Batbayar Battseren, Wolfram Hardt, Implementation and Enhancement of a Mission Scheduling Component for Autonomous UAV, Mai 2026, Chemnitz

Chemnitzer Informatik-Berichte

ISSN 0947-5125

Herausgeber: Fakultät für Informatik, TU Chemnitz
Straße der Nationen 62, D-09111 Chemnitz