Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Engineering Multi-Robot Mission Coordination in the Service Robot Domain

Gabriel Siqueira Rodrigues

Tese apresentada como requisito parcial
para conclusão do Doutorado em Informática

Orientadora
Prof.ª Dr.ª Genaína Nunes Rodrigues

Coorientador
Prof. Dr. Patrizio Pelliccione

Brasília
2025

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Engenharia de Coordenação de Missões Multi-Robô no Domínio da Robótica de Serviço

Gabriel Siqueira Rodrigues

Tese apresentada como requisito parcial
para conclusão do Doutorado em Informática

Orientadora
Prof.ª Dr.ª Genaína Nunes Rodrigues

Coorientador
Prof. Dr. Patrizio Pelliccione

Brasília
2025

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Doutorado em Informática

Coordenador: Prof. Dr. Rodrigo Bonifácio

Banca examinadora composta por:

Prof.ª Dr.ª Genaína Nunes Rodrigues (Orientadora) — CIC/UnB
Prof. Dr. Patrizio Pelliccione (Coorientador) — Gran Sasso Science Institute
Prof. Dr. Marco Autili — University of L'Aquila
Prof.ª Dr.ª Elisa Nakagawa — ICMC/USP
Prof.ª Dr.ª Célia Ghedini Ralha — CIC/UnB

CIP — Catalogação Internacional na Publicação

Rodrigues, Gabriel Siqueira.

Engineering Multi-Robot Mission Coordination in the Service Robot Domain / Gabriel Siqueira Rodrigues. Brasília : UnB, 2025.
244 p. : il. ; 29,5 cm.

Tese (Doutorado) — Universidade de Brasília, Brasília, 2025.

1. multi-robot systems, 2. mission coordination, 3. service robots,
4. robot simulation, 5. lightweight simulation

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Engineering Multi-Robot Mission Coordination in the Service Robot Domain

Gabriel Siqueira Rodrigues

Tese apresentada como requisito parcial
para conclusão do Doutorado em Informática

Prof.ª Dr.ª Genaína Nunes Rodrigues (Orientadora)
CIC/UnB

Prof. Dr. Patrizio Pelliccione (Coorientador)
Gran Sasso Science Institute

Prof. Dr. Marco Autili
University of L'Aquila

Prof.ª Dr.ª Elisa Nakagawa
ICMC/USP

Prof.ª Dr.ª Célia Ghedini Ralha
CIC/UnB

Prof. Dr. Rodrigo Bonifácio
Coordenador do Doutorado em Informática

Brasília, 2 de Março de 2025

Resumo

Título:*Engenharia de coordenação de múltiplos robôs no domínio de robôs de serviço*

Os robôs transformaram indústrias como a automotiva ao assumirem tarefas tediosas, repetitivas e perigosas dentro de ambientes fabris. Tradicionalmente, esses robôs são grandes máquinas operando em ambientes controlados. No entanto, nos últimos anos, houve um crescimento significativo no desenvolvimento de robôs móveis projetados para atuar em espaços compartilhados com humanos, incluindo hospitais, hotéis e restaurantes. Conhecidos como robôs de serviço, esses sistemas são capazes de navegar em ambientes centrados em humanos e realizar tarefas como manipulação de objetos e inspeção do ambiente usando câmeras e sensores. Para explorar plenamente o potencial dos robôs de serviço é essencial estabelecer mecanismos eficazes de coordenação. No entanto, orquestrar múltiplos robôs em espaços compartilhados com humanos apresenta desafios substanciais de engenharia devido à natureza dinâmica desses ambientes e às incertezas inerentes que os caracterizam. Além disso, embora a garantia de qualidade seja crucial para esses sistemas, a simulação de sua lógica de coordenação é particularmente complexa devido à forte dependência de fatores ambientais. Esta tese avança o estado da arte na engenharia de sistemas multi-robô no domínio dos robôs de serviço, com foco especial em arquitetura e simulação. Para isso, esta pesquisa introduz uma nova arquitetura, chamada MissionControl, projetada para facilitar o desenvolvimento e a coordenação de missões MRS. O MissionControl reduz a complexidade do sistema ao definir um modelo de componentes que estrutura o sistema em componentes encapsulados e reutilizáveis, com responsabilidades bem definidas. Além disso, ele integra modelos de componentes e de tempo de execução com processos de coordenação, priorizando modificabilidade e integrabilidade. Uma implementação prototípica do MissionControl foi desenvolvida para comunicação entre robôs e ROS para execução de tarefas. A arquitetura foi avaliada por meio de experimentos no simulador MORSE e inspeções sistemáticas baseadas em diretrizes de análise de arquitetura de software. Os resultados experimentais indicam uma melhoria nas taxas de sucesso das missões e uma redução nas falhas causadas por esgotamento de recursos, mesmo sob condições de incerteza. Por fim, esta tese contribui ainda com um design de simulação leve chamado HMR Sim, projetado para avaliar a lógica de coordenação. O

framework de simulação proposto é eficiente em termos de recursos e suporta a execução de testes por meio de uma linguagem específica de domínio, permitindo a validação sistemática dos comportamentos dos sistemas multi-robô.

Palavras-chave: sistemas multi-robôs, coordenação de missão, robôs de serviço, simulação de robôs, simulação leve

Abstract

Robots have transformed industries such as automotive manufacturing by taking over tedious, repetitive, and hazardous tasks within factory settings. Traditionally, these robots are large machines operating in highly controlled environments. However, in recent years, there has been a significant rise in the development of mobile robots designed to assist in human-shared spaces, including hospitals, hotels, and restaurants. Known as service robots, these systems can navigate human-centric environments and perform tasks such as object manipulation and environmental inspection using cameras and sensors. To fully leverage the potential of service robots in social settings, effective coordination mechanisms must be established. However, orchestrating multiple robots in shared human spaces presents substantial engineering challenges due to the open-ended nature of these environments and the inherent uncertainties they introduce. Furthermore, while quality assurance is crucial for such systems, simulating their coordination logic is particularly complex due to its strong dependence on environmental factors. This thesis advances the state-of-the-art in engineering autonomous multi-robot systems in the service robot domain, particularly in architecture and simulation. It proposes novel architectural solutions to manage the complexity and uncertainty inherent in dynamic environments. To this end, this research introduces a new architecture, MissionControl, designed to streamline the development and coordination of multi-robot missions. MissionControl mitigates system complexity by defining a component model that structures the system into encapsulated reusable components with clear responsibilities. In addition, it integrates both the component and runtime models with coordination processes, prioritizing modifiability, integrability, and autonomy. A prototype implementation of MissionControl was developed for inter-robot communication and Robot Operating System for task execution. The architecture was evaluated through controlled experiments in the MORSE simulator and systematic inspections based on software architecture analysis guidelines. Experimental results indicate an improvement in mission success rates and a reduction in failures due to resource depletion, even under uncertain conditions. Lastly, this thesis contributes to a lightweight simulation design called HMR Sim, which is tailored to evaluate coordination logic. The proposed simulation framework is resource efficient and supports the execution

of tests using a domain-specific language, enabling systematic validation of multi-robot systems behaviors.

Keywords: multi-robot systems, mission coordination, service robots, robot simulation, lightweight simulation

Acknowledgements

And now we have reached the end of a journey, and what a long journey it was. This path was not traveled alone, and I am deeply grateful to everyone who supported me along the way.

I would like to express my deepest gratitude to my advisor, Prof. Dr. Genaína Rodrigues, who has been my mentor since my undergraduate studies. Thank you for your unwavering guidance and patience throughout all these years. I am also incredibly grateful to my co-advisor, Prof. Dr. Patrizio Pelliccione, for his invaluable academic guidance, which was instrumental in helping me clarify the contributions of this work.

I extend my sincere thanks to the members of my examination committee, Prof. Dr. Marco Autili and Profa. Dra. Elisa Nakagawa, for their time and feedback. A special thank you goes to Profa. Dra. Célia Ghedini Ralha for her detailed review, which contributed significantly to the quality of the final version of this thesis.

I also wish to acknowledge Prof. Dr. Rodrigo Bonifácio and Profa. Dra. Claudia Nalon, who served as the head of the department during my final years. I deeply appreciate their patience and support during the times I needed it most.

My time in the lab would not have been the same without my colleagues. I am grateful to Ricardo Caldas, Gabriel Araujo, Vicente de Moraes, and Eric Gil, my close collaborators in those robotics adventure - thank you for the partnership.

I owe a special debt of gratitude to my family. To my mother, Elizabeth Siqueira, and my wife, Nágila Ramos, thank you for your unconditional support and for your understanding during the many times I was unavailable while working on this research.

Finally, this work was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, by FAPDF under Calls 03/2018, 04/2021 and CNPq process 313215/2021-9.

Dedication

To all the students navigating the uncertainties of research - I hope you too will soon be putting the final touches on your thesis.

Contents

List of Figures

List of Tables

Acronyms

BDD Behavior-Driven Development. 26

BT Behavior Tree. 4, 15–17, 29, 44–46, 49, 85, 90

CPS Cyber-Physical Systems. 13, 14

DD Design Decisions. 33

DEECo Distributed Emergent Ensembles of Components. 6, 13, 37, 47, 48, 66

DSL Domain-Specific Language. 8, 37

EBCS Ensemble-Based Component Systems. 13, 14

ECS Entity-Component-System. 10, 23–25, 72–74, 78

HTN Hierarchical Task Network. 4, 7, 11, 12, 29, 43, 90, 91, 93

IA Instantaneous Assignment. 14, 30, 39, 92

iHTN Instantiated HTN. 11, 12, 32, 34, 42, 43, 60, 90, 91

MAS Multi-Agent Systems. 9, 15

MR Multi-Robot Tasks. 14, 30, 92, 93

MRS Multi-robot Systems. 1–7, 9, 10, 13, 29, 64, 90, 97

MRTA Multi-Robot Task Allocation. 14, 92, 93

NPC Non-Playable Character. 15

ROS Robot Operating System. 6, 7, 15, 17–19, 48, 49, 63, 64, 66, 70, 74–77, 79, 82, 84, 85

SR Single-Robot Tasks. 14, 93

ST Single-Task Robots. 14, 30, 92

TA Time-Extended Assignments. 14, 93

UAV Unmanned Aerial Vehicle. 89, 90, 93

UGV Unmanned Ground Vehicle. 90, 93

Chapter 1

Introduction

The rising costs of healthcare and the increasing burden of personnel burnout are pressing global issues and are expected to worsen due to an aging population. In recent years, we have witnessed a growing interest in *Service Robots*, a particular robot that performs useful tasks for humans or equipment, excluding industrial automation applications [1–3]. Such robots can replace humans in dangerous and tedious tasks in various domains, such as [4]: healthcare[5], hospitality[6], and retail [7]. In recent decades, robots have been employed, mainly in industrial settings, with a significant impact on cost reduction. In an industrial setting, robots typically operate by performing a repetitive task within a confined space specifically designed for their operation. In recent years, there has been a notable increase in the availability of mobile robots adequate for service robot applications, suggesting that mass production of these robots is on the horizon. This anticipated reduction in unit costs presents a significant opportunity to deploy robots in areas such as healthcare, helping to address the critical challenges of cost and personnel strain.

Multi-robot Systems (MRS) [8] refers to the use of multiple robots in a given application. Heterogeneous MRS [9] refers to the deployment of robots with various characteristics within a system. In a heterogeneous MRS, the robots collaborate and combine their capabilities to perform specific missions. The primary motivation for using heterogeneous MRS is its ability to execute a diverse range of missions that require a spectrum of robot capabilities. By employing robots with varying capabilities, it is possible to achieve this without requiring any single robot to be overly complex or to possess an exhaustive set of capabilities.

1.1 Motivation

The implementation of service robots holds the promise of beneficial effects on service industries by replacing humans in dangerous tasks, reducing risk and harm, replacing humans in tedious tasks, freeing resources for more critical activities, and lowering labor expenses, making services more accessible. Furthermore, these benefits are crucial in the coming years due to the aging populations in many countries.

Our vision is that organizations will use off-the-shelf robots from various providers to create end-user solutions, similar to how they currently use off-the-shelf personal computers and networking infrastructure to implement cost-effective information systems. Off-the-shelf robots with generic capabilities can be produced in large numbers, benefiting from economies of scale, which, in the long term, will lower prices and enable widespread adoption. However, off-the-shelf components are not sufficient to provide applications, as each environment instance, e.g., each hospital and care home, has its own characteristics and policies. The characteristics that vary across environment instances include the set of available robots (possibly from different providers), the environment map (e.g., how the building is organized), and other installed equipment, such as elevators and automated doors. We may also have different policies, for example, in which situation a person can access a given zone. The varied attributes of an environment can result in distinct and changing requirements for applications that are unique to each environment instance. Additionally, because of the intricate nature of the entire system, creating software tailored specifically for an environment instance from scratch may be expensive, error-prone, and economically inefficient. Consequently, the development of software to manage robots within an environment may be a bottleneck to adoption.

1.2 Problem

The successful deployment of MRS in the service robot domain is significantly dependent on managing the software complexity, controlling the robots, and the development costs. This thesis adopts a software architecture approach to the problem, examining how to decompose complex software into components with well-defined responsibilities that can support diverse missions and integrate heterogeneous robots.

The software architecture serves as the fundamental blueprint for a software system, defining its components, their relationships, and the principles guiding its design and evolution [10]. By defining an architecture for mission coordination, we intend to identify a standard set of problems in this domain and encapsulate them as components. This

approach allows a developer to tackle the complexity of the software by breaking the complexity of the entire system into smaller, more manageable problems.

A well-defined architecture is crucial for achieving the desired quality attributes, which represent the system's non-functional requirements. These attributes, such as performance, security, reliability, usability, modifiability, and scalability, are not directly related to what the system does, but rather how well it does it. The architectural decisions made early in the development lifecycle have a profound impact on the system's ability to satisfy these quality attributes.

Modifiability is an architectural quality attribute that refers to the ease with which the system can be adapted, extended, or updated to accommodate new requirements, functionalities, or changes in the operational environment. The modifiability benefits are twofold: on the one hand, it enables quick, efficient modification of the system to accommodate changing requirements. Accommodating change is fundamental, as the system must be in sync with the evolving dynamic environments in which service robots operate. On the other hand, modifiability allows software to be reused in similar but slightly different environments, let us say, in different hospitals, by providing defined extension points where the system can be adapted to cater for differences between environments. This can lower the overall cost of the system and, in consequence, ease the barrier to adopting service robots in practice.

Integrability is another architectural quality attribute, crucial when dealing with contributions from multiple sources and the need to incorporate diverse hardware and software components. In complex MRS, different teams or organizations may be responsible for developing specific capabilities, such as perception, navigation, manipulation, or high-level planning. Furthermore, the system might need to interact with existing infrastructure or incorporate off-the-shelf robotic platforms and software libraries. A robust architecture must provide clear guidelines and standards for seamlessly integrating these disparate components into the overall system. This includes defining common communication protocols, data formats, and interface specifications. Strong integrability ensures that modules can work harmoniously together, enabling the creation of a cohesive, functional multi-robot system that leverages the expertise and resources of various contributors. Without a focus on modifiability and integrability, the development and maintenance of complex MRS in the dynamic service domain would become prohibitively difficult, hindering their widespread adoption and limiting their potential impact. Therefore, the underlying system architecture must be designed with these qualities as core principles to ensure the long-term viability and adaptability of multi-robot solutions.

Moreover, testability is a crucial quality attribute that dictates how easily software can be tested. Well-defined encapsulation within software components significantly enhances

testability by limiting testing to the component's public interface and isolating it from internal implementation details. This allows for focused unit tests that verify the component's behavior in isolation. However, to ensure the entire system functions correctly, integration tests are essential. These tests verify the interactions and data flow between different encapsulated components. Simulation, particularly in complex systems such as multi-robotics, is a valuable tool for improving testability. By creating controlled, reproducible virtual environments, developers can simulate scenarios, edge cases, and failure modes that might be difficult or dangerous to test in the real world, thereby facilitating more comprehensive and efficient testing of both individual components and their integration.

Service robots can serve a diverse range of applications, from conversational robots that respond to inquiries in lobbies, engaging solely in human-initiated interactions, to robots that perform security patrols and execute structured tasks within their environment [11]. In this thesis, we focus on structured missions, in which robots execute a sequence of tasks. Cleaning and delivery are illustrative examples of such missions.

1.3 Methodology

This thesis adopts a mixed-methods research approach [12], integrating quantitative and qualitative methodologies to comprehensively address the engineering challenges of MRSs in the service robot domain.

Initially, an exploratory research method [12] was used. This phase involved developing scenarios in which MRS could be applied in the service robot domain, in particular, the Lab Samples Logistics scenario, which served as a motivating example to concretize the challenges and opportunities within multi-robot mission coordination.

Through this case study and an extensive literature review, the research investigated the suitability and integration of various coordination approaches, examining how diverse methodologies such as Hierarchical Task Network (HTN), Behavior Tree (BT), and Ensemble-based systems could fit into an overall mission coordination problem. Building upon this exploratory phase, the research led us to the question:

RQ: *How to architect mission coordination in the service robot domain?*

To answer this question, we proposed MissionControl, a novel architecture designed to enhance the development and coordination of multi-robot missions in dynamic and uncertain environments. In MissionControl, we focus on modifiability and integrability quality attributes. These attributes guarantee that systems developed with the architecture can

be extended with new missions and tasks. The integrability quality attribute guarantees that the system can be extended to new environments with different surrounding systems.

The evaluation of this proposed architecture strategically combines quantitative and qualitative methods. The quantitative evaluation involved controlled experiments in a simulator to assess the architecture's effectiveness in promoting mission coordination, measured by improvements in mission success rates and reduced failures under uncertainty. Concurrently, a qualitative evaluation, conducted through systematic inspections and guideline-based analysis of architectural decisions, assessed the architecture's critical quality attributes, particularly modifiability and integrability. This dual approach provided a holistic validation of MissionControl, leveraging the complementary strengths of both paradigms, in line with the principles articulated by Creswell [12].

Moreover, when researching MRS coordination, we identified a gap in existing robotic simulators when applying to test and evaluate mission coordination. This led us to an additional research question.

RQ: *How to design a suitable robot simulator for mission coordination?*

Despite the acknowledged benefits of simulation for cost-effective and automated testing in robotics, its practical adoption remains limited due to computational costs and a lack of suitable tools. While physical simulators offer high fidelity and seamless transition to real-world deployment, they are computationally intensive, hindering large-scale multi-robot scenarios. In contrast, lightweight simulators, though efficient, sacrifice fidelity, making the transition to physical robots challenging and diminishing their effectiveness as testing tools. To bridge this gap, we propose a lightweight simulation approach designed specifically for testing multi-robot mission coordination.

1.4 Objectives

In summary, the aim of this thesis is to tackle the bottleneck of producing software to coordinate MRS in a service robot domain. We look into a software engineer perspective and focus on two more specific goals:

- To describe a software architecture for mission coordination focusing on modifiability and integrability
- To propose a simulation approach suitable for validating the mission coordination.

1.5 Contributions

The first contribution of this thesis was to describe scenarios for service robots and identify their challenges. In collaboration with other researchers, these scenarios were compiled into RoboMax, a repository of reusable exemplars.

The second and main contribution of this thesis is the MissionControl, an architecture for the development of applications in the domain of service robots. The architecture consists of a component model, runtime models, and processes to coordinate mission execution. The secondary contribution is the implementation of a prototype on top of an ensemble-based component framework [13, 14] for inter-robot communication. At the same time, the robot task execution was implemented in the Robot Operating System (ROS) middleware [15].

The experiments have been designed to consider three types of uncertainties [16]: *Resources/Capabilities Uncertainty*, *Environment Uncertainty*, and *Model or State Uncertainty*. Compared with a baseline approach, gains in execution quality were observed: increased mission success rate and a decreased number of robots that failed due to a completely depleted battery, even with some degree of uncertainty in the environment, robot models, and sensors.

In summary, the contributions of this thesis are as follows:

C1 An architecture for the coordination of missions in the service robots domain, constituted of components and runtime models as well as processes for coordinating the execution of MRS missions. The architecture description is accompanied by a concrete framework named MissionControl, which is implemented on top of the Distributed Emergent Ensembles of Components (DEECo) component framework [13, 14] for inter-robot communication. In contrast, robot task execution was implemented using the Robot Operating System [15]. (Chapter 3)

C2 A first artifact for the controlled experiment in the Lab Samples Logistics specification [11], where a workflow was designed and implemented, integrating ROS middleware and the Modular Open Robotics Simulation Engine (MORSE) simulator. For repeatability, experiments were conducted, and the collected data were statistically analyzed in Jupyter Notebooks [17]. The simulation artifact and data analysis scripts are also publicly available as Open Science material [18]. (Chapter 4)

C3 A second artifact of an internal inspection following a guideline-based software architecture analysis process [10]. The inspection process analyzed the conformity of MissionControl with a comprehensive set of guidelines through the lens of modifiability and integrability. As such, the artifact juxtaposes design decisions from the

classic software architecture literature [10] with those from robotic software development [19]. For reproducibility purposes, the artifact provides a comprehensive set of analyzed guidelines and the inspection process results [18]. (Chapter 4)

C4 A simulator tailored for MR coordination, aiming to balance computational efficiency with code portability. This approach enables the simulation of numerous robots in a single environment, facilitates minimal code modifications for real-world deployment, and provides integrated automated testing tools. The preliminary results of the evaluation have also been provided for performance and integration with ROS 2 purposes. (Chapter 5)

1.6 Publications

- Mehrnoosh Askarpour, Christos Tsigkanos, Claudio Menghi, Radu Calinescu, Patrizio Pelliccione, Sergio Garcia, Tim J. von Oertzen, Manuel Wimmer, Luca Berardinelli, Matteo Rossi, Marcello M. Bersani, and Gabriel S. Rodrigues. RoboMAX: Robotic Mission Adaptation eXemplars. In Proceedings of the 16th International. Symposium on Software Engineering for Adaptive and Self-Managing Systems. ACM, 2021. In this paper, the author contributed to the description and classification of robotic missions.

- Gabriel Rodrigues, Ricardo Caldas, Gabriel Araujo, Vicente de Moraes, Genaína Rodrigues, Patrizio Pelliccione, An architecture for mission coordination of heterogeneous robots, Journal of Systems and Software, Volume 191, 2022.

1.7 Thesis Outline

This thesis is structured to address the engineering challenges of multi-robot mission coordination, particularly in the service robot domain. The remainder of the thesis outline is as follows:

Chapter 2 covers essential knowledge relevant to MRS, including task decomposition (Hierarchical Task Networks), Robot Operating System, Behavior Trees, coalition formation, task allocation, software architecture quality attributes, and simulation techniques like Entity-Component-System and Behavior-Driven Development. Chapter 3 introduces *MissionControl*, a novel architecture designed to streamline the development and coordination of multi-robot missions. It details its component model, which structures the system into encapsulated, reusable components, and explains how it integrates with coordination processes to prioritize modifiability, integrability, and autonomy. Chapter 4

presents a comprehensive assessment of MissionControl. This includes a *quantitative evaluation* through controlled experiments in a simulator to measure improvements in mission success rates and reductions in failures, as well as a *qualitative evaluation* examining its modifiability and integrability based on software architecture analysis guidelines. Chapter 5 introduces *HMR Sim*, a lightweight simulation design developed as a prototype to evaluate coordination logic. It aims to support large-scale multi-robot scenarios efficiently while maintaining code portability to physical robots, and provides integrated tools for automated testing using a Domain-Specific Language (DSL). Chapter 6 discusses existing research and approaches in areas such as heterogeneous robot coordination architectures, mission specification languages, skill-based robotics, multi-robot task allocation, and methods for addressing uncertainty and enhancing quality attributes in robotic systems. Chapter 7 summarizes the main contributions of the thesis, including the MissionControl architecture and the HMR Sim lightweight simulation approach. It discusses their impact on engineering multi-robot mission coordination and outlines directions for future research.

Chapter 2

Background

The engineering of an autonomous system is a complex endeavor, primarily due to the necessity of executing sophisticated tasks within dynamic and uncertain environments. This complexity is particularly acute in the service robot domain, where MRS operate in non-structured, human-shared spaces. Successfully tackling this challenge requires drawing expertise from multiple distinct academic and technical disciplines.

The literature reflects this multidisciplinary nature: Artificial Intelligence (AI) emphasizes decision-making and mission planning, while Software Engineering (SE) focuses on architectural stability, robustness, and maintainability. Although this thesis adopts an SE perspectivecentered on the MissionControl architecture and the HMR Sim simulatorit first establishes the computational foundations for mission coordination. We therefore review concepts from AI and multi-agent systems, particularly those related to mission planning, which underpin MRS design and operation.

In this context, MRS refers to a heterogeneous group of autonomous robots that must coordinate to execute missions in shared environments. This concept builds on the foundations of Multi-Agent Systems (MAS), where multiple intelligent agentsphysical or virtualperceive, reason, and act autonomously to achieve specific goals [20].

Agents can be classified as reactive (responding directly to stimuli), deliberative (acting through planning), or hybrid (combining both). A Multi-Agent Systems (MAS) arises when such agents interact, sharing sensory information and pursuing common or conflicting objectives under decentralized and asynchronous control [21]. As part of Distributed Artificial Intelligence, MAS research focuses on coordinated, concurrent action and problem solving, providing the conceptual foundation for collective robotic behavior in MRS. Therefore, an MRS, particularly within the context of the thesis, refers to autonomous systems composed of a heterogeneous set of robots that must be coordinated to execute missions within an environment.

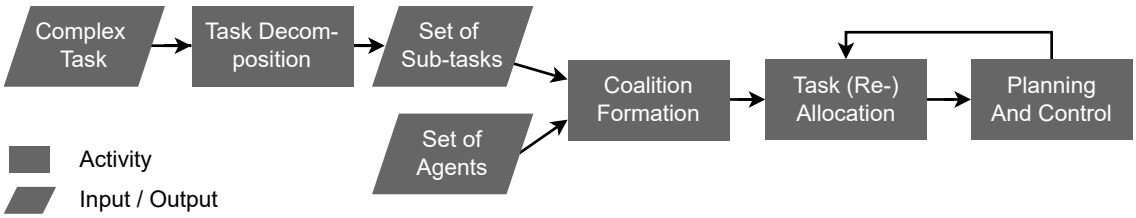


Figure 2.1: Workflow for the MRS [9, 22]

The remainder of this chapter provides the theoretical background necessary to understand our contributions to engineering MRS coordination.

2.1 MRS Workflow

The systematic design of MRS for accomplishing complex missions requires a structured approach. Kiener et al. [9, 22] describe a workflow that encompasses four main activities as illustrated in Figure 2.1. Firstly, *task decomposition* involves breaking complex missions into simpler, more manageable subtasks. Subsequently, *coalition formation* involves assembling suitable teams of robots. This is followed by *task allocation*, where individual subtasks are assigned to specific robot teams. Finally, *task execution/planning and control* involves meticulous planning and execution of a sequence of actions by each team within the environment to achieve their assigned subtasks and, ultimately, the mission's overall objective.

Regarding the activities of the MRS workflow, this thesis focuses on: (i) task decomposition as the input to MissionControl, (ii) coalition formation and task allocation as the core activities within MissionControl, and (iii) the output interface of MissionControl with the planning and control activities. Accordingly, the background discussion will emphasize these MRS workflow activities and the underlying technologies employed in MissionControl, as our contribution **C1**, where appropriate.

Different works in the literature proposed approaches with varying degrees of autonomy. For example, the proposal by Kiener et al. [22] employs a human designer to execute task decomposition and coalition formation, while task allocation and robot planning and control are performed autonomously by the robot teams.

2.1.1 Task Decomposition

Task decomposition is identified as the crucial first step in the systematic workflow for managing MRS as they tackle complex missions. This process is fundamentally defined by the goal of breaking down complex missions into simpler, more manageable subtasks [9].

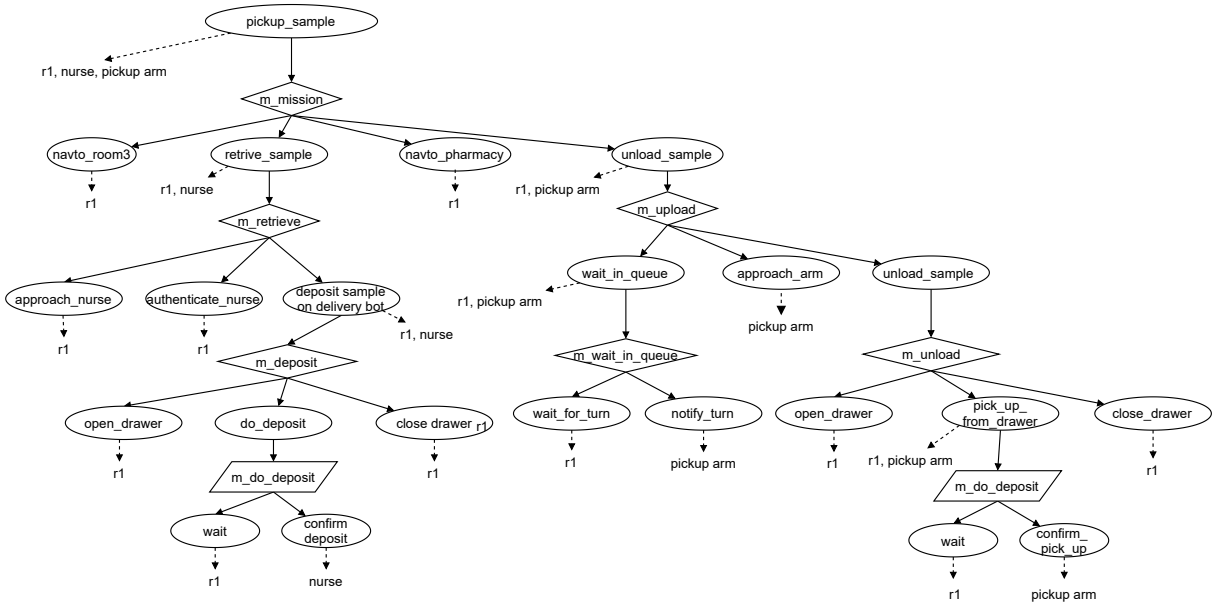


Figure 2.2: iHTN of the Lab Samples Logistics Exemplar

The mission itself represents a high-level abstract goal. The task decomposition activity generates a comprehensive plan, specifying (i) what tasks should be performed to accomplish the goal, and (ii) the dependencies that exist between these tasks. The resulting tasks are elementary and executable. When formalized using a hierarchical model such as HTN, the plan explicitly defines the execution order.

Planning is a discipline of Artificial Intelligence that aims to develop generic algorithms that enable autonomous systems to choose and organize their actions to achieve a goal by anticipating their effects [23]. Hierarchical Task Networks [24] is a formalism for task decomposition and planning. The Instantiated HTN (iHTN) formalism was proposed in Lesire et al. [25] to formalize a multi-robot collaborative mission. The iHTN tree represents a plan for an instance of a mission and can be computed by an HTN planner, such as Shop2 [26]. The iHTN is a tree of abstract and concrete tasks (found in its leaves).

Figure 2.2 illustrates an iHTN for the 'Lab Samples Logistics' mission. Tasks (ellipses) are efforts that a set of agents (robots or humans) must undertake. A task can be abstract or concrete. Methods refine abstract tasks. *Methods* are linked to tasks of a lower level and a type of ordering. The ordering can be sequential (diamond) or unordered (parallelogram). Sequential ordering requires that tasks be executed in sequence, so that each depends on the execution of the previous task. Unordered methods indicate that there is no dependency between tasks, and they can be performed in parallel (if assigned to different robots) or sequenced due to resource constraints.

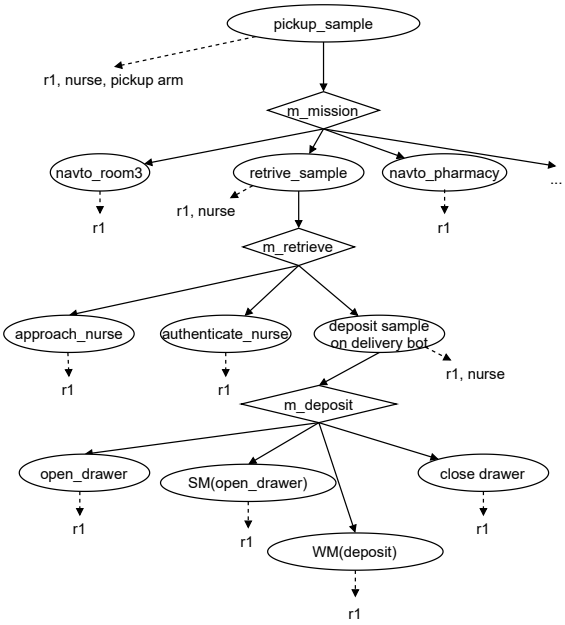


Figure 2.3: R1 Local Plan

2.1.1.1 HTN

A global mission specified in iHTN with tasks assigned to different agents (robots and people) can be broken down into a local plan for each agent, using an algorithm (provided by Lesire et al. [25]). Figure 2.3 illustrates the local plans for the 'r1' role in the 'Lab Samples Logistics' mission. Only the tasks assigned to the agent from the Global Plan are included in the local plan. In addition to assigned tasks, synchronizations are added to the local plan. Synchronizations are *sent* and *wait for messages* (SM, WM tasks in Figure 2.3), where (i) an agent waits for a notification before starting the execution of a task that has dependencies with tasks assigned to another agent, and (ii) an agent sends a notification to other agents after completing a task on which other agents depend. These synchronizations allow the execution of a subset of possible multi-robot collaborations in which the tasks assigned to different robots have interdependencies, but are not executed simultaneously by more than one robot. The tree nature of the model makes it efficient for repairing the plan, as repairs can be made locally in the branch where the failure occurs.

In previous work, we proposed the Deployment Variability Model (DVM), a structure to hierarchically model for planning at runtime via the GoalD framework[27] for adaptive deployment of heterogeneous systems. Like iHTN, it is a hierarchical model and can be repaired locally in the event of failure. In the DVM, each node represents a goal and the known list of strategies to achieve these goals. Each alternative can depend on sub-goals. Unlike iHTN, DVM keeps in each node an ordered list of alternative strategies for

achieving a (sub-) goal. In case of a change, this list is reevaluated, and a new strategy can be chosen.

2.1.2 Coalition Formation

After breaking down a complex task into smaller tasks, these smaller tasks should be assigned to a robot or a group of robots for execution. Coalition or team formation divides agents into coalitions or groups to execute the mission. These agents can be either non-cooperative or cooperative [9]. In this work, we focus on cooperative coalition formation, as this is the setting where many practical service robot systems would fall. Collaborative coalition agents are aware of each other and share the same goals. Their individual actions collectively lead to the achievement of the common goal.

2.1.2.1 Ensembles

Ensemble-Based Component Systems (EBCS) [13, 14] represent a class of component-based architectural paradigms designed to address the challenges inherent in engineering modern Cyber-Physical Systems (CPS) [28]. These complex systems necessitate strong requirements related to autonomicity, open-endedness, dynamicity, and self-adaptation, particularly in environments relying on opportunistic communication structures, such as mobile ad hoc networks. The core construct within this paradigm is the autonomous component ensemble. Although EBCSs was not originally proposed for MRS, many of the challenges present in acrshortcps exist in MRS, which are particular cases of CPS.

Distributed Emergent Ensembles of Components (DEECo) [13] is a popular realization of EBCS and is the specific framework that we use in this thesis, due to its simplicity. Other related and similar frameworks or languages focusing on component ensembles or attribute-based communication include Helena[29], another framework for building ensemble-based systems. The main difference between Helena and DEECo is that in Helena, a component explicitly indicates which ensembles it belongs to. Helena defines an ensemble as a set of roles, where each role is specified by attributes and operations, and a set of role connectors. jRESP[30], is another ensemble-based framework that implements the Software Component Ensemble Language (SCEL). In jRESP, components (called nodes) dynamically form ensembles by relying on the attribute-based communication paradigm. SCEL is described as a low-level generic theoretical model. TCOEL/TCOEF[31] utilizes the Trait-based COmponent Ensemble Language (TCOEL) and the Trait-based COmponent Ensemble Framework (TCOEF). TCOEL/TCOEF extends DEECo's model by allowing for the specification of collaboration hierarchies (nested ensembles), a feature DEECo previously lacked. Furthermore, the general topic of software engineering for col-

laborative software systems, including the research line focusing on autonomic component ensembles, has been an active area of research fueled by projects such as ASCENS[32].

In EBCS, the system is formed by autonomous components that are bound dynamically. *Ensembles* are a dynamic group of cooperating autonomous components that share a common goal.

Component is an autonomous entity that relies solely on its knowledge, i.e., a representation of its partial view of the whole system, to guide its decisions. The component consists of knowledge and processes. Knowledge is a set of fields that represent the internal state of a component. A process is defined by a function and a set of mappings for input and output. The input mappings of the process are fields of knowledge that are passed to the function as arguments, and the output mapping specifies the destination field of the return of the function. A component can be either a coordinator or a member of an ensemble. Each device in a CPS can deploy one or more components.

An *ensemble* is a group of components that consists of a single coordinator and multiple members. An ensemble mediates communication between the coordinator and members and is defined by the roles of the accepted coordinator and members, a membership condition, and knowledge exchange mappings. A *role* is a selection of knowledge fields required by the ensemble. The *membership condition* is a predicate on knowledge fields that dynamically determines which components should be members of the ensemble. The *exchange mappings* are rules that determine the knowledge exchanges that should occur between the member components. A knowledge mapping is either coordinator-to-members or members-to-coordinator.

2.1.3 Task Allocation

To coordinate a robot mission, a Multi-Robot Task Allocation (MRTA) problem is introduced. An MRTA problem consists of receiving a set of tasks and assigning these tasks to a fleet of robots [33] with their interrelated task utilities and constraints [34]. There is extensive literature on MRTA [35, 36].

Gerkey and Matari proposed a widely accepted taxonomy for MRTA problems [33] based on the three characteristics, as follows: (i) Single-Task Robots (ST) vs. Multi-Task robots (MT). ST robots can perform only one task at a time, while MT robots can simultaneously work on multiple tasks. (ii) Single-Robot Tasks (SR) vs. Multi-Robot Tasks (MR): SR require exactly one robot to complete, while MR require multiple robots. (iii) Instantaneous Assignment (IA) vs. Time-Extended Assignments (TA): In IA, tasks are allocated as they arrive, while in TA, tasks are scheduled over a planning horizon. Particularly in MissionControl, we apply single-task robots, multi-robot tasks, and instantaneous assignments, further presented in Section 3.5.

2.1.4 Planning and Control

In MAS, planning and control determine the sequence of actions, or policy, that agents must perform to complete their assigned task, after complex tasks have been decomposed into sub-tasks and allocated to cooperating groups of agents[9].

Following the foundation of MAS, planning and control in MissionControl is encapsulated into the task execution stage, providing an interface with the required skill implementation in the underlying robotic infrastructure. This is further presented in Section 3.6. In the next subsection, we provide further background on BTs and ROS as key underlying elements of robot control platforms.

2.2 Behavior Trees

A Behavior Tree [37, 38], which originated in the game industry, offers a structured approach to designing complex agent behavior. Initially used for Non-Playable Character (NPC), it replaced finite-state machines (FSMs) and hierarchical finite-state machines (HFSMs) due to its modularity and reduced error proneness. Encapsulating sub-behaviors within branches in BTs fosters a more organized and maintainable design, facilitating scalability for more complex behaviors [38]. BTs provide explicit support for task hierarchies. This means that a large, complex task can be naturally decomposed into a number of subtasks, which may in turn contain their own subtasks, and so on. For example, in a game, the high-level task of "Fighting" can be broken down into subtasks like fighting with a bow or a sword. Fighting with a bow might further involve subtasks such as grasping an arrow, placing it on the string, pulling the string, and releasing it. The hierarchical tree structure organizes the state transition logic, with the fundamental actions serving as leaves. This inherent structure facilitates the management and analysis of complex tasks by humans and algorithms [38, 39].

BTs primarily focus on creating reactive agents. These agents respond to sensory input in real-time, adapting their actions without necessarily formulating long-term plans. In essence, BTs support complex agent behavior by providing a structure where the long-term goals defined by the tree's architecture and reactive acting (immediate response to environment changes via ticks and task switching logic) are seamlessly integrated within a modular and highly reusable system. This reactive nature makes them well-suited for dynamic environments. The parallels between creating convincing NPCs and developing autonomous robots are significant. Both require agents to perceive their surroundings and react appropriately. Consequently, BTs have found widespread adoption in robotics research and industry [39]. BTs are structured as trees, where each node represents either an action to be performed or a condition to be checked. This hierarchical structure

dictates the flow of execution. The tree begins at the root node and branches out into various node types. Composite nodes, also known as control flow nodes, manage how their child nodes are executed. At the tree's leaves, we find Action nodes, representing tasks the robot can perform (like moving, grasping, or sensing), and Condition nodes, which evaluate the environment or robot state (for example, "is object in range?" or "battery low?"), returning either success or failure. The BT operates through repeated "ticks," starting at the root and traversing the tree. During each tick, nodes are executed based on their type and the results of their children. Each node, during its execution, can have one of three statuses: Running (executing now), Success (completed successfully), or Failure (failed to complete).

2.2.1 Action Nodes

These are the leaf nodes of the tree and are divided into two types:

Action : It is represented by a rectangle. It can execute various operations, returning Running until it reaches a final state, Success or Failure.

Condition : This type of node is represented by an ellipse. It checks a condition and returns Failure or Success.

2.2.2 Control Nodes

These are internal nodes responsible for how the tree is traversed according to the statuses of their child nodes. All types of internal nodes can have any number of children, except the Decorator, which can have only one. Child nodes can be control nodes or action nodes. In the classical BT formulation, there are four categories of control flow nodes:

Sequence : It is represented by a square with a rightward arrow. This type of node executes the child nodes in order, from left to right, and returns Success when all children return Success, otherwise, it returns the child's status, Running or Failure, and stops execution. For example, a robot's "Patrol" behavior might be a Sequence: first, "Navigate to Waypoint 1," then "Wait at Waypoint 1," then "Navigate to Waypoint 2," and so on. If the robot cannot navigate to a waypoint, the entire patrol sequence fails.

Fallback : It is represented by a question mark. The Selector (or Fallback) node, on the other hand, also executes children from left to right, but stops as soon as one

succeeds, returning success immediately. The fallback returns Failure if all child nodes return Failure; otherwise, it returns the child's status, Running or Success, and stops execution. A robot trying to "Pick Up Object" might use a Selector: first, "Try Grasp with Arm 1," then "Try Grasp with Arm 2" if the first fails. If either arm succeeds, the pick-up action is considered successful. Only if both fail does the selector itself fail.

Parallel : It is represented by a square with two parallel rightward arrows. This node executes all its child nodes and returns Success if a number M of children return Success, returns Failure if more than N - M children return Failure, and returns Running in any other case. A robot "Cleaning a Room" might use a Parallel node to simultaneously "Vacuum" and "Dust," continuing as long as both tasks are considered running or successful.

Decorator : It has only one child node, and its behavior can be determined by the Behavior Tree developer to add semantics or change the return status of a node. A classic example would be the Inverter, which inverts the status of the child node.

BTs integrate seamlessly with ROS2. This integration allows BT nodes to interact with ROS topics, services, and actions, enabling sophisticated robot control. For instance, a condition node within a BT might subscribe to a sensor topic like "is_object_detected," while an action node could publish commands to Nav2's NavigateToPose action. ROS actions are particularly useful for managing long-running tasks within a BT, providing mechanisms for preemption and feedback. Services, on the other hand, facilitate synchronous communication, enabling tasks such as querying the robot's current pose. This ROS integration empowers BTs to leverage the extensive ROS ecosystem, combining high-level logic with low-level control and perception. Several libraries facilitate this integration. pytree and behavior_tree_cpp are prominent examples, providing the core functionality for defining, executing, and visualizing BTs.

2.3 ROS Middleware

ROS [40] is a widely adopted middleware for the development of robotic software. As a middleware, it provides tooling for component-based development of software for robots. The ROS middleware is complemented by a robust ecosystem of reusable components. The name ROS stands for Robot Operating System, although this name can be misleading as ROS is not a real operating system, but it is rather a middleware that runs on top of a real operating system, popularly Linux.

While ROS1 established a strong foundation, ROS2 has emerged as the current standard, offering significant improvements in areas like real-time capabilities, multi-robot support, and communication reliability. In both ROS1 and ROS2, robot control software is developed as components called nodes, which communicate using a publish-subscribe pattern. The nodes exchange messages on topics, enabling a modular and flexible architecture that facilitates easy integration and reconfiguration of functionalities. Each node can be independently developed and tested, fostering collaborative development and simplifying the overall complexity of the system. This publish-subscribe mechanism creates a loosely coupled architecture, where nodes are not directly dependent on each other but rather on the messages exchanged via topics. This decoupling enhances robustness, as the failure of one node typically does not impact the operation of others, provided that they are not critically reliant on the data published by the failing node. ROS2 builds upon these core concepts, introducing features such as DDS (Data Distribution Service) for interprocess communication, enhancing real-time performance, and improving support for distributed systems, making it suitable for more complex and demanding robotic applications.

ROS2 provides a rich set of tools and libraries that facilitate various aspects of robot software development, including hardware abstraction, message passing, parameter management, and visualization. For instance, the hardware abstraction layer (HAL) allows developers to interact with different robot platforms through a unified interface, regardless of the underlying hardware specifics. This significantly reduces the effort required to port the software across different robots. The message passing system, based on predefined message types, ensures interoperability between nodes and promotes code reusability. The parameter server acts as a central repository for configuration parameters, allowing for dynamic reconfiguration of nodes without requiring recompilation. Finally, tools like RViz provide powerful visualization capabilities, enabling developers to monitor and debug the robot's state and behavior in real-time.

A significant strength of the ROS ecosystem lies in its vast collection of reusable packages. These packages, contributed by the open-source community, provide pre-built functionalities for common robotics tasks, such as navigation, perception, manipulation, and control. Using these existing packages can significantly accelerate the development time and reduce the need to reinvent the wheel. For example, packages like Nav2 offer sophisticated navigation capabilities, while packages like MoveIt! provide control for robotic arms to manipulate objects. By integrating these readily available components, developers can focus on the specific aspects of their application, rather than spending time implementing fundamental functionalities from scratch. This reuse not only saves time but also promotes code quality and consistency, as these packages are often well-tested

and actively maintained by the community.

Beyond the publish-subscribe mechanism, ROS2 offers other communication paradigms, notably services and actions. Services provide a request-response mechanism for synchronous communication between nodes. One node can act as a service server, listening for requests from other nodes (clients). Upon receiving a request, the server processes it and sends a response to the client. This is particularly useful for tasks that require immediate feedback, such as asking for the current joint state of a robot arm. Actions, on the other hand, are designed for long-term, goal-oriented tasks. They provide a mechanism for asynchronous communication with feedback and cancellation capabilities. An action client can send a goal to an action server, which then executes the task. During execution, the server can provide feedback to the client, providing updates on progress. The client can also request the server to cancel the goal before it is complete. Actions are well-suited for tasks like navigation, where the robot needs to reach a specific goal location, and the user might want to monitor the progress or cancel the navigation if needed.

For example, to move the robot using Nav2, we can send a navigation goal to Nav2 in ROS2. First, the script initializes the ROS2 node and creates the action client, waiting for the Nav2 action server to become available. Then a PoseStamped message is constructed, containing the desired goal pose. The pose.position and pose.orientation fields are set to the coordinates and orientation of the destination. The script then sends the goal to the Nav2 action server named NavigateToPose. Wait for the goal to be accepted and then for the navigation to complete. Feedback can be monitored during navigation, and upon completion, the script checks if the goal was successfully reached or if it failed, printing the reason for failure if applicable. This process involves asynchronous communication with the Nav2 action server, allowing for monitoring and potential cancellation of the navigation task.

2.4 Architecture Quality

An architecture can accommodate a problem to a certain extent if it possesses key attributes that are essential to the system for which it was designed [10]. Some quality attributes are associated with the way the system is developed, such as modifiability, integrability, and testability. Other quality attributes are associated with the properties that the system exhibits in the end environment, such as performance and availability.

Integrability expresses the ease of integrating independently developed components into a system, while modifiability refers to the ease with which a software system can be changed or adapted to new requirements or circumstances. Testability, in turn, is the

degree to which a software system facilitates the establishment of test criteria and the execution of tests to determine whether those criteria have been met. As most of the cost of software development is spent on maintenancei.e., in changes after initial deployment, modifiability and integrability are key attributes that define long-term software success. Moreover, these attributes are fundamental to enabling architectural reuse: to fit another environment, the architecture must be modified to suit the new application and integrated with existing systems in that environment.

Architectural tactics are design decisions that help achieve specific quality attribute responses [10]. Unlike general good practices that are universally beneficial, tactics often involve trade-offs, and designers should be conscious of their application. For instance, designing a system for modifiability can be challenging, as many tactics that promote it may negatively affect other attributes, such as performance. One such tactic is the *use of an intermediary* [10], where an extra layer or component mediates communication between others. While this tactic promotes modifiability by hiding component complexity and providing a simplified interface, it can decrease performance. Applying modifiability tactics indiscriminately can thus be cost-ineffective, introducing unnecessary overhead where it is not needed. Because it is impossible to foresee all future changes, selecting architectural tactics requires a careful evaluation of likely change classes. A sound strategy is to identify probable change requests (e.g., system extension for a new mission) and design a clear path for that change using appropriate tactics.

Quality attributes define measurable system properties that guide architectural decisions. Each attribute is typically described through a scenario specifying the stimulus, environment, artifact, response, and response measure. Examples include:

- **Performance**: maintaining acceptable response times and throughput under load.
- **Availability**: ensuring the system remains operational and accessible.
- **Modifiability**: facilitating easy and cost-effective evolution of the system.
- **Integrability**: enabling easy combination of independently developed components and systems.
- **Testability**: supporting efficient verification and validation of system behavior.
- **Security**: protecting against unauthorized access or malicious use.
- **Reliability**: maintaining correct service despite faults or failures.

Tactics are concrete design strategies that operationalize these quality attributes. They bridge the gap between abstract quality goals and architectural structures or patterns. Examples include:

- **Performance tactics**: caching, concurrency, load balancing, and resource management.
- **Availability tactics**: redundancy, replication, failover mechanisms, and health monitoring.
- **Modifiability tactics**: encapsulation, separation of concerns, parameterization, and interface abstraction.
- **Integrability tactics**: use of standard interfaces and protocols, service discovery mechanisms, and middleware support for data translation.
- **Testability tactics**: dependency injection, logging and monitoring hooks, and the separation of testable units.
- **Security tactics**: authentication, authorization, encryption, and input validation.
- **Reliability tactics**: checkpointing, rollback, and exception handling.

These tactics are often instantiated through architectural patternssuch as layered, event-driven, or microservice architecturesthat provide the structural foundation for applying them. Figure 2.4 summarizes the relationship between quality attributes, tactics, and architectural patterns.

2.5 Simulation

Robotics simulators play an important role in robotics research as tools for testing the efficiency, safety, and robustness of new algorithms [41]. They provide a virtual proving ground that can be instrumental in understanding how future robots should be designed and controlled for safe operation and improved performance [42]. Physics simulators enable the vast majority of robotics research because they provide an environment that is cheap and allows users access to a variety of desired robots without the potential to degrade or break the physical platform, which is important since physical robots are often expensive, fragile, and scarce [43]. Gazebo [44], Simbad [45], CoppeliaSim [46], MORSE [47], and Dragonfly [48] are some examples or simulators tailored for exercising robotic scenarios.

In robotics, simulation involves creating computer models to replicate the behavior of a robot and its surroundings. This virtual testing environment allows researchers and developers to refine robotic systems before deploying them in real-world scenarios. As part of our fourth contribution (**C4**), we present in this section the background topics related to that contribution.

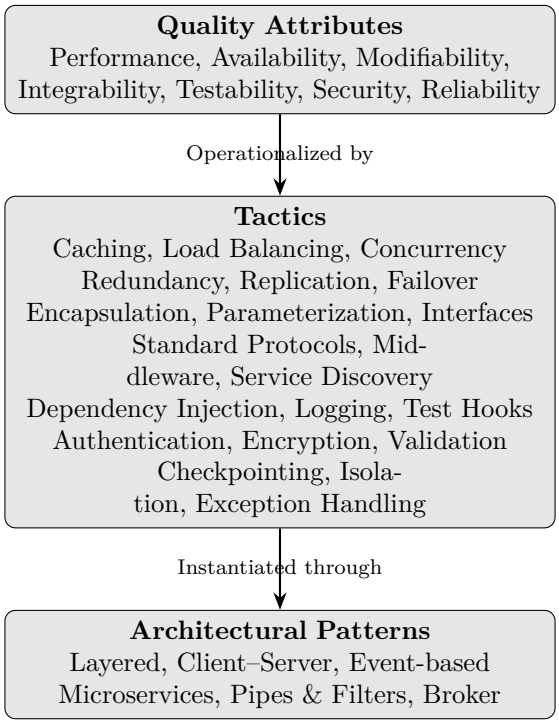


Figure 2.4: Relationship between quality attributes, tactics, and architectural patterns, according to [10].

2.5.1 Simulation Techniques

A well-established simulation technique is discrete time with a fixed increment interval [49]. In this model, the state of a system at time t_{i+1} is a function of the state of the system at time t_i. Each variable that composes the system's state is a function of variables and states up to the previous moment. The simulation time increment between t_i and t_{i+1} is always the same and predefined.

If the time t_{calc} required to compute the system state t_{i+1} from state t_i is less than the increment time t_{incr}, then the simulation will be computed faster than real-time; similarly, if $t_{calc} > t_{incr}$, then the simulation is computed slower than real-time. These situations are known as off-line simulation, because there is no synchronization between simulation time and real time. This is an acceptable situation for this project, where the goal is to obtain the desired simulation in the shortest possible time.

This simulation technique is suitable for simulating systems that change constantly, such as the temperature of a room over time, or a signal received by a sensor that operates at a known frequency. However, the simulation "clock" is synchronized, and all state functions are processed at each time increment, which can lead to unnecessary calculations. For example, in a simulation involving a function that changes the temperature of a room every 200 ms, and a sensor that records the room temperature with readings every 100 ms, the function that changes the temperature must be executed at all increments, which

must be at most 100 ms to support the sensor reading. In this scenario, half of the calls to the temperature change function do not affect the system state, but still have to be processed.

Another simulation technique is *discrete event simulation (DES)*. An event e has a time t and a function f that change the simulation state and potentially create other events, which will be added to an event queue. Each state s_{i+1} is the result of processing the event at the beginning of the queue over the state s_i. The event queue is a priority queue ordered by the time t of each event, where the time of each newly generated event can be equal to or greater than the time of the event that generated it. The simulation time corresponds to the time t of the current event that is being processed, and since the events are ordered by time, it will only be incremented when all events at that time have been processed.

Unlike fixed-increment time simulation, where the same functions are executed at known time intervals, in DES, different functions change the simulation state, and the simulation time at state s_{i+1} depends not only on state s but also on the event being processed. This new time may not increase uniformly throughout the simulation. This technique is suitable for simulating systems that change infrequently over time, such as the inventory of a warehouse or the operation of service robots within a warehouse.

Fixed-time increment simulation can be implemented using the discrete event technique, provided that the events are created with times that have a constant interval. Another interesting feature that can be achieved with discrete events is to separate the function into subsystems that are executed independently and asynchronously. Returning to the example of the room that changes temperature and has the sensor, each sensor reading event can create the next reading event for time t + 100ms; similarly, each temperature change event creates a new change event for time t + 200 ms. In this way, the problem of state change functions having to be executed ahead of time is avoided.

2.5.2 Entity-Component-System

The development of real-time applications and games demands a flexible and efficient entity management system. Entity-Component-System (ECS)[50] is a software design pattern widely used in games, typically in real-time interactive systems, such as games, that promotes a clean separation of identity (entities), data properties (components), and computational behaviors (systems).

In this pattern, simulation objects are transformed into entities. An entity is related to a specific concept (e.g., a wall, a robot, a person), with associated data. Entities can be created and destroyed during simulation execution and are used only as identifiers. Each entity identifies a collection of components.

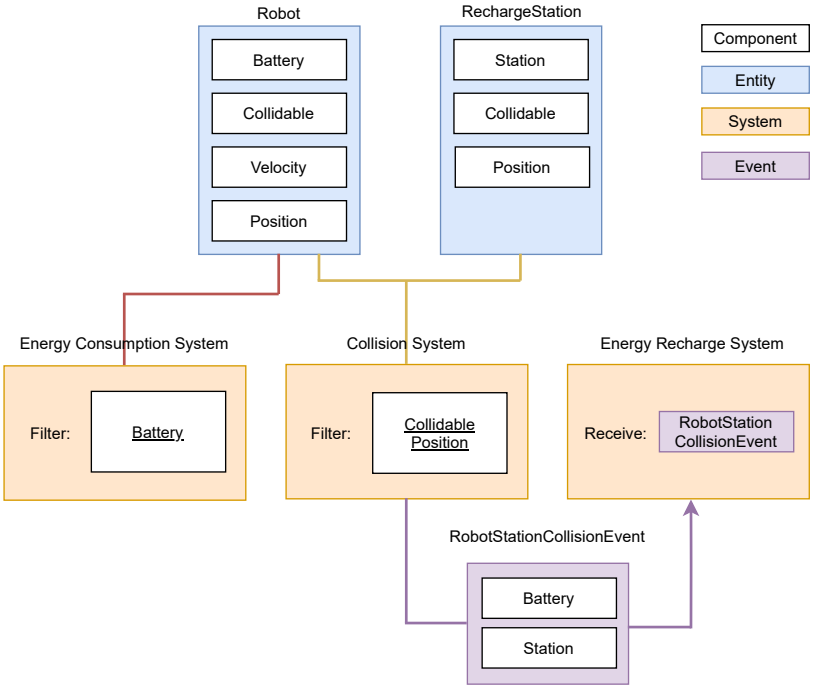


Figure 2.5: Example of a system built on the ECS pattern

A *component*, stores data, but typically does not implement any logic. Components can also be added or removed during simulation execution, altering the state of the entity to which they belong. Components represent some attribute, state, or capability of the entity (e.g., Position, Velocity, Sensors, etc.). In this way, the ECS pattern manages to separate data from the application's logic.

The simulation logic resides in the *systems*, which modifies the data of the components according to their purpose. In other words, instead of focusing directly on an entity, systems are interested in groups of components that have the same aspects of the system and that will be processed to perform a specific task. Each system acts independently of other systems on a set of components that interest it. By changing component values, the systems effectively change the state of the simulation. The simulation state is the set of states of all components of all entities present in the simulation. It is altered only by systems, each altering a small part of this global state.

Figure 2.5 shows the operation of a system that uses ECS. In this example, there are two entities: Robot and RechargeStation. The Robot entity has a Battery, Collidable, Velocity, and Position component, and the RechargeStation entity has the Station, Collidable, and Position components.

The example presents two systems. `Energy Consumption System` is responsible for consuming (decrementing) the total energy of all entities that have the `Battery` component, in which case only the `Robot` entity has this component. `Collision System` is

responsible for checking and notifying collisions between entities in the simulation. If a collision occurs between a `Robot` entity and a `RechargeStation`, it means that this Robot is in contact with the recharge station and should start recharging its battery. The Energy Recharge System then controls the recharging process by incrementally increasing the level of the battery.

By separating responsibilities into systems, ECS favors modularity. Each system (or set of systems) and its associated set of components can be added or removed from the simulator independently as needed. For example, a communication system between different robots can be implemented as a component that stores a message queue and can be added to each robot, associated with two systems: one system that delivers the messages from one robot to another, and another system that processes the messages of each robot. Note that if the processing is not suitable for a simulation, simply replace that system with another that is suitable. Additionally, if a simulation does not make use of this messaging system, for example, if battery recharging is not of interest, simply remove it completely from the simulator, making the simulation lighter.

Another advantage of using the ECS pattern is the flexibility to add or remove entities' capabilities during simulation execution. Since each entity is simply a collection of components, it is possible to associate certain capabilities of robots (e.g., sensors, actuators) with the presence or absence of certain components in that entity. For example, given the existence of a camera component and an associated system that simulates image capture, any entity that has this component will have the ability to collect images via the camera component. Removing a component from the entity removes that attribute or capability from it.

Despite these advantages, as pointed out by Wiebush, the use of ECS can bring about complications of compatibility between independently developed systems, such as the use of incompatible components and the difficulty in knowing which system is responsible for a particular functionality and how to use it [51]. Details of how these problems were felt during the development of the project and measures taken to mitigate them are discussed in Chapter 5.

To implement the ECS pattern in our work, we used the Esper library, a lightweight ECS library in Python with focus on performance [52]. It creates a World class that maintains a list of entities and all components for each entity. A component can be any structure in Python; in the case of the project, classes were used. It is also possible to add systems to the World class, which are implemented as functions, conventionally called processes. Some of the project's systems are added to the World. Esper also provides functions that facilitate obtaining components or sets of components from saved entities, creating and removing entities and components, and managing the execution of systems.

2.6 Behavior-Driven Development

Behavior-Driven Development (BDD) is a software development technique that uses natural language to describe how code should behave. BDD is an agile development technique that originated from Test-Driven Development (TDD) and Domain-Driven Design (DDD). BDD is realized by a set of software engineering practices aimed at improving code quality and delivering correct software [53, 54]. BDD focuses on product behavior and encourages collaboration among teams.

The BDD process begins in the requirements gathering phase. In this phase, software requirements are discussed among stakeholders from different teams, and a responsible person, usually from the testing area, will describe these requirements in files that can be understood by everyone involved in the process. These files will guide product development and serve as a form of documentation to prevent information loss throughout the process. In the end, it will be validated whether all behaviors described in the scenarios of each file are working correctly. This validation can be done automatically through test automation. In this way, each scenario of a feature is a test case.

Test cases, composed of inputs, sets of steps, and outputs, are described using a format suggested by BDD, based on user stories. These test cases will describe, in various scenarios, the behavior of the business rules of each feature.

The process of developing tests is similar to TDD. First, tests are created that will fail, then the code is written until these tests pass, and finally, the code is refactored. This cycle repeats. The main difference is that tests in BDD are behavioral tests.

One problem that BDD tries to solve is communication between teams and other stakeholders, and the loss of information about feature requirements throughout the various processes. For example, using BDD, when a business person requests a new feature, they will meet with the development and testing teams and explain the requirements of this new feature. In this meeting, everyone involved will translate these specifications, using the Gherkin language, into scenarios and rules that can be understood by everyone involved. The various scenarios and rules described will guide the development and serve as test cases.

According to BDD, it will help the team focus on identifying, understanding, and implementing the features that really matter to customers.

2.6.0.1 Scenario Specification Language: Gherkin

BDD proposes using a language that is understandable by all stakeholders to describe software requirements, as described above. One such language is Gherkin [55].

The behavior of software features will be described in several files that end with the suffix `.feature`. These files will be used to guide developers as part of the documentation and will also be used to test the various behaviors of the product. Each `.feature` file will describe a feature, and this feature can be divided into more than one file. These files will be composed of one or more scenarios.

Scenarios are made up of several rules, and each rule can be mapped to a method that will verify if it is executed correctly.

A *feature* can be specified as follows:

Feature: short description of the feature

Background: group rules common to all scenarios

Scenario: describe a specific use case scenario

Given a precondition

And another pre-condition

When an action occurs

And another action occurs.

Then a verifiable condition that is expected to occur.

The listing 2.1 shows an example of the `.feature` file.

```
Feature: Robot Displacement

Background:
        Given a simulation

Scenario: The robot stands in the same place
        Given a robot in position 0.0, 0.0
        When the robot does not move
        Then a robot is in 0.0, 0.0

Scenario: The robot moves to a position
        Given a robot in <initial_position>
        When the robot moves to <move_to_position>
        Then a robot is in <final_position>

Examples:
        | initial_position | move_to_position |
          ↪ final_position |
```

```
| 0.0, 0.0          | 1.5, 1.5          | 1.5, 1.5
  ↪         |
| 5.0, 5.0          | 12.5, 10.0        | 12.5, 10.0
  ↪       |
```

Code Listing 2.1: Example of a *.feature* file.

Chapter 3

The MissionControl Architecture

An architecture for mission coordination of heterogeneous robots, Full article published in Journal of Systems and Software, Volume 191, 2022.

In this chapter, we present *MissionControl* architecture. This architecture aims to provide abstractions for engineering heterogeneous MRS focusing on the architectural perspective of coordination of missions of heterogeneous robots.

To decouple between coordination and robot task execution, MissionControl is divided into two major components: *Mission Coordination* and *Task Execution*. Mission coordination is built on top of ensemble-based systems [14]. Ensembles provide abstractions to form dynamic groups of robots employing membership functions and knowledge exchange mappings between robots and the mission coordinator. The coordinator keeps an updated knowledge base about the properties of the available robots (e.g. capabilities, position, battery level, battery consumption rate), receives mission requests from users, and realizes the coalition formation and task allocation for received requests.

To allow the system to execute a variety of missions planned by a HTN planner [24], we created a component model in which a skill implementation is a BT [56], and a mission is executed by dynamically activating these skills. This approach allowed us to use an HTN planner for deliberation while using BTs for reactive behavior. By dividing the behavior of robots into independent componentisable skills, we create an opportunity for (i) reuse of skills across applications and different robots models that share some characteristics and (ii) for different skills to be independently developed, potentially by different teams.

To suitably form the robots coalition, the coordinator evaluates the set of tasks for a given role, selecting the robots that yield the best utility. This evaluation is executed following the mission at hand, the collected knowledge about the available robots, and

skills descriptors, i.e. components that provide estimates for specific tasks types. After forming the coalition, the selected robots receive their local mission plan, which is fulfilled by tasks. Then, the planned tasks are performed by activating an associated skill from a library of skills available to the robot.

Before further delving into the MissionControl architecture, we start by discussing the flow of processes and artifacts for mission coordination (Sect.3.2). Then, in Section 3.3, we present an overview of the MissionControl architecture. In Section 3.4 we explain how the architecture is realized on top of the ensemble's component model. In Sections 3.2 and 3.6 we provide more details on the mission coordination and task execution processes of MissionControl, respectively. Whenever appropriate, we also provide the design decisions behind the MissionControl architecture and their corresponding rationale.

3.1 Lab Samples Logistics

In a hospital setting, the staff spends a significant amount of time transporting supplies and samples around the hospital. These activities can be automated, allowing personnel to focus on more specialized care activities. One particular logistic task involves the transport of laboratory samples from their collection points to the laboratory for analysis.

The lab sample scenario can be described as follows. A nurse is responsible for collecting the sample and can request delivery to the laboratory, identifying the room where collection should take place. Robots should transport samples from the patient rooms to the laboratory. The system must include a robot with a securely locked drawer, which must then navigate to the collection room, identify the nurse, approach her, open the drawer, await the deposit, close the drawer, and then navigate to the laboratory carrying the sample. In the laboratory, the sample can be collected by a robotic arm or laboratory personnel. The robotic arm picks up samples, scans the barcode in each sample, sorts and loads the samples into the entry module of the analysis machines. Figure 3.1 illustrates the map of the hospital in the scenario.

We assume that each robot can perform only one task at a time and that tasks can be requested spontaneously in an unknown timeline. In this way, the system allocates each task independently, taking into account knowledge about the state of the system at the moment the request is handled, that is, the allocation follows Single-Task, Multi-Robot, Instantaneous Assignment (ST-MR-IA) [34] model. When the system receives a task, it can have a varying number of robots available, in different positions, and with different battery levels. If the system assigns a task to a robot that does not have enough battery to complete the task, the robot can reach a low-battery critical failure state in which it cannot move autonomously and should be rescued by an operator.

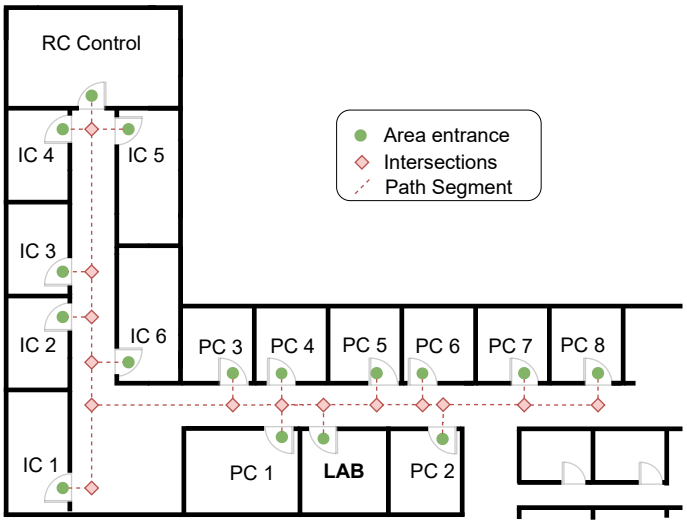


Figure 3.1: Lab Samples Scenario

3.2 Overall System Design

Mission coordination involves receiving requests from users, forming a coalition, and distributing plans. Figure 3.2 shows the Mission Coordination integrated with Task Execution.

The lanes represent the user, the coordinator, and the robots. The coordination responsibility is divided into three phases: (i) Mission Initialization, with the communications and processes that occur when a request from the user is received, (ii) Coalition Formation, which is executed when a role in the mission needs to be assigned, and (iii) Supervision / Execution when robots perform the mission plan.

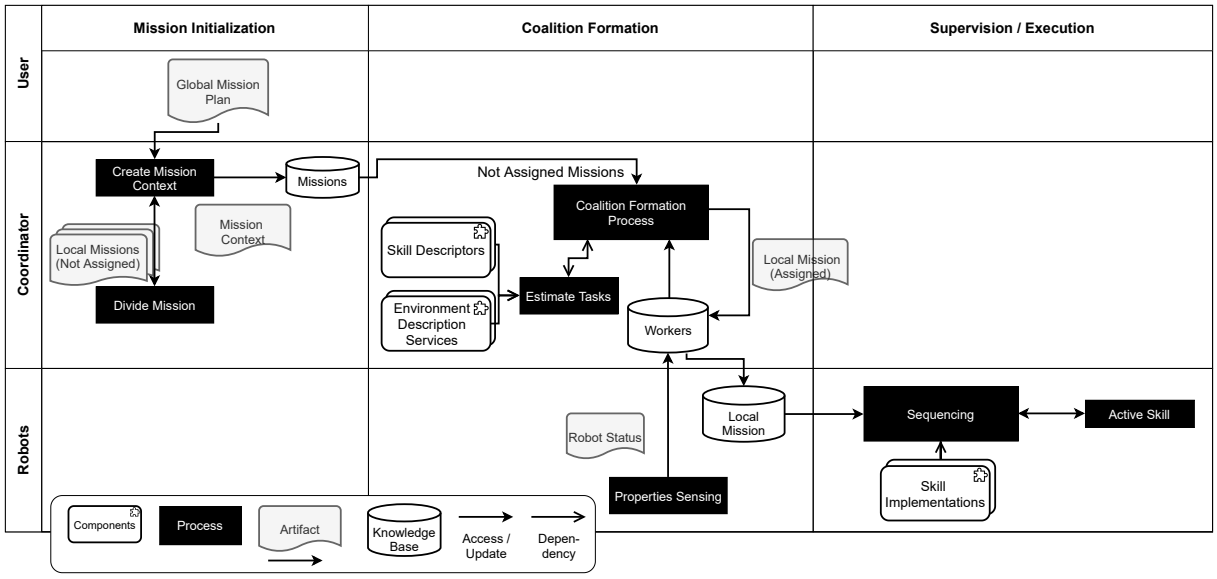


Figure 3.2: Mission Coordination and Task Execution

Mission initialization is triggered when a user sends a request with a Global Mission plan to the coordinator. The Global Mission Plan is the output of the mission planning process. Mission planning, also called task decomposition process, is the activity that generates a plan suitable for the known state of the world. This plan specifies (i) what needs to be accomplished in terms of what tasks should be performed, and (ii) what dependencies exist between these tasks. In MissionControl the output of mission planning is the Global Mission Plan, it is the input for mission coordination, and is in the format of an iHTN (Sect. 2.1.1.1).

In the *Create Mission Context* activity, the coordinator instantiates a mission context. Then, the plan is divided into local missions (using the divide algorithm of Lesire et al. [25]). Finally, the coordinator adds the mission context to the set of missions in its knowledge base. The local missions within the mission context do not initially have an assigned robot, and, therefore, the mission context has initially pending assignments.

The *Coalition Formation Process* is activated for missions with local missions not already assigned. The Coalition Formation Process tries to organize a coalition by searching for an assignment for each local plan without an assigned robot. When forming the coalition, the coordinator queries for available workers in the 'Workers' database. The Workers database is kept up-to-date with periodic updates from robots ('Properties Sensing' Process)

If a coalition is formed, the system distributes local missions for each robot (Figure 3.2, bottom lane).

Once a robot is assigned a plan, it executes the plan by performing the *Sequencing* process. In this process, the robot executes the mission by iterating over the tasks of the plan and activating the corresponding skill from its library of Skill Implementations. Active skill has low-level logic to control the robot sensors and actuators while performing a task. More details on task execution will follow (Sect. 3.6).

3.3 MissionControl Core Quality Attributes

From the architecture design perspective of MissionControl , we should ensure that it addresses important architecture attributes for which the system was designed, following the principles of Bass et al. for the design of software architecture [10]. In particular, *modifiability* (i.e, ability of the architecture to be modified to fit a new application) and *integrability* (i.e ability of the architecture to integrate with existing systems) are key attributes at stake in MissionControl . Modifiability and integrability are key attributes to allow a complete architecture to be reused between applications. As to fit in another environment, the architecture needs to be modified to suit the new application and inte-

grate with existing systems in that environment. However, designing a system for such attributes can be tricky, as one may not foresee clearly the required changes. A design strategy for those attributes is to identify a class of change requests that are likely to occur (i.e., the system should be extended for a new kind of mission) and design into the system a clear path for that change by using appropriate architectural tactics [10]. Modifiability tactics aim at controlling the complexity of making component changes by increasing their cohesion, reducing their coupling, and deferring their binding. Instead, integrability tactics aim to reduce the impacts on components whenever they are added, integrated into sets, or reintegrated once changed.

Key Architectural Quality Attributes In MissionControl, integrability and modifiability are prioritized so that new or existing missions and their constituent parts may be seamlessly composed and reused across mission applications. In the rest of this section, we enumerate the Design Decisions (DD) taken to support these quality attributes in these highlighted boxes.

Figure 3.3 offers an overview of MissionControl. The architecture provides a runtime environment and a component model.

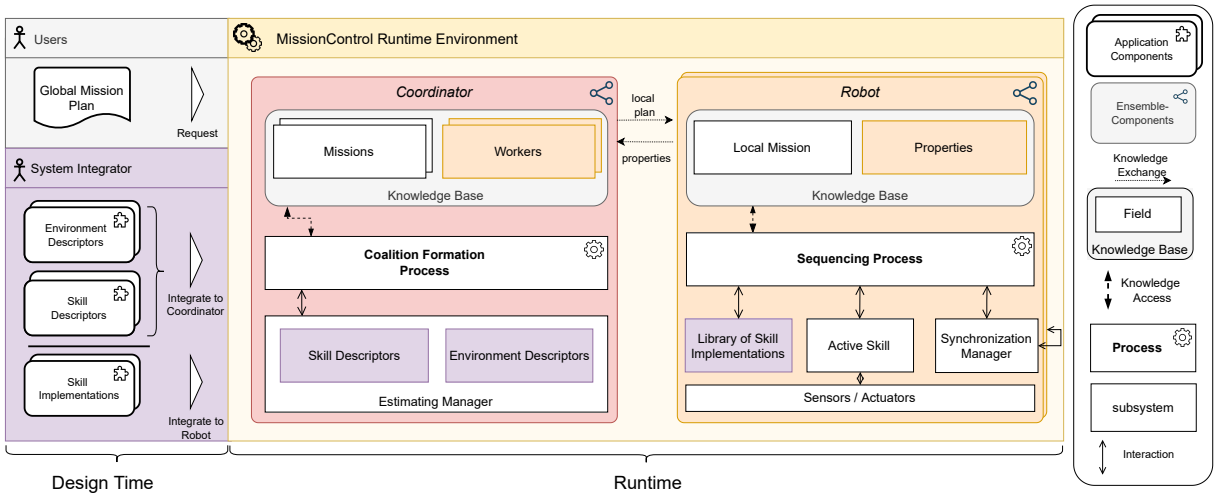


Figure 3.3: MissionControl Components

The runtime environment encompasses the processes for coordinating the mission, while the component model provides extension points, allowing MissionControl to be tailor-made to the application, environment and robots for which the system will be deployed. The application components are *Skill Descriptors*, *Skill Implementations*, and *Environment Description*. They are integrated into the system by the *System Integrator*. The *Environment Descriptors* provides the data required by skill descriptors/implementations that are specific to the end application environment (e.g., a map describing the

routes that the robots can take). Through the Environment Descriptors interface, MissionControl abstracts the consultation of information necessary for the coordination process, reducing the difficulty of integration in different environments. The *Skill Descriptors* provides estimates of the cost/utility of performing a task with a given robot, while the *Skill Implementations* are components that provide the control required to perform tasks by a robot. The contribution to modifiability of the Skill Implementations and Skill Descriptor is twofold. On the one hand, they facilitate the extension of the system to new skills. However, they isolate modifications related to specific behaviors from the components that provide that behavior.

MissionControl distributes responsibilities between the coordinator and the robot. We consider that in a service robot system, it is convenient to delegate functions to a central component, deployed in a server, connected to power, and with a more stable wired network. Such a central component tends to have a longer up-time, greater computational power, and is easier to upgrade (compared to upgrading multiple robots). The central node aggregates the information from the environment and chooses the best robot to execute parts of the plan. In its distributed control, MissionControl gives autonomy to robots in the execution of the mission. In this way, the robot receives the plan at a high level of abstraction (in terms of tasks), is responsible for making the local control of the execution of its plan, and also responds to the coordinator at a high level of abstraction (in terms of plan progress). The advantages of giving autonomy to robots are twofold: (i) heterogeneity is simpler, as the coordinator treats the different robots with a uniform interface, and (ii) less communication traffic compared to a fully centralized approach, as the robot only reports the mission status from time to time and does not need to coordinate frequent decision making.

(DD.1) Centralized vs distributed control In MissionControl, we chose a hybrid between centralized and distributed control, distributing responsibilities between a central node (Coordinator) and distributed components (Robots).

Runtime activities are encapsulated in MissionControl as two high-level entities, namely *coordinator* and *robot*. The *coordinator* receives missions from users in the form of requests, stores them in its *Knowledge Base*, and assigns local mission plans to *robots*. The main content of the request is the Global Mission Plan, which is the decomposition of the mission into elementary tasks and is formalized as an iHTN (see Section 2.1.1.1). In the core of this assignment, there is the *coalition formation process*, which is responsible for selecting robots to participate in each mission. This process is supported by *Estimating Manager*, a subsystem used to estimate the cost and utility of assigning a specific

local mission to a given robot. The Estimating Manager is further extended by two application components, i.e. *skill descriptors* and *environment descriptors*. The binding between *coordinator* and *robots* is executed dynamically, following the Ensemble-Based Component System approach. Deferring binding to runtime is a well-known tactic for modifiability [10]. Here, it is employed to allow for changes to the set of available robots transparently.

A robot performs its local mission by performing the tasks within its local plan and synchronizing with other robots. The local mission is a tree structure where the leaves are either elementary tasks or synchronizations (as previously explained in Section 2.1.1.1). The sequencing process first chooses the skill implementation from *library of skills* for each task and then activates it. Finally, *active skill* controls the robot hardware and low-level controlling processes while returning the progress of the execution process that writes it into the robot's knowledge base. We further explain the execution process of the task of MissionControl in Section 3.6. Skills favor modifiability by allowing the extension of a given deployment of MissionControl by adding the associated skill implementation and descriptions, without requiring further modification of other parts of the system.

The coordinator and robot synchronize through knowledge exchanges. Figure 3.3 shows two knowledge exchanges in MissionControl: (i) the robots update the coordinator with their 'properties' (e.g., skills, position, battery level, battery discharge rate, etc.); (ii) the coordinator updates the robots with a local mission.

The underlying principles of our architecture are mission, task, and skill. Missions express high-level abstract goals of what needs to be achieved (e.g., fetch a lab sample for a nurse in room 3) and decompose them into elementary executable tasks. The *missions* in MissionControl is a tree that refines the high-level *abstract tasks* into concrete *elementary Tasks*. Skills represent the abilities of a robot to realize a task and abstract away the details about how a robot can perform the task. Elementary tasks in a mission are realized by applying appropriate skills (e.g., the task 'navigate to room 3' is rendered achievable by applying the skill 'navigation').

Up to this point, we provided an overview of the main components and abstractions and how they interplay to realize multi-robot missions.

3.3.1 Reasoning

An architecture can be more or less fit to a problem if it exhibits quality attributes that are key for the system for which the architecture was designed. Programmability quality attributes, such as modifiability, integrability, are the ones that are associated with how the system is developed (in opposition to runtime quality attributes that are associated with the attributes that the system exhibits in the end environment) and

are concerned with the cost and risk associated with implementing changes. As most of the cost of software development is spent in maintenance, that is, in changes of the software after the initial deployment, modifiability and integrability are key attributes that will define the success of software in the long run. Moreover, the quality attributes of modifiability and integrability are the key attributes that allow a complete architecture to be reused between applications. In order to fit in another environment, the architecture needs to be modified to suit the new application and integrated with existing systems in that environment. Designing a system with a focus on modifiability can be tricky, as we cannot foresee perfectly which changes will be required, which renders it difficult (if even possible) to evaluate candidate architectural tactics. Many tactics that promote modifiability (e.g., use of intermediary) imply trade-offs with other quality attributes (e.g., introducing an intermediary can lead to a decrease the performance). Therefore, applying architectural tactics that promote modifiability indiscriminately can be cost-ineffective, as we can introduce costs by applying tactics in parts of the system in which these tactics will not be required.

A strategy for design regarding modifiability is to identify a class of change requests that are likely to occur (i.e., the system should be extended for a new kind of mission) and design into the system a clear path for that change by using appropriate architectural tactics. To create an architecture that can be modified and reusable between different missions, we analyze a set of robot missions in the service robot domain described in the literature. From the descriptions, we identified the differences and commonalities between the missions. Then we derived a generic mission description, that is, a description that abstracts away the specifics of different missions, and identified key points that are likely to change.

In MissionControl , we opted for an ensemble-based system architecture instead of a layered architecture approach, as, for example, in [57]. While in a layered model, communication can occur only between adjacent layers, an ensemble-based approach allows communication between components that form an ensemble. As such, the coordinator and the robots form ensembles, i.e., dynamic groups to achieve missions, in which the coordinator component is the coordinator of the ensemble, and the robots are members of the ensemble. The ensemble defines knowledge exchange mappings, i.e., communications that occur between components.

3.4 MissionControl Ensembles

As anticipated before, our model for coordination of the mission is based on the concept of ensemble-based systems [14]. Ensembles provide abstractions for simplifying the

(i) dynamic binding between the coordinator and available robots and (ii) implementation of the coordination process, separating the decision-making processes and the communication between the components. In MissionControl , the Mission Coordination ensemble has a coordinator and a dynamic set of robots as components.

(DD.2) Architectural Pattern MissionControl is an ensemble-based system architecture where the components are bound together at run-time by forming an ensemble for coordinating missions and have processes that operate on the local knowledge base.

Listing 3.1 presents an excerpt of the constructs for these components in the DEECo DSL [14] specification. The coordinator is sketched in lines 4-15 with its Coalition Formation Process (line 10). Following the ensemble paradigm, processes operate on the knowledge base of the ensemble component, i.e., they have fields in the knowledge base as input and output. The robot is outlined on the lines 20-40, it features the Worker role and the *sequencing process*. The *sequencing process* sequentially iterates tasks in the local mission plan, activating the related skill, and monitoring the progress until the task is completed (more details about the sequencing process and related subsystems are presented in Section 3.6).

The ensemble specification is presented in the Listing 3.1 (lines 42-56). The role of the ensemble specification is threefold: to define (i) the roles of the member and coordinator, (ii) the membership condition, and (iii) knowledge exchange mappings to occur between the coordinator and members. In our model, the coordinator of the ensemble is the Mission Coordinator, and robots are the members of the ensemble (i.e., *Robots* are the components that have the *Worker* role).

The membership condition (line 46) is evaluated against a pair coordinator-member, and if evaluated to *True*, then the members (i.e., robots) can join the ensemble. The robot members selected by the membership function will be candidates to perform the missions received by the coordinator. The ideal membership function can be subject to organizational policies, e.g., a hospital could separate robots working in a given section of the hospital to minimize the chance of spreading infectious diseases. For the scope of this work, we consider a simple policy: the coordinator is initialized with a set of skills of interest that are expected to appear in the requested missions. Any robot that has at least one skill in that set is to be a member of the Mission Coordination ensemble. In the ‘lab samples logistics‘, the coordinator is initialized with required_skills as ‘operate_drawer‘, and robots capable of operating a drawer are the ones that are the members of the ensemble that are responsible for executing the missions. Consequently, they are

candidates in the allocation process.

Knowledge exchanges between the coordinator and robots are carried out through knowledge mappings. In MissionControl , we use knowledge exchanges for (i) registering the properties about the robot in the knowledge base of the coordinator (line 51), and (ii) distributing the 'local mission' for the robots (line 55).

3.5 Mission Coordination

(DD.3) Task Decomposition For integrability purposes, MissionControl provides a service interface to integrate with an external mission planner. Moreover, it favors modifiability by decoupling responsibilities between task decomposition and mission coordination.

The Coalition Formation Process is supported by the estimate tasks subprocess that uses skill descriptors and environment descriptors that are part of the component model and are extended with concerns that impact each task type. The coalition formation process uses the coordinator's knowledge about the available robots that are updated by the robots that are members of the coordinator ensemble.

Estimating Manager The estimate manager provides estimations for the coalition formation process ('Estimate Tasks' in the Coordinator lane in Figure 3.2). The Estimating Manager provides the method ESTIMATE that receives a candidate worker and a list of tasks and returns a *Bid*. A Bid has three attributes: *worker*, the candidate robot, *partials*, estimates for each task in *task_list*, and *estimate*, the aggregation of the estimates of all tasks in *task_list*. The property estimate of the execution of tasks (e.g., time and battery charge required) is calculated by applying the Skill Descriptor estimate function to the context of task execution. To do so, the task execution context comprises task parameters (e.g., destination position), information about the robot (e.g., robot's origin position) and information about the environment (e.g., length of the shortest route between the origin and destination).

The Skill Descriptor Register keeps the instances of Skill Descriptors and is responsible for finding the appropriate Skill Descriptor for each task. Figure 3.4 shows the Estimating Manager, the Skill Descriptor Registers, and an exemplar of a skill descriptor.

Coalition Formation Process - The coalition is formed by selecting a subset of robots to execute roles in a mission. More specifically, the coalition formation process for each

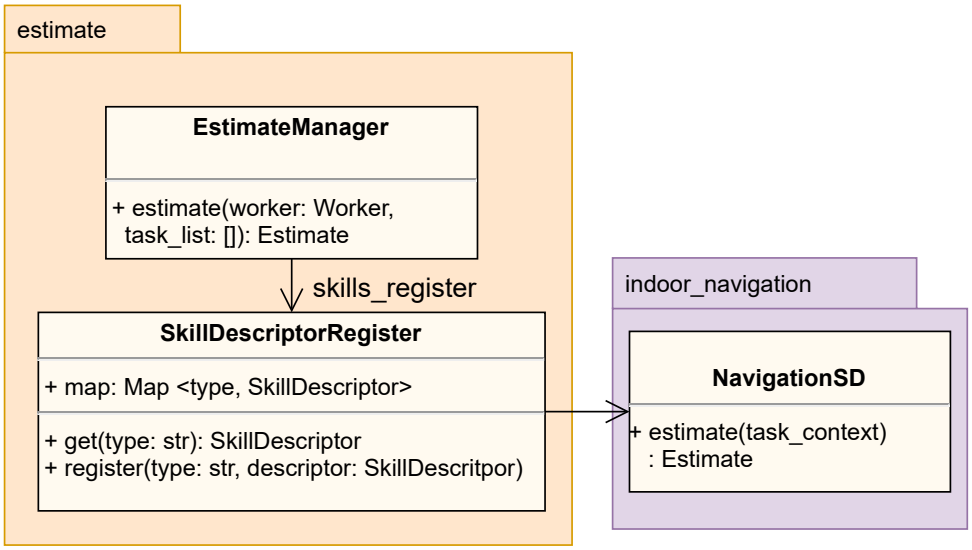


Figure 3.4: Estimate Manager

mission with pending assignments takes the set of currently not assigned workers and executes Algorithm 1.

In this work, we focus on missions that have been specified for instantaneous assignment, where an external process (e.g., a human user) requests a task in an uncertain timeline that may involve multiple robots (known in the literature as Single Task, Multi-Robot [33]). Then, the system assigns a set of robots among the available ones to execute the mission. We chose to realize coalition formation and task allocation simultaneously, i.e., we selected the robots to form the coalition based on the expected qualities of the execution of the tasks at hand by the selected robots. This is based on the premise that we will have quality information about the environment when we receive the mission request to choose a near-optimal set of robots to carry out the mission. There are classes of missions for which IA is not optimal, i.e., when we have a set of rooms to be cleaned, and we need to distribute these rooms among robot teams (Multi-Task, Multi-Robot model). In this work, we do not cover missions beyond IA.

(DD.4) Coalition Formation / Task Allocation Coalition formation and task allocation are performed simultaneously and delegated to a well-defined module within the architecture. Future variants could be integrated to improve performance or to support missions with other allocation requirements.

In a nutshell, Algorithm 1 is auction-based: for each not-assigned local mission, the coordinator creates a set of bids with a bid for each known available worker. The bid is viable if the related worker has all the required skills and sufficient resources (e.g., battery charge). The bids are then ranked, and finally, the best-ranked bid is selected.

Algorithm 1 Coalition Formation Process

Require: *mission* with pending local missions and *workers* a list of currently available robots

```
1: function CREATE COALITION(mission, workers)
2:     plan_rank_map ← {}
3:     pending_local_mission ← PENDING(mission)
4:     for local_mission in pending_local_mission do
5:         viable_bids ← []
6:         task_list ← FLAT_PLAN(local_mission.plan)
7:         candidates ← GET_COMPATIBLE_WORKERS(task_list, workers)
8:         for worker in candidates do
9:             bid ← ESTIMATE(worker, task_list)
10:            is_viable ← CHECK_VIABLE(bid)
11:            if is_viable then
12:                viable_bids.insert(bid)
13:            end if
14:        end for
15:        if viable_bids = ∅ then
16:            return false
17:        else
18:            ranked_bids ← RANK(viable_bids)
19:            plan_rank_map[local_mission] ← ranked_bids
20:        end if
21:    end for
22:    selected_bids ← SELECT_BIDS(plan_rank_map)
23:    SET_ASSIGNMENTS(pending_local_mission, selected_bids)
24:    return true
25: end function
```

Algorithm 1 receives as input a mission context and the set of available workers. The fields in each worker object are the same as the Worker role of the robot ensemble component, as shown in Listing 3.1.

First, the PENDING function receives local missions with pending assignments. Local missions are in a pending state if they are not concluded or do not have an assigned worker. For each pending *local_mission*, the function searches for an assignment as follows (lines 4-21). First, *task_list* receives a *flat* version of the local mission, i.e., a list of the elementary tasks contained in the local mission (lines 6. Next, the set of compatible workers is assigned to *candidates* and 7). Compatible workers are the ones who have all the skills required to perform the local mission, i.e., have a skill for each task in *task_list*. Then, for each candidate *worker*, we estimate the cost/utility of the allocation (line 9). The ESTIMATE function is a call for the estimate manager, which is a subsystem that supports the coalition formation process and is extended by

Code Listing 3.1: DSL excerpt from ensemble specification

```
role MissionsCoordinator:
    missions, active_workers, required_skills

component Coordinator features MissionsCoordinator
    knowledge
        missions = []
        active_workers  = []
        required_skills = []

    process coalition_formation
        inout missions
        inout active_workers
        function:
            ...
        scheduling: periodic( 100ms )

role Worker:
    skills, local_mission, location, battery_level,
        ↪ battery_consumption_rate, avg_speed

component Robot features Worker
    knowledge:
        skills = []
        local_mission = ...
        location = ...
        battery_level = ...
        battery_consumption_rate = ...
        avg_speed = ...
        task_status = ...

    process properties_sensing
        inout location, battery_level
        function:
            ...
        scheduling: periodic( 1000 ms )
    process sequencing
        in local_mission
        inout task_status
        function:
            ...
        scheduling: periodic( 100ms )

ensemble MissionsCoordination:
    coordinator: MissionsCoordinator
    member: Worker

    membership:
        coordinator.required_skills ∩ member.skills ≠ ∅

    knowledge exchange:
        # member to coordinator
        coordinator.workers ← member
        mission_assigned = mission_member_is_assigned(coordinator, member)
        update_mission_progress(mission_assigned, member.local_mission)
        # coordinator to member
        member.local_mission ← get_member_local_mission(coordinator, member
            ↪ )
    scheduling: periodic( 100ms )
```

the environment and skill descriptors. The function ESTIMATE returns *bid*. After creating estimates for the assignment, we check if it is viable (line 10), i.e., if it satisfies restrictions, such as if the robot has enough battery to perform all tasks in *task_list*. If restrictions are satisfied, the *bid* is inserted into *viable_bids* (lines 11-13). Then, if no viable bid was obtained for any of the local missions with pending assignments, the algorithm returns $false$ (lines 15-17). In that case, any robot will be assigned, even for other local missions within the mission that have viable allocations. That was a design choice, as we opted not to assign robots to missions if part of the mission cannot be executed. Otherwise, if *viable_bids* is not empty, the viable bids are ranked (line 18) and the rank is placed on a map that relates local missions that are not assigned with the list of ranked bids (lines 19). Finally, the best-ranked bids for each local mission are selected and an assignment is realized for each local mission in *pending_local_mission* (lines 22-24).

3.5.1 Mission Coordination: Component Model

The previous section described the Mission Coordination Runtime Environment of MissionControl . It provides a domain-independent model for the coordination of the missions. The components described in this section extend the runtime environment for specific missions and related tasks.

Global Mission Plan The global mission plan is received in requests from users. In MissionControl, the Global Mission Plan specifies the mission in terms of a decomposition of tasks, in which each task is associated with an intermediate state that the system must achieve in order to conclude the mission. For example, Laboratory Sample Logistics iHTN (Figure 2.2) has a task *navto_room*3 which means that the robot must reach the *room*3 location (that is, the nurse location defined in the mission creation). It's important to note that tasks in MissionControl are described in terms of what needs to be achieved, or in other words, the intermediate state that a successful execution of the task guarantees. For example, back to tasks *navto_room*3, it specifies the end state ($location = room3$), but not other information, such as, for example, the route that a robot will take to reach this location. This is because each candidate robot may have different initial positions, so the route is calculated later in the process, during the coordination process. It is a tree that refines abstract tasks into elementary executable tasks. Elementary tasks have types that are unique identification labels within a domain (e.g., navigate, approach a person, authenticate a person, and operate a drawer). For a received request to be valid, its contained plan should have elementary task types within the set of supported task types, i.e., the task types for which the system has equivalent skill descriptors and implementations.

The Global Mission Plan is expected to be generated with the help of an HTN planner (see, e.g., Shop2 [26]). We extended iHTN [25] by allowing tasks to be assigned to roles rather than specific agents. Mission roles are either managed or non-managed. Non-managed roles are agents that the system has no control over, i.e., it cannot send plans for them to execute. In laboratory samples (Fig. 2.2), the nurse is a non-managed role, while 'r1' is a managed one. In addition, the managed roles must be assigned by the coordinator.

While the Global Mission Plan specifies what must be performed, the Skill and Environment Descriptors are used to extend the system with the capability of estimating the viability and cost of executing the tasks in the plan by a given robot.

Skills Descriptors Components that provide estimates of the properties of the execution of a task by a given robot in a specific task context. The skill descriptor interface contains a method *estimates* that receives the task context and returns estimates for that task. The skill descriptor can have Environment Description as dependencies that support the estimation process by providing information about the environment. The dependencies between the skill descriptors and the environment descriptors are resolved by the runtime environment.

Environment Descriptors These components encapsulate the information about the environment required by the Skill Descriptors. The interface of the environment descriptor is an empty interface. This is because the information required by the skill descriptors is domain-specific. Figure 3.5 presents an example of an environment descriptor and a skill descriptor. RoutesED is an environment descriptor that is instantiated with the environment map and resolves queries about routes between two points. NavigationSD is a skill descriptor that uses the route descriptor to perform estimates of navigation tasks.

(DD.5) Coordination Extension Points To favor the reuse of MissionControl in different environments and across different missions, we define points of extension where estimates can be extended and adapted for the end-application mission and environment. The extension points allow the System Integrator to add new components without having to change significant parts of the system. The coordinator extension points are: skill descriptors and environment descriptors.

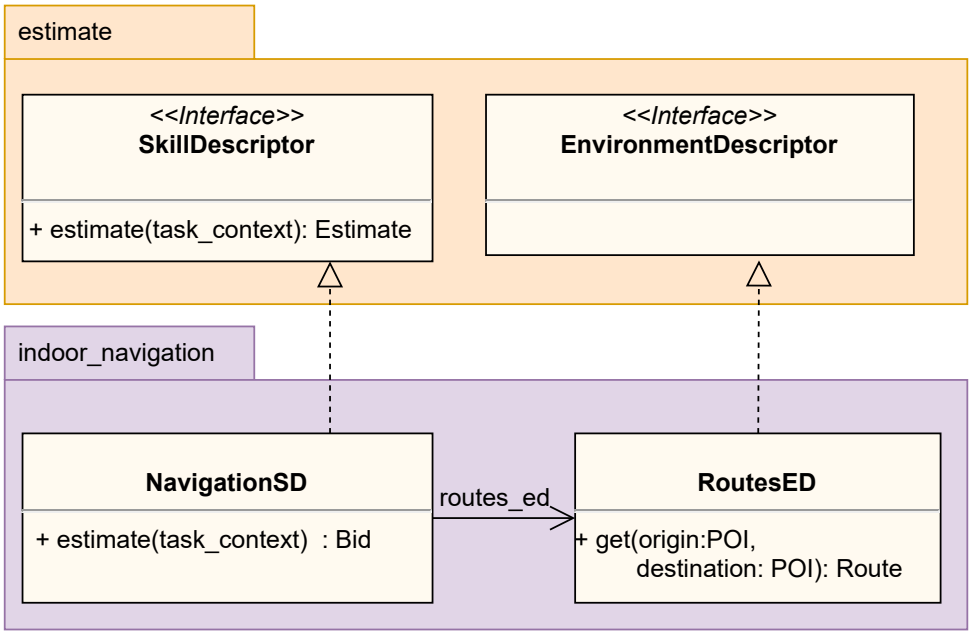


Figure 3.5: Skill and Environment Descriptors Interfaces and implementation examples

3.6 Task Execution

The coalition formation process, described in the last section, assigns local plans to robots. Then, in the execution phase, the robots carry out the local plans. Task Execution is represented in the rightmost lane in Figure 3.2.

The robot performs its tasks by selecting the next task from its local plan and activating an appropriate skill implementation. Skill implementations encapsulate the different behaviors that the robot can apply to execute tasks, while the runtime environment is concerned with the life cycle of these behaviors. The Task Execution assumes that the Local Mission is present in the knowledge base of the robot. Then, the Sequencing process is responsible for determining from the local mission which of the next tasks should be executed. When a new task is selected, the Sequencing process looks for a compatible skill implementation for that task in the library of skill implementations and activates it. Synchronizations (i.e., Send Message/ Wait Message) are built-in skill implementations, while other skills must be integrated into the system. Task Execution in MissionControl is based on concepts from Behavior Trees. In traditional BT applications, the overall behavior of an agent is specified in a single BT.

3.6.0.1 Task Execution: Runtime Environment

In relation to task execution, the MissionControl runtime environment encompasses sequencing and management of the life cycle of the active skill. This runtime environment

is extended by the library of skill implementations, that the low-level control for each skill that the robot supports.

Sequencing The sequencing process is performed periodically, and it is responsible for (i) selecting the next task that must be executed from the current local plan, (ii) loading the skill for the selected task, and (iii) ticking (i.e., passing the control to) the active skill until the task reaches an end. Ticking is a concept of behavior trees in which the control is passed to the root node of the tree, and according to the conditions of the BT, the control is switched between subtrees [56].

Each execution of the process (Algorithm 2) is as follows. If there is no active plan, it does nothing (line 6). If there is no active task, obtain the next task (line 10) that must be performed according to the local plan and load the equivalent skill from the skill library (line 12). When there is a task in progress, or just after loading a new skill, tick the BT of the active task, obtain the result (line 15), update the local mission (line 16), and set the status of the task in the knowledge base (line 18).

Algorithm 2 Sequencing Process

```
Require:
 1: local_mission: manages local plan, provides the tasks into the correct order
 2: active_skill_ctrl: manages the life-cycle of the skills
 3: task_status: interface with the knowledge base
 4:
 5: function SEQUENCING(local_mission, active_skill_ctrl, task_status)
 6:     if local_mission.HAS_NO_PLAN( )then
 7:         return
 8:     end if
 9:     if active_skill_ctrl.IS_IDLE( )then
10:         next_task ← local_mission.NEXT_TASK()
11:         skill_impl ← skill_library.QUERY(next_task)
12:         active_skill_ctrl.LOAD(skill_impl)
13:         task_status.SET_VALUE(status)
14:     end if
15:     tick_status ← active_skill_ctrl.TICK()
16:     local_mission.UPDATE(tick_status)
17:     ts ← local_mission.GET_TASK_STATUS()
18:     task_status.SET_VALUE(ts)
19: end function
```

Active Skill While a skill is active, i.e., during the execution of a task, the active skill receives ticks. In these ticks, the active skill can interface with low-level processes to

control the robot behavior. This continues until the BT reaches a final state of success or failure.

In each tick, the runtime environment traverses the active skills in the BT, checking conditions, and possibly starting actions. The result of the BT tick can be: (i) *In progress*, when the task is being performed; (ii) *Success end*, when the task was completed, and finally, (iii) *Fatal failure*, when the task cannot be completed.

Synchronization - Besides elementary tasks, which are performed by skills, the local plan also has synchronization tasks as described in Section 2.1.1.1. These synchronizations are of the form wait message and send message. These synchronizations are performed by the Synchronization manager.

3.6.0.2 Task Execution: Component Model

Skills Implementations The skill implementation encapsulates the complexity related to executing a specific type of task, providing a uniform interface for mission coordination. Skill Implementations are deployed to a 'library of skills', and are associated with a task type. At the mission level, a task might be seen as not started, in progress, completed, or in failure. The complexity of monitoring the environment and diagnosing the presence and type of failure must be implemented as part of the skill.

The skill-based design facilitates the system design process, breaking the robots' behaviors into different skills. Yet, in an uncertain environment, every single skill can be complex, as it may need to apply different behaviors in order to handle different emerging situations. For example, the ability to navigate may not only navigate from one room to another on the same floor, one may also have to take an elevator or ask a person for permission to pass. To perform these different behaviors, skill providers may have to make use of several low-level sensors, actuators, and processes. For allowing better management of behaviors at the skill levels, we integrated BTs at task execution. In our approach, each skill implementation has a BT that is loaded when the skill is activated. Then, during the task execution, the system periodically ticks the BT and evaluates the progress of the task. Actions in the BTs can control lower-level processes in the robot. BT promote the composition of reusable behaviors, which can be useful in a skill-based design as it allows parts of skills to be reused between skills (e.g., parts of a skill for approaching and following a person can be reused, such as parts related to locating and determining the position of a person in relation to the robot).

(DD.6) Skill-based system To favor the reuse of MissionControl in different missions and with a heterogeneous set of robots, we followed a skill-based design to implement robot behaviors.

To evaluate MissionControl, we implemented a prototype realizing the architecture in Python (version 3.8). We implemented the core data model, processes, and the estimate manager in pure Python, without dependencies on specific ensemble frameworks. Then, we implemented a version of the DEECo component model for Python, highly based on an existing implementation named PyDEEco [58][1] and implemented an integration layer between MissionControl and the DEECo component model. We created an integration layer to maintain a separation between the implementation of the ensembles and the core of MissionControl. This separation was done to increase testability and to facilitate the creation of future ports on top of other ensemble providers.

Figure 3.6 shows the main modules of the implementation of MissionControl as a concrete framework and the dependencies between the modules. The application package represents any end application that uses the MissionControl. The framework implemented by MissionControl has five modules: *data_model*, *coordination*, *estimating*, *execution* and *deeco_integration*. The data model contains the types and basic algorithms for processing missions and, therefore, is a dependency for all other modules. The module *estimating* has the implementation of the estimate manager, and the interfaces for the Skill Descriptor and Environment Descriptors. The module *coordination* contains the coalition formation process and the data structures to support it. The *execution* module contains the sequence process, the skill library, the interface required for skill implementations, and supporting types. Finally, the module *deeco_integration* is where the Coordinator and the Robot components as well as the definition of the mission coordination ensemble, are implemented (as discussed in Section 3.4). We introduced the deeco_integration module to create a separation between the data model and algorithms and DEECo, which increases the testability and modifiability of the modules.

3.7 Implementation

An implementation of a DEECo component model, such as PyDEEco, provides abstractions of the ensemble, such as ensemble formation, knowledge exchange, and function scheduling. PyDEEco also provides an internal simulator that was handy during the development, as it allowed us to execute an ensemble-based system as a local single process application. This simulation functionality allowed us to create integration tests with

[1] https://github.com/d3scomp/PyDEECo

standard test libraries, without any special environment setup. However, pyDEEco lacked some core features of the DEEco component model that our architecture required, so we forked the pyDEEco into the MissionControl repository and implemented the missing features (it is represented by the *deeco* module in Figure 3.6).

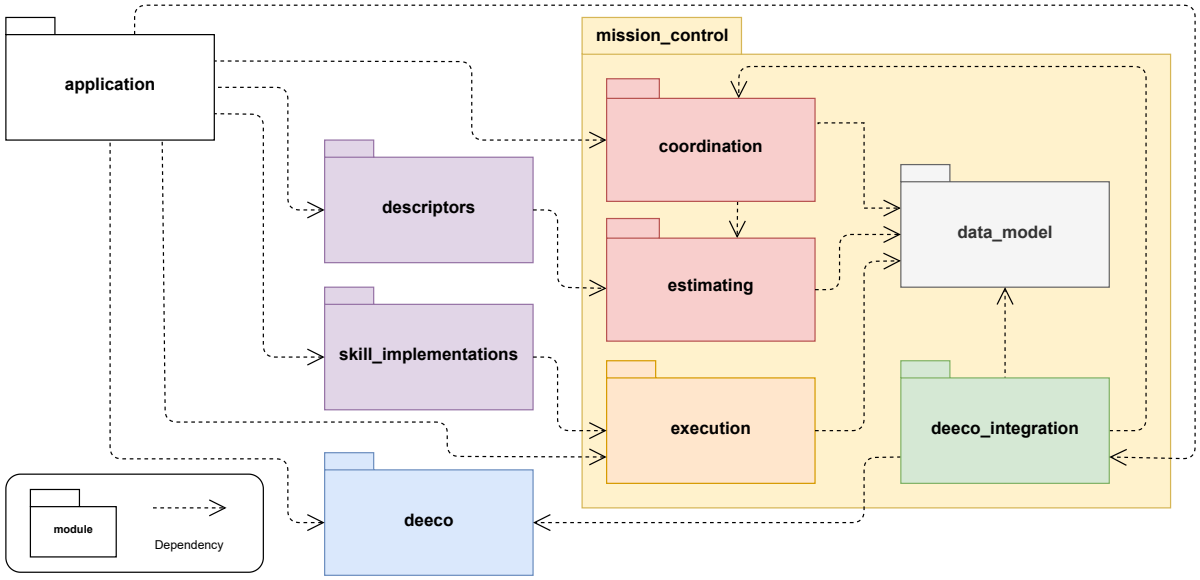


Figure 3.6: Dependencies Between MissionControl Modules

The application module implements the main function and instantiates and wires the components together. The instantiation of the architecture can have complex dependency graphs, as, for example, the coalition formation process depends on an instance of the estimate manager, which in turn depends on different skills descriptors, which, in turn, depend on environment descriptors. To ease the process of wiring the components together, we used a Dependency Injection (DI) [59] container, which automatically instantiates dependencies and wires them together. The robot version of the application uses py-trees [2], and ROS [15] to implement the active skill. ROS provides middleware and inter-process communication functionalities, while py-trees is a Python Implementation of behavior trees.

(DD.7) Make the implementation independent of specific frameworks and middleware whenever possible - We decided to implement the data model and main algorithms as standalone code, and created specific modules that provide bindings for ROS and DEECo. This favors modifiability and integrability by making MissionControl independent of the ROS and DEECo frameworks.

[2] http://wiki.ros.org/py_trees_ros

To evaluate the architecture, we implemented some application components. These components are not part of the MissionControl Runtime Environment, but are an example of components extending it. The indoor_navigation package has three components: Routes Environment Descriptor , Navigation Skill Descriptor , and the Navigation Skill Implementation . The route environment descriptor is based on a topological map [60], which uses a graph of points-of-interest (POIs, e.g., 'laboratory', 'intensive care 1') and linear segments between these points. A graph search algorithm is used to generate a route between two locations. The resulting route has a total distance (in meters) and a list of waypoints (intermediate points in the path used to orient the navigation). The route environment descriptors are instantiated with a specific building graph. The descriptor of navigation skills depends on the descriptor of the environment of the route. The total length of the route is used to estimate the time required to reach a destination from the origin point (that is, the position of the robot before the navigation task).

The Navigation Skill receives a route already planned and should follow the list of waypoints to get to the destination. We implemented the control of following the list of waypoints in the BT of the skill, while the navigation between two adjacent points in the list of waypoints and obstacle avoidance are provided by the move-base ROS node[3].

[3] http://wiki.ros.org/move_base

Chapter 4

Evaluation of MissionControl

An architecture for mission coordination of heterogeneous robots, Full article published in Journal of Systems and Software, Volume 191, 2022.

The evaluation of MissionControl has two goals. First, as shown in Section 4.1, through a controlled experiment [61, 62] we aim to collect evidence that our architecture promotes effective mission coordination of heterogeneous robots. Second, as shown in Section 4.2, we evaluate the modifiability and integrability of the architecture.

4.1 Quantitative Evaluation

The experiment described in this chapter is based on the scenario described in Section 3.1, and can be divided into four phases, which are discussed in the following subsections: (i) definition of objectives (Sect.4.1.1) (ii) setup of experiments (Sect. 4.1.2), (iii) data collection and analysis (Sec. 4.1.3), and (iv) presentation of results (Sec. 4.1.4).

4.1.1 Evaluation Goal 1

We structured the definition of objectives following the Goal Question Method (GQM) paradigm [63]. This standard helped us derive metrics to evaluate the feasibility and efficiency of the MissionControl relative to a baseline approach. The derivation process is performed by formulating questions that concretely explore the reasoning behind the evaluation. To that extent, evaluating the MissionControl consists of collecting and computing the derived metrics. According to the GQM paradigm, we describe the evaluation's overall goal and the metrics.

Goal 1: Evaluate whether MissionControl promotes effective mission coordination of co-operative heterogeneous robots.

- *Question 1)* How reliable is MissionControl in forming coalitions of heterogeneous robots for the completion of missions compared to the baseline?
- *Question 2)* How time-efficient are the heterogeneous robot teams in MissionControl compared to the baseline?
- *Question 3)* How effective is MissionControl in preventing battery failures compared to the baseline?

To answer questions from 1 to 3, we set up an experiment that consists of running a series of trials on randomly generated scenarios. To allow for comparison, we executed each scenario with two treatments: MissionControl and a *baseline* approach.To the best of our knowledge, there are no other applicable approaches to serve as a baseline for this evaluation. Therefore, we used as the baseline an algorithm that randomly assigns robots to roles on the mission plan.

Metrics For each trial run, we collected two direct measures: (i) the end-state of the trial run, and (ii) the time-to-conclude (TTC) of trial runs that ended in success. The end-state can be either a success or a failure, and in the case of failure, we also collect the cause of failure.

These measures constitute a sample of the expected behavior of the system, for which we can perform statistical hypothesis tests to answer the questions. At the end of our controlled experiment, we want to assess whether MissionControl will fulfill the following expectations:

1. A higher success rate than the baseline.
2. A lower mean time-to-conclude than the baseline.
3. A lower mean failure rate due to a low battery than the baseline.

4.1.2 The Experimentation Scenarios

To evaluate these expectations, we conducted a t-test to compare the sample of trials coordinated with MissionControl against those conducted using the baseline. This analysis yielded a confidence measure that reflects the t-tests support for our hypothesis. It is important to note that we employed a conventional 0.95 confidence level as the benchmark throughout the controlled experiment.

We evaluate our approach in the mission specification for laboratory sample logistics described in Section 3.1, where the system has to collect the sample from a nurse and deliver it to the laboratory. In addition, the experiment is performed in a simulated environment by running a set of trials, where each test is defined by a scenario and a mission plan. A scenario consists of an initial position for a nurse requesting a sample delivery and an initial configuration of each robot. The mission plan is generated according to the scenario using MissionControl or the baseline algorithm. The simulated world, the robots, and the nurse are further described as follows:

Simulated World. The simulation environment is an indoor hospital layout with 15 rooms and a separate laboratory area. The environment has 36.3 *meters* by 34.8 *meters* (LxW). Figure 4.1 shows a higher-level view of the environment and the topological map, where small circles and a label mark the entrance of the hospital areas. The laboratory is labeled LAB, while the others represent patient rooms.

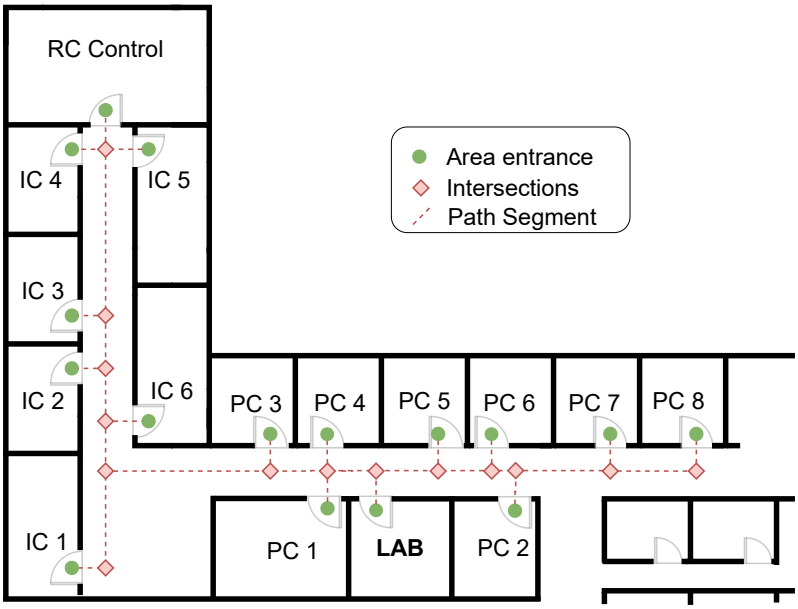


Figure 4.1: Lab Samples Logistics - Extract of the hospital floor plan and topological map

Robots and Nurse. The simulated world is populated with a nurse, 6 initial mobile robots, and a static robotic arm, located in the lab, that can fetch samples from the mobile robots. The scenarios are created by choosing the initial position of the nurse and the initial configurations of the robots available. Each robot in the scenario has four initial settings: (i) a set of skills, (ii) initial location (i.e., the entrance of one of the 15 rooms), (iii) initial battery level, and (iv) battery consumption rate. During trials, the robots have an average speed of 0.13 m/s.

The scenario is defined by a set of controlled variables that define the initial configurations of the robot set and the nurse. To generate the set of scenarios for the experiment,

we randomly generate sets of alternative values for each of the scenario‘s initial configurations (i.e., positions of each agent, robots, and nurse, and configurations of robots), and combine them. For each initial configuration of the scenario, we generate three random sets of values and then generate a total combination of the sets, totaling 81 (3^4) unique scenarios. All other factors related to the simulated robots, the simulated humans, and the map were kept constant throughout the scenarios.

The robots are assigned to tasks according to a mission plan that is generated by the treatment that is the object of evaluation in the experiment. The treatment is either MissionControl, which executes as described in Sections 3.3 and 3.7, and a baseline algorithm that randomly assigns robots to roles in the mission specification.

As outputs of the trial run, we collect the end-state of the trial and the time-to-conclude (TTC) in case the end state is a success. These are the dependent variables of the experiment. The end state of a trial can be (i) *success*, (ii) *no-skill failure*, (iii) *low battery failure*, (iv) *timeout sim failure*, and (v) *timeout wall failure*. The *success* end-state occurs when the lab sample reaches its destination before a failure occurs. The *no-skill failure* occurs when a robot tries to execute a task for which it does not have a proper capability. The *low battery* failure occurs when the assigned movable reaches a battery level below 5%. The *timeout sim failure* occurs when, after the simulation reaches 15 min on the simulated clock, it does not reach an end-state. And finally, a *timeout wall failure* occurs when the experiment executor machine marks 45 minutes of wall-clock (i.e., real-world time, not within the simulation) executing the same trial, without reaching an end-state, which normally indicates a failure in the simulation setup. The TTC is measured only for the trials that were successful and is the difference between the time stamp of the request and the time when the lab sample reaches its destination.

4.1.3 The Experimentation Pipeline

Conducting an experiment to assess the coordination of heterogeneous robots and an environment of uncertainty is not a trivial task, as we need to create a significant diversity of scenarios, configure the environment suitably to represent the scenario, and run the trials in sufficient numbers for the results to have statistical significance. Therefore, to execute the experiments, we created a pipeline comprised of three phases: generation, execution, and analysis. Figure 4.2 illustrates the execution pipeline of the experiment.

The purpose of experiment generation is to create scenario trials by assigning a plan to the robots. We created two trials for each generated scenario, one for the approach and one for the baseline. We generated the approach plan by (i) instantiating the coordinator and components for each robot, according to the scenario, (ii) triggering a request to the coordinator, according to the position of the nurse defined in the scenario, (iii) waiting

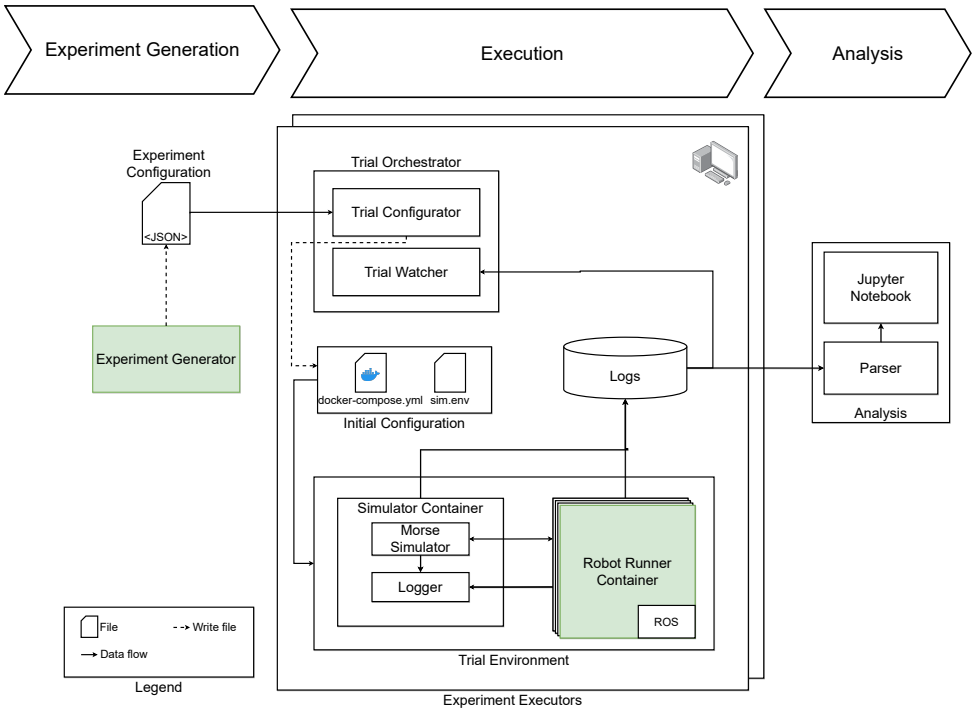


Figure 4.2: The framework of the experimentation pipeline

for the system to form the ensemble and handle the request, and finally (iv) parsing the plan assigned to the robots and writing it to the trial. The experiment generation process produces an Experiment Configuration file containing the parameters for executing each trial (162 trials in total, 81 for each treatment).

The second phase of our experiment pipeline is the execution of the experiment, comprised of two major components: the trial orchestrator and the trial environment. The Trial Orchestrator creates an appropriate simulation environment for each trial, consisting of a simulator and a robot runtime controller for each robot in the scenario.

The Trial Environment comprises the simulation container. Our simulation environment uses the Morse simulator [64] instantiated with the map depicted in Figure 4.1. Also, as part of the simulation container, there is the logger service that receives logging information from the other sections of the system and then stores the data in a file, also logging the simulation time and the sender node. Additionally, we extended the simulation environment with instrumentation to collect the trial end-state and TTC. To evaluate *success end-state* and *TTC*, we implemented a new entity inside the simulator, *Inventory*, which is instantiated at the final destination, and it registers the entrance of lab samples as soon as they are delivered. The Inventory receives the sample from the robot arm in the lab, sends the event log data, and requests the simulation end by sending a keyword to the logger service. In addition, we implemented a battery module that simulates battery consumption and identifies low battery failures. When the end of the

simulation is signaled, or the simulation exceeds the time limit, the Trial Watcher then shuts down the simulation, stores the log content in a proper file with the trial ID, and logs the execution's wall-clock time.

The execution phase is quite demanding from the computational resource perspective, as it was executed in parallel by 8 Dell OptiPlex 3040 desktops with Intel i5 6th gen, 8 GB of RA,M and took 28 hours to conclude. Each of the 8 machines independently executed the completed set of trials, resulting in the 81 scenarios running 8 times for both the MissionControl and the baseline approach. So, for both treatments, there were 648 trial runs, 1296 executions in total. The experiment repository[1] has the implementation of the pipeline, as well as instructions for deploying and executing.

Finally, in the analysis phase of our pipeline, we collect the data in the form of log files for each trial, parse it, and create a dataset to be manipulated by statistical software. From this dataset, we plotted the graphs and performed the hypothesis tests that are presented in Section 4.1.4. The data obtained at run-time along with the Jupyter Notebooks [17] used during the analysis are available in a public repository [18].

4.1.4 Results

How reliable is MissionControl in forming coalitions of heterogeneous robots for concluding missions compared to the baseline? To answer the first question, we compared the number of successful trial runs of our approach with the baseline treatment. In 648 executions, the mission coordinated by the baseline had 354 successes, while MissionControl had 558, which yielded an increase of 57.627 % in the success rate compared to the baseline. To check whether the success rate obtained by MissionControl was statistically significantly higher than that of the baseline, we performed a paired t-test [65] comparing both treatments. More specifically, the number of successes obtained by MissionControl and the baseline in each scenario was paired, and a set ($\overline{D}$) was generated with the differences between the number of successes, for each scenario ($\overline{D}_i = S_{MC,i} - S_{BL,i} \mid i$ the index of the scenario). The null hypothesis H_0 is that the difference in the mean success rate is equal to zero ($\mu_D = 0$), that is, MissionControl has the same mean success rate as the baseline, and the alternative hypothesis H_1 is that the mean difference is higher than zero ($\mu_D > 0$), that is, MissionControl has a higher success rate than the baseline). The result of the paired t-test was that H_0 should be rejected in favor of H_1, with the confidence of $(1 - \alpha) = 0.9995$, indicating a high level of confidence that MissionControl leads to a higher mean success rate than the baseline.

Figure 4.3 shows the distribution of the end states for both the MissionControl and the baseline in each of the 81 generated scenarios. MissionControl has fewer failure states

[1] https://github.com/lesunb/morse_simulation

in general. In particular, the trials treated with MissionControl did not present failures in relation to the chosen robot not having the necessary skills for the task (no-skill failure). These no-skill failures are due to the coalition formation process filtering robots that do not have the required skills before estimating the bids. We also got fewer occurrences of failures due to a low battery level for the scenarios treated with MissionControl. This decrease in failures is due to the approach that checks for resource restrictions in the coalition formation process. Figure 4.4 summarizes the distribution of the number of successes per scenario as a violin plot. We can observe that our approach has a higher concentration of a higher number of successes per scenario compared to the baseline. In summary, MissionControl completely avoided one class of failure (no-skill failure) while reducing the occurrence of others (low-battery and timeout failures). The findings support that a MissionControl-based system can form coalitions capable of executing missions, reducing the incidence of failures compared to the baseline approach. We provide a more in-depth discussion of the observed failures in trials treated with the MissionControl at the end of this section.

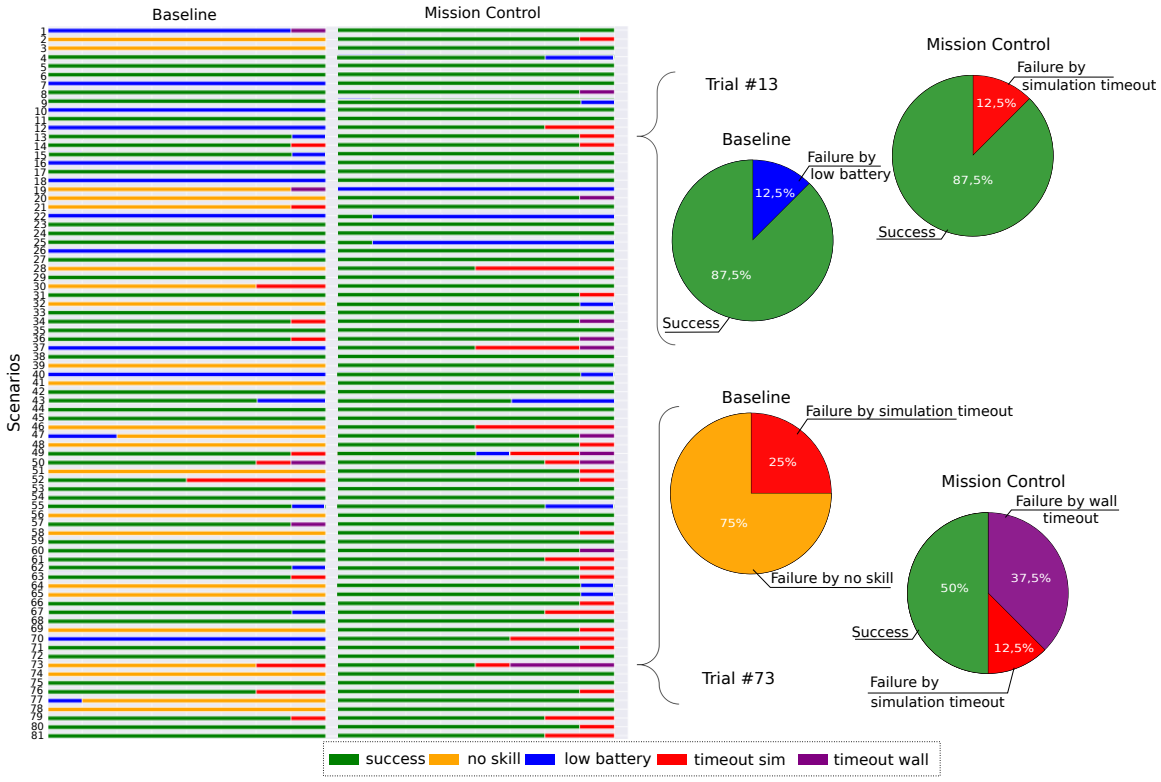


Figure 4.3: Trials results

How time-efficient are teams of heterogeneous robots in MissionControl compared to the baseline? To answer the second question, we compared the average time within the

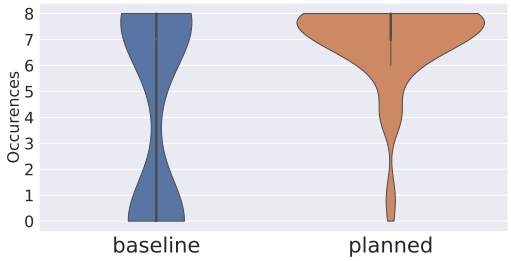


Figure 4.4: Violin plot with the number of successes per scenario.

simulation for the conclusion of successful missions between the approaches and the baseline. In 648 executions, $TTC_{MissionControl}$ was 268.495 s, while $TTC_{Baseline}$ was 318.28 s, resulting in a decrease of 16.64 % on the average time required to complete missions.

To verify the statistical significance of the results, we performed a t-test (unpaired). The test was performed following a method for inference on the difference in the means of two distributions with unknown variance [65], taking as samples the set of time-to-conclude for either of the treatments (a total of 354 values for the baseline, and 558 values for MissionControl). The null hypothesis H_0 is that the difference in the mean TTC equals zero ($\mu_{diffTTC} = 0$), i.e., MissionControl have the same mean TTC as the baseline, and as an alternative hypothesis H_1, that the mean difference is less than zero ($\mu_{diffTTC} < 0$), i..e, the MissionControl has a lower mean time-to-conclude than the baseline. We found with confidence $(1 - \alpha) = 0.9995$ that H_0 should be rejected in favor of H_1, concluding with statistical significance that the mean of TTC is lower for MissionControl than for the baseline.

In Figure 4.5, we have a visualization of the time distribution in seconds. We can see that our MissionControl, in orange, has a higher concentration of missions that are completed in less time. It should be noted that there is a concentration of trials that yields a TTC between 350s and 450s.

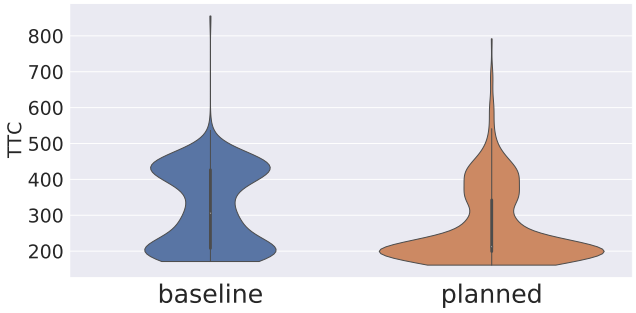


Figure 4.5: Violin plot with time distribution (TTC in seconds)

We investigated the data and concluded that this concentration is related to the influence of factors in the scenario, especially the initial location of the agents. As explained previously, the 81 scenarios were generated by a total combination of three levels of four independent factors. One of such factors is *initial location*, which defines the room where the set of simulation agents (that is, the nurse and robots) will be at the scenario initialization. By grouping the TTC for the trials with the same initial position, we obtain three groups (a, b, and c) for the planned and baseline trials. Figure 4.6 shows a box plot for TTC of the trials in each distance group. The groups a and b have a similar distance traveled by the robot, but the group a requires the robot to maneuver a sharper turn than the group b. The group c is the shortest distance scenario. The set of scenarios in group a is responsible for the TTC of around 350 and 450 seconds. Figures for the other factors are available in our analysis repository [18]. They were generated from Jupyter Notebook scripts [17].

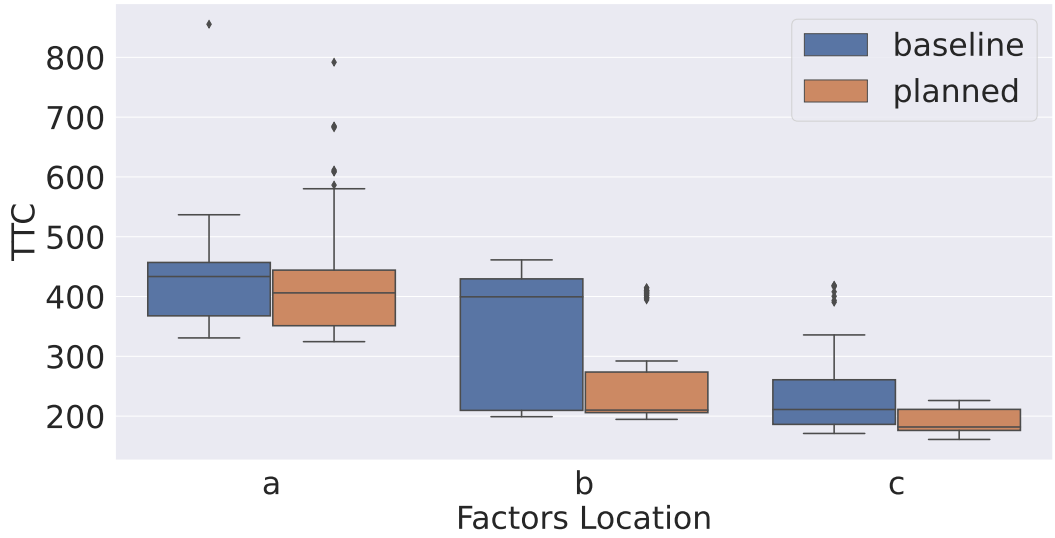


Figure 4.6: TTC for trials grouped by initial location (in seconds)

How effective is MissionControl in preventing failures from lack of battery compared with the baseline? To answer the third question, we compared the occurrences of failures due to the battery reaching a critically low level. In 648 executions, our approach had a total of 35 low battery failure scenarios, whereas the baseline had 97 low battery failures, which resulted in a reduction of 63.92 % for low battery events compared to the baseline.

Then we checked whether MissionControl was able to reduce the mean failure rate due to a low battery with statistical significance. We performed a paired t-test pairing the difference of the number of failures due to low battery in each scenario ($\overline{D}_i = BF_{MC,i} - BF_{BL,i} \mid i$ the index of scenario). The null hypothesis H_0 is that the difference in the

mean success rate is equal to zero ($\mu_D = 0$), that is, MissionControl have the same mean failure rate of the baseline, and the alternative hypothesis H_1 is that the mean difference is smaller than zero ($\mu_D < 0$), that is, MissionControl has a lower mean failure rate due to low battery than the baseline. The result of the t-test is that H_0 is to be rejected in favor of H_1 with confidence in $(1 - \alpha) = 0.95$. In conclusion, MissionControl reduces the mean rate of failures due to battery level compared to the baseline.

Qualitative Analysis of Failures The results for *Question* 1 indicate an overall improvement in the rate of success of the approach when compared to the baseline, corroborating that checking for skills and resources before forming the coalition can avoid failures. However, we still observed a non-negligible rate of 13.89% of failures. The next step was to further analyze the execution of the trials planned by the approach to identify the cause of these failures and the limitations of the MissionControl. The verification was twofold: first, we verified the logs generated by the mission coordinator, verifying if the coalition formation processes were correctly executed; second, we analyzed the logs generated by the robots assigned to tasks, and inspected the logs generated in trails associated with failures (90 in total). Table 4.1 summarizes the findings.

	Executed	**Success**	**Failure**
Mission Coordination (81 scenarios)			
The ensemble was Formed	81	81 (100%)	0
Robots were correctly evaluated	81	81 (100%)	0
Best evaluated robot was assigned	81	81 (100%)	0
Task Execution (8 runs of 81 scenarios)			
Simulation Environment Setup	648	633 (97.69%)	15 (2.31%)
navto_room	633	630 (99.53%)	3 (0.47%)
retrieve_sample	630	615 (97.62%)	15 (2.38%)
navto_lab	615	564 (91.71%)	51 (8.29%)
unload_sample	564	558 (98.94%)	6 (1.06%)
Total	648	558 (86.11%)	90 (13.89%)

Table 4.1: Success rate of different phases of the experiment

To verify the Coalition Formation phase, we implemented a script to extract information from the logs generated by the mission coordinator and check three properties: (i) the ensemble was formed correctly, specifically, if all robots in the scenario were inserted on the knowledge base of the mission coordinator; (ii) robots were correctly evaluated, i.e., for all robots with the required skills, the time and required battery were estimated and the estimated values are what it is expected; and finally, (iii) the best-evaluated robot was assigned, i.e., the best-ranked robots received the assignments. After identifying and

fixing an issue, all 81 scenarios passed the 3 checks, which gave us confidence that the implementation of the Coalition Formation Process was executed correctly. We then further analyzed the Execution phase by analyzing the logs generated during the execution, especially the trial runs that resulted in mission failure. We analyzed the logs generated and classified them. In the 648 trial runs of the approach, we observe 90 failures. We categorized the failures into two groups: (i) simulation failure and (ii) task execution failure. The simulation failures occurred when the simulation was not loaded correctly, and the generated log files were empty. Simulation failure occurred in 15 of the 648 trial runs. We evaluated that these failures were related to the simulation environment and not to the mission coordination or task execution. Task execution failures occurred when an assigned robot that had the skill to execute the task started to execute a task according to its plan, but was not able to complete the task successfully. The trials that occurred with task execution failures ended up in a low-battery or timeout-sim state, depending on whether the battery of the assigned robot was depleted or not reach a timeout before the simulation. By observing the last tasks started on a failed trial, we determined in which task the failure occurred. Table 4.1 presents the task execution failures aggregated at the top-level decomposition of the plan for the lab samples example (iHTN on Figure 2.2). The majority of task execution failures occurred during the *navto_lab* task (51 of the 90 failures). Specifically, these failures occurred in the 8 trial runs of scenario 19, the only scenario for which the approach did not obtain any success. By analyzing the time taken by the assigned robot, we concluded that in these scenarios, the estimates were too optimistic for the task *navto_lab*. One possible explanation is difficulties while maneuvering out of the nurse room (IC 6, in the case of scenario 19) where the robot needs to realize a very close turn while exiting the room to get to the corridor that leads to the lab (the same maneuver is present in 73.3% of trials that ended on failure while representing only 33.3% of the total of trials).

After carefully checking the coalition formation process and analyzing each of the failures that occurred in the trials in which MissionControl was applied, we concluded that the observed failures were due to (i) simulation environment failures, which are beyond our control and responsibility, (ii) limitations on the skill implementation, and (iii) limitations of the skill descriptors. The limitations on skill implementation and descriptor are beyond the scope of this work. By out of scope, we mean that the errors are caused by the simulation environment and its interaction with the extensions we build for the experiment, as well as the environment implemented to control each robot. In any case, those failures were not caused by MissionControl, which makes it a rather robust approach.

4.2 Qualitative Evaluation

We recall that key quality attributes of MissionControl as a software architecture is to promote modifiability (i.e. ability of the architecture to be modified to suit a new application) and integrability (i.e. ability of the architecture to integrate with existing and independently developed systems). Therefore, we formulated a second evaluation goal to check how the architecture adds to these quality attributes.

4.2.1 Evaluation Goal 2

Goal 2: Evaluate the modifiability and integrability of MissionControl.

- *Question 1)* How modifiable is the architecture?
- *Question 2)* How integrable is the architecture?

To perform the evaluation of both modifiability and integrability, we first perform a "tactic-based" evaluation inspired by [10] and, then, a "guideline-based" evaluation inspired by [19].

The process of the "tactic-based" evaluation consists of reviewing the available artifacts (i.e., architecture documentation and the code base of the implementation in our case) through a questionnaire that focuses on a single quality attribute at a time. The questionnaire contains items that query the application of architectural tactics to support the quality attribute. For each item, (i) we register whether our design decisions (DD.1 to DD.7 in Chap. 3) satisfy a specific tactic, (ii) we register the location in the architecture where the tactic can be found (i.e., in which module), and finally, (iii) we describe the rationale and assumptions. While inspecting the code base as preparation for evaluation, in many cases, we identified improvements to be realized. In cases where the modifications were feasible and within the scope of this work, we implemented the improvements, and then a new evaluation iteration was realized. These improvements were mostly clarifications on the documentation and refactoring on the code base regarding re-organization of the code units, but not including functionality (e.g., renaming and moving implemented units). Furthermore, while evaluating the architecture from the modifiability and integrability point of view, some tactics overlapped. In particular, *reduce coupling* for modifiability is closely connected to *limit dependencies* for integrability. To avoid duplication, while evaluating modifiability, we focused on impact changes in the MissionControl runtime environment, and while evaluating integrability, we focused on the MissionControl component model, that is, the interfaces and the support for integration of skill descriptor, environment descriptor, and skill implementation components.

For what concerns the "guideline-based" evaluation, we performed an evaluation that is more specific to the robotic domain. Specifically, we evaluated the various modules that execute the mission in relation to compliance with the guidelines described in [19]. The set of 49 guidelines contains foundations for juxtaposing design decisions in robotic software within traditional software quality attributes (e.g., reliability, maintainability, safety) from the ISO/IEC 25010 standard (ISO/IEC, 2010)[2]. Our proposed architecture is concerned with leveraging modifiability (a sub-characteristic of maintainability in ISO/IEC 25010) and integrability (similar enough to portability in ISO/IEC 25010) as key quality attributes for multirobot management software. Therefore, for what concerns the evaluation based on guidelines, we analyze the compliance of our architectural decisions with the empirical guidelines through the lenses of *maintainability* and *portability*.

The analysis process follows an in-house inspection of the architectural decisions against the set of good practices. Three co-authors designed and executed the inspection[3] [18]. We attribute the roles of architect analyst to two co-authors who participated intensively in coding, and external inspector to a co-author external to the implementation but aware of the high-level architectural decisions. The inspector was supposed to go through each and every guideline to verify its compliance with the guidelines. Checking compliance here stands for querying the architect analysts whether the guideline is relevant and whether and how the guideline is reflected in the implementation, asking for examples when possible.

In the remainder of this section, we report the outcome of the evaluations based on "tactic-based" and "guideline-based" evaluations by answering Questions 1 and 2.

4.2.1.1 *Question 1)* How modifiable is the architecture?

Tactic-based evaluation: The questionnaire for the modifiability quality attribute has 7 evaluation elements grouped in three tactic groups: (i) increase cohesion, (ii) reduce coupling, and (iii) delay binding. After some refactoring interactions, all the questions were satisfied. The following is a brief discussion of each tactic group.

The first tactic group, i.e., increase cohesion, has the intent to reduce the probability that a change request requires changes in multiple modules. The part of the questionnaire devoted to evaluating this first tactic group consists of two questions that focus on the cohesion of software modules. We evaluated these two questions for each module of the MissionControl module (cf. Figure 3.6). The most related design decisions are DD.1-DD.3, DD.5, and DD.6; they help to split the responsibilities between components.

[2]ISO/IEC 25010:2011 Systems and software engineering Systems and software Quality Requirements and Evaluation (SQuaRE) System and software quality models.

[3]All the process and decisions are further documented in our replication package

The second tactic group, reduce coupling, has four items that are focused on increasing encapsulation and abstraction. The most related design decisions to this tactic group are DD.2, DD.5 and DD.6. The use of dependency injection for the wiring of the components (Section 3.7) was also important to avoid components relying on the specific implementation of others.

Finally, the defer binding tactic group section of the questionnaire has only one item that assesses whether the binding is systematically deferred in the system, so that a functionality can be replaced last in life cycle. The most related design decisions are DD.2, DD.5, and DD.6. Binding is threefold deferred in MissionControl:

- through the use of service interface for mission decomposition, so missions can be adapted without changing MissionControl,
- through the use of ensembles that allows for dynamic binding between the coordinator and the robots at runtime, and
- through the use of a component model, which permits to (i) add skills and environment descriptors, and (ii) bind the components during initialization thanks to the skill implementations; this allows these components to be changed at runtime without modifying MissionControl.

In summary, after some refactoring, the code base satisfies all the items in the questionnaire. Details about the analysis are available in our online appendix [18].

Guideline-based evaluation: When it comes to analyzing to what extent MissionControl promotes modifiability through the lens of software architecting practices in robotics, we highlight three[4] guidelines:

M1. *"When possible, core algorithms, libraries, and other generic software components should be ROS-agnostic;"*

M2. *"Group nodes and interfaces into cohesive sets, each of them with its own responsibilities and well-defined dependencies"*

M3. *"Decouple ROS nodes from variations in the execution environment"*

As for M1D. ROS-agnostic core algorithms, design decision 6 (aka DD.6) on defining MissionControl as a *skill-based system* induced our architects to implement a single interface between mission management and ROS-based skill management (e.g. SkiROS [66]).

[4]Three (3) out of 16 guidelines concern modifiability. Among those 16, the M1-M3 guidelines were considered the most useful for robotics professionals working with ROS-based applications.

The single interface grants ROS-agnostic mission coordinators with benefits for further modifications. It also complies with DD.7 on the independent implementation of mission coordination.

Moreover, regarding M2, DD.6 breaks robotic mission management into coordination (i.e., coalition formation) and execution. Thus, it favors MissionControl users in clustering the coalition formation process and its dependencies (i.e., mission, skill, and environment information) separately from the sequencing of actions, skill activation, and respective dependencies (i.e., sensors and actuators) as shown in Figs. 3.3 and 3.2. The separation of concerns leveraged by DD.6 results in the possibility of independent extensions to the coalition formation and skill implementations.

Regarding M3, that is, decoupling ROS nodes from variations in the environment, it emerges as the effect of grounding MissionControl in DD.5, on *coordination extension points*. The environment descriptors feed the coordination layer with application-specific knowledge, resulting in more precise and domain-dependent coalition formation. This feature alleviates MissionControl's users from encoding and modifying environmental information in the ROS nodes that are responsible for skill behavior.

In addition to the three guidelines reported above, we have evaluated how MissionControl's design decisions stand against 16 guidelines for architecting modifiable robotic software. MissionControl supports or favors a set of 9 guidelines for users implementing heterogeneous multi-robot applications. One guideline is relevant to modifiability, but not to multi-robot applications. Despite their relevance to MRS, four guidelines are not relevant to mission coordination, for example *"Keep number of nodes as low as possible [...]."* and *"Use ROS standard messaging protocols [...]."*. Finally, two guidelines refer to the persistence of the data. They deemed relevant to the multi-robot domain and to mission coordination, but they were not followed in MissionControl. For long-term and short-term persistence of data, MissionControl employs in-memory mission-, skill-, and environment descriptors in opposition to a centralized ROS node for data management, as suggested in the guidelines.

The complete list of modifiability guidelines and analysis is available in our online Appendix [18].

4.2.1.2 *Question 2)* How integrable is the architecture?

Tactic-based evaluation: The questionnaire for the quality attribute of integrability has a total of ten items, organized into three tactic groups: (i) *limit dependencies*, (ii) *adapt*, and (iii) *coordinate*. An overview of the evaluation for each tactic group follows.

The *limit dependencies* tactic group has five questions and evaluates the use of tactics such as encapsulation and the use of an intermediary to limit the number of potential

dependencies between components of the architecture. An architecture that limits the number of dependencies can reduce the impact needed to integrate a new component, thus favoring integrability. In MissionControl, we provide clear extension points with well-defined interfaces and fewer dependencies between the architecture and the extending components. Design decisions that address the limit decision group are DD.2, DD.3, DD.5, and DD.6. All items queried in the limit dependencies tactic group are satisfactorily supported in MissionControl.

The *adapt* tactic group evaluates the ability of the system to change the interface of the components to make it integrable in the architecture. This is achieved by applying tactics related to component configurability, service discovery (both supported by MissionControl), and interface tailoring (which we only partially support). The tailoring interface item refers to adding and hiding interfaces statically (i.e., at compile time). Although in the MissionControl there is no mechanism for tailoring interfaces at compile-time, we believe that this could be achieved in our architecture by using the adapter design pattern, with minimal impact on our architecture due to the use of dependency injection. However, we are not certain that it would suffice in every scenario; therefore, we judge this item as partially supported. For this group, 2 out of 3 questions were answered as fully supported, while one was partially supported. The most relevant design decisions that address this tactic group are DD.2, DD.3, DD.5, and DD.6.

Finally, the *coordinate* tactic group is assessed using two questions. The first question is related to the use of orchestration to manage components. MissionControl applies such a tactic in the estimating module to call the descriptors, and in the sequencing process to call the skill implementation in the required mission order. The orchestration mechanism permits independence between components that are related to different skills; this is realized through an external orchestration mechanism that calls the different skills. The second question in this tactic group is concerned with resource management that governs access to computing resources. This item is partially satisfied as MissionControl realizes the coalition formation taking resources into account. However, resource management is enforced during the execution of tasks by skill implementation. In a future version of MissionControl, it could supervise skill implementations according to available resources at runtime. For example, if the robot is running short on battery, the robot could slow down. The design decisions that address this tactic group are DD.4, DD.5 and DD.6.

In summary, in the tactic-based integrability assessment, MissionControl satisfies 8 of 10 items, and partially satisfies 2 of them. This reflects the use of different tactics in MissionControl to enable and promote the integration of new components into the architecture. Among those items that are only partially satisfied, resource management is planned for future work. Details about the analysis are available in our online ap-

pendix [18].

Guideline-based evaluation: The validation team analyzed 16 guidelines that promote integrability for software architecture design in robotics, with remarks to two supported guidelines:

I1. *"Identify variation points of the system in advance, and design the system so that it can be extended by third-party users without modifying its core nodes'*

I2. *"ROS nodes should be agnostic of the underlying communication mechanisms (e.g., network protocols, deployment topology)."*

MissionControl provides interfaces for external mission planners (DD.3), reuse in different environments (DD.5), and skill-based mission management (DD.6). Such decisions have an impact on the integration of planners, environment specifications, and skill implementations. Referring to guideline I1, identifying the variation points in advance was a fundamental building block of our architecture. This is highlighted by the purple subsystems in Fig. 3.3, i.e. *Skill Descriptors*, *Environment Description Services*, and *Library of Skill Implementations*.

MissionControl favors ROS nodes decoupled from the underlying communication mechanisms (I2.). This is a result of the design decision 2 (DD.2) on the ensemble-based architectural pattern. To this extent, DEECo [13] plays an important role. The skill implementations are independent of the topology or address of other robots, enabling runtime binding between peers. Therefore, integrating new skill implementations at the ROS level is freed from the underlying communication mechanisms between robot peers.

In summary, 16 guidelines were analysed against MissionControl's design decisions for architecting integrable robotic software. MissionControl supports or favors 7 guidelines. The other 9 guidelines are not relevant to MissionControl's scope since they are concerned with what and how skills should be designed on top of ROS nodes. This is not a primary concern of MissionControl, given DD.1. (i.e., *separation of responsibility between coordinator and robots*). From the seven (7) favored guidelines, six guidelines were deemed useful by practitioners [19] and all of them are present in MissionControl. The last guideline is concerned with long- and short-term data persistence. The complete list of integrability guidelines and analysis is available in our online appendix [18].

4.3 Threats to Validity

Construct Validity. In-house evaluation of architectural decisions poses threats to validity. Given the unique nature of our proposal, we decided to design a rigorous pro-

cess, which was revised and approved by all co-authors, including experts in software architecture research. Moreover, we make available in a replication package a detailed documentation of questions, working documents, conversations, processes, and results for peer review.

Reliability To build confidence that our architecture promotes autonomous mission coordination and execution, we offer the instance of MissionControl used in our evaluation as a publicly available artifact[5]. The implementation comes with a suite of passing tests asserting that our proposed architectural decisions are followed in the implementation through varying scenarios[6]. In the public repository, we provide automated tooling for executing the experiments, collecting data, and assistance for parsing and analyzing the collected data.

Internal validity We account for the efficiency of the MissionControl in terms of successful mission executions against a baseline algorithm. To this extent, successful missions are measured by whether all tasks assigned to individual robots are performed correctly. Although robot teams may vary in capabilities, they share a repository of navigation.

External validity The trials' configurations include only one mission specification in one simulated environment. Although MissionControl is designed to be independent of the semantics of the input model, given the separation of concerns, the generalization of our approach is impaired by the lack of experimentation with different missions in other environments, which would raise other types of uncertainties. Also, with only 10 implemented skills, including one human-robot collaboration and one robot-robot, a more comprehensive evaluation with other types of collaboration is left to future work.

[5] https://github.com/lesunb/hmrs_mission_control
[6] https://github.com/lesunb/hrms_mission_control_evaluation

Chapter 5

Lightweight simulation for Mission Coordination

Tests in simulated environments, rather than in real-world physical environments, can significantly reduce the cost of testing and increase the opportunities for test automation [67]. And it is known that many bugs in real-world systems could be found in simulation [68, 69]. However, a recent study has revealed that while many practitioners and participants are aware of the theoretical advantages of incorporating simulation into their engineering processes, few of them systematically utilize simulation for testing purposes[67]. Several factors contribute to this under-utilization. Firstly, the computational cost of executing scenarios can be significant, posing a challenge for practical implementation. Secondly, the absence of suitable tools to run automated tests within simulated environments further hinders its widespread adoption. [67]

Most existing simulators can be categorized into two classes: physical simulators and lightweight high-level simulators. Physical simulators, such as Gazebo [44], CoppeliaSim [46], MORSE [47], MuJoCo [70] and MVSim [71] are built upon physics engines. These simulators model the sensor readings and actuator effects of each robot. Physical simulators are suitable for developing control nodes due to the high fidelity of the simulation. They offer the advantage of executing code very similarly to what a physical robot would run, providing a smooth transition between simulation and real robot, and allowing for testing the same code that will be deployed in the system facing the user. However, physical simulators come with a drawback: they are computationally intensive and difficult to set up. Moreover, due to the computational intensity of the simulations, it is extremely challenging to work with scenarios involving multiple robots in a large-scale environment, such as an entire hospital building. Compared to physical simulators, lightweight high-level simulators, such as Simbad [45], Dragonfly [48], and Robocode [72], are less computationally intensive and difficult to set up. They often abstract away significant

portions of the underlying physics, simplifying the simulation and enabling faster execution, especially in complex or large-scale scenarios. However, this simplification comes at the cost that the same software that runs in the simulator cannot be easily transitioned to a physical robot. Consequently, these simulators serve as disposable prototyping tools. The downside of a disposable approach is that bugs can be introduced while writing the code that will execute on the physical robot, thereby partially negating the advantage of using simulation as a test tool.

Although physical simulators facilitate a seamless transition from simulation to physical robots, and high-level simulators offer lightweight simulation capabilities, there exists a gap in the literature that seeks to strike a balance between computational intensity and ease of migration to physical robots. To address this gap, we propose a lightweight simulation approach tailored for testing and experimentation in multi-robot mission coordination. This simulator efficiently handles lightweight simulation while preserving the same interfaces that coordination logic encounters in a physical robot.

Our lightweight simulation approach aims to achieve the following objectives:

(i) Enable lightweight execution that supports dozens of robots in the same scene. (ii) Allow the code controlling the robot to be deployed to real robots with minimal modifications. (iii) Provide integrated tooling for writing automated test scenarios.

However, it is imperative to emphasize that we do not intend to replace the use of physical simulators in the engineering of robotic systems. Physical simulators cannot be replaced in the development of low-level control algorithms that enable the robot to execute actions. With lightweight simulation, our aim is to provide an additional tool tailored for testing higher-level coordination aspects. The remainder of this chapter will dive into the core components of lightweight simulation, including its entity management system, event handling, and scenario execution pipeline.

5.1 Concerns in Lightweight Simulation

In engineering mission coordination, many concerns are involved. For simplicity, we consider five layers of concerns as illustrated in Figure 5.1: coordination, sequencing, skill, robotic actions, and environment. Coordination is responsible for assigning missions to a robot, sequencing is responsible for interpreting the mission plan and choosing the tasks to execute, and the skill is responsible for executing a task while relying on robotic actions to sense the environment and actuate on it. Robotic actions abstract the capabilities of the robot through control loops that operationalize these capabilities by reading signals from sensors and generating signals to control the actuators. Finally, the environment

is the environment of the robot, in which it does not have direct control, but can sense through sensors and act upon through actuators.

The distinction between robotic actions and skills may not be obvious at first glance, but it is important in this chapter. Skills are built on top of robotic actions to perform abstract tasks. The realization of an abstract task may require the execution of many actions and decisions. For example, a navigation task may use a hospital map to look at a route to reach a given room in a hospital. During the navigation route, the robot should reach the intermediate Points of Interest (POI). Also, as part of the navigation skill, the robot may need to interact with elevators and automated doors and coordinate with other robots to queue to access a resource. Although a navigation skill may use robotic actions as a building block, the skill may evolve into more than a single action.

In a low-level simulator like Gazebo, the most common setups involve running a physical simulation engine alongside plugins that simulate signals from sensors and the physical effects of actuators. For example, consider the robot navigation subsystem, which enables a mobile robot to reach various locations within the vicinity. Navigation is a crucial robotic skill in various service robot scenarios. In a low-level simulator, the simulator generates readings from various sensors, such as motor encoders and LiDAR sensors. These readings are then processed by control loops that generate signals to control actuators, such as motors. In addition, the simulator creates a rotating wheel that interacts with the floor surface, resulting in movement. These low-level details are essential when exercising the Control Nodes that control the actuators, allowing the robot to execute actions like reaching a destination. For example, Nav2 is the reusable package that contains ROS control nodes and provides navigation to mobile robots. Nav2 provides an ROS action server that receives destinations and, through a control loop, guides the robot to reach that destination. Whether in a physical robot or a simulated robot in a physical simulator, Nav2 can be used to handle robot navigation. However, in our experience, this level of detail is not ideal when engineering coordination logic, the physical simulation is computing intensive, and also requires a lot of setup in the simulator, as we need to set, for example, the materials of the surfaces.

During engineering and developing the coordination level of the system, we need to simulate multiple robots in a vast environment over an extended period. In such large scenarios, we believe that simulating the scene at the physics level is not feasible. Therefore, we propose the development of lightweight simulations (the rightmost column in Figure 5.1). These lightweight simulations have two primary design goals: first, to enable the execution of scenarios with minimal computing resources, and second, to allow the coordination components to be portable from the simulation to a physical robot. To achieve a lightweight simulation in terms of computing resources, we abstract the robotics

actions at a high level. For example, in navigation, we simulate the robot moving on a map without the low-level details required by a physical simulator. The implementation of this system draws inspiration from the mechanics implementations in video games. Instead of simulating the intricate details of wheels, motors, and the physics of a moving robot, we focus on simulating the robot's movement from the perspective of an external observer.

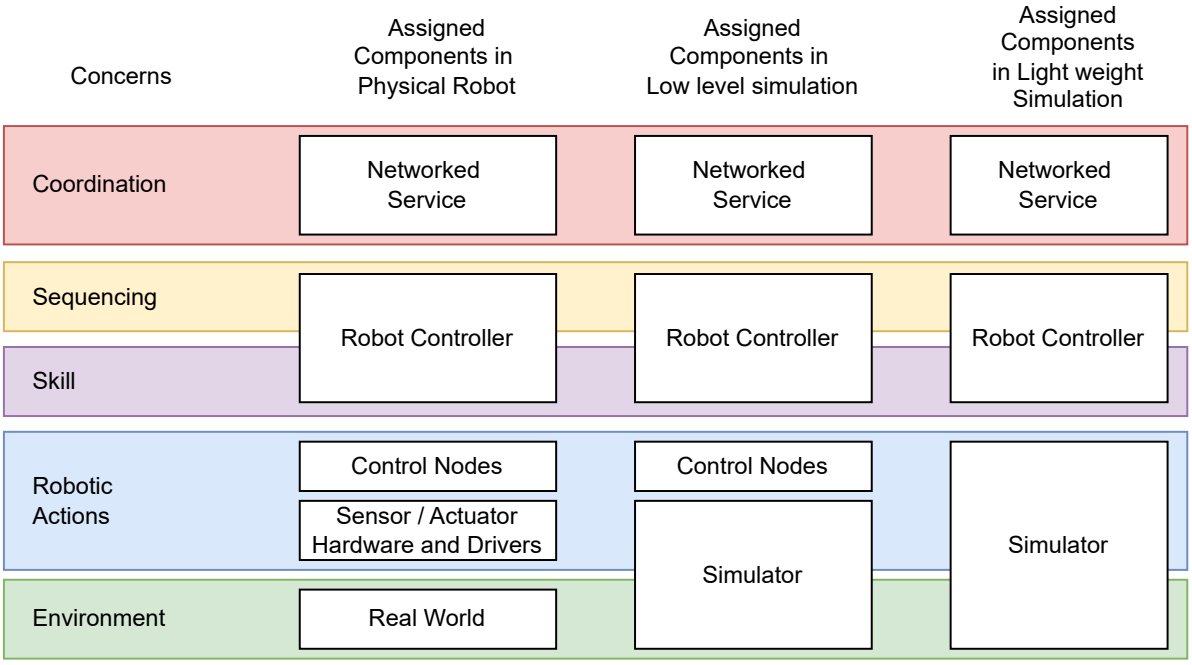


Figure 5.1: Concerns of simulation in robotics from an architectural viewpoint.

(DD.1) Lightweight simulation Simulation systems simulate robotic actions enough to exercise skills and abstract away low-level physical details if possible.

With *lightweight simulation*, we aim to test coordination logic in a great number of scenarios and in a large environment with many robots and other actors like humans, with modest computational infrastructure. For coordination, the implementation details of the system's low-level control are not as relevant, with their skills and capabilities being more important. For example, assuming that a robot has the ability to pick up objects of up to 'x' kg that are within a distance of up to 'y' meters, the way this operation is performed and the actual configuration of the mechanism that enables it are completely abstracted, and only the effects of this capability are simulated.

We argue that while engineering multi-robot mission coordination, the software engineers will be most concerned with the top three layers: coordination, sequencing, and skills. However, to fully exercise these layers, we need to interact with the robotic actions

layer. In that way, the main job of the simulator is to provide an abstraction of the robotic actions layer enough to exercise and test the functionalities concerning the level of the top three layers.

(DD.2) Portability of coordination logic The simulator provides interfaces that mimic control nodes that a skill would interact with in a physical robot.

In our work, achieving a lightweight simulation should not compromise the portability of components from simulation to physical robots. To this end, we propose mocking the interface between the robot controlling logic and the simulator using the same interfaces that a skill would interface with controlling nodes, such as Nav2. This portability is inherent in physical simulators, as they execute the same control nodes as are executed in physical robots. However, to the best of our knowledge, this portability is not yet possible in existing lightweight simulators.

5.2 Architecture for Lightweight Simulation

As previously stated, our main goals while proposing a lightweight simulation tailored for mission coordination were to contribute a simulation that was both computationally light and portable to allow the software components to be used in physical robots, either without or with minimum changes. Moreover, while designing the architecture, we defined two additional goals: extensibility and testability.

By extensibility in simulation, we want to easily incorporate new simulation aspects, like new robotic actions and external actors like humans or automated entities like elevators and automated doors. Theoretically, in a physical simulator, one can easily add new types of robots just by modeling the new robot in the simulation. For example, a new legged robot can be added by adding the mechanics of the leg and the motors. However, following a lightweight approach, to include a legged robot, one would need to create an extension in the simulator, implementing a new abstract system to simulate how a legged robot moves. Because of this, extensibility is also a goal in our approach. To ensure *extensibility*, we adopted an ECS architecture. This allows for the addition of new simulation aspects by incorporating new systems and components into entities (as introduced in Section 2.5.2).

(DD.3) Extensibility of the simulation To achieve extensibility, we adopted an ECS architecture. This architecture enables a simulation to be easily extended by adding new components and systems. In addition, ECS facilitates the creation of specialized simulations by selectively choosing systems.

5.2.1 Entity-Component System

Given that the simulator architecture is based on ECS, the concepts of entity, component, and system are fundamental within HMR Sim. Both the objects within a simulation and the simulation itself are constructed through composition: objects composed of different components, and the simulation through systems and configuration.

Entities in the simulation are generally related to physical objects. An entity can represent, for example, a robot, a wall, an independent sensor, etc. Within the simulation, entities have a unique identifier. This identifier associates the entity with a set of components that belong to it. At the beginning of the simulation, an entity typically has a specific set of components that can be modified throughout the simulation. Entities can also be created or removed during the simulation.

Components store data about the entities to which they belong. For example, the presence of a Velocity component indicates the ability to move and stores the entity's current velocity; the presence of a Camera component indicates that the entity has a camera sensor and can store the last recorded image, etc. Components can be defined by the user as classes and are loaded automatically using an appropriate naming system. Section 5.3.1 covers the creation of components and how they become available in the simulator.

It is important to note that any ability associated with a given component is defined by the system that acts on that component, so that with the same component, it is possible to assign different behaviors to an entity. For example, if a robot has a Camera component, depending on the system used, this camera can represent a normal camera, an infrared camera, a night vision camera, etc. A consequence of this fact is that the commitment to reality (within what is acceptable in the simulation) is largely the responsibility of the systems used in the simulation. The user must employ systems in the simulation that guarantee the degree of realism expected by them in the behaviors they implement.

5.3 Implementation: the HMR Sim

To validate the concept of lightweight simulation, we developed a prototype named HMR Sim. It is implemented in Python. The HMR Sim implements the ECS architecture

and provides components and systems that can be used to implement some scenarios and as examples for extensions of the simulator for scenarios not covered.

Moreover, to achieve lightweight simulation, HMR Sim employs the discrete event simulation (DES) technique using the SimPy library. This framework leverages processes for event management and execution. The simulation environment facilitates interactions between multiple processes and the environment through events.

Python was chosen because of its widespread adoption and readability, which facilitated collaboration on the project. Python is also well supported in ROS2. However, the execution speed is not as fast as it would be if developed in a compiled performance language such as C++, which is also well supported in ROS. Nevertheless, the perceived readability advantages of Python outweigh this limitation for a prototype.

The tests are implemented in pytest[73] and pytest-bdd[74] using our own infrastructure to test robotic scenarios. The tests are dual-folded in HMR Sim. On the one hand, tests were used during development to test the simulator itself, and can be used to test use case-specific extensions to the simulator. The tests can also be used to test coordination components intended for use in user-facing applications interacting with physical robots.

5.3.1 Entities and Components

HMR Sim utilizes the `esper` library [75] to manage entities and their components within the simulation. Each entity in the map or the entities that pass through the configuration is represented by an integer, stored within the `World` class of `esper`, and can be accessed through the simulation instance in the `world` attribute, as well as by the systems. Entities can also be added directly to the simulation using the *Simulator.add_entity* method.

Entities possess a set of components. Components can be added or removed from entities using methods from the `esper` library. There are also methods to verify the existence of a component in an entity, search for specific components of entities, search for all components of a certain type in the `World`, etc. Components are simply Python classes that must have the same name as the file that declares them (one per file). In the ECS pattern, it is conventional that the components do not have implemented logic, and it is suggested that only the *__init__* and *__str__* methods are implemented in a component. `@dataclass` can be used.

The identifier of a component is the name of its class. Thus, if the component `simulator.components.MyComponent` is declared, to use it in a system, it can be imported normally (e.g. `from simulator.components.MyComponent import MyComponent`) and used this import when referring to the component (searching for it in the `World`, for example).

Systems are constructed as Python functions and can be processes of the simpy library or functions accepted as systems by Esper (see Sections 2.5.1 and 2.5.2 for details on the libraries). The systems must be initialized and added to the simulator before the execution phase.

A simulation can be programmed through a configuration object (Python dictionary) or a mixture of the two options. Maps are XML files constructed with the JGraph library [76], with some restrictions. Any program compatible with this library can be used, such as the online and open-source editor `diagrams.net` [77]. It's also possible to create entities in the simulation through EntityDefinition (see project documentation).

5.3.2 Systems

Systems are an essential part of the simulator, as they are responsible for driving the simulation forward. Systems are also very versatile, and can be used not only to represent a part of the simulation but also to extract information from it. There are two different types of systems that can be used: a system compatible with systems from the `esper` library, referred to as esper systems or step systems, and systems compatible with the `simpy` library, referred to as DES systems. Both are optional, and the choice of which to use depends on what is being simulated.

The central concept of HMR Sim is that systems are built to utilize a set of components and represent a part of the simulation. Each set of related systems (and their components) confers new capabilities to the simulator and therefore to the simulation being performed. For example, systems can represent sensors, movement, communication between robots, etc. Much of the flexibility that the simulator provides lies in the relative ease of creating and using systems. If a system is not suitable for your needs, it can simply be replaced by another or modified, without having to alter other parts of the simulation. For example, if the goal of a simulation is to compare two robot management algorithms, two different systems (that use the same components) will be created, one for each algorithm. Testing algorithm A or B becomes a simple matter of adding system A or B to the simulation.

5.3.3 Interface with coordination layers

To achieve the portability goal in HMR Sim by (i) the runtime of the controlling logic is independent of the simulator runtime, allowing the designer to use the same tools he uses while building the components for a physical robot, and (ii) the interface between the robot controlling software and the HMR Sim is done through ROS.

As of today, the ROS middleware is the most popular middleware for developing control software. As briefly introduced in Section 2.3, ROS provides mechanisms for

developing software for controlling robots as a graph of interdependent nodes, which are software components that execute in their own control loop and can communicate with other nodes through a publish-subscribe mechanism. When developing software for a robot, we typically have some nodes that subscribe to topics that are fed by sensors, and nodes that publish to topics that abstract actions of a robot that are subscribed to by the drivers or lower-level control components.

In HMR Sim, we do not make assumptions about the runtime environment of the software controlling the robots. By using ROS as an interface, the software controlling the robot runs in a separate process from the simulator and can even run on different machines or, more practically, in Docker containers. We advise using Docker containers to run the robot environment, although nothing in the simulator requires it. By using a container, the engineers can produce an insulated environment similar to what will be used in a real robot.

5.3.4 Sensors and Actuators

Systems representing sensors and actuators confer different abilities to entities in the simulation. Actuators can abstract entire components of the robot (e.g., grippers, platforms, tweezers, etc.) and can be modulated through reactive DES systems, which react to commands. Sensors, on the other hand, due to their more constant behavior, can also be approximated by Esper systems. The construction of these systems is done according to the requirements of the robot's abilities; therefore, the implemented and tested systems are more like proofs of concept.

A generic sensor system has been developed (`SensorSystem`). It can be initialized with a frequency and a sensor type (a component) that has an area of operation (`sensor_range`) and a channel to receive events (`reply_channel`, a `simpy.Store`). The idea is that for each type of sensor within the simulation, an instance of this system would be added to the simulation environment and would be responsible for capturing all entities within the sensor's area and sending them via an event to the sensor of each entity through the sensor's channel. This abstracts the data capture phase. An extra system that implements the sensor itself, processes the data, and sends information still needs to be built.

Actuators are more specific. The `hospital_scenario` simulation implements a gripper actuator, capable of picking up interactive objects and dropping them somewhere else. Objects are marked as interactive with an annotation on the map. To assist in the process of picking up and dropping objects, which in this simulation effectively removes the interactive entity from the simulation and then reconstructs it in another location on the map, the object management system can be used (`ManageObjects`). The gripper

system was also used to test the creation of instructions for the control system, exporting two instructions.

5.4 Navigation System

In HMR Sim we want the simulation to be lightweight but still realistic from the point of view of coordination. Roughly speaking, a robot that is able to reach positions in a simulated environment, if an analogous robot would do the same in analogous physical characteristics, would be able to do the same in an analogous physical environment. But robots can move in various ways depending on how they are constructed. To better represent the movement capabilities of a robot, we implemented systems that model specific kinematic constraints. For example, an omnidirectional robot is one that can move freely in any direction, including sideways and diagonally, without needing to reorient itself. This is often achieved through specialized wheel designs like mecanum wheels or spherical wheels. While a differential drive robot, in contrast, moves by varying the rotational speed of two independently driven wheels. This allows it to move forward, backward, and turn by controlling the relative speed and direction of each wheel. It cannot directly move sideways, but it can achieve more complex movements by combining forward motion with turns. Other common robot locomotion systems include: Ackermann steering (similar to a car), legged robots, and aerial robots (drones).

In the HMR Sim, we can support various robot kinematics by implementing a kinematic system and components that move according to that kinematics. Figure 5.2 illustrates a robot simulation with a robot that moves using a differential kinematic. First, a system controlling the robot sends a move command specifying the coordinates of a target point. The Nav2System receives the command and creates an event representing the command in the simulation using a ROS2 node. Then, Nav2System inserts the command into the simulation. In each simulation tick, the navigation systems for each specific kinematic process the entities that contain their related components. In the example shown in Figure 5.2, the robot is a differential base, so it is processed by DifferentialBaseKinematicSystem. This system updates the robots position. In the case of DifferentialBaseKinematicSystem, the kinematics are implemented by a PIDController.

5.4.1 Rendering

Rendering is the process of creating a visual representation of a simulated scenario on a screen. Rendering a simulation is very useful, especially when constructing a simulation, as it allows for instant feedback for developers. However, rendering can slow down a simulation as (i) it adds up computational cost, (ii) adds additional complexity as we

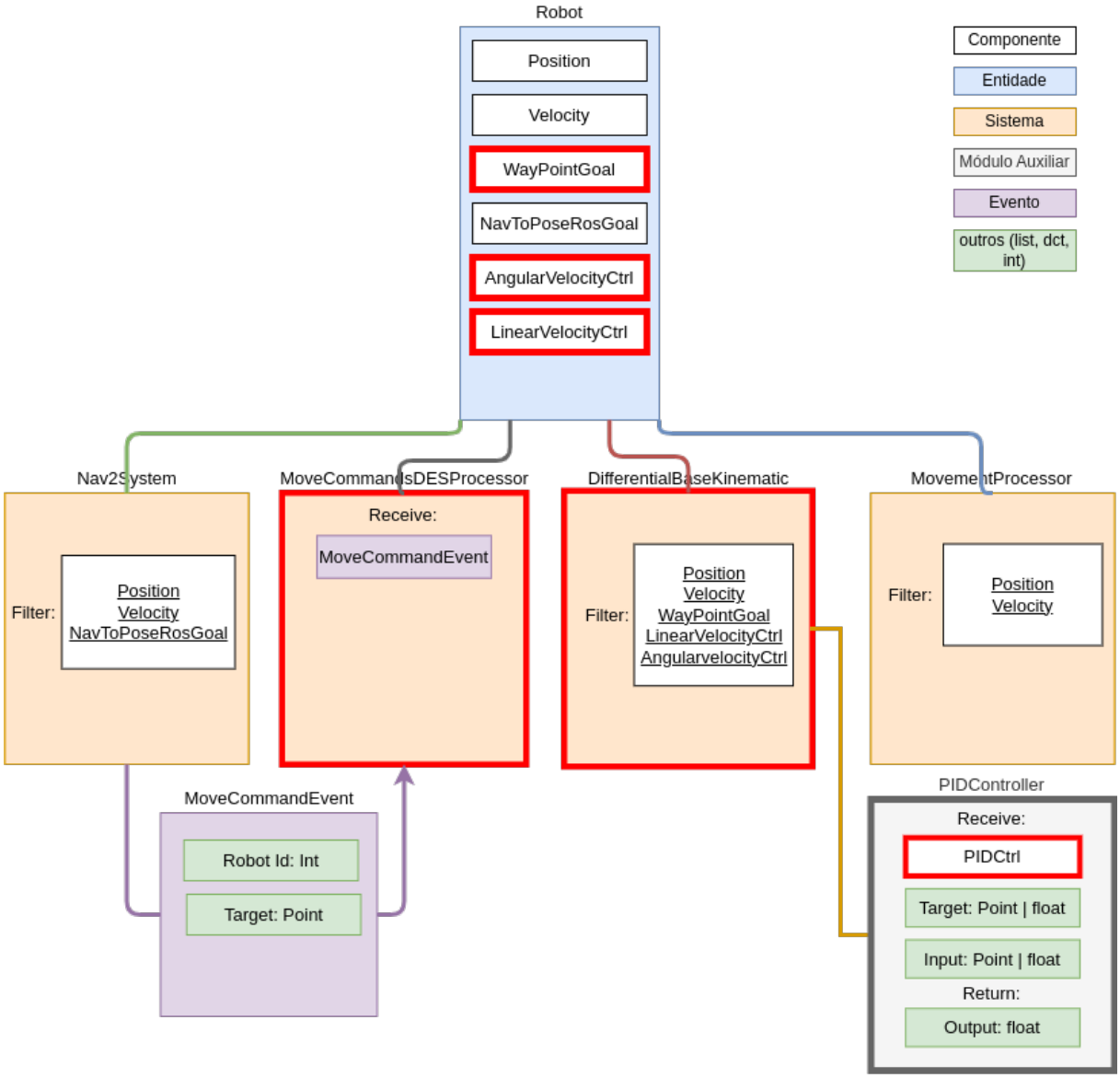


Figure 5.2: Navigation Mechanisms

need to add a visual representation of the elements in the scene, and (iii) it requires us to execute a scene at a speed that is intelligible to humans. Due to these drawbacks, we decided as a design goal to make rendering optional in HMR Sim, allowing developers to use rendering during the development of a simulation scenario and disabling it while executing tests. This allows us to benefit from rendering when it is useful, and also to disable it to allow fast execution of simulations.

Optional rendering can be achieved in an ECS architecture by making rendering a system that can be added to a simulation when rendering is wanted. To render a simulation, we create a real-time representation of the scene as it develops, by rendering on the screen a sequence of frames that represent the simulation, in which each frame is a

snapshot of the scene at a given moment. To do so, the rendering system must, in each step (or with some other frequency) of the simulation, introspect into the components of the scene and get the properties that are of interest for the visual representation, for example, the position of a robot in a map, and then transform this representation into a representation in a visual scene.

5.5 Test in HMR Sim

Unit tests can be used to test units in software controlling the robot, similarly to any other software. Although ROS provides capabilities to test nodes, integration tests, especially involving multiple robots, can be challenging to execute. To perform tests with HMR Sim, it was proposed to use the principles of BDD (Behavior Driven Development). In this project, BDD involves describing the behavior of robotic systems through basic scenarios. Then, these descriptions will guide the development of these systems and serve as documentation for future developers of the project, avoiding the loss of important information. Finally, it will be validated whether all behaviors described in the scenarios are working correctly through acceptance tests. This evaluation was done using automated tests. The scenarios will serve as test cases, with the rules described by `Then` being the acceptance criteria.

(DD.4) Testability We enhance testability by creating scenarios decoratively and adding asserting directives to assess that a desired state was achieved. Moreover, it was supported for using the simulation integrated with a test library so it is possible to execute simulation scenarios as test cases.

To support BDD in HMR Sim, we must be able to describe test scenarios and execute them. Tests in the context of the HMR Sim serve two primary purposes: (i) test the robot behavior that the simulation is exercising and (ii) test the inner workings of the systems within the simulator.

The second purpose may come as a surprise, as the primary goal of HMR Sim is to allow one to exercise coordination logic and assert if it behaves as expected. However, to test the coordination logic, we often need to create some simulation infrastructure that will simulate the external systems that will create inputs to the coordination logic. For example, in the Lab Samples Logistic Scenario, we need to simulate a nurse submitting requests and depositing a sample. Moreover, to assert that the system is robust through testing, the engineer may want to exercise the system under exceptional scenarios like the nurse trying to cancel the mission at different stages, or when there is no available

route to the laboratory, or when the robot has dropped its battery level, etc. To create different test scenarios, one often needs a non-trivial simulation infrastructure, and this infrastructure is also bug-prone. In that way, one also needs to test the simulation infrastructure to establish that the basic functionalities of the simulator are working. This includes checking if it is possible to instantiate a simulation, configure its scenario (add systems, components, entities, etc.), execute it, and obtain the expected result.

To implement automated tests of robot behavior, BDD scenarios will involve only the description of the behavior of a robotic mission through basic scenario steps and the validation of the behavior of these scenarios in an automated way.

The main components of behavior tests in the HMR Sim are:

- .feature scenarios in Gherkin language - readable descriptions of scenario setups, executed actions, and expected end state.
- test files in Python that operationalize the execution of a specific scenario, creating a binding between the simulation infrastructure and the feature description.
- simulation infrastructure - systems and components that simulate the world, the robot's participation in the world, together with the coordination logic.
- test framework - that eases the process of creating test files.

To illustrate each of these components, we depict a simple scenario where a static robot is in the environment in an initial position. After a while, the robot should stay in the same position. In Listing 5.1 we present a simple .feature file describing this scenario in Gherkin.

```
Feature: Single Static Robot

Scenario: Single static robot
    Given a map with one room
    And a robot with id 'robot' in position '1,1' inside the
    room
    When the robot stay still
    Then the robot with id 'robot' is in '1,1'

Scenario: Multi static robots
    Given a map with one room
    And a robot with id 'robot' in position '1,1' inside the
    room
    And a robot with id 'robot2' in position '25,25' inside
```

```
    the room
    And a robot with id 'robot3' in position '42,42' inside
    the room
    When the robot stay still
    Then the robot with id 'robot' is in '1,1'
    Then the robot with id 'robot2' is in '25,25'
    Then the robot with id 'robot3' is in '42,42'
```

Code Listing 5.1: Feature with simple scneario: a single static robot (‘single_static_robot.feature‘)

And in Listing 5.2 is the content of a test that operationalizes the Gherkin scenario.

```
scenarios('../features/single_static_robot.feature')

@given("a map with one room", target_fixture="simulation")
def map_with_one_room():
    config = {
        "context": "tests/data",
        "map": "one_room_map.drawio",
        "FPS": 60,
        "DLW": 10,
        "duration": 10
    }
    simulation = Simulator(config)
    return simulation

@pytest.fixture
def scenario_helper(simulation):
    return ScenarioCreationHelper(simulation)

@pytest.fixture
def assertion_helper(simulation):
    return AssertionHelper(simulation)

@given(parsers.parse(("a robot with id '{robot}' in position"
                " '{position}' inside the room")))
def a_robot_in_given_position(scenario_helper,
                              robot, position):
    pos = position.replace(' ', '').split(',')
    x = int(pos[0])
```

```
    y = int(pos[1])
    scenario_helper.set_position(robot, x, y)

@when("the robot stay still")
def stay_still(simulation):
    simulation.run()

@then(parsers.parse(("the robot with id '{robot}' is"
                     " in '{position}'")))
def check_the_robot_is_in_position(assertion_helper,
                                   robot, position):
    pos = position.replace(' ', '').split(',')
    pos = (int(pos[0]),int(pos[1]))
    assert assertion_helper.is_in_position(robot, pos)
```

Code Listing 5.2: Test file (‘single_static_robot.py‘

The execution of the scenario is possible due to the HMR Sim core functionality as well as some test infrastructure. The test infrastructure is composed of a set of methods developed to assist in the creation and assertions of scenarios. The test infrastructure comprises two primary classes, `ScenarioCreationHelper` and `AssertionHelper`, which inherit from the `TestHelper` class. The ScenarioCreationHelper helps instantiate the scenario through helper methods and is typically used to instantiate steps of type "given" in a Gherkin-structured scenario. The "when" steps will typically just trigger the simulation execution. While the `AssertionHelper` provides helper methods that allow a developer to create assertions of the entities in the scene. AssertionHelper will typically appear in steps with the “then” keyword.

5.6 Preliminary Results

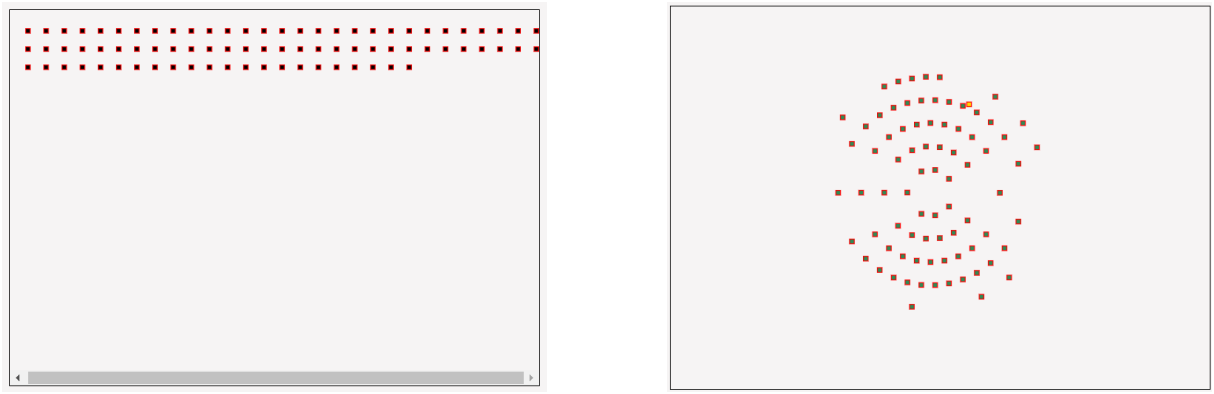
In order to provide evidence of the design decisions implemented in HMR Sim, we illustrate in this section two scenarios where HMR Sim was assessed: (i) in a drone swarm simulation and (2) in a ROS2 navigation scenario. The purpose of this assessment is to evaluate HMR Sim for its performance and for its ability to integrate with ROS2 while preserving its lightweight features.

5.6.1 Scenario 1 - Drone Swarm Simulation

This section describes a drone swarm simulation that tests the performance of a simulator by gradually increasing the number of drones from 20 to 200. In this scenario, we

measure the processing time for each second of the simulation. While this scenario is not directly related to service robots, it serves as a benchmark for performance evaluation, as it involves controlling multiple robots simultaneously.

A controller directs a large number of drones to specific positions within a defined configuration. Each drone attempts to reach its target position without colliding with other drones. Upon reaching its position, the drone informs the controller and maintains its position. If a collision occurs, the drone notifies the controller. The controller waits for a response from all drones or a maximum of 15 seconds before ending the simulation. Figure 5.3 illustrates the initial and final state in an instantiation of this scenario with 80 drones. Figure 5.4 illustrates the systems, entities, components, and events employed in this scenario.



(a) Initial Setting

(b) Final Setting

Figure 5.3: Initial and final setting for a swarm of 80 drones; a drone is represented by a red square

A disturbance system introduces velocity variations to hovering drones, requiring corrective action. Visualizations illustrate the simulation's structure and drone movement, showing the initial and final drone configurations.

Table 5.1 presents a summary of the simulator's performance with varying numbers of drones (*Swarm size*). "Simulation Time" refers to the total simulation time, while *Min. Time 1s* and *Max. Time 1s* represents the shortest and longest clock times required to compute 1 second of simulation, respectively. *Average* indicates the average clock time in seconds needed to compute 1 second of simulation. The variation in the time required to compute one second of simulation is related to the positions of the drones throughout the simulation. In this particular simulation with 80 drones, the simulation time is approximately synchronized with the clock time. All scenarios were executed on the same machine, a Dell OptiPlex 3040 with Intel i5 6th gen, 8 GB of RAM.

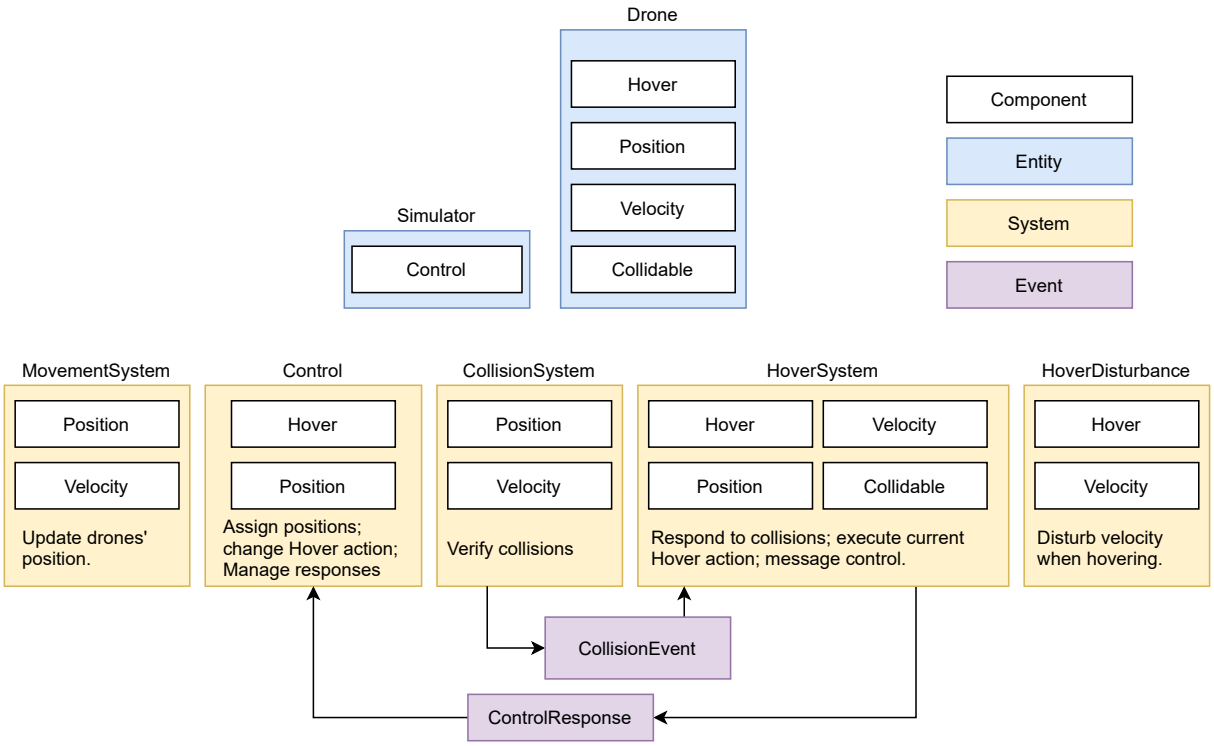


Figure 5.4: Drones Swarm Scenario

5.6.2 Scenario 2 - Follow Path skill with Behavior Tree

Following the MissionControl concept model, we can implement robotic skills on top of robotic actions using Behavior Trees (section 2.2). Thanks to the robotic actions exposed as ROS services, we can leverage off-the-shelf Behavior Trees libraries like Pytrees and BehaviorTree.CPP without any additional support in the simulator. All we need to do is create a Behavior Tree action that encapsulates the robotic action exposed as a ROS service. In this section, we present the *Follow Path* scenario. In the simulation, the robot is a differential base with navigation capabilities to reach waypoints in a 2D room. The

Swarm size	Simulation Time	Min. Time 1s	Max. Time 1s	Average
20	8s	0.0979s	0.1778s	0.1414s
40	10s	0.2700s	0.3969s	0.3458s
60	20s	0.4404s	0.8829s	0.6042s
80	23s	0.6978s	1.3350s	0.9698s
100	22s	0.9491s	2.3663s	1.4146s
150	25s	1.7435s	3.0516s	2.3176s
200	32s	3.1433s	4.4711s	3.7685s

Table 5.1: Simulation performance evaluation as the time required to process 1s of the simulation in the drone swarm scenario

simulator exposes a ROS2 action to move the base that implements the nav2 package interface. The map of this scenario is depicted in Figure 5.5.

As an initial step, to test the ROS2 interface, while the simulator is running, we can send commands through the terminal as illustrated in listing 5.3. The command makes the robot (named *R2D2*) move to the desired position (x:400, y:350 in the listing).

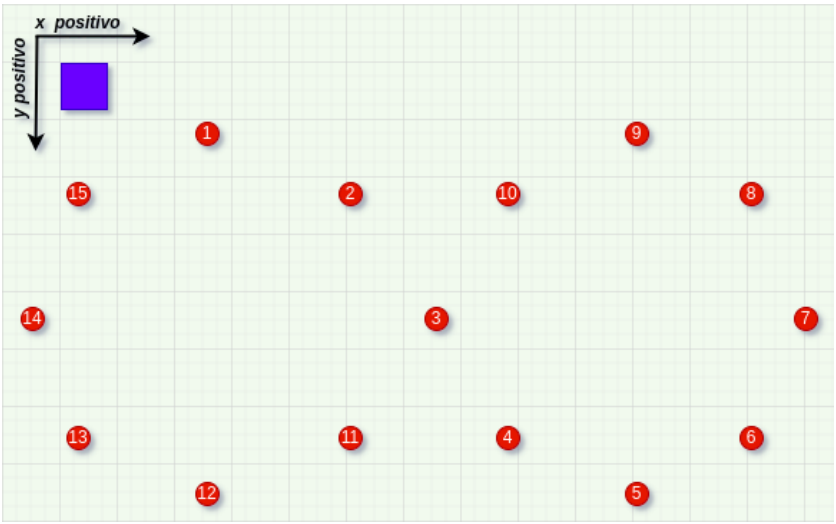


Figure 5.5: Map used in the simulation with waypoints forming a lemniscate

Code Listing 5.3: Sending ROS2 commands to the simulated robot using the host computer terminal.

```
$ ros2 action send_goal navigate_to_pose/R2D2 nav2_msgs/
   ↪ action/NavigateToPose "{pose: {  pose: { position: {x:
   ↪ 400, y: 350, z: 1} } } }" --feedback
```

Moreover, to demonstrate our ability to build a robotic skill on top of a robotic action, we implemented a simple robotic skill named *follow path* that enables the robot to navigate the environment in a specific trajectory, following a path specified as an ordered list of waypoints. This skill is implemented on top of nav2's *navigate_to_pose* action described above. Figure 5.6 is a graphical representation of the behavior tree for *follow path* skill. This BT alternates between selecting the next point to reach and executing the navigation to that point, using the blackboard to communicate the position to reach. The listing 5.4 shows the code that implements the BT using the popular *Py Trees* library.

Code Listing 5.4: Follow Path skill implemented on Py Trees.

```
def create_root() -> py_trees.behaviour.Behaviour:
    way_points = [
      (135.0, 82.0), (235.0, 124.0), (295.0, 211.0),
```

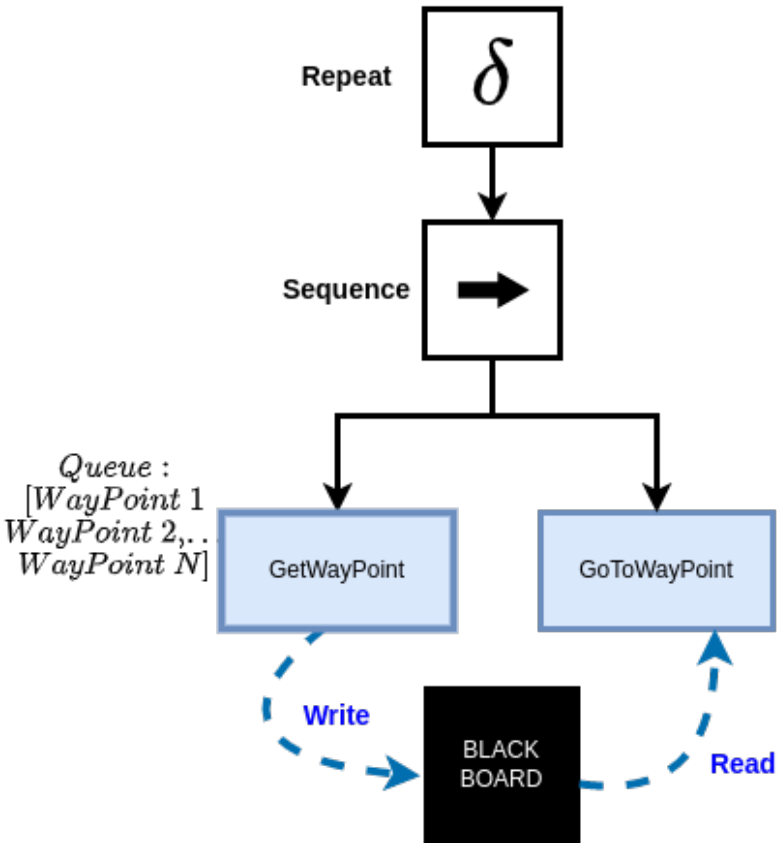


Figure 5.6: Follow Path skill - Graphical representation of the *Behaviour Tree*.

```
 (345.0, 294.0), (435.0, 333.0), (515.0, 294.0),
 (553.0, 211.0), (515.0, 124.0), (435.0, 82.0),
 (345.0, 124.0), (235.0, 294.0), (135.0, 333.0),
 (45.0, 294.0), (13.0, 211.0), (45.0, 124.0)
 ]
sequence = py_trees.composites.Sequence(
                "GoToWayPointSequence")
get_waypoint = GetWayPoint(
                 way_points=way_points, name="GetWayPoint
                    ↪ ")
robot_name = "R2D2"
nav_to_action_client = py_trees_ros.action_clients
  .FromBlackboard(
    action_type=NavigateToPose,
    action_name="navigate_to_pose/" + robot_name,
    name="GoToWayPoint",
```

```
        key="waypoint",
        generate_feedback_message=lambda msg: "remaining {0}"
            .format(msg.feedback.distance_remaining)
    )
sequence.add_children([get_waypoint, nav_to_action_client
    ↪ ])
root = Repeat(
    name="GoToWayPointRepeatDecorator",
    child=sequence,
    num_success=len(way_points)
)
return root
```

Figure 5.7 illustrates the scenario executed, with 15 waypoints that the robot must reach in sequence. The hover robot moves following a differential kinematic model.

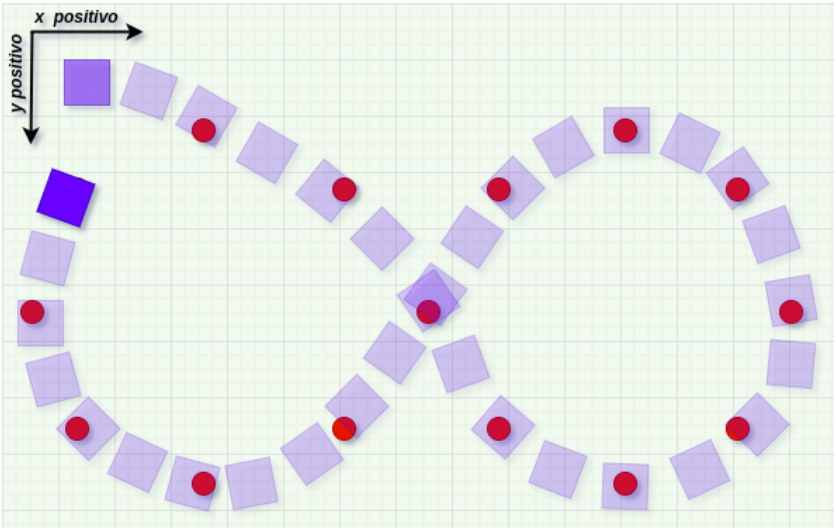


Figure 5.7: Follow Path skill - Result of the simulation.

5.7 Final Remarks

This chapter introduced a lightweight simulation approach designed specifically for testing and experimenting with multi-robot mission coordination. Addressing the limitations of both computationally intensive physics simulators and overly simplified educational tools, our approach offers a balance between fidelity and scalability. By focusing on high-level scenario representation and maintaining consistent interfaces with real robot systems, the simulator enables rapid prototyping and validation of coordination logic

across a wide range of scenarios. The use of Gherkin language for scenario specification further simplifies test creation, while the lightweight architecture supports the simulation of numerous robots simultaneously. While not intended to replace detailed physical simulators, HMR Sim serves as a valuable complementary tool, tailored for testing the coordination logic and allowing for a Behavior Driven development cycle. The ongoing development of HMR Sim aims to provide a robust platform for multi-robot systems engineering, ultimately streamlining the process of developing and validating complex coordination algorithms.

Chapter 6

Related Work

An architecture for mission coordination of heterogeneous robots, Full article published in Journal of Systems and Software, Volume 191, 2022.

Architectures for Coordinating Heterogeneous Robots

Several architectural approaches have been proposed to support the coordination of heterogeneous robot teams, especially in unstructured or uncertain environments. For example, previous work explored the coordination of UAV in search and rescue missions or environmental exploration tasks. These approaches typically target fixed-function behaviors such as *take off*, *scan region*, or *land*, and assume a stable set of robots and tasks. Examples include architectures focused on the deployment of UAVs in outdoor settings or mixed aerial-ground coordination for planetary exploration scenarios [78–81]. These works address relevant concerns such as environmental uncertainty and human interaction, but offer limited support for extensibility or dynamic task reconfiguration.

More recent architectural contributions, such as the one proposed by Camara et al. [82], adopt model-based adaptation to select configurations that satisfy non-functional requirements using formal analysis techniques. However, these approaches usually act at the configuration level rather than coordinating the execution of autonomous missions in a fleet of heterogeneous robots. Similarly, distributed supervision frameworks like Hidden [25, 83] and SERA [57] offer valuable mechanisms for mission decomposition and resilience, but are limited in their capacity to dynamically and autonomously assign tasks based on real-time resource availability and contextual factors. Other efforts, such as SHAGE [84], ORCA [85], and RobMoSys [86], emphasize self-management, modularity,

and open development ecosystems. Although these provide solid foundations for building robotic software, they often lack systematic support for autonomous mission coordination or do not explicitly address challenges such as coalition formation or runtime variability in multirobot settings as carried out in MissionControl.

Denguir et al. [87] proposed a generic framework for mission planning and execution with a heterogeneous multi-robot system. The Denguir et al. Framework and MissionControl architecture share the objective of coordinating heterogeneous MRS, but they differ significantly in their focus, particularly regarding core quality attributes and target domains. The MissionControl architecture explicitly targets the service robot domain, involving robots that assist humans in environments such as hospitals, hotels, and restaurants. The design decisions (DDs) of MissionControl are fundamentally driven by software architectural quality attributes, primarily modifiability and integrability. The Denguir et al. framework focuses on the coordination of Unmanned Ground Vehicle (UGV) and UAV in missions like exploration, surveillance, or search and rescue in dynamic and unstructured environments. The framework's primary emphasis is on executing the mission using a decentralized control strategy to enable robots to adapt to environmental changes and maintain formation stability in both 2D and 3D spaces.

Languages and Models for Mission Specification

Although languages and models for robotic mission specification are out of the scope of this work, they are an important aspect in the engineering process of the mission coordination. The ability to model robotic missions at appropriate abstraction levels is central to effective coordination. The current version of MissionControl takes as input the global mission plan in the form of an iHTN. Alternatively, global mission plans could be specified as BTs, which have become a popular formalism in robotics to describe agent behaviors due to their modularity and reactivity. In most of the existing literature, BTs are used to define complete agent control policies [39]. In contrast, the approach adopted in this thesis restricts BTs to the task execution layer, ensuring responsiveness at the operational level, while using HTN at the higher, deliberative layer to provide flexible mission-level reasoning.

Complementary to MissionControl, the MutRoSe framework [88] provides a mission specification language designed to simplify the task assignment process in multirobot systems. It introduces a high-level mission model that accounts for task dependencies and real-world variability, and supports automated decomposition into robot-level tasks. While MutRoSe focuses on the generation of a global mission plan, our work addresses

the subsequent coordination and assignment of these tasks, thereby operating at a complementary level of the mission engineering process.

Pelletier et al. proposed a formal framework [89, 90] which provides a formalism based on Petri Nets utilizing Skill Petri Nets (SPNs) to formally specify and verify hierarchical skill composition. While MissionControl focuses on the overall architecture for mission coordination, it employs a Sequencing process to execute the mission by sequencing tasks derived from an iHTN plan. A formal approach like [89] may be advantageous for the rigorous specification and verification of hierarchical skill composition, enabling the use of formal methods for powerful verification capabilities like model-checking logic properties, offering controller code synthesis, and maintaining compatibility with multiple elementary skill frameworks. An alternative way to specify and plan robotic missions is to use temporal logics such as Linear Temporal Logic (LTL). The specified mission in a temporal logic language is then input into a planner, which synthesizes the mission plan [91–93]. A downside of using temporal logic is that it requires temporal logic specialists to write the mission specifications. Patterns for mission specification [91] have been proposed as a way to fill the gap between natural language for description of missions and temporal logic formalization of the mission specification, relaxing the need for temporal logic specialists to specify the missions. The current version of MissionControl we use iHTN as the representation of a planned mission which allows MissionControl to be easily integrated with an HTN planner, which is an alternative to Petri Net and Temporal Logic-based approaches. However, the mission specification and planning are transversal to MissionControl, and in the future, we may explore extending MissionControl with support to alternative mission planning approaches.

The work by Lestingi et al. [94] focuses specifically on increasing confidence in complex, interactive scenarios by proposing a formal development framework for service robot applications. This methodology employs Stochastic Hybrid Automata (SHA) to model the systemincluding extensions for human behavior featuring non-linear fatigue dynamics and stochastic free willand uses Statistical Model Checking (SMC) to formally verify properties and estimate the probability of successful mission completion. The resulting coordination mechanism is implemented via a peer-to-peer control infrastructure where orchestrators exchange tasks, extending the framework from single-robot to multi-robot scenarios and demonstrating scalability. While Lestingi et al. employ formal methods (SHA/SMC) primarily for quantifying reliability and safety against probabilistic requirements, this thesis concentrates on providing a reusable software architecture optimized for architectural adaptability and coherence in managing coordination logic within dynamic environments.

Skill-Based Approaches to Robotic Coordination

Skill-based abstraction for robot behavior has been a recurring theme in the literature on robotic systems engineering. Rovida et al. [66] proposed a structured approach to modeling robotic software based on formal definitions of skills and their interaction with physical and computational resources. Their work demonstrates how redundancy, fault detection, and alternative skill paths can be integrated into mission execution. However, while this model supports flexible and robust individual robot behavior, it does not explicitly deal with task allocation or mission coordination among multiple robots. In contrast, our architecture builds upon similar skill abstractions but integrates them into a larger coordination framework that supports task assignment and coalition formation in multi-robot scenarios.

While this thesis primarily addresses the architectural challenges and runtime processes necessary for Multi-Robot Mission Coordination, particularly within the service domain, other crucial research focuses on the foundational representation of robot functionality itself. For example, Silva et al[95] contribute a formal method for modeling functions of heterogeneous robots, designed to resolve the issue of inconsistent function descriptions. Based on models initially developed for manufacturing, this approach uses ontologies and standards (such as VDI 3682 and ISA 88) to formally define the abstract function as a Capability and its technology-dependent executable implementation as a Skill. This rigorous, semantic separation enables sophisticated planning and supports advanced formal reasoning about functions, such as matching required capabilities against provided capabilities and inferring knowledge. This focus on the consistent, structured description and reasoning about robot abilities provides a framework that is complementary to coordination architectures like MissionControl, which rely on defined robot capabilities and skills during runtime processes like Coalition Formation and task execution.

Multi-Robot Task Allocation and Planning

The coordination of robot teams also depends heavily on the strategies used for MRTA. Gerkey and Mataris seminal taxonomy [33] provides a useful framework for classifying MRTA problems. According to this taxonomy, the allocation model explored in this thesis can be classified as a single-task, multi-robot, instantaneous assignment (ST-MR-IA) problem. That is, each robot is assigned one task at a time, and multiple robots may be required for a single task. Allocation decisions are made instantaneously, based on the current context, rather than projecting future task requirements.

Although MRTA has been widely studied in the literaturewith surveys offering extensive coverage of algorithms and problem formulations [9, 35, 36, 96]many approaches emphasize offline optimization under static constraints. This static view often falls short in dynamic service environments characterized by fluctuating resources and mission goals. Recent contributions have proposed more adaptive methods, including heuristic-based coalition formation for urgent tasks [97] and formal methods for task scheduling [98]. For instance, Flores proposed KANOA [98], an end-to-end methodology tackling the Multi-Robot Task Allocation and Scheduling problem (classified as single-task, multi-robot time-extended assignment or (SR-MR-TA). This involves finding individual robot plans that are Pareto-optimal with respect to multiple, potentially conflicting, optimization objectives (e.g., maximizing mission success probability, minimizing idle time, minimizing travel cost). The solution synthesis is designed for the design phase of development, where formal guarantees are provided before deployment. Furthermore, research addressing the complexity of delivery systems utilizes multi-objective optimization techniques: for instance, [99] focuses on optimal route scheduling for a Heterogeneous Robotic Delivery System (HRDS), consisting of UGV and UAV, by aiming to maximize customer satisfaction while minimizing delivery costs. Ferreira et al. [100] tackle MRTA in the class SR-MR-TA in a two-stage process. In the first stage is realized the task decomposition using a heuristic search algorithm. In the second stage, task allocation is realized by applying multi-objective optimization using a distributed genetic algorithm with mimetism and knowledge sharing. Garcia et al [101] introduce *Multi-3*, a framework designed to explicitly solve multi-missions and multi-instance tasks. Multi-3 extends HTN to model multi-instance tasks (tasks repeated for multiple locations or objects) and allows for the concurrent execution of multiple missions. To manage this increased complexity and improve resource utilization, Multi-3 introduces the concept of fragments, which are atomic sequences of elementary tasks derived from the mission specification. The system uses an online auction-based strategy to dynamically assign these fragments to idle robots. This dynamic assignment permits robots to seamlessly interleave tasks across different missions or instances, effectively reducing robot idle time and lowering overall mission completion time. *Multi-3* extends what in MissionControl is the coalition formation process, and therefore it is complementary to MissionControl in scenarios where we can optimize resources by treating the allocation as a multi-instance task allocation problem. Yet, most of these works isolate the allocation algorithm from the broader software architecture. Our contribution, by contrast, focuses on the architectural integration of task allocation as a means to support variability, extensibility, and runtime adaptability in heterogeneous robotic missions. MRTA approaches are complementary to MissionControl, which can be extended to use a different task-allocation component implementing a variety of

approaches.

Addressing Uncertainty and Enhancing Quality Attributes

A recurring challenge in mission coordination is the ability to operate under uncertainty, including varying environmental conditions, dynamic robot availability, and mission-specific constraints. In our approach, autonomy is enhanced through the automated formation of robot coalitions and task assignment based on real-time constraints. Other recent efforts address uncertainty through different technical lenses. For example, Filippone et al. [102] adopt a self-adaptive architecture to react to environmental changes, while Tavares et al. [103] propose a multi-robot system architecture focusing on plan recovery for dynamic environments. Jun and Son [104] propose a hierarchical control loop for supervising large-scale heterogeneous multi-robot systems in dynamic environments, with the dynamics of the system modeled through a formal method called hybrid automata. Although these methods differ in scope, they share the goal of increasing system robustness in unpredictable contexts.

Beyond uncertainty, several works have emphasized other quality attributes important for mission-critical robotic systems. Explainability [105], verifiability and testability [106], decentralized trust [107], and runtime monitoring [108] have all been explored as means to increase trustworthiness and maintainability. Furthermore, uncertainty-aware behavior trees [109] were investigated as a way to handle uncertainty while executing a mission in the presence of uncertainty. These works are complementary to the approach proposed in this thesis for mission coordination. Future work will investigate how such attributes can be addressed within MissionControl by incorporating appropriate architectural tactics and design patterns.

State-of-the-Practice in Service Robotics

Coordination strategies for service robots have also emerged in commercial platforms. FetchCore™[1] and MobilePlanner™[2] are two prominent examples that support map generation, robot fleet monitoring, and task scheduling. These systems often provide intuitive user interfaces and streamlined workflows, making them effective for structured environments such as warehouses or manufacturing lines. However, these platforms are typically designed for narrow application domains, depend on vendor-specific ecosystems, and are

[1] https://fetchrobotics.com/resources/fetchcore-software-brochure/
[2] https://automation.omron.com/pt/br/products/family/Mobile%20Planner

difficult to extend to accommodate the variability, autonomy, and uncertainty present in dynamic service scenarios. As such, while informative, they offer limited inspiration for the kind of mission coordination system pursued in this work.

Chapter 7

Conclusion

In this thesis, we delve into the intricate challenges of engineering coordination within multi-robot systems, specifically in the domain of service robots. We took the vision of a software engineer developing end-user-facing solutions atop existing off-the-shelf solutions. In our vision, such an engineer needs to design software on top of generic capabilities provided by the robot vendor or by open-source projects to coordinate the robots in the execution of useful missions for the end-user. In particular, we focused on two specific contributions (i) the software architecture for the mission coordination and (ii) how to efficiently test the coordination by leveraging a purpose-built simulator.

The central contribution of this thesis is MissionControl, an architecture for the coordination of missions of heterogeneous robots. MissionControl increases the level of autonomy of component-based applications in robotics operating in an uncertain environment. Our work leverages modularity and decoupling between coordination and the robot (or set of robots), which are considered key issues for robotics. In MissionControl, missions are received as hierarchical task networks and MissionControl automatically assigns robots available to the received missions. To decide on assignments, the coordinator MissionControl takes into account the required skills for the mission and the actual skills of the robots, such as the constraints of battery charge, while trying to optimize the time required to complete missions. MissionControl provides a component model that allows easy integration of new skills into the system. MissionControl makes no assumptions on the mission specification, in addition to the fact that tasks in a mission specification have a counterpart in the library of skills, which allows a system implemented with MissionControl to execute a variety of missions that use skills within the library. The system manages dynamics by taking advantage of the ensemble paradigm that allows the entry and exit of robots from the ensemble and the formation of the coalition between the available robots based on knowledge at the moment of the mission's start. Additionally, we integrate a reactive layer, based on behavior trees, which allows the robot to react to

emergent events.

The experiments carried out have shown that a system implemented with MissionControl was able to achieve higher success rates compared to a baseline approach. Moreover, we presented a qualitative evaluation that evaluated how the tactics employed contributed to the architectural quality attributes in MissionControl.

Despite the contributions presented, we recognize that further efforts in this domain should continue, as there are still various challenges to be tackled in the engineering of multi-robot mission coordination. The empirical evaluation represents a primary limitation of the work. Although the thesis addressed the bottleneck of producing coordination software for heterogeneous MRS, the quantitative assessment was restricted to controlled experiments conducted within a single simulated environment. As a consequence, external validity remains, and the transferability of the findings to broader mission types, physical deployments, or more diverse operational conditions cannot yet be fully asserted. This constraint is compounded by the relatively narrow set of implemented collaborations and the use of only ten robot skills, which prevents a comprehensive assessment of the architectures capacity to handle the full breadth of heterogeneous robot behaviours. Furthermore, the absence of extensive comparative benchmarking against multiple state-of-the-art MRS architectures limits the ability to rigorously situate MissionControl within the wider landscape of existing approaches.

From an architectural perspective, the framework intentionally prioritized modifiability and integrability; however, this choice entailed the explicit deferral of other quality attributes. We should note that aspects such as safety and availability were not deeply integrated into the design, even though they are central to the development of dependable robotic systems. Similarly, the architecture would benefit in the future from stronger formalization and the inclusion of explicit formal guarantees.

The systems resilience and autonomy are further constrained by its control model. MissionControl currently adopts a centralized approach for task allocation and scheduling, which introduces a single point of failure and places overall system resilience at risk. Addressing this limitation will require extending the architecture toward decentralized, distributed, or hybrid coordination schemes.

Moreover, the systems runtime adaptation capabilities remain limited. Full support for adaptive scheduling and self-adaptation under uncertainty is yet to be realized. Correspondingly, more comprehensive supervisory and recovery mechanisms will be necessary to strengthen the systems ability to maintain continuity and autonomously manage unforeseen events. Future research may further increase system autonomy by adding new processes, such as a supervisory process capable of handling mission errors like those described in Chapter 4.

Finally, we have introduced an ongoing initiative to develop a lightweight simulation methodology specifically designed to facilitate testing and experimentation in multi-robot mission coordination. This approach seeks to bridge the existing gap between physical simulators and conventional non-realistic lightweight simulators, the latter of which typically do not support seamless code migration from simulation to real-world robotic platforms. In this context, we outlined the principles underpinning lightweight simulation and presented a prototype, referred to as HMR Sim. However, it is important to note that HMR Sim remains under active development and, as of yet, lacks a comprehensive systematic evaluation.

References

[1] Horizon 2020 Call ICT-2016. Service Robots, 2021. 1

[2] Lionel P. Robert, Marcelo Fantinato, Sangseok You, and Patrick C. K. Hung. Social robotics business and computing: Editorial for special issue of information systems frontiers. *Information Systems Frontiers*, 26(1):18, July 2023. 1

[3] Jochen Wirtz, Paul G Patterson, Werner H Kunz, Thorsten Gruber, Vinh Nhat Lu, Stefanie Paluch, and Antje Martins. Brave new world: service robots in the frontline. *J. Serv. Manag.*, 29(5):907–931, November 2018. 1

[4] Horizon 2020. Robotics 2020 Multi-Annual Roadmap, 2015. 1

[5] David Silvera-Tawil. Robotics in healthcare: A survey. *SN Comput. Sci.*, 5(1), January 2024. 1

[6] Nripendra P. Rana, Nusaiba Begum, Mohd. Nishat Faisal, and Anubhav Mishra. Customer experiences with service robots in hotels: a review and research agenda. *Journal of Hospitality Marketing & Management*, 34(2):145174, September 2024. 1

[7] Nivin Vinoi, Amit Shankar, Reeti Agarwal, and Rsha Alghafes. Revolutionizing retail: The transformative power of service robots on shopping dynamics. *Journal of Retailing and Consumer Services*, 82:104085, January 2025. 1

[8] Avinash Gautam and Sudeept Mohan. A review of research in multi-robot systems. In *2012 IEEE 7th International Conference on Industrial and Information Systems (ICIIS)*, pages 1–5, 2012. 1

[9] Yara Rizk, Mariette Awad, and Edward W. Tunstel. Cooperative Heterogeneous Multi-Robot Systems: A Survey. *ACM Computing Surveys*, 52(2):1–31, 2019. 1, 10, 13, 15, 93

[10] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2021. 2, 6, 7, 19, 20, 22, 32, 33, 35, 61

[11] Mehrnoosh Askarpour, Christos Tsigkanos, Claudio Menghi, Radu Calinescu, Patrizio Pelliccione, Sergio Garcia, Tim J. von Oertzen, Manuel Wimmer, Luca Berardinelli, Matteo Rossi, Marcello M. Bersani, and Gabriel S. Rodrigues. RoboMAX: Robotic Mission Adaptation eXemplars. In *Proceedings of the 16th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2021. 4, 6

[12] John W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications, Thousand Oaks, CA, 4th edition, 2014. 4, 5

[13] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. DEECO: An ensemble-based component system. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering - CBSE '13*, page 81. ACM Press, 2013. 6, 13, 66

[14] Tomas Bures, Frantisek Plasil, Michal Kit, Petr Tuma, and Nicklas Hoch. Software Abstractions for Component Interaction in the Internet of Things. *Computer*, 49(12):50–59, 2016. 6, 13, 29, 36, 37

[15] Stanford Artificial Intelligence Laboratory et al. Robotic operating system. https://www.ros.org, May 2025. Accessed: March 2025. 6, 48

[16] Sara Mahdavi-Hezavehi, Paris Avgeriou, and Danny Weyns. A classification framework of uncertainty in architecture-based self-adaptive systems with multiple quality requirements. In *Managing Trade-Offs in Adaptable Software Architectures :*, pages 45–78. Morgan Kaufmann, Oxford, England, 1st edition, 2016. 6

[17] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90, 2016. Accessed: March 2025. 6, 55, 58

[18] Gabriel Rodrigues, Ricardo Caldas, Gabriel Araujo, Vicente Moraes, Genaina Rodrigues, and Patrizio Pelliccione. Replication Package. https://zenodo.org/record/6516005#.YnGxuNNBzb0, May 2022. 6, 7, 55, 58, 62, 63, 64, 66

[19] Ivano Malavolta, Grace Lewis, Bradley Schmerl, Patricia Lago, and David Garlan. How do you architect your robots? state of the practice and guidelines for ros-based systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '20, page 3140, New York, NY, USA, 2020. Association for Computing Machinery. 7, 61, 62, 66

[20] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 3rd edition, 2010. 9

[21] Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems*. Cambridge University Press, Cambridge, England, December 2008. 9

[22] Jutta Kiener and Oskar von Stryk. Towards cooperation of heterogeneous, autonomous robots: A case study of humanoid and wheeled robots. *Robotics and Autonomous Systems*, 58(7):921–929, 2010. 10

[23] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, Oxford, England, May 2004. 11

[24] Kutluhan Erol, James A. Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In Barbara Hayes-Roth and Richard E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 2*, pages 1123–1128. AAAI Press / The MIT Press, 1994. 11, 29

[25] Charles Lesire, Guillaume Infantes, Thibault Gateau, and Magali Barbier. A distributed architecture for supervision of autonomous multi-robot missions: Application to air-sea scenarios. *Autonomous Robots*, 40(7):1343–1362, 2016. 11, 12, 32, 43, 89

[26] Dana Nau, Tsz-Chiu Au Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404, 2003. 11, 43

[27] Gabriel S. Rodrigues, Felipe P. Guimarães, Genaína N. Rodrigues, Alessia Knauss, João Paulo C. de Araújo, Hugo Andrade, and Raian Ali. GoalD: A Goal-Driven deployment framework for dynamic and heterogeneous computing environments. *Information and Software Technology*, 111:159–176, 2019. 12

[28] Kyoung-Dae Kim and P. R. Kumar. CyberPhysical Systems: A Perspective at the Centennial. *Proceedings of the IEEE*, 100:1287–1308, May 2012. 13

[29] Rolf Hennicker and Annabelle Klarl. Foundations for ensemble modeling–the helena approach: Handling massively distributed systems with elaborate ensemble architectures. In *Specification, Algebra, and Software: Essays Dedicated to Kokichi Futatsugi*, pages 359–381. Springer, 2014. 13

[30] Rocco De Nicola, Gianluigi Ferrari, Michele Loreti, and Rosario Pugliese. *A Language-Based Approach to Autonomic Computing*, pages 25–48. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. 13

[31] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Frantisek Plasil, Filip Krijt, Jiri Vinarek, and Jan Kofron. A language and framework for dynamic component ensembles in smart systems. *International Journal on Software Tools for Technology Transfer*, 22(4):497–509, 2020. 13

[32] Martin Wirsing, Matthias Hölzl, Nora Koch, and Philip Mayer, editors. *Software Engineering for Collective Autonomic Systems*, volume 8998 of *LNCS*. Springer International Publishing, 2015. 14

[33] Brian P. Gerkey and Maja J. Matari. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, 23(9):939–954, 2004. 14, 39, 92

[34] G. Ayorkor Korsah, Anthony Stentz, and M. Bernardine Dias. A comprehensive taxonomy for multi-robot task allocation. *The International Journal of Robotics Research*, 32(12):1495–1512, 2013. 14, 30

[35] Alaa Khamis, Ahmed Hussein, and Ahmed Elmogy. Multi-robot Task Allocation: A Review of the State-of-the-Art. In Anis Koubâa and J.Ramiro Martínez-de Dios, editors, *Cooperative Robots and Sensor Networks 2015*, volume 604 of *Studies in Computational Intelligence*, pages 31–51. Springer International Publishing, 2015. 14, 93

[36] Ernesto Nunes, Marie Manner, Hakim Mitiche, and Maria Gini. A taxonomy for task allocation problems with temporal and ordering constraints. *Robotics and Autonomous Systems*, 90:55–70, 2017. 14, 93

[37] Michele Collendanchise and Petter Ögren. *Behavior Trees in Robotics and Al*. Number 6 in Chapman & Hall/CRC Artificial Intelligence and Robotics Series. CRC Press, Taylor and Francis Group, CRC Press is an imprint of the Taylor and Francis Group, an Informa Business, 2019. 15

[38] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. A survey of behavior trees in robotics and AI. *Rob. Auton. Syst.*, 154(104096):104096, August 2022. 15

[39] Razan Ghzouli, Thorsten Berger, Einar Broch Johnsen, Andrzej Wasowski, and Swaib Dragule. Behavior trees and state machines in robotics applications. *Software Engineering, IEEE Transactions on*, 49(9):4243–4267, September 2023. 15, 90

[40] Anis Koubaa, editor. *Robot operating system (ROS)*. Studies in computational intelligence. Springer International Publishing, Cham, 2023. 17

[41] Aaron Staranowicz and Gian Luca Mariottini. A survey and comparison of commercial and open-source robotic simulator software. In *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '11, New York, NY, USA, 2011. Association for Computing Machinery. 21

[42] HeeSun Choi, Cindy Crump, Christian Duriez, Asher Elmquist, Gregory Hager, David Han, Frank Hearl, Jessica Hodgins, Abhinandan Jain, Frederick Leve, Chen Li, Franziska Meier, Dan Negrut, Ludovic Righetti, Alberto Rodriguez, Jie Tan, and Jeff Trinkle. On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward. *Proceedings of the National Academy of Sciences*, 118(1), December 2020. 21

[43] Jack Collins, Shelvin Chand, Anthony Vanderkop, and David Howard. A review of physics simulators for robotic applications. *IEEE Access*, 9:5141651431, 2021. 21

[44] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004. 21, 68

[45] Louis Hugues and Nicolas Bredeche. Simbad: an autonomous robot simulation package for education and research. In *International Conference on Simulation of Adaptive Behavior*, pages 831–842. Springer, 2006. 21, 68

[46] Eric Rohmer, Surya PN Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326. IEEE, 2013. 21, 68

[47] Gilberto Echeverria, Nicolas Lassabe, Arnaud Degroote, and Séverin Lemaignan. Modular open robots simulation engine: Morse. In *2011 IEEE International Conference on Robotics and Automation*, pages 46–51. IEEE, 2011. 21, 68

[48] Paulo Henrique Maia, Lucas Vieira, Matheus Chagas, Yijun Yu, Andrea Zisman, and Bashar Nuseibeh. Dragonfly: a tool for simulating self-adaptive drone behaviours. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 107–113. IEEE, 2019. 21, 68

[49] Jean Bélanger, P Venne, and Jean-Nicolas Paquin. The what, where and why of real-time simulation. *Planet Rt*, 1(1):25–29, 2010. 22

[50] Patrick Redmond, Jonathan Castello, José Manuel Calderón Trilla, and Lindsey Kuper. Exploring the theory and practice of concurrency in the Entity-Component-System pattern. *Proc. ACM Program. Lang.*, 9(OOPSLA2):1–28, October 2025. 23

[51] Dennis Wiebusch and Marc Erich Latoschik. Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems. In *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 25–32. IEEE, 2015. 25

[52] Ben Moran. Esper: An entity component system (ecs) for python. https://github.com/benmoran56/esper. Accessed: March 2025. 25

[53] Dan North & Associates Limited. Introducing bdd. https://dannorth.net/introducing-bdd/, 2006. Accessed: March 2025. 26

[54] Gojko Adzic. *Specification by Example: How Successful Teams Deliver the Right Software*. Manning, 2011. 26

[55] Cucumber Contributors. Gherkin language reference. https://cucumber.io/docs/gherkin/reference/. Accessed: March 2025. 26

[56] Alejandro Marzinotto, Michele Colledanchise, Christian Smith, and Petter Ogren. Towards a unified behavior trees framework for robot control. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5420–5427, Hong Kong, China, May 2014. IEEE. 29, 45

[57] Sergio García, Claudio Menghi, Patrizio Pelliccione, Thorsten Berger, and Rebekka Wohlrab. An architecture for decentralized, collaborative, and autonomous robots. In *ICSA*, pages 75–84. IEEE Computer Society, 2018. 36, 89

[58] Vladimír Matna. *Integration Paradigms for Ensemble-Based Smart Cyber-Physical Systems*. Doctoral thesis, Charles University, 2018. 47

[59] Martin Fowler. Inversion of Control Containers and Dependency Injection pattern. http://www.martinfowler.com/articles/injection.html, 2004. Accessed: March 2025. 48

[60] Ioannis Kostavelis and Antonios Gasteratos. Semantic mapping for mobile robotics tasks: A survey. *Robotics and Autonomous Systems*, 66:86–103, April 2015. 49

[61] Douglas C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, Inc, Hoboken, NJ, 8th edition, 2013. 50

[62] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. 50

[63] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. *The Goal Question Metric Approach*. Encyclopedia of Software Engineering, Wiley, 1994. 50

[64] Séverin Lemaignan, Gilberto Echeverria, Michael Karg, Jim Mainprice, Alexandra Kirsch, and Rachid Alami. Human-robot interaction in the MORSE simulator. In *Proceedings of the Seventh Annual ACM/IEEE International Conference on Human-Robot Interaction*, HRI '12, pages 181–182. Association for Computing Machinery, 2012. 54

[65] George C. Runger Douglas C. Montgomery. *Applied Statistics and Probability for Engineers*. Wiley, 7th edition, 2018. 55, 57

[66] Francesco Rovida, Matthew Crosby, Dirk Holz, Athanasios S. Polydoros, Bjarne Großmann, Ronald P. A. Petrick, and Volker Krüger. SkiROS—A Skill-Based Robot Control Platform on Top of ROS. In Anis Koubaa, editor, *Robot Operating System (ROS)*, volume 707, pages 121–160. Springer International Publishing, Cham, 2017. 63, 92

[67] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. A study on challenges of testing robotic systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, October 2020. 68

[68] Christopher S. Timperley, A. Afzal, D.S. Katz, J.M. Hernandez, and C. Le Goues. Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In *International Conference on Software Testing, Verification and Validation, IEEE*, pages 331–342. ICST, 2018. 68

[69] Thierry Sotiropoulos, H. Waeselynck, J. Guiochet, and F. Ingrand. Can robot navigation bugs be found in simulation? an exploratory study. In *International Conference on Software Quality, Reliability and Security*, QRS'17, pages 150–159. IEEE, 2017. 68

[70] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *IROS*, pages 5026–5033. IEEE, 2012. 68

[71] José-Luis Blanco-Claraco, Borys Tymchenko, Francisco José Mañas-Alvarez, Fernando Cañadas-Aránega, Ángel López-Gázquez, and José Carlos Moreno. Multivehicle simulator (mvsim): Lightweight dynamics simulator for multiagents and mobile robotics research. *SoftwareX*, 23:101443, 2023. 68

[72] Robocode Contributors. Robocode Tank Royale Docs. https://robocode-dev.github.io/tank-royale/. Accessed: March 2025. 68

[73] Pytest Contributors. Pytest documentation. https://docs.pytest.org/en/stable/. 74

[74] Pytest-BDD Contributors. Pytest-bdd documentation. https://pytest-bdd.readthedocs.io/en/stable/. 74

[75] Benjamin Moran. Esper. https://github.com/benmoran56/esper, 05 2020. 74

[76] JGraph. https://github.com/jgraph. Accessed: March 2025. 75

[77] Drawio. drawio. https://app.diagrams.net. Accessed: March 2025. 75

[78] J. Gancet, G. Hattenberger, R. Alami, and S. Lacroix. Task planning and control for a multi-UAV system: Architecture and algorithms. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1017–1022, Edmonton, Alta., Canada, 2005. IEEE. 89

[79] J. Cacace, A. Finzi, V. Lippiello, M. Furci, N. Mimmo, and L. Marconi. A control architecture for multiple drones operated via multimodal interaction in search & rescue mission. In *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pages 233–239, Lausanne, Switzerland, October 2016. IEEE. 89

[80] Jose Luis Sanchez-Lopez, Martin Molina, Hriday Bavle, Carlos Sampedro, Ramón A. Suárez Fernández, and Pascual Campoy. A Multi-Layered Component-Based Approach for the Development of Aerial Robotic Systems: The Aerostack Framework. *Journal of Intelligent & Robotic Systems*, 88(2-4):683–709, December 2017. 89

[81] Martin J. Schuster, Marcus G. Muller, Sebastian G. Brunner, Hannah Lehner, Peter Lehner, Ryo Sakagami, Andreas Domel, Lukas Meyer, Bernhard Vodermayer, Riccardo Giubilato, Mallikarjuna Vayugundla, Josef Reill, Florian Steidle, Ingo von Bargen, Kristin Bussmann, Rico Belder, Philipp Lutz, Wolfgang Sturzl, Michal Smisek, Moritz Maier, Samantha Stoneman, Andre Fonseca Prince, Bernhard Rebele, Maximilian Durner, Emanuel Staudinger, Siwei Zhang, Robert Pohlmann, Esther Bischoff, Christian Braun, Susanne Schroder, Enrico Dietz, Sven Frohmann, Anko Borner, Heinz-Wilhelm Hubers, Bernard Foing, Rudolph Triebel, Alin O. Albu-Schaffer, and Armin Wedler. The ARCHES Space-Analogue Demonstration Mission: Towards Heterogeneous Teams of Autonomous Robots for Collaborative

Scientific Sampling in Planetary Exploration. *IEEE Robotics and Automation Letters*, 5(4):5315–5322, October 2020. 89

[82] Javier Cámara, Bradley Schmerl, and David Garlan. Software architecture and task plan co-adaptation for mobile service robots. In *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 125–136. ACM, 2020. 89

[83] Thibault Gateau, Charles Lesire, and Magali Barbier. HiDDeN: Cooperative plan execution and repair for heterogeneous robots in dynamic environments. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4790–4795, 2013. 89

[84] Dongsun Kim, Sooyong Park, Youngkyun Jin, Hyeongsoo Chang, Yu-Sik Park, In-Young Ko, Kwanwoo Lee, Junhee Lee, Yeon-Chool Park, and Sukhan Lee. Shage: A framework for self-managed robot software. In *Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems*, SEAMS '06, page 7985, New York, NY, USA, 2006. Association for Computing Machinery. 89

[85] Tobias Kaupp, Alex Brooks, Ben Upcroft, and Alexei Makarenko. Building a Software Architecture for a Human-Robot Team Using the Orca Framework. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 3736–3741, Rome, Italy, April 2007. IEEE. 89

[86] RobMoSys: Composable Models and Software for Robtics Systems - Towards an EU Digital Industrial Platform for Robotics. http://robmosys.eu, (2017-2020). 89

[87] Mohsen Denguir, Ameur Touir, Achraf Gazdar, and Safwan Qasem. Toward a generic framework for mission planning and execution with a heterogeneous multi-robot system. *Sensors*, 24(21):6881, 2024. 90

[88] Eric Bernd Gil, Genaína Nunes Rodrigues, Patrizio Pelliccione, and Radu Calinescu. Mission specification and decomposition for multi-robot systems. *Robotics and Autonomous Systems*, 163:104386, May 2023. 90

[89] Baptiste Pelletier, Charles Lesire, and Karen Godary-Dejean. A formal framework for the specification and verification of robotic skills composition for autonomous behaviors. *Robotics and Autonomous Systems*, page 105041, 2025. 91

[90] Baptiste Pelletier, Charles Lesire, and Karen Godary-Dejean. Net-based analysis of robotic skills. In *Application and Theory of Petri Nets and Concurrency: 46th International Conference, PETRI NETS 2025, Paris, France, June 22–27, 2025, Proceedings*, volume 15714, page 354. Springer Nature, 2025. 91

[91] Claudio Menghi, Christos Tsigkanos, Thorsten Berger, and Patrizio Pelliccione. PsALM: Specification of Dependable Robotic Missions. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 99–102. IEEE, 2019. 91

[92] Claudio Menghi, Christos Tsigkanos, Mehrnoosh Askarpour, Patrizio Pelliccione, Gricel Vázquez, Radu Calinescu, and Sergio García. Mission specification patterns for mobile robots: Providing support for quantitative properties. *IEEE Transactions on Software Engineering*, 49(4):2741–2760, 2022. 91

[93] Gricel Vázquez, Anastasia Mavridou, Marie Farrell, Tom Pressburger, and Radu Calinescu. Robotics: A new mission for fret requirements. In *NASA Formal Methods Symposium*, pages 359–376. Springer, 2024. 91

[94] Livia Lestingi, Cristian Sbrolli, Pasquale Scarmozzino, Giorgio Romeo, Marcello M Bersani, and Matteo Rossi. Formal modeling and verification of multi-robot interactive scenarios in service settings. In *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*, pages 80–90, 2022. 91

[95] Luis Miguel Vieira da Silva, Aljosha Köcher, and Alexander Fay. A capability and skill model for heterogeneous autonomous robots. *at-Automatisierungstechnik*, 71(2):140–150, 2023. 92

[96] Alaa Khamis, Ahmed Hussein, and Ahmed Elmogy. *Multi-robot Task Allocation: A Review of the State-of-the-Art*, pages 31–51. Springer International Publishing, Cham, 2015. 93

[97] Miao Guo, Bin Xin, Yipeng Wang, and Jie Chen. A local-search-based heuristic for coalition formation in urgent missions. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 54(11):69246935, November 2024. 93

[98] Gricel Nidteja Vazquez Flores. *Scheduling of Missions with Constrained Tasks for Heterogeneous Multi-Robot Systems*. PhD thesis, University of York, April 2024. 93

[99] Zhihuan Chen, Shangxuan Hou, Zuao Wang, Yang Chen, Mian Hu, and Rana Muhammad Adnan Ikram. Delivery route scheduling of heterogeneous robotic system with customers satisfaction by using multi-objective artificial bee colony algorithm. *Drones*, 8(10):519, 2024. 93

[100] Barbara Arbanas Ferreira, Tamara Petrović, and Stjepan Bogdan. Distributed mission planning of complex tasks for heterogeneous multi-robot systems. In *2022 IEEE 18th International Conference on Automation Science and Engineering (CASE)*, pages 1224–1231. IEEE, 2022. 93

[101] Juan Antonio Piñera García, Gianluca Filippone, Marco Autili, and Patrizio Pelliccione. Multi-3 adaptive coordination of robots. *(preprint) Available at SSRN 5635645*, 2025. 93

[102] Gianluca Filippone, Juan Antonio Piñera García, Marco Autili, and Patrizio Pelliccione. Handling uncertainty in the specification of autonomous multi-robot systems through mission adaptation. In *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 24. ACM, April 2024. 94

[103] Carlos Joel Tavares da Silva and Célia Ghedini Ralha. Multi-robot system architecture focusing on plan recovery for dynamic environments. In *2023 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1668–1673, 2023. 94

[104] Chanyoung Ju and Hyoung Il Son. A hybrid systems-based hierarchical control architecture for heterogeneous field robot teams. *IEEE Transactions on Cybernetics*, 53(3):1802–1815, 2021. 94

[105] Marcello M. Bersani, Matteo Camilli, Livia Lestingi, Raffaela Mirandola, Matteo Rossi, and Patrizia Scandurra. A conceptual framework for explainability requirements in software-intensive systems. *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*, pages 309–315, 2023. 94

[106] Ricardo Caldas, Juan Antonio Pinera Garcia, Matei Schiopu, Patrizio Pelliccione, Genaina Rodrigues, and Thorsten Berger. Runtime Verification and Field-Based Testing for ROS-Based Robotic Systems. *IEEE Transactions on Software Engineering*, 50(10):2544–2567, October 2024. 94

[107] Renan Lima Baima, Loïck Chovet, Eduard Hartwich, Abhishek Bera, Johannes Sedlmeir, Gilbert Fridgen, and Miguel Angel Olivares-Mendez. Trustful coopetitive infrastructures for the new space exploration era. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, SAC 24, pages 612–621. ACM, April 2024. 94

[108] Marco Stadler and Michael Vierhauser. Romosu: Flexible runtime monitoring support for ros-based applications. In *2023 IEEE/ACM 5th International Workshop on Robotics Software Engineering (RoSE)*, pages 53–60, 2023. 94

[109] Mehran Rostamnia, Gianluca Filippone, Ricardo Caldas, and Patrizio Pelliccione. Towards adaptable and uncertainty-aware behavior trees. In *2025 IEEE/ACM 7th International Workshop on Robotics Software Engineering (RoSE)*, pages 9–16. IEEE, 2025. 94