WiRoM: a High-level Mission Planning System for Heterogeneous Multi-Robot Simulations

Joakim Moss Grutle

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Bergen University College

Department of Informatics,
University of Bergen

June 2020

Western Norway
University of
Applied Sciences

Abstract

Robots are complicated machines. Today we have a lot of different types of robots, each with different types of sensors and actuators which have their own practical (and impractical) applications. Utilizing the benefits of having multiple heterogeneous robots is useful for creating a more diverse repository of executable tasks, some that might be impossible to execute for a single robot or homogeneous multi-robots. Having a more diverse repository of tasks would allow robots to execute complex missions and increase the efficiency of mission execution. Planning new missions for heterogeneous multi-robots is a complicated and convoluted assignment for most users because of the robotics and programming knowledge that is required to plan such complex missions. This thesis will present **WiRoM**, a web-based mission planning system for heterogeneous multi-robot setups that have been designed and developed by combining the core concepts of Model-Driven Software Engineering with robot programming. By abstracting the mission planning to a higher level and implementing the user interface as a low-code platform, we can make mission planning, task development, and task allocation more trivial and efficient for users.

Acknowledgements

Firstly, I would like to thank my supervisor Prof. Adrian Rutle for his engagement in the project and thesis. He provided continuous guidance, knowledge, and assistance throughout my work, and his contributions are irreplaceable. I need to thank my co-students and other friends from my studies, they have been one of the driving factors for making my study period as great as it has been. I would also like to thank my family and childhood friends for backing my choices and providing encouragement and motivation throughout my entire study. Finally, I would like to thank my girlfriend for her continuous perseverance and support allowing me to work on my studies and conduct this research.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Chapter 1

Introduction

1.1 Background

Robots have become a more abstract construct ever since they were introduced. If you mention the word **robot** to someone, they may think of something completely different then you do. Many people might think of a science fiction humanoid machine walking around serving real humans, while others think of something more trivial like e.g. a lawnmower, a toy, a car, or maybe even the robot-dance popularized in the 60s.

The Oxford English Dictionary provides this definition of the word **robot**: ***"A machine capable of carrying out a complex series of actions automatically, especially one programmable by a computer"*** [1]

Traditionally, robots have been programmed or hardwired to solve some specific actions in a static environment, for instance as a robot factory assembly line or an autonomous vacuum cleaner. As technology is moving forward we get more advanced robots which are continuously increasing the efficiency and the scope of problem-solving. The optimal scenario would be to have a single type of robot that could solve all given problems with minimal human programming. Such a robot does not currently exist, but working towards this goal by combining existing solutions and technologies would help move the domain towards this optimal scenario.

Robots have become tools used by humans in everyday life, both industrial and private. Different robots have different actions and purposes, where some robots are very simple, and some are more advanced. But what happens when we have a *complex mission*, where the possibilities of physical

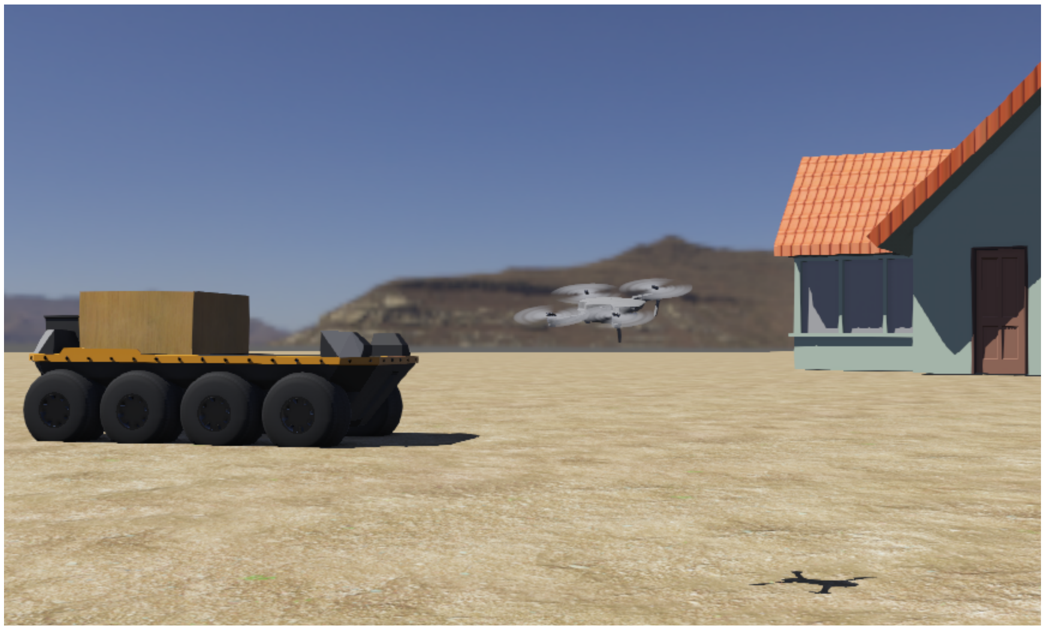


Figure 1.1: Screen-capture from the simulated environment in WiRoM

human interaction are slim, and no robot exists that possess all the required components to perform such a mission? One cannot simply ask a rover to fly or a drone to swim, we have to remember that robots are not only complex but also limited and fragile tools.

A *multi-robot* system can make it possible for multiple robots to cooperate to solve missions that couldn't be solved by a single robot alone. A system where we could combine *heterogeneous robots*, where each robot has different applications, could offer more actions and tasks for more advanced mission execution. Raising the *abstraction levels* of such a system and having a simple user interface would provide a foundation for the system to allow users with different levels of experience to utilize the system. Having a safe and low-cost environment for mission planning was also desired, and this could be provided by a *simulation tool*. Planning missions for simulated environments would provide proper missions that could be transferred to real-life scenarios.

This thesis present **WiRoM** (**W**eb **i**nterface based **Ro**bot **M**ission planner) as a new solution to the heterogeneous multi-robot mission planning problem. Figure 1.1 shows a screen-capture from the simulated environment used in WiRoM, where we can see a drone and a rover in action. This solution will be a *prototype* since no other system currently exists that does it in the same way. The system will be implemented as a *high-level web application*, where a user can plan missions directly from the browser. The user interface will be designed as a **low-code platform**, letting users with minimal robotics and programming experience plan, execute, and observe heterogeneous multi-robot missions directly from the browser.

1.2 Problem Description

There will be challenges that need to be resolved when developing *WiRoM*. One challenge is the problem of programming *heterogeneous robots* at the lower levels, and running them in a simulated environment. Each robot will have individual components that need to be controlled, e.g. wheels, arms, cameras, etc. Some *language or framework* that allows for efficient robot programming will need to be located and utilized to solve this challenge.

Going between the abstraction levels, i.e. transforming high-level mission to low-level programming will also be a challenge. This will be resolved by using **Model-Driven Software Engineering** (MDSE) core concepts in combination with robot programming and simulation tools to *bridge the gap between the high and low abstraction levels*.

Another challenge will be to create a **high-level interface** which is capable of complex mission solving. Users with little experience should be able to plan complex missions without having to spend a lot of time learning how to program robots. Users should also avoid having to download and install unnecessary software in order to make it work. This challenge will be resolved by developing the interface of the system as a web interface by locating a web development framework and use it to implement the system as a *low code platform*.

Additionally, there will be challenges with *task allocation* in heterogeneous multi-robot setups. This problem is often introduced as the **Multi Robot Task Allocation** (MRTA) problem. There are two main approaches to task allocation, which is **automatic task allocation** and **manual task allocation**. It is desired for the system to find some combination of automatic and manual task allocation to increase the efficiency and optimization of task allocation, and have the possibility for the system to be used for further testing of different task allocation algorithms.

1.3 Motivation

New technology and solutions are developed every day to solve different types of challenges that humans have in the real world. Robots become more integrated into our lives, and there are many real-life problems and challenges (missions) that can be solved better using robots. Developing a usable and functional prototype of WiRoM would provide more research that would be useful for further work and/or research, which the main motivating factor for providing a proper solution to the problems stated in this thesis.

1.4 Research question

The problem description provides the foundation for the research questions asked in this thesis. We are looking to create a new solution as a functional prototype for the heterogeneous multi-robot mission planning problem. The following research questions will be answered in the thesis:

Main research question

RQ1 *How well do Model Driven Software Engineering core concepts combined with robot programming enhance the practicality and usability of a high-level heterogeneous multi-robot mission planning system?*

Sub-questions

RQ2 *How well do the concepts and principles of low-code platforms enhance the user-friendliness of the system*

RQ3 *How efficiently are tasks allocated to multiple heterogeneous robots within the system?*

1.5 Method

The research in this thesis will be conducted as a **case study**. *Case studies* are defined by Yin [2] as *"an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries* *between phenomenon and context are not clearly evident"*. One of the main preconditions for conducting *case study* research is that the research question is concerned about *how* some phenomena occur. This thesis will create WiRoM as a prototype solution to the heterogeneous multi-robot mission planning problem, using several different concepts to reach the desired results. Data collection will be done by user-testing the system and developing a practical mission example using the system. This data will be used to evaluate the system, and assess the validity of our results.

1.6 Thesis Outline

- Chapter 2 provides the theoretical background for the thesis. This chapter starts by defining the field of study in the context of the thesis, before it goes into the theory of Model-Driven Software Engineering, Distributed Autonomous Robotics Systems, Multi-Robot Task Allocation, and finally Methodology to discuss the relevant theory and information which is the theoretical background for the thesis.

- Chapter 3 looks at related work to the thesis, providing information on similar projects preceding the one provided in this thesis.

- Chapter 4 is the design and implementation of the system that is provided in the thesis. The chapter goes through each step of the project and provides insight into why different choices were made and how the different parts were designed and implemented.

- Chapter 5 presents the different user roles of the system and the which cases they will act upon the system, as well as the workflow of the system for said users and cases. This chapter shows how the system is intended to be used by users with varying amounts of experience.

- Chapter 6 discusses the research strategy and method used to collect data for evaluating the system. The data collection methods are then presented and the results are discussed alongside the topics from the research questions.

- Chapter 7 provides the conclusion of the thesis. The conclusion provides the answer to the research question, before wrapping up the thesis and system as a solution to the presented problems.

- Chapter 8 present the suggested improvements and opportunities for further work on the system.

Chapter 2

Theoretical Background

2.1 Field of study

This section will present and define the different topics that cover the basic knowledge that is in the scope of this thesis. These topics will be explained shortly concerning the problem of this thesis to provide the reader with some basic insight into these topics and clarify the meaning and purposes of the topics in this context. The more advanced topics of this thesis will be discussed after we have defined the field of study.

2.1.1 Robotics

Robotics is defined in the *Oxford English Dictionary* as "The branch of technology that deals with the design, construction, operation, and application of robots" [3]. This tells us that robotics is considered as the technological domain that acts upon robots, including the robots themselves. The robot definition proposed in the introduction by the *Oxford English Dictionary* defines robots a"machines capable of carrying out actions automatically" and are "programmable by a computer", but what does this tell us? The book *Elements of robotics* by M. Ben-Ari and F. Mondada [4] explains that it depends on how one would define the complexity of the actions that the robot performs because otherwise it could just be defined as an automaton. It is also difficult to define what a computer is, maybe the computer could be another robot? The book claims that there is one part of robots that fall outside of this definition, which is the sensors of a robot. These sensors make the robot capable of acting in complex environments and adapts its action based on the data gathered by the sensors. Acting out actions using robots with multiple sensors and data would, therefore, pose a difficult challenge.

When considering a heterogeneous multi-robot setup, one would not only focus on one type of robot but multiple different types of robots. There are many different classifications of robots as can be seen in [4], and they all need to be programmed to operate. We can see why it would be useful and efficient to have a high-level system that provides a way of programming said robots, without having to learn all the specifics of each robot.

2.1.2 Tasks

A **task** is formally defined in the *Oxford English Dictionary* as "A piece of work to be done or undertaken" [5]. A task can be considered as one specific part of a larger project, working towards some goal. Tasks in robotics can be considered as small sub-problems that need to be solved by robots. These tasks should be big enough such that there is significant progress towards the overall goal when a task is complete. Tasks should also be small and independent enough such that they have the possibility of being distributed to the robot that can execute the task the best. Defining tasks can be done by combining robot actions at a lower level, and then distribute the tasks to the robots that can perform said actions the best. Having a set of such tasks working towards a goal will be considered as a mission, where tasks are executed by multiple heterogeneous robots to reach the goal of that mission.

2.1.3 Missions

A **mission** is formally defined in the *Oxford English Dictionary* as "An important assignment given to a person or group of people, typically involving travel abroad" [6]. A mission is an assignment (or set of smaller assignments) with a specific goal that needs to be reached. The *Oxford English Dictionary* definition is based on human missions, which typically go abroad. In the robotics case, this would be similar, having robots or groups of robots solving missions in some environment. Solving a mission would mean solving multiple different sub-assignments, or in our case tasks. A mission consists of some set of tasks, all of which need to be executed to reach the goal of the mission. Letting users define missions and tasks for each robot will provide an easy and high-level solution to execute missions using robots. The problem with robotics is that it is very costly and requires prior knowledge within this domain, but by using simulation tools, one can avoid such problems.

2.1.4 Simulation

Zlajpah [7] defines simulations as “Simulation is the process of designing a model of an actual or theoretical physical system, executing the model, and analyzing the execution output”. Simulations are widely used in many different fields, such as research from engineering and computer science to economics and social science. Digital simulation tools are a big part of the simulation field practiced within the software and robotics domain. Such simulations can make it possible for a user to execute scenarios with systems that would be expensive, dangerous, or maybe even impossible to execute in the real world. They could also assist in scaling the system and provide a solid foundation for testing other parts of the reality it simulates. Simulations of any system are not one-to-one comparable to real life, and different tools will have different pros and cons, so choosing the correct tool for a project is important. Creating simulated models of real-world missions would provide a solid foundation for verifying and validating whether that mission would be suitable as a mission that could be solved in the real world.

2.1.5 Verification and validation

Simulations can be seen as models of some real-life scenarios. R. Sargen mention in the paper [8] that there are two generally considered definitions of simulated model **verification and validation**. Model verification is often defined as “ensuring that the computer program of the computerized model and its implementation are correct”. Model validation is usually defined to mean “substantiation that a computerized model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model”. Two main concepts are considered to be applied to robot mission planning, computerized mission verification, and operational mission validation. Computerized mission verification would be an automatic process that could verify whether a mission has been implemented correctly and is a valid mission (i.e. it can be executed). There are tools and modeling languages that allow developers to create models and perform dynamic testing for automatic verification. Operational validation of the simulation model would be done by simply executing a mission in a simulation tool and observe whether the goal of the mission was reached properly, proving that it's intended application is valid.

2.2 Model Driven Software Engineering

This section will discuss the theoretical background behind *Model-Driven Software Engineering* (MDSE)[1], discussing the core concept and definitions, highlighting the most relevant parts for solving our challenges (Section 1.2). The book *Model-Driven Software Engineering in Practice: Second Edition* by M. Brambilla, J. Cabot, and M. Wimmer [9] will be the main material in addition to other sources that are cited throughout this section.

Model-Driven Software Engineering (MDSE) is a methodology for software development where models are at the core of the development process. In MDSE, a model is usually a diagram created using a modeling language, for instance, the Unified Modeling Language (UML) [10]. The purpose of using such models in software engineering is to provide a specification of the software and raise the abstraction level of the software. One of the main principles of MDSE is raising abstraction levels, this helps reduce the complexity of the system for users and simplify the development process. Besides, MDSE contributes to concepts such as model transformation and automatic code generation, using models to help build the codebase [9].

2.2.1 What is a model?

In short, a model can be defined as a simplified or partial representation of reality. A model will never describe reality in its entirety but will capture the main characteristics of reality. Models have central importance in many scientific contexts. Humans use models to understand and teach difficult or complex ideas presented in different scientific scenarios. For instance, imagine how hard it would be to explain concepts like atoms or the universe without creating a model to capture the features of such a concept. It is also important to have well-defined rules and semantics within certain domains,

[1]We use the term MDSE, but there are many different names and definition such as MDE, MDD, MBE, etc. as seen in [9], and although they have some differences, we focus on the core concepts and will therefore not separate these definitions unless it is required for some cases.

such that the idea that is being modeled isn't being misrepresented by the model [9].

Models can have different purposes:

- Descriptive models, which are models that explain a reality or a concept
- Prescriptive models, which are models that determine the scope and details of how to study a problem
- Models that define how a system should be implemented

New technology and concepts are making it possible to use such models to develop software, and have models at the core of the engineering process [9].

2.2.2 MDSE concepts

MDSE is a methodology because it applies the advantages of its core concepts to the software engineering domain. Figure 2.1 shows an overview of the main aspects of the MDSE methodology. These are the core aspects of MDSE represented at different levels of reality. We describe the model by going through each row (core aspects) and its columns (levels):

- The first row is the model concept. The model of the application is defined at the application level. The next level is the application domain, where the modeling language is defined. The final level of modeling is the meta-level, which is the modeling language that describes the modeling language, or the meta-modeling language.
- The second row is the automation concept. Automation is defined in the application level as automatic model transformation or code generation. The transformation definition is defined at the application level, while the meta-level define the transformation language used to create the transformation.
- The third row is the realization concept. The realization is defined by artifacts such as code at the application level. The implementation platforms of specific domains are defined at the application domain for the artifacts [9].

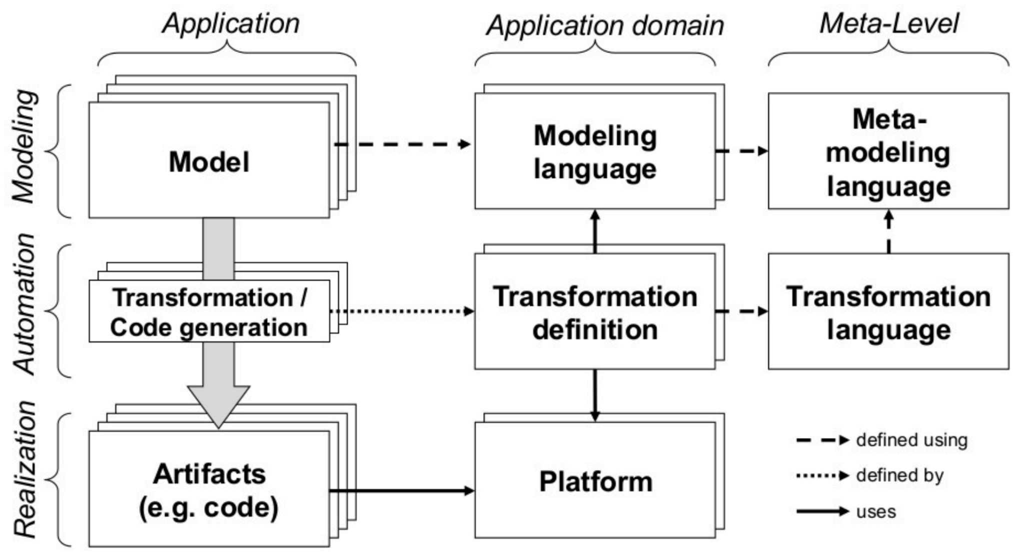


Figure 2.1: Overview of the MDSE methodology (top-down process) [9]

The core flow of MDSE is from the application models down to the running realization, through subsequent model transformations. This allows the reuse of models and execution of systems on different platforms. Indeed, at the realization level, the running software relies on a specific platform (defined for a specific application domain) for its execution [9].

2.2.3 Model Driven Architecture

The *Object Management Group* (OMG) defined a comprehensive proposal for applying MDSE practices to systems development [11]. This proposal goes under the name of *Model-Driven Architecture* (MDA). MDA itself is not an OMG specification but rather an approach to system development which is enabled by existing OMG specifications. The entire MDA infrastructure is based on a few core components and their definitions: System, model, architecture, platform, viewpoint, view, and transformation. These components are defined in MDA in line with MDSE and are an incarnation and representation given by OMG to the various aspects of system definition [9].

Figure 2.2 shows the three levels of abstraction that are defined in MDA. These levels are defines as:

- Computation-Independent Model (CIM): The model at the highest abstraction level. This model represents the context, requirements, and purpose of a target system without explaining the computational parts of the system. The CIM presents an exact solution for the system, and what the solution is expected to do. The CIM should not show the specification at the lower levels, keeping the model-independent from the implementation process for the system. The CIM is also referred to as the business model or domain model, and may not (in principle) even need a map to a software-based implementation citeMDSEbook.

- Platform-Independent Model (PIM): This model is at the level of the system where the behavior and structure are defined, regardless of the implementation platform. The PIM will model the parts of the CIM that can be solved using a software-based solution, and further refines the requirements for the software system presented in the CIM. The PIM is more specific than the CIM, but still exhibit a sufficient degree of independence from the lower levels, implying that it can map to one or more implementation platforms [9].

- Platform-Specific Model (PSM): The model at the lowest abstraction level. This model is not executable, but contain the required information regarding the structure and behavior of a system on a specific platform. The developers can use this platform in combination with the PIM to implement the executable code for the system [9].

2.2.4 Domain-Specific Languages

Domain-Specific Languages (DSL) are languages with individual and specific syntax and notation that is used to develop a system within a specific domain. A DSL is the opposite of a *General-Purpose Language* (GPL) which is a language that is created such that it can be applied to many different domains. A. Nordmann et al. published the paper [12] which highlights two fundamental characteristics of well-designed DSLs: "*their expressive power targeted a specific domain and the definition of formal notations intuitively understandable for domain experts while being machine-processable, eventually yielding executable models of robotics applications*". A DSL has to be powerful within its domain, and also intuitive, otherwise, there would most likely be no actual utilization of that DSL within that domain.

A survey on DSLs in robotics was performed in [12]. They analyzed 41 different DSLs, to find that the most common way of creating DSLs was to use the Eclipse Modeling Framework (EMF) [13]. EMF provides a toolchain for creating DSLs and metamodels using the popular Eclipse IDE. Following EMF was other custom toolchains and also the approach of extending a GPL. The latter is interesting because it means creating a DSL from a GPL like Java, Python, C++, etc. Such a DSL is called an internal DSL because it extends the syntax and notation from the GPL to perform more specific action within the domain at hand. This would open up for having all the benefits of the GPL available when developing different parts of the application.

2.2.5 Low-code development

Low-code development and *low-code development platforms* (LCDP) are two terms that derive from the usage of low-code applications to perform software development. Mendix [14] is a low-code platform that is considered as one of the leading enterprises within the field. Figure 2.3 shows a screenshot

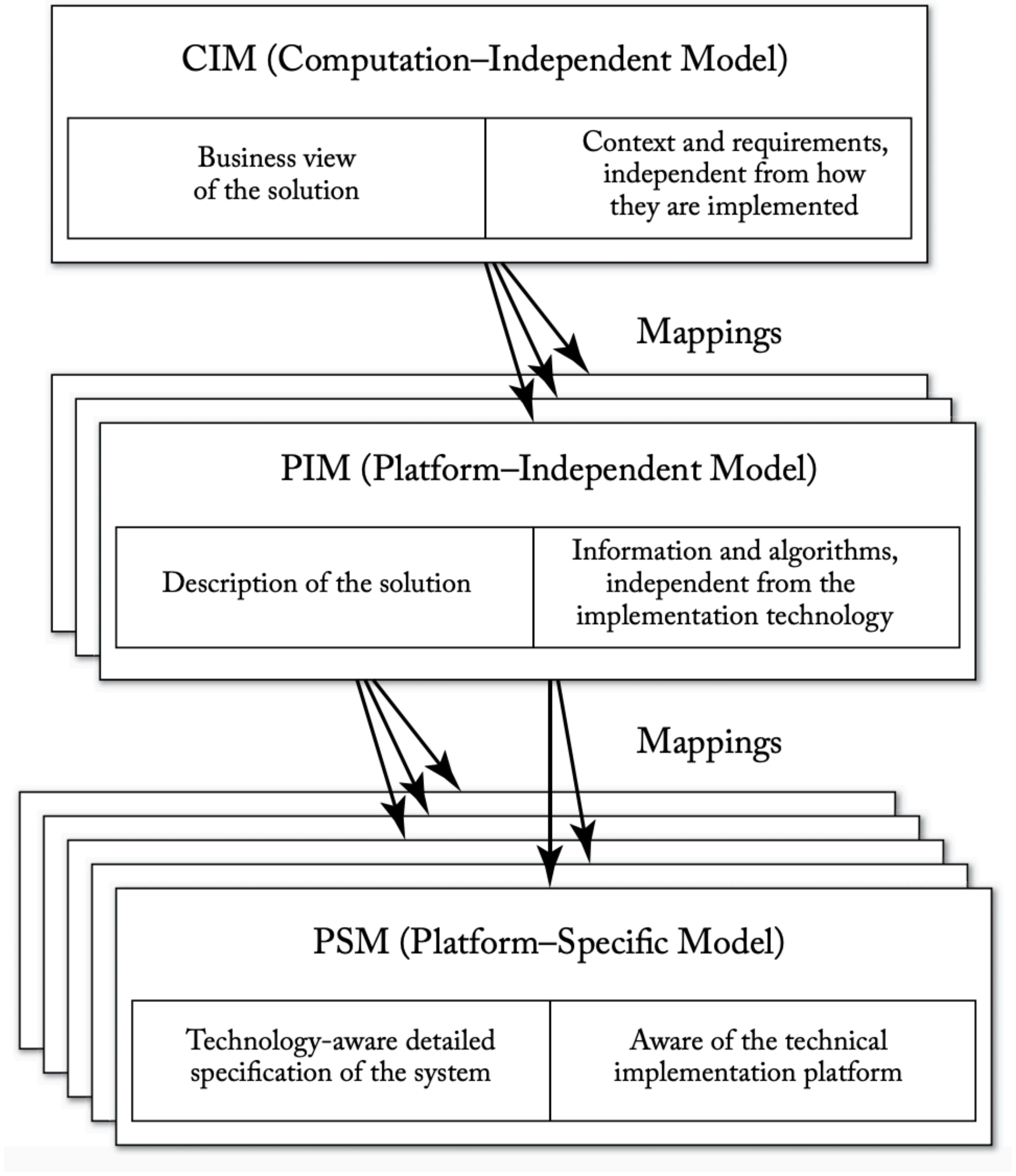

CIM (Computation–Independent Model)
Business view of the solution
Context and requirements, independent from how they are implemented
Mappings
PIM (Platform–Independent Model)
Description of the solution
Information and algorithms, independent from the implementation technology
Mappings
PSM (Platform–Specific Model)
Technology-aware detailed specification of the system
Aware of the technical implementation platform

Figure 2.2: The three levels of modeling abstraction codified in MDA [9]

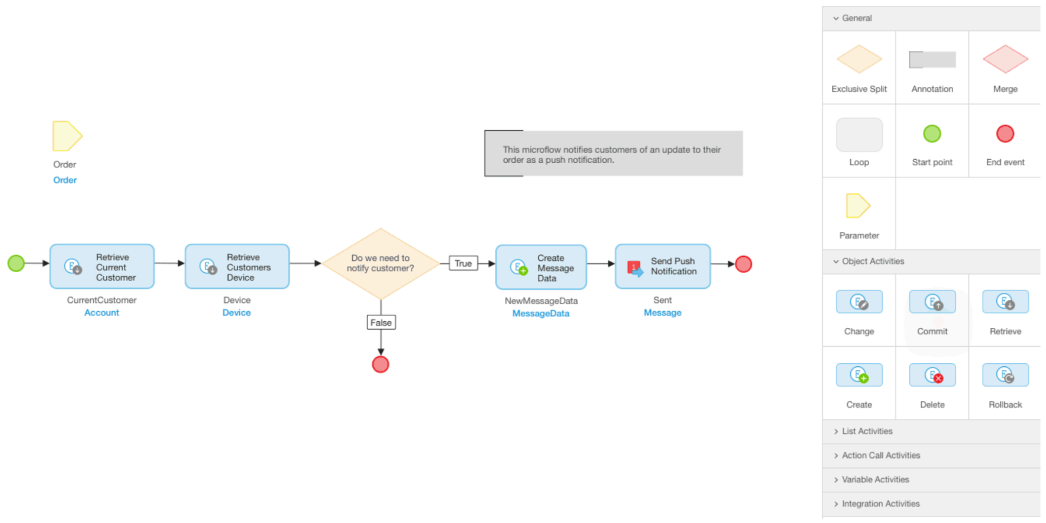


Figure 2.3: Screenshot from the Mendix development platform

from one of Mendix low-code platform development tools, which operates much like a modeling tool. Mendix focuses on the connection between low-code and model-driven development. In a blog-post that was posted on their website by Johan den Haan, he mentions that model-driven development is the most important concept in low-code platforms [15]. Additionally, *modeling-languages.com* mentions that low-code is just a new *buzzword* [16] for model-driven engineering, because the definitions of the two are so similar, and low-code sell better because it is ”new”. Now whether it is just a new buzzword or not is not important to us, but the fact that low-code has a strong correlation to MDSE is the important part. Low-code can be seen as a method of developing software that uses MDSE principles and concepts.

LCPDs are software development platforms where a user can develop software using something like a graphical user interface, reducing the amount of manual coding needed when making new applications [17]. Such platforms come with several benefits. It allows for a wider range of users, such as users without much experience or users who are experts in other fields can take part in the development process of software, since the threshold for contribution has been lowered. The cost and time of production may also go down, since an application may be quick to set up using LCDP, and only require minimal manual coding for providing specific functionality. LCDP also has some drawbacks. Each LCDP is usually very domain-specific, meaning it can only be applied when developing software within a certain domain. It may also require some learning for users before they can use the platform, which may not be worth it if it's only for short term results.

2.2.6 MDSE and robotics

There are many users from multiple scientific and technological domains that are experts within their respective fields of work. They can benefit greatly from being able to transfer their knowledge to robotics and vice versa. The appliance of models and MDSE principles can have a big impact on robotic software systems, mainly by making robotics more accessible for domain experts from other domains, increasing the level of work and research that can be performed within the robotics domain [18].

The cost of creating new robotics products is significantly related to the complexity of developing software control systems that are flexible enough to easily accommodate frequently changing requirements:

1) more advanced tasks in highly dynamic environments

2) in collaboration with unskilled users

3) in compliance with changing regulations

Recent initiatives aimed at developing the MDSE approaches that simplify the static and dynamic reconfiguration of a robot control system according to specific application requirements and operational conditions [18].

D. Brugali [18] discuss different parts of MDSE within the robotics domain. One of the points made in this paper is that one of MDSEs core principles, which is automation (automatic code generation), does not provide the biggest benefits when it comes to robotics development. This is because they argue that robotics go through a lot of change, making automatic code generation tools obsolete whenever new (and possibly groundbreaking) technology is introduced within the robotics domain. They also argue that code generation leads to higher training costs and organizational changes, reducing the benefits of using such an approach for developing robotics.

They rather claim that the architectural parts of MDSE provide the main benefits to robotics. We already provided a short introduction to MDA in Section 2.2.3 and the core concepts of that approach. The paper [18] shows that using MDA creates a clear difference between functional components and the Platform-Independent Models (PIMs) of a software system. Transforming PIMs to Platform-Specific Models (PSMs) when programming robots provide resilience to the constant changes in the robotics world, because of the PIMs independence from elements like functional components and code generators. The paper concludes that the architectural model was a central artifact of all the different activities during the development process (analysis, design, implementation, configuration, and documentation).

MDSE conclusion Using Model-Driven Software Engineering as a methodology by creating models as a means of developing the system is not that interesting for this project. On the other hand, using the principles of MDSE as a foundation for creating a Mission Planning system for heterogeneous multi-robot setups will be a beneficial approach to reach the goals of the system.

As discussed in Section 2.2.6, principles from MDA would be the most useful and relevant for a robotics project. Creating high-level models that can be mapped or transformed into lower-level models (CIMs to PIMs to PSMs) could be a good approach to develop how the system moves between the different abstraction levels of the application. The high-level models can be created as an internal DSL, using a modern GPL or multiple GPLs at different abstractions levels. This would allow for easier access to simulation tools and also bring the benefits of using a modern GPL with its features and frameworks.

Having the internal DSL implemented as a Graphical User Interface (GUI) using principles from Low-Code Development Platforms would provide a simple and user-friendly interface. It was also required for DSLs to be powerful within its domain, and using MDSE concepts such as automatic code generation and model transformation would make the application powerful at the lower levels of the application. The only problem is that these parts of the system could become outdated (as stated in Section 2.2.6), and should, therefore, be at such a level that it is independent of the code generator, making it a simple process to replace this at a later stage if so desired.

2.3 Multi-robot task allocation

Missions and tasks can become complex, especially when they are to be allocated to heterogeneous multi-robot setups. This problem is known as the *Multi-Robot Task Allocation* (MRTA) problem. There are many different types of tasks that robots can perform, with many different types of ways to allocate them.

2.3.1 The Multi-robot task allocation problem

MRTA is a complex problem, especially when it comes to heterogeneous robots with many different capabilities and functions that will work together to solve some human-defined mission. MRTA problems addresses the problem of assigning tasks to robots in a way that increases the efficiency and reliability in such a system. Figure 2.4 shows an illustration of the MRTA problem. In this problem, it is given:

1. R: a team of mobile robots $r_i; i = 1, 2, ...n$
2. T: a set of tasks $t_{ij}; j = 1, 2, ...n_t$
3. U: a set of robot utilities, u_{ij} is the utility of robot i to execute task j

For the general case, the problem is to find the optimal allocation of a set of tasks to a subset of robots, which will be responsible for accomplishing it: $A : T \to R$. [19]

This problem can be defined as an optimization problem, or more specifically, a variation of the optimal assignment problem. Solving the MRTA problems using optimal assignment algorithms is widely considered as an efficient method of allocating tasks to robots, and the extent of the efficiency depends on the optimization of the algorithm. In [19] they perform a brief overview of the MRTA field and review the state-of-the-art MRTA solutions to different types of MRTA problems.

Many methods have been proposed for solving the MRTA problem. The MRTA problem can be seen as a range of different problems, depending on the approach. Some of these problems are the *Discrete Fair Division Problem*, the *Multiple Traveling Salesman Problem*, the *Alliance Efficiency Problem*, and (as mentioned) the *Optimal Assignment Problem.* All of these are well-known problems that have been researched over a long time, and many different solutions have been proposed to try to solve these problems.

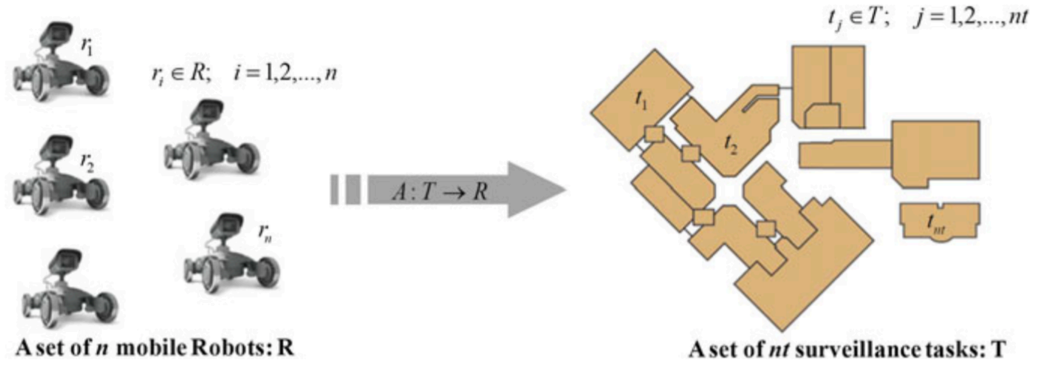

r_1
$r_i \in R; \quad i = 1,2,\ldots,n$
r_2
r_n
A set of n mobile Robots: R
$A: T \rightarrow R$
$t_j \in T; \quad j = 1,2,\ldots,nt$
t_1
t_2
t_{nt}
A set of nt surveillance tasks: T

Figure 2.4: MRTA problem [19].

Organizational approaches

There are two different organizational approaches when it comes to task allocation for a multi-robot setup. The first one is a **centralized** approach. This approach is when there is some controller that performs the task allocation based on the information available in the system and communicates with all the robots which task they are to perform. The second approach is a **decentralized** or *distributed* approach. This approach would have all the robots communicate with each other to decide who performs which task. Both approaches have their pros and cons, e.g. centralized is more efficient, especially for smaller systems. The problem with the centralized approach is its robustness to faults, which is not a problem with the decentralized approach because each robot will continue its work if one of the others breaks down.

MRTA approaches

The first and most popular MRTA approach is the **market-based** approach, which consists of having each robot in a system bid on tasks that they can perform. An auctioneer announces the available tasks, then each robot calculates their utility and lets the auctioneer know their bid for each task. The auctioneer then uses some optimization strategy to decide who wins each bid. The market-based approach is efficient, robust (can utilize both organizational approaches), scalable, and well able to operate in unknown or dynamic environments. The disadvantages of market-based approaches lie in the inconsistency and informality that come with calculating the utility for each robot since different utility functions might fit well for different approaches.

The second MRTA approach is the **optimization-based** approach. Optimization is the branch of applied mathematics focusing on solving a specific problem to find the optimal solution for this problem out of a set of available solutions. Many different approaches have been proposed for different MRTA problems: *stochastic, linear programming, population-based, and hybrid optimization solutions.* The optimization-based approach performs as well as the market-based approach, while also being more robust.

2.3.2 Taxonomy

To help organize this work and identify the theoretical foundations of what they describe largely ad hoc approaches, B. P. Gerkey and M. J. Mataric proposed a taxonomy for MRTA problems [20]. This taxonomy, which is now widely used, provides a common vocabulary for describing MRTA problems. Korsah, Dias, and Stentz claim that this taxonomy is limited in scope. Their publication [21] provides an even more comprehensive and complete taxonomy called iTax that explicitly handles the issues of interrelated utilities and constraints which applies to a much larger space of important task allocation problems.

Defining a task allocation problem

We begin by looking into Gerkey and Mataric's work. In [20] they show that there are axes that are considered in a task allocation problem: the type of robot, the type of task, and the type of allocation. Defining the respective axes and finding the appropriate setup for the current task allocation problem scenario will provide more insight into how this problem can be solved. A visual representation of the axes can be seen in Figure 2.5.

Two different types of robots are considered for task allocation problems: single-task robots (ST) and multi-task robots (MT). ST robots are only capable of solving a single task, while MT robots can perform multiple different tasks at the same time. Having a setup of MT robots would provide more challenges to the task allocation problem than with ST robots because the number of possible allocation scenarios would increase greatly by the capabilities of each robot.

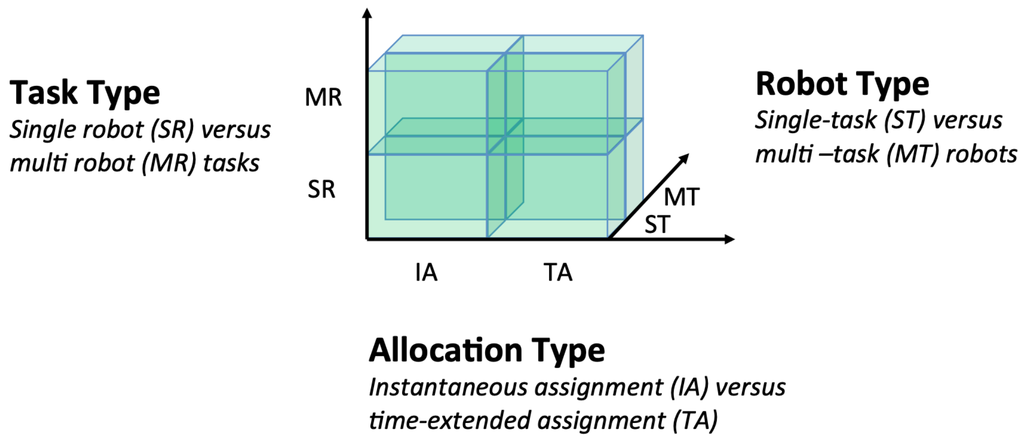


Figure 2.5: Visual representation of the three axes of Gerkey and Mataric's taxonomy [21]

There are also two types of tasks: single-robot tasks (SR) and multi-robot tasks (MR). SR tasks are tasks that require one robot to be able to be executed, while MR tasks would require the combined efforts of more than one robot to be able to solve that one specific task. The same happens with MR tasks as with MT robots, making the number of possible allocations increase with the number of robots needed for each MR task.

Finally, there is also considered two types of allocation: instantaneous assignment (IA) and time-extended assignment (TA). The process of IA would be to assign one task at a time to each robot, not planning for future tasks. TA on the other hand would assign multiple tasks to robots, needing future planning to decide which robots should perform which task after their first.

The three different axes for the task allocation problem has two different types each. This gives us a total of 2^3 or 8 possible scenarios.

Utility

The MRTA problem is mostly considered as an optimization problem, considering how one is trying to use task allocation algorithms to optimize the performance of the overall system. To achieve this, one would need some sort of performance value, or in this case a utility value. The utility value is a concept in economics, game theory, and operations research, as well as in multi-robot coordination. Having a utility function for each robot is based on the notion that each individual can internally estimate the value (or the cost) of performing an action or a task.

The formal MRTA taxonomy paper [20] provides a function that can be used to calculate the utility based on two factors:

- The expected quality Q of which the robot is able to execute the task (e.g. accuracy and fault-handling)
- The expected resource cost C, given the spatio-temporal requirements of the task (e.g., the amount of battery required to drive the motors)

So given robot R and task T, if R is able to execute T, then one can define on some standardized scale Q_{RT} and C_{RT} such that one can calculate the utility score of that robot. This results in a combined, non-negative utility measure:

$$U_{RT} = \begin{cases} Q_{RT} - C_{RT} & \text{if R is capable of executing T and } Q_{RT} > C_{RT} \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

The calculated utility value in Expression 2.3.2 will however be inexact due to unforeseen factors like noise on sensors, faults in hardware, or environmental problems. When we talk about optimal allocation in this sense we base this on the information we have available when the task allocation algorithm runs in the system. Calculating the utility is supposed to be flexible because the requirements for calculating utility changes a lot based on the setup, so being able to include all of the most important aspects for setups within the system is important to get the optimal result.

iTax

Korsah, Dias, and Stentz [21] describe more thoroughly the types of tasks that can be present within a system. Figure 2.6 shows an illustration of the different levels of complexity for a task. This is based on terminology for MRTA that was proposed by Zlot [22]:

- The elemental tasks are tasks that consist of one single (atomic) action.
- Simple tasks are tasks with multiple actions required, that are all given to a single robot.

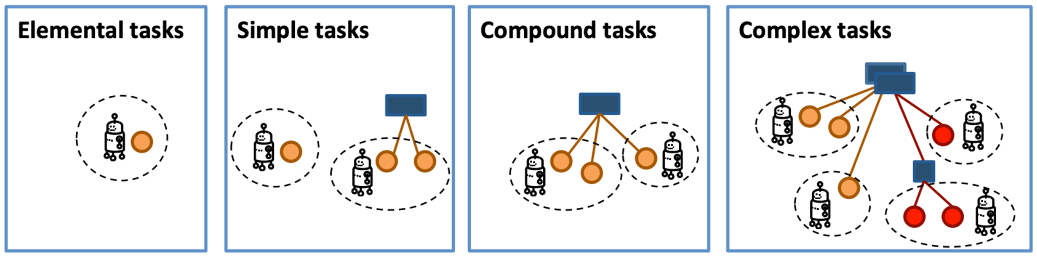


Figure 2.6: Illustration of Zlot's task types [21].

- Compound tasks are more advanced tasks that have a set of actions that can't be allocated to one single robot, requiring multiple robots.
- Complex tasks are similar to compound tasks, except it is unknown before the allocation process which actions should be allocated together to resolve the task.

Dotted circles indicate potential valid allocations of tasks to robots. Shaded circles represent elemental tasks while shaded rectangles represent decomposable tasks, whose decomposition into elemental tasks is illustrated by a tree-like structure. The superimposed trees in the rightmost figure illustrate multiple possible ways of decomposing the example complex task.

This definition is used to determine how advanced a system is. The more advanced the tasks, the more advanced the system. Additionally, iTax proposes a two-level taxonomy in which the first level comprises a single dimension defining the degree of interdependence of robot-task utilities. The second level provides further descriptive information about the problem configuration, utilizing Gerkey and Mataric's taxonomy.

iTax represents the degree of interdependence with a single categorical variable with four possible values, listed below and illustrated in Figure 2.7.

- No Dependencies (ND): These are task allocation problems with simple or compound tasks that have independent robot-task utilities. That is, the effective utilities of a robot for a task do not depend on any other tasks or robots in the system.
- In-Schedule Dependencies (ID): These are task allocation problems with simple or compound tasks for which the robot-task utilities have intra-schedule dependencies. That is, the effective utilities of a robot for a task depend on what other tasks the robot is performing.

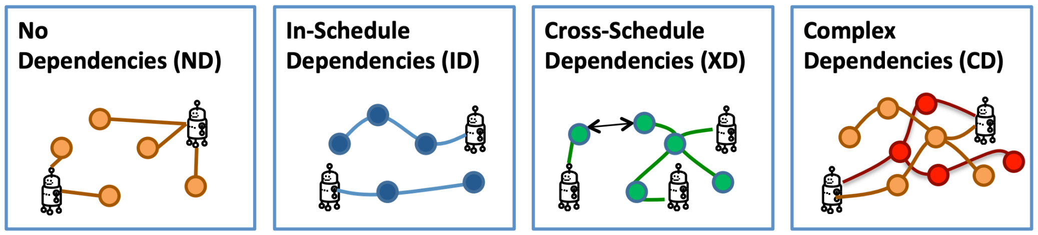

No Dependencies (ND)
In-Schedule Dependencies (ID)
Cross-Schedule Dependencies (XD)
Complex Dependencies (CD)

Figure 2.7: Examples illustrating the four high-level categories of the new taxonomy [21].

- Cross-Schedule Dependencies (XD): These are task allocation problems with simple or compound tasks for which the robot-task utilities have inter-schedule dependencies (in addition to in-schedule dependencies for each robot). That is, the effective utilities of a robot for a task depend not only on its schedule but also on the schedules of other robots in that system.
- Complex Dependencies (CD): These are task allocation problems for which the robot-task utilities have inter-schedule dependencies for complex tasks (in addition to any in-schedule and cross-schedule dependencies for simple or compound tasks). That is, the effective utilities of a robot for a task depend on the schedules of other robots in the system in a manner that is determined by the particular task decomposition that is ultimately chosen.

Shaded circles represent tasks and solid lines represent robot routes. Arrows between tasks indicate constraints. The superimposed routes in the rightmost figure illustrate multiple possible task decompositions.

A problem will be defined using a prefix using this two-level approach. This means that a task allocation problem like e.g. MT-SR-IA will also fit within one of the four categories illustrated in Figure 2.7. If this MT-SR-IA is a problem with in-schedule dependencies (ID), then this problem would be categorized as a **ID** [**MT-SR-IA**] task allocation problem. The label **ID** [**MT-SR-IA**] refers to the category of problems with in-schedule dependencies for which we need to perform instantaneous allocation (TA) of multi-robot tasks (SR) to single-task robots (ST).

The types of task, robot, and allocation processes are not the only components to be considered when analyzing the task allocation problem. iTax specifies that there may exist some constraints in the context of the given task or mission which may decide whether the task can be solved by a specific robot (or any robot). These constraints include concepts such as capability, time, proximity, or simultaneous execution, and can be considered as a set of joint side constraints. An example of a constraint would be that a robot must be capable of executing a task within a certain time-frame. It might also need to be in proximity of some objects to perform the task and not interfere with simultaneous executions. These are just some typical constraints that need to be considered together with other factors that can affect the task allocation process.

The XD [ST-SR-IA] MRTA problem in this thesis

Defining the task allocation problem for the system will be entirely dependent on how complex the missions that will be the target for the system should be, and the way the tasks and robots are implemented at the lower levels. Developing WiRoM to execute overly complex missions are not needed to show an example of the possibilities of the system. The same thing applies to the MRTA problem because it could be an entire thesis by itself. We want to go for a simple task allocation problem in the prototype, to show that it is possible, and argue that the system can be extended, such that it can solve more complex MRTA problems.

We want to avoid multi-robot tasks (MR) and multi-task robots (MT), as well as not focusing on future planning (TA), such that most of the time is spent developing a proper functional prototype of WiRoM. This means the system will be developed such that we can solve the task allocation problem as an ST-SR-IA problem. The complexity of missions that can be solved using the ST-SR-IA will still be very extensive and will provide a good prototype for solving such problems.

There will be some cooperation between different robots on missions level, i.e. the missions consist of multiple tasks that need to be performed by multiple robots. Otherwise, we would not have a multi-robot mission solving system. This is not the same as having multi-robot tasks (MR) because it does not provide simultaneous task solving, however, it fits nicely into the category of having Cross-Schedule Dependencies when solving missions. Even though each robot and task is allocated one to one, they still need to cooperate to execute the final mission goal together. The task allocation problem for the system will be labeled as a **XD [ST-SR-IA]** problem.

The ST-SR-IA problem can be seen as an optimal assignment problem in the following way: if we have m robots, n prioritized tasks, and utility estimates for each of the mn possible robot-task pairs, assign at most one robot to each task. The amount of robots is not large, and a centralized approach can, therefore, be used to collect the robots' utilities and then calculate the optimal task allocation in $O(mn^2)$ time using for instance Kuhn's Hungarian method [23].

All of the previous MRTA solutions for **XD [ST-SR-IA]** discussed in the iTax paper [21] have been using a market-based approach with a centralized controller, because this makes the most sense for such a problem. A market-based approach is therefore the desired solution to the MRTA problem in this project. iTax proposes the mathematical model below for the XD [ST-SR-IA] problem. N is the set of robots, M is the set of tasks, T is the time-limit for a constraint and K is the set of joint side constraints.

Maximize:

$$\sum_{i\in N}\sum_{j\in M} u_{ij}x_{ij} \tag{2.2}$$

Subject to:

$$\begin{gathered}\sum_{i\in N}\sum_{j\in M} t_{ij}x_{ij} \le T_k, \quad \forall k \in K \\ \sum_{i\in N} x_{ij} \le 1, \quad \forall j \in M \\ x_{ij} \in \{0,1\}, \quad \forall i \in N \quad \forall j \in M\end{gathered} \tag{2.3}$$

This problem wants to maximize (2.2) subject to the constraints shown in (2.3). In (2.2), u_{ij} is the utility estimate and x_{ij} is whether a task is allocated to a robot. The first expression in (2.3) tells us that the time t it takes for robot i to perform task j should be less than or equal to the time limit for that constraint. The second expression says that the sum of all task j assigned to a robot i should be less than or equal to one, meaning each robot is only assigned one task, and each task has been assigned to only one robot. The final expression is a constraint that tells us that each robot and task pair should be either 0 or 1, so either the robot is allocated to the task (0) or not (1).

This is supposedly one of many mathematical models for this problem, but this is the one presented in the paper because it fits the problem at hand. Having such a model shows how one should approach this MRTA problem from a mathematical standpoint, and will be useful for the development process of the task allocation algorithm in the system.

MRTA conclusion Going through the literature has given a good indication of how to go forward with creating a task allocation algorithm for the system. Our approach is in the form of a **XD [ST-SR-IA]** problem that can be solved using a *market-based* approach with a *centralized* controller. The robots bid using a utility estimate combined with the considered cross-schedule dependencies, and the mission planner allocates the tasks to the best fitting robots.

2.4 Distributed Autonomous Robotic Systems

The Distributed Autonomous Robotic Systems (DARS) is an international symposium. Searching for and finding relevant publications from these books can provide a lot of knowledge about several different aspects of robots that are related to the topics that are presented in this thesis.

2.4.1 DARS review

The first book was published in 1994 [24] and the next book will be the 15th version of the book, published after the next international symposium which is to be held in Kyoto, Japan in November of 2020 [25]. These books contain many publications, some more relevant to the thesis than others. Reading through the titles and abstracts of these publications and compare them to a set of keywords will filter out all unwanted publications, making it easier to focus on the publications that contain relevant information.

Keywords for assessing relevance: Heterogeneous, multi-robot, mission planning, task allocation, high-level, mission execution, simulation.

The publications [26] [27] [28] [29] [30] provide research on exploratory missions. These types of missions have clear practical use cases in the real world and have clear benefits of being executed by robots. All except [27] used simulations to provide an evaluation for the work done by the researchers. Instead [27] discusses different types of exploratory missions, categorizing different types of missions and their purposes, hazards, etc. It proceeds to explain why some missions are enhanced by or even require robots for optimal performance. This paper help creating many different scenarios for exploratory missions that can be executed by the mission planning system [29]. The publication [26] performs topological exploration using different types of external factors like human interaction and odor to explore the terrain, providing feedback on findings. The project [28] simulates the usage of heterogeneous robots to perform exploration, by using several different *Marsus*-bots for exploring the terrain, while a "motherbot" is used as a power-station which can move around and provide power to the *Marsus*-bots, showing the benefits of using a heterogeneous setup for such missions. The paper [30] provides a path planning solution using real ocean data for multi-robot environmental monitoring, taking into account all the different variables in such an environment to collect data using robots.

Several publications in DARS confront the *task allocation problem* from many different perspectives, and publications like [31] [32] provide more insight into specifically *multi-robot* task allocation. These two publications execute the task with the given task allocation in simulations to get their results. Both provide solutions to the multi-robot task allocation problem within environments that are similar to real-life setups. Both also use market-based solutions to reach their conclusion. [31] uses a decentralized heterogeneous multi-robot setup, which is also able to select the right amount of robots needed based on priority. [32] focuses more on real-time task allocation, stating that solving the task allocation problem itself does not mean we get perfect task allocation, since the world around them robots constantly change. The paper provides a possible solution to the overall problem, tested in a US NAVY simulator. One other notable task planning paper [33] focuses on the high-level social capabilities of a robot, i.e. the ability to collaborate with humans. The propositions from this paper could be interesting to test using the mission planning system created during this thesis.

One thing to be noted is that some of the latest DARS symposiums have been focusing a lot on robotic swarms, implying that a lot of the publications brought forward in the most recent symposiums have been performed on decentralized homogeneous multi-robot setups. This is different from a heterogeneous setup, in that all of the robots in a swarm are the same type of robot. This suggests that their relevance needs to be assessed on the other topics of the publication and whether the topics can be easily compared or transferred to a heterogeneous setup. Publications [34] [35] [36] present some robot swarm topics, but these topics can be considered as multi-robot topics because the robots being either homogeneous or heterogeneous does not matter to the topic. [34] discusses the scalability and fault tolerance of robot swarms. The paper starts out stating that swarm scalability and fault tolerance is often taken for granted, saying that this might not always be the case. The paper argues that if a single robot fault or breaks down in a robotic swarm, then this one issue can be detrimental to the entire mission of the swarm unless the swarm is prepared to handle such a problem. The paper states that increasing the size of a homogeneous swarm will increase the possibility of faults and breakdowns, which would be detrimental to the swarm.

Another paper on swarms is [35], and it provides several different solutions for communication within a swarm of robots, helping the robots provide each other with information and solve missions together. The proposed solutions are based on the type of communication devices that are available for robots like infra-red, WiFi, Bluetooth, etc. These solutions will work regardless of the type of robot, meaning they can be heterogeneous as long as they possess the same type of communication device. [36] uses machine learning to create more complex task solving using swarms. By using supervised learning as a tool for teaching the swarm how to act and react to different scenarios, one could create a multi-robot setup that could be completely decentralized and self-sustaining. The paper provides one possible solution on very simple robots, which could be extended to different setups.

DARS conclusion DARS provides good insight into many different aspects of distributed autonomous robotic systems. This will be a good resource for many different parts of WiRoM, especially for more specific solutions to problems that are presented when using robots.

2.5 Research methodology

In the introduction, we presented the method of the thesis as a *case study*. This section will define case studies and resolve how the research process should be conducted as a case study in software engineering.

The paper *Research Methods in Computer Science* by Serge Demeyer [37] proposes that the dominant method of research in software engineering is case studies. We have already seen one definition of case studies by Yin

[2] in Section 1.5. Demeyer mentions that case studies are dominant in the software engineering domain, but claims that the term *case study* might be used a bit too liberal because the extensiveness of research might not be what one would consider as a proper case study. Whether this thesis is eligible to be conducted as a case study was assessed as follows.

The research questions for this thesis were laid out in the introduction. The main research question and both of the sub-questions are all *how*-questions (see Section 1.4), meaning they want to get an answer to how phenomena occur. Each question focuses on the topic of one issue to a specific case, within the respective context. In Yin's book about case study research [2], he explains that "*how* and *why* questions are more explanatory and likely to lead to the use of case studies, histories, and experiments as the preferred research strategies. This is because such questions deal with operational links needing to be traced over time, rather than mere frequencies or incidence." Yin adds to this by stating that collecting data without affecting the behavioral events and examine the contemporary events while trying to answer *how* and *why* questions will indicate that the research is being conducted as a case study. What this means is that the subject of study (in this case the users of the system) should not be manipulated by the researcher when collecting data, but rather observed and interviewed afterward. This means that the way we collect data for the research in this thesis will be a determining factor for deciding whether we are conducting a case study or not.

Collecting data for research is usually done in one of three approaches: qualitative methods, quantitative methods, or a mixed approach. Creswell [38] describe these approaches as: - Qualitative research is an approach for exploring and understanding the meaning of individuals or groups ascribe to a social or human problem. - Quantitative research is an approach for testing objective theories by examining the relationship among variables. - Mixed methods research is an approach to an inquiry involving collecting both quantitative and qualitative data, integrating the two forms of data, and using distinct designs that may involve philosophical assumptions and theoretical frameworks.

Answering the research questions proposed in this thesis will require more than one source of data to increase the validity of the answers, which is also normal when conducting case studies. To answer the research questions (Section 1.4), we will have to see if the prototype that was developed using these concepts works, and how well the system as a whole work in practice.

This can be done by observing the system in its environment, looking at the types of missions that can be solved by it. It can also be done by observing and interviewing users of the system to gather data about how well the system is. Two different approaches to gathering data have therefore been proposed to get the data that is needed to answer the questions presented in this thesis: user testing and mission example.

This type of data collection is mostly non-numerical, having the system as the subject and missions and humans affecting the system. This method of data collection leans away from the quantitative methods and more towards qualitative methods. The research question is also more typical for qualitative methods according to Creswell [38]. He explains that *how*-questions are one of the main types of qualitative research questions. Having *how*-questions combined with one specific issue for one topic also strengthens the question as a qualitative-type question. Case studies are generally considered to use qualitative methods, but this depends on the case at hand. Some cases might use a mixed approach, or even lean towards quantitative methods. As mentioned earlier Yin [2] claimed that when dealing with a *how*-question, observation and interviewing is a central part when performing the case study, and this is exactly the desired method to perform data collection for answering the questions.

Research methodology conclusion Case studies are regarded as the most dominant research methodology in software engineering, and this thesis does not seem to fall outside of this assumption. This thesis provides *how*-questions as research questions, and the method for data collection is qualitative, meaning data will be collected through observation and interviewing. For this reason, our choice of conducting the research as a case study is eligible.

Chapter 3

Related work

3.1 Criteria

In this chapter, we review work that is related to the work in this thesis. The related work will focus mostly on the state of the art projects that use concepts that are similar to the ones presented in this thesis.

This thesis will not follow any literature review methodologies like Systematic Literature Review (SLR) or Systematic Mapping Study (SMS) [39] to find and review related work. The literature will be reviewed, but not systematically, which means we will be having a more loose (but defined) approach to locating relevant information and reviewing it.

Some search criteria and exceptions were defined for work related to this thesis:

Inclusion criteria: Robotics, simulation, DSL, heterogeneous, high-level, mission planning, MRTA, low-code

Exclusion criteria: Non-functional (no working example), homogeneous (exact same robot, not types of robots), older than 2010

Exceptions: Not all inclusions criteria need to be fulfilled in a single search because this will be too strict. For instance, using all of the inclusion criteria in the search query would give no results when searching in academic databases, because of low-code being such an infrequent keyword. It is, however, desired to cover all of the inclusion criteria using multiple different sources, and sources that cover more criteria are considered as more relevant.

Databases used for searching for relevant work are Google Scholar and the Bergen University Library (Oria, all sources).

3.2 Task definition language

WiRoM is inspired by an earlier project called the **Task Definition Language** (TDL). TDL is a high-level robot programming DSL that was created to be able to program heterogeneous multi-robot setups for a mission planning system [40] (GitHub page: [41]). The main focus of the TDL was to create some domain-specific language that defined the actions of robots and have them perform missions in a simulated environment. It also had a simple web interface that could control robots and start the simulation. TDL also provides a solution to the MRTA problem by using an auction-based algorithm (see Section 2.3).

TDL and WiRoM have a lot in common, and WiRoM can be considered as a continuation or a different take on the problems solved in TDL. The main difference is that WiRoM will put more focus on the higher abstraction levels. The users should not only be able to run missions from a web interface but also plan missions without the need of learning an entirely new language like TDL. WiRoM will also be aimed towards less experienced developers, and we will, therefore, put more focus on this part of the application, unlike the web interface implementation in TDL.

3.3 PROMISE

PROMISE [42] is a novel language that enables users to specify missions on a high level of abstraction for autonomous multi-robots. **PROMISE** stands for sim**P**le **RO**bot **MI**ssion **S**p**E**cification, and it is a high-level DSL developed as a user-friendly way to define multi-robots missions, while having well-defined semantics. PROMISE uses a combination of textual syntax and graphical syntax to plan missions. It is integrated into a software framework that allows executing the specified missions on simulators, as well as actual robots. Their work was illustrated using running examples and user testing.

In [42] they mention that Model-Driven Engineering is a core technology for robotic systems, emphasizing the DSL parts of MDE. The upper levels of the language are created such that they are platform-independent and highly customizable. The platform-independent models are compiled down to an intermediate language, which is then interpreted by robots. Such a system allows the specification of complex missions by providing executable tasks and operators.

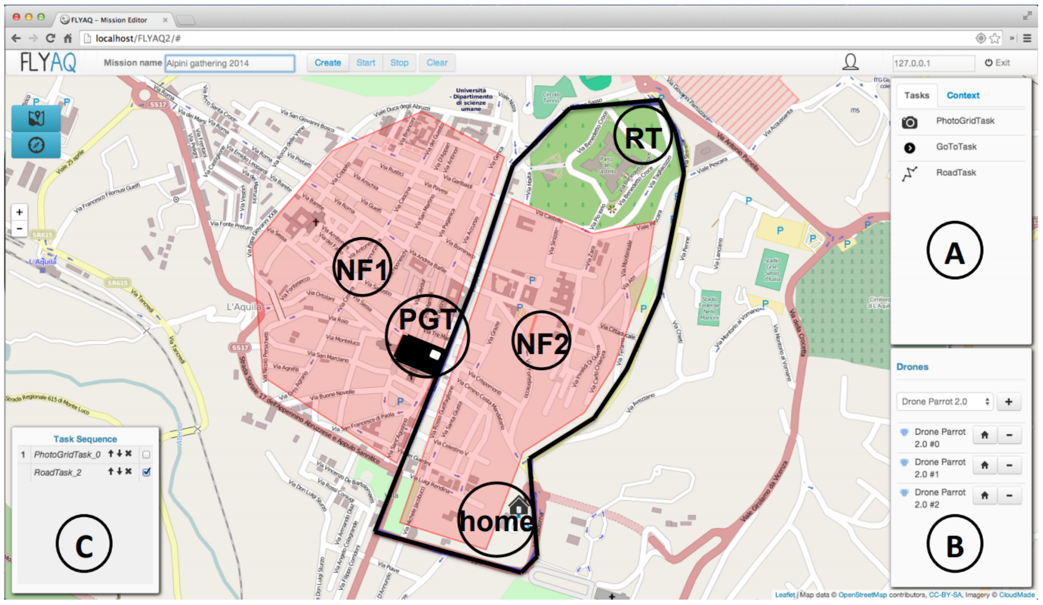


Figure 3.1: Screenshot from the FLYAQ mission planning interface

PROMISE is related to the WiRoM prototype in many different ways, as they both provide high-level heterogeneous mission planning systems. They both also provide the possibility of graphical planning and execution in simulated environments. The main difference is that WiRoM is going to be a web-based solution, using a low-code approach to the graphical interface. Much inspiration can be drawn from PROMISE when developing WiRoM.

The paper on PROMISE [42] was published by S. Garci et al. in October 2019, which is **after** the work on WiRoM had already started. This gives us an indication that the WiRoM solution is relevant to the problems that exist in the robotics domain.

3.4 FLYAQ

FLYAQ [43] was created as a mission planning system for allowing non-expert users to specify and generate missions using autonomous drones. FLYAQ provides a DSL for mission planning called *Monitoring Mission Language* (MML) using a graphical user interface as can be seen in Figure 3.1.

Planning mission using FLYAQ is done by specifying a mission using the modeling constructs provided by the language, then specify the context, and lastly the map (location) of where the mission should be performed. The mission is then represented using an intermediate language named *QBL*, and QBL models are interpreted during run-time by robot controllers. The generation from MML to QBL has been implemented using model transformation.

FLYAQ has become very popular and has been extended numerous times. In [44] they add support the high-level specification of adaptive and highly-resilient missions because one of the weaknesses of FLYAQ is that the missions are planned at design time, and wasn't resilient when facing unforeseeable or emergent situations. The paper [45] also discusses FLYAQ with the context of MDE and Mobile Multi-robots, where they also extended FLYAQ to be applied to underwater robots. Other work extending FLYAQ include [46] [47] [48] [49].

It was mentioned that the WiRoM project was inspired by TDL (see Section 3.2), and TDL used FLYAQ as one of its main sources for relevant work [40]. In TDL they had implemented a web interface, and this web interface was highly inspired by the FLYAQs web interface, using a map to define the mission. Similarly to FLYAQ, the interface in WiRoM will be developed as a web interface that can be used by non-expert users but will be able to support multiple types of heterogeneous robots from the beginning, allowing for focus on extending other parts of the system.

3.5 The ERGO framework

The **European Robotic Goal-Oriented Autonomous Controller** (ERGO) [50] is a space robotics project which is one part of six space robotic projects in the frame of the PERASPERA SRC. The objectives of these projects are to deliver key technologies for orbital and explorational missions within 2023/2024 (`https://www.h2020-ergo.eu/project/objetives/`). The main objective of the ERGO project is to develop a fully functional autonomous system that can solve missions such as Mars exploration using heterogeneous multi-robot setups, with the possibility of being able to operate at several different levels of autonomy [51].

ERGO uses a model-based approach to perform the Missions planning for the robots. The mission planner uses PDDL 3.0 as a standard language for the scheduling and planning community. PDDL stands for *Planning Domain Definition Language*, and this language was made to standardize artificial intelligence planning languages. PDDL is used for modeling the inputs (domain and problem), which allows the generation of suitable code skeletons, and glue code that can be combined with code developed by the user into an application's executable. Another part of the ERGO framework that uses a model-based approach is that the ERGO agent is conceived as a TASTE [52] component. TASTE is a tool-chain targeting heterogeneous embedded systems, using a model-based development approach, to let the user focus on the functional code while the TASTE tools are responsible for putting everything together [51] [52] [53].

The ERGO framework was field-tested in November 2018, in an environment on Earth which is similar to the environment on Mars, which was the Moroccan desert. The project ended after this achievement, but it is stated on the ERGO homepage [50] that the project will be continued in the second phase of the PERASPERA project [54].

ERGO and WiRoM are looking to solve similar issues using similar solutions. The ERGO framework provides a lot of new and state of the art technology that has been field-tested, and is, therefore, a good source for inspiration.

3.6 Lowcomote

Low-code related work was hard to come by, but we believe that **lowcomote** can provide some insight into the world of Low-Code Development Platforms.

Lowcomote has published an article [55], which discusses the pros and cons parts of the Low-Code Development Platforms, and shows its proposed solutions. Lowcomote is a project dedicated to highlighting and fixing some of the limitations of LCDPs. Lowcomote recognizes three main limitations that hamper the use of LCDPs: Scalability, fragmentation, and software-only systems. LCDPs are good for creating small applications, specifically software, using a defined toolkit that is tailored to a certain type of application or domain [55].

The article [55] mentions that LCDPs have been especially successful for the development of domain-specific applications in four market segments: database applications, mobile applications, process applications, and request-handling applications, with Internet of Things (IoT) being the potential fifth market in the future. The desire is therefore to extend beyond these markets, into other potential domains, both within the software engineering and other

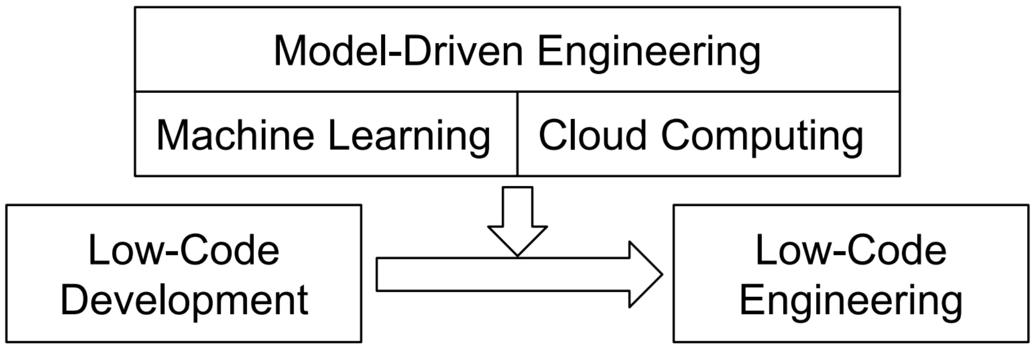


Figure 3.2: Lowcomote in a nutshell [55] (page 2)

engineering domains.

Lowcomote identifies as an Innovative Training Network, which means the project intends to train professionals to create new LCDPs where the limitations of older LCDPs are minimized, and the benefits are further enhanced. Lowcomote claims that these individuals will upgrade the current landscape of LCDPs to Low-Code Engineering Platforms. Figure 3.2 shows how Lowcomote intends to combine Model-Driven Engineering (MDE), Machine Learning and Cloud Computing with existing LCDPs to obtain the desired LCDPs.

This thesis will present a solution that can be considered as a Low-Code Platform for the heterogeneous multi-robot mission planning problem. It is desired for WiRoM to follow some of the principles presented in Lowcomote, to become a modern, well rounded, and scalable solution to the problems presented in the thesis. This is to make sure that the research is relevant and the project can be utilized or extended by other actuators in the future.

3.7 Additional work

Other related work was also reviewed, but will not be discussed as thoroughly, because they are not as related to this thesis, but are still relevant.

The Buzz language [56] is analogous to TDL, because it is also a DSL used to program heterogeneous robots. This DSL was created to be a novel programming language for large robot swarms containing many different robots. It is built upon other popular robot programming technologies.

RobotML [57] is a Robotic Modeling Language. RobotML is made to ease the design of robotic applications, simulation, and deployment to multiple target execution platforms. RobotML provides a high-level solution to programming heterogeneous robots, by using abstraction to hide lower-level details.

Robotbenchmark [58] is a web-based robot programming learning tool that allows users to execute robot controllers in a simulated environment straight from the browser. Robotbenchmark acts as a learning tool, offering a series of robot programming challenges that address various topics across a wide range of difficulty levels.

A. Hussein and A. Khamis presented in [59] a market-based approach used for solving the MRTA problem. They propose an approach that will be used to find the best allocation of heterogeneous robots to heterogeneous tasks, in the context of Multi-robot Systems (MRS).

PsAIM was presented in [60] as a proposed pattern catalog for specification patterns in robotic systems. PsAIM is a tool that uses these proposed patterns to assist developers in designing complex missions, such that they can be executed.

Additionally, we have the **Robotics DSL Zoo** [61] created by Nordmann et al. provide a list of available DSLs that have been developed for programming and controlling robots of all forms and shape. The index on their website provides all the publications in the Robotics DSL Zoo, as well as the paper [61] where they provide the survey on these publications.

3.8 Related work conclusion

There exist many different projects that are related to the work in this thesis. TDL, PROMISE, FLYAQ, the ERGO framework are all some of the most relevant projects to look at for inspiration when implementing the application. Lowcomote will assist in developing the high abstraction levels as a low-code platform, bringing abstraction to even higher levels than most we have seen. This means languages like TDL or others mentioned in the Robotics DSL Zoo could be applied at the lower abstraction levels when programming robots using WiRoM. Other related work was also presented to provide a solid foundation.

Chapter 4

Design and Implementation

4.1 Technological requirements

Defining a set of requirements for choosing the different technologies will assist in the process of rationalizing the choices of technologies for WiRoM. It's ideal to locate the technologies that would fulfill the requirements and provide an efficient development process when developing WiRoM

Firstly, we need to define some requirements based on the challenges that were described in Section 1.2. The following technological requirements are presented for this thesis:

R.1 A high-level programming language with a framework and/or library for programming robots being applicable by a viable simulation tool.

R.2 A powerful simulation tool with a wide repository of robots and support for heterogeneous setups with feature support for WiRoM, with a minimal gap between simulation and reality.

R.3 A powerful and simple to use web application framework with good library support for developing a high-level low-code platform web interface.

Having defined these technological requirements, the next section will be dedicated to reviewing the technology that is needed to fulfill these requirements.

4.2 Technological review

The initial programming framework proposition was to use something like *Johnny-Five* [62][63] or *Cylon.js* [64] to program the the lower levels of the robots. Both are based on *JavaScript* and *Node.js*, [65] which means they are high level libraries, have good platform support and are simple to use. The problem with these frameworks was that there was no obvious way to simulate the missions with multiple advanced robots in a simple and inexpensive way. Therefore they were dismissed, and it was decided to look at requirement **R.1** and requirement **R.2** the other way around, beginning with finding a compatible simulation tool rather than frameworks and programming languages. Hence, it will be presented the same way in this thesis, starting with requirement **R.2** then requirement **R.1**, before moving on to requirement **R.3**.

4.2.1 Requirement 2: Simulation software

We have already discussed some of the benefits of using simulations in section 2.1.4. Looking into several listings of robotics simulation tools from different sources such as [66] [67], three different simulation tools stood out: **V-REP, Gazebo** and **Webots**. All three would be feasible for requirement **R.2**, as they all support heterogeneous setups and have large robot repositories while having the possibility of using higher-level languages to control robots within the simulation. Webots already stands out, as it is the simulation tool that was used to create robotbenchmark, the web-based robot programming learning tool we briefly mentioned in Section 3.7, meaning this simulation tool is already applied in practice as a web application. Webots also support a wider variety of programming languages than Gazebo, and has superior documentation compared to V-REP [1], making Webots the preferred simulation tool for this thesis.

Webots is a graphical robotics simulation tool that has been used by the industry and universities for research and education purposes ever since the beginning of the software. The Figure 4.1 below show an illustration from the *Webots reference manual.* The development of Webots started in 1996 by Dr. Olivier Michel at the EPFL in Lausanne but was taken over by Cyberbotics Ltd. in 1998. Cyberbotics are still developing Webots, updating the software

[1] This has to be taken as a subjective remark: I found it much easier to find information in Webots' documentation compared to V-REP's when I was doing research about the two

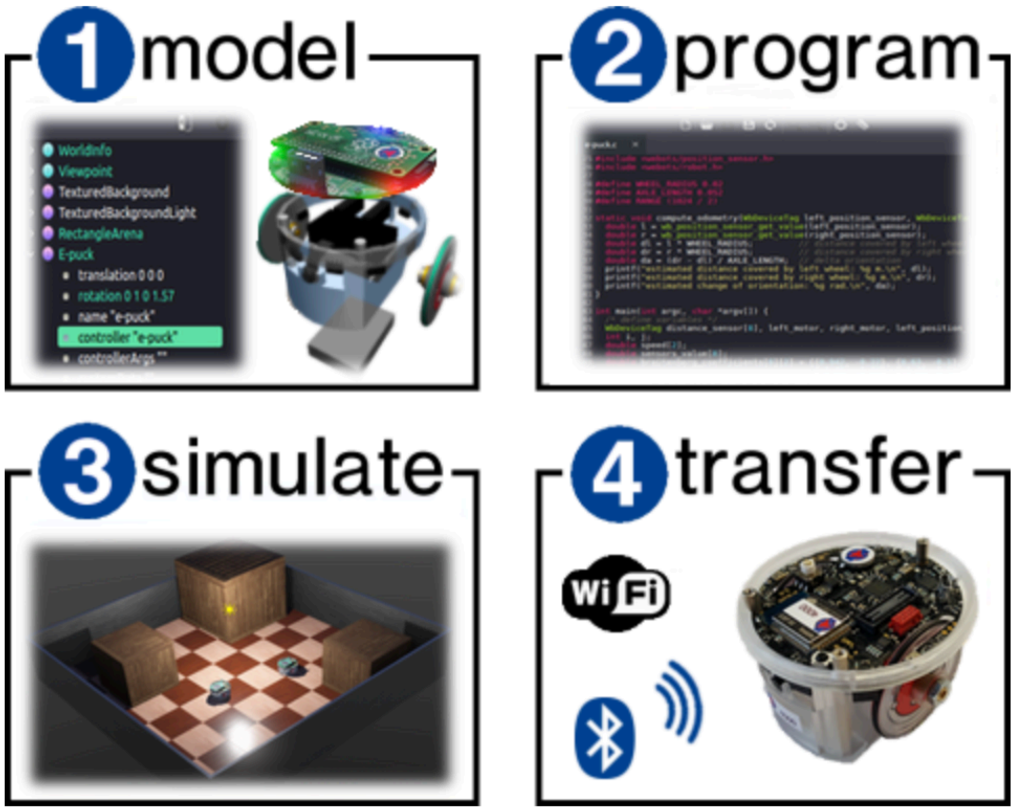


Figure 4.1: Webots illustration from the Webots reference manual: `https://cyberbotics.com/doc/reference/index`

several times a year, continuously adding robot models and features to the software. Webots was made open source after 20 years as proprietary software in the R2019a update. The codebase was then published to GitHub [68], hoping to make the software more accessible for research advancement [69].

Webots has good world-building and robot customization GUI, where one can interact with and get information about all the different parts of the simulated world. Because of robotbenchmark (Section 3.7), we also know that *Webots* supports streaming directly to the browser, meaning that this specific simulation tool can be integrated into the application, avoiding the need to interact with the simulation tool when executing missions.

This thesis will stick to simulations, being well aware of the gap between the simulation and reality. Luckily *Webots* has good support for bridging this gap as well. It is possible to cross-compile code to some supported robots, or create a remote control plugin for robots which are not compatible with cross-compilation at this point in time. This is mostly for future work, as we will focus on the simulated environments in this thesis.

4.2.2 Requirement 1: The programming language and framework

Using a high-level GPL/DSL will be the best approach for this project. Some of the ambitions for the heterogeneous system is that it also should be scalable and easily replaceable with similar setups and tools at the lower levels, otherwise it wouldn't be a proper abstraction, just a hard-coded robot controller created for one specific system and setup. The choice of robot programming method and tools is therefore an ambiguous choice, so this choice will be more based on the power and features of specific tools that can be applied for this system, in addition to personal experiences and preferences.

There are many languages that fulfill the requirement **R.1**. It is therefore desired to prioritize the efficiency of development when choosing a language and framework. The robot programming itself will not be the main focus of the thesis, we will focus on bringing robot programming to higher abstraction levels, but having good functions programmed will nonetheless be crucial for assessing the benefits of WiRoM.

As mentioned, *Webots* fills the technological requirement **R.2** for the project by having a diverse robot repository [70], with the support of heterogeneous multi-robot setups. Webots also has it's own high-level robot programming framework, called the controller framework (also used in robot-benchmark), which can be utilized by multiple different languages like *C, C++, Java, Python, MatLab* or technologies like the Robot Operating System *ROS* [71]. This is a simple framework that allows the user to control and program robots using a wide variety of packages that are included in the framework. The fact that the framework has multiple different language options open up for users with different amounts of experience with different languages to explore robot programming.

Python is one of my most preferred languages, because of previous experience with the language. *Python* has quickly become one of the most popular programming languages according to the popular online community Stack Overflow, as can be seen in Figure 4.2 which was posted in a blog on

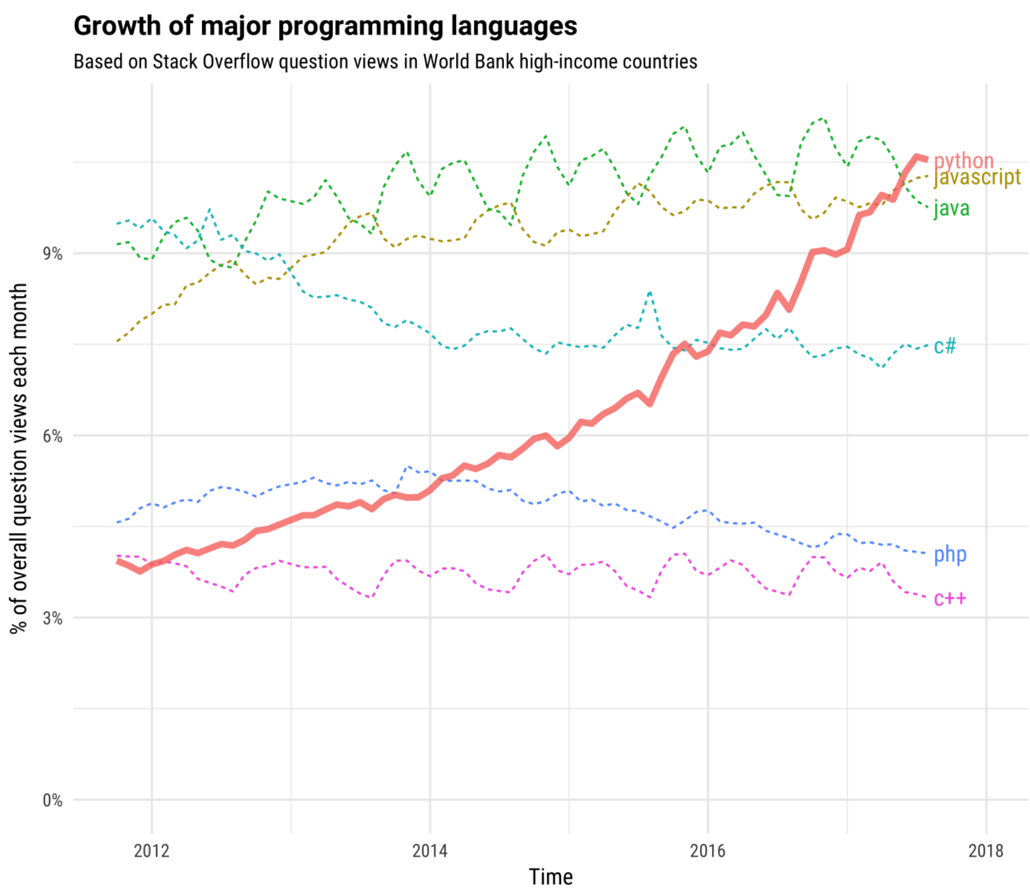


Figure 4.2: Python growth over time presented by Stack overflow [72]

their website [72].

Python is powerful and quick to use for development with its simple syntax and wide repository of libraries and frameworks at hand that can be utilized. *Python* fits the requirement well, being efficient and powerful to use for the developer. Using the Webots controller framework with *Python* fulfills the requirement **R.1**, and its additional benefits and previous experience with the language makes it the chosen programming method for the actions performed in the simulation [72].

4.2.3 Requirement 3: Web interface technology

WiRoM should provide a user-friendly interface that is intuitive for new users with varying amounts of experience with robotics and/or programming. Making sure that the web interface is created properly is, therefore, one of the main interests when developing WiRom.

There are a lot of different options to use for creating the web interface since a lot of frameworks will fit the requirement **R.3**. Just like selecting a language and framework for requirement **R.2**, a lot of it comes down to personal preference, but unlike requirement **R.2**, I do not possess a lot of preferences for which framework to use (not enough to make a preferred choice). Some research will follow to find out which framework that fits our requirements the best.

E. Wohlgethan published the paper [73] in 2018, comparing the three major *JavaScript* frameworks used today: **React, Vue** and **Angular**. The paper concludes that the difference is very small between the three, but some of the frameworks are better than others for certain projects. React seems to be the one that sticks out by being the most popular, both supported by the industry and the community, having the most downloaded packages, and highest presence in jobs and forums. Having a lot of good packages available will be a benefit when creating the GUI and visualizing different mission planning components. React is also component-based and has good state handling, which makes it powerful and easy to use for large applications where the state of the components is important. This makes it simple to store missions and missions states, splitting up functionality to different components.

Considering the points brought up in [73], *React* is considered as the best choice for developing the web interface of WiRoM, as it fulfills the requirement **R.3** better than the main competitors *Angular* and *Vue.*

Technology conclusion Locating technologies was not a simple assignment. Selecting the simulation tool and robot programming framework required a lot of documentation review (especially for simulation tools), and we ended up finding *Webots* and *Python* as the technologies that best fit the requirements. *React* has shown to be the best choice for creating the web interface, but there will always be a challenge when developing using new technologies.

4.3 Development process

We want to briefly present the development process and underlying technological choices that were done in order to erase any doubts surrounding these parts of the thesis.

No specific methodology was selected for going forward with developing the system, but some form of *Scrum* and *Kanban* combination was used in practice. A *Trello* board was used to keep control of the tasks that were supposed to be done in the project. The work was also divided into *two-week sprints*. This meant that at the end of each sprint (every two weeks) the developer met with the supervisor to go through the work of the previous sprint and they planned the next sprint until the next meeting.

All of the technological choices were made based on my own opinions and experiences, combined with the requirements presented in Section 4.1.

We use *Mac OS* as the operating system, meaning the project will be optimized towards Mac OS. We also have access to Windows, so the system will be tested by on that Windows to make sure that it works. *Visual Studio Code* was used for developing the project because it fits well with both React.js (JS in general) and Python. This also allows for using the same editor for both of the codebases. *Git* was used as version control of the system, where *GitHub* [74] was the choice of Git tool, and not bothering with branching because of this being a solo-project.

4.4 Mission abstraction levels

The missions will be structured in such a way that the functionality is distributed over multiple levels of abstraction. Defining each level of the mission planning system and examining its purposes will help to develop a better solution of the system, while also contributing to the analysis found in the thesis.

Figure 4.3 shows a meta-model of the relations between the different components at different abstraction levels of WiRoM. The dotted line separates the levels of abstraction, and the higher abstraction levels are on the left, while the lower abstraction levels are on the right. The lowest abstraction level contains the **simpleaction** and **robot**, the middle abstraction level is the **task**, and the highest abstraction level is the **mission**.

Simpleactions and **robots**, can be seen all the way to the right in Fig-

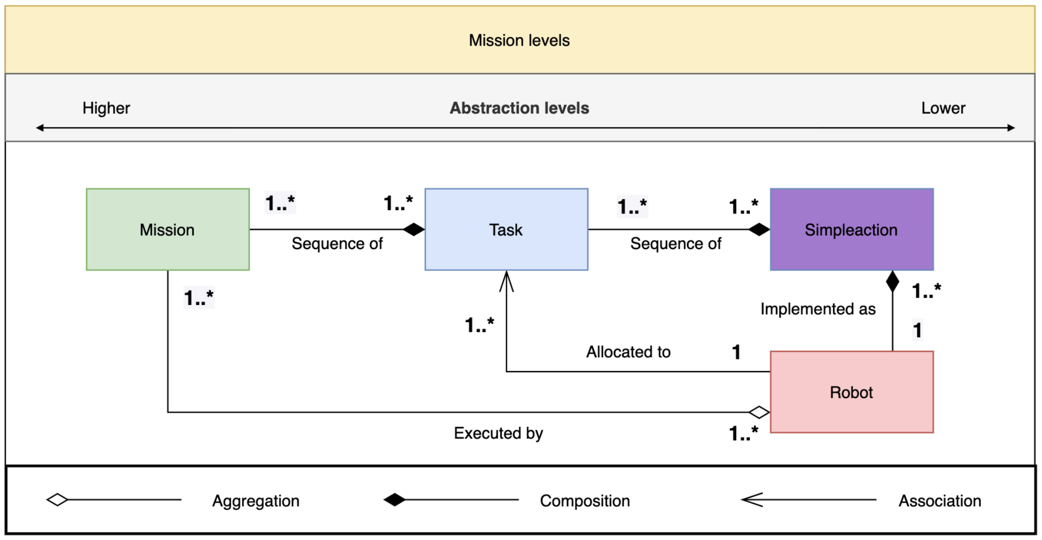


Figure 4.3: Meta-model of the abstraction levels and relations between missions, tasks, simpleactions and robots

ure 4.3. A simpleaction is a basic function or action that is implemented for a robot, that can be used to execute some behavior for that robot. Simpleactions can range from very basic functionality, e.g. *blink light*, to more advanced, e.g. *go to location*. Some simpleactions are also programmed only to enable some static functionality for a robot e.g. *enable collision avoidance*, such that the robot will now have collision avoidance on when moving around the environment. The simpleactions that are available in the system would depend on which robots are added to the system, and how their functionality can be implemented as simpleactions.

The mission planning system will come with a set of existing robots and simpleactions but can be extended by a developer to provide more functionality. When adding a new robot to the system which has not been previously applied, one would need to implement all the new simpleactions for that robot's functionality. One benefit of using simpleactions is that it works in the same way as an Application Programming Interface (API), meaning that a developer can use any language and framework to program that simpleaction, as long as the function name is properly defined. For instance, a simpleaction called *go forward* can easily be converted to call for a function in any programming language for any robot, as long as it is implemented as a function that makes the robot go forward. This could be added either using the same naming or a default naming scheme (e.g. in python: *go forward* would be parsed to *go_forward()*) or manually create some relation from a simpleaction in the code to the simpleaction in the web interface.

On the level above the simpleactions, we find the **task**, as can be seen in the middle of Figure 4.3. A task is in practice only a sequence of simpleactions that a robot can perform. Each of the simpleactions is executed in the order of the sequence they are defined in. For instance, if we have a task called *scout room* it could consist of the simpleactions: *go forward*, *turn left*, *go forward* and *turn left*. This task would make the robot go around in a room executing one and one simpleactions in the given sequence, such that the robot can scout a small area. In Section 2.3.2 we defined the MRTA problem for this thesis as an **XD[ST-SR-IA]** problem, meaning that one task can only be allocated to one robot, as can be seen in Figure 4.3. One robot can be allocated to multiple tasks, but not with simultaneous execution or forward planning, but dynamically, one task at a time. In order for two tasks in a mission to be executed simultaneously, we would need two different robots, each allocated to one of the two tasks.

The final and highest abstraction level is the **mission**, found to the far left in Figure 4.3. A mission is a sequence of tasks that are to be executed by one or multiple robots. For instance a mission called *scout and fetch sample* could consist of the tasks *scout area* and *fetch sample*. *Scout area* and *fetch sample* are sequenced tasks, which again consist of sequenced simpleactions to perform the mission. One robot would scout an area looking for a sample, sending the location of the sample to another robot which will then go out and fetch that sample.

Missions and underlying tasks and simpleactions are intended to act at such a high abstraction level that they can easily be used by inexperienced users and changed or modified by developers and domain expert users, while also being practical and efficient, as is a part of the research questions (Section 1.4) that are asked in this thesis. The system will also provide the users with default missions, tasks, and simpleactions that can be modified or used for other missions.

4.5 System architecture and design

Developing WiRoM will require different layers of design and programming. Modeling the systems architecture and design will help both developers and other users to understand how the system works and is implemented. The system is shown using two models, one more general model of the architecture of the entire system (Figure 4.4), and one model that shows the design of the mission planning web application, and how the missions, tasks, and simpleactions interact with the system (Figure 4.5).

Figure 4.4 shows the *overall architecture* model. In this model, each pointed arrow represents levels of abstraction, each abstraction level pointing towards the abstraction level it acts upon.

At the top, we find the **web application interface**, which contains the *mission planner* and all its components, as well as the *simulation stream.* The simulation stream is separate in this case because this is a specific extension of the selected simulation tool (Webots), which is why it is directly connected to the simulation software.

The two next levels are the **Python server** and the **robot programming framework**. The Python server receives all messages from the web application interface, processes them, and forwards them in the right format to the robot programming framework. The robot programming framework is in this case the Webots controller framework (also Python, see Section 4.2.2).

Below the robot programming framework, we have the **simulation software** and **middleware/ native API, robot/ hardware and components**. This thesis only focuses on using simulation software (Webots). The simulation software is technically much higher abstraction level than something such as the hardware, but in this case, they are both controlled by some robot programming framework, which is why they are both at the same level. This level contains the robots, and these robots receive instructions from the robot programming framework and execute them within their environments.

One of the main purposes of raising the abstraction levels is that any given abstraction level is interoperable, meaning that if you select any level of the system, all of the levels it is pointing to can be replaced, without affecting the selected level. The only exception to this would be the simulation stream, because of the streaming clients' direct connection to Webots.

Figure 4.5 shows the **mission planner web application** architecture model. There are two different arrows in this model, *green* and *black.* Black arrows show *manual flow* in the system, i.e. parts of the system that need user interaction in order to act between them, while green arrows represent the *automatic flow* of the system, i.e. parts of the system that need no human interaction between them.

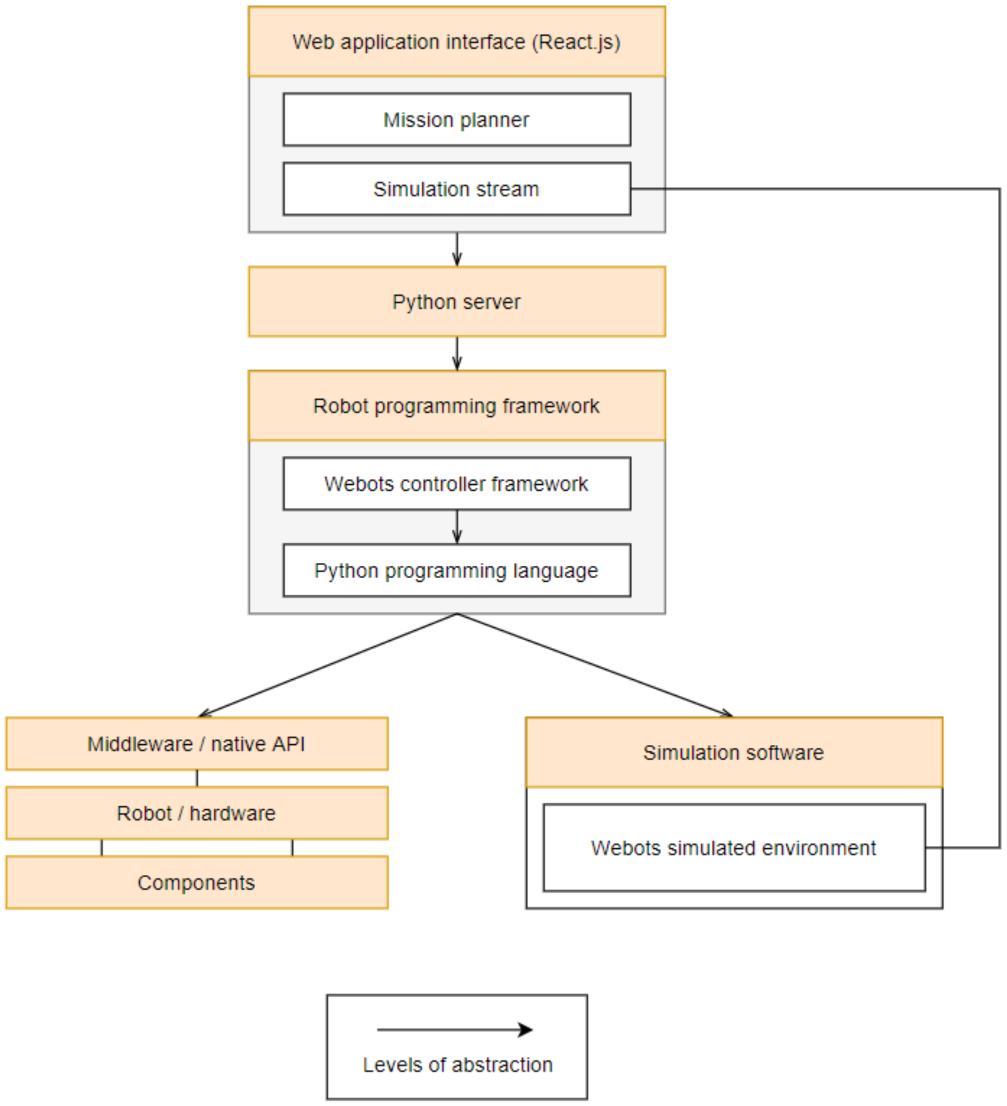

Web application interface (React.js)
Mission planner
Simulation stream
Python server
Robot programming framework
Webots controller framework
Python programming language
Middleware / native API
Robot / hardware
Components
Simulation software
Webots simulated environment
Levels of abstraction

Figure 4.4: Overall system architecture model

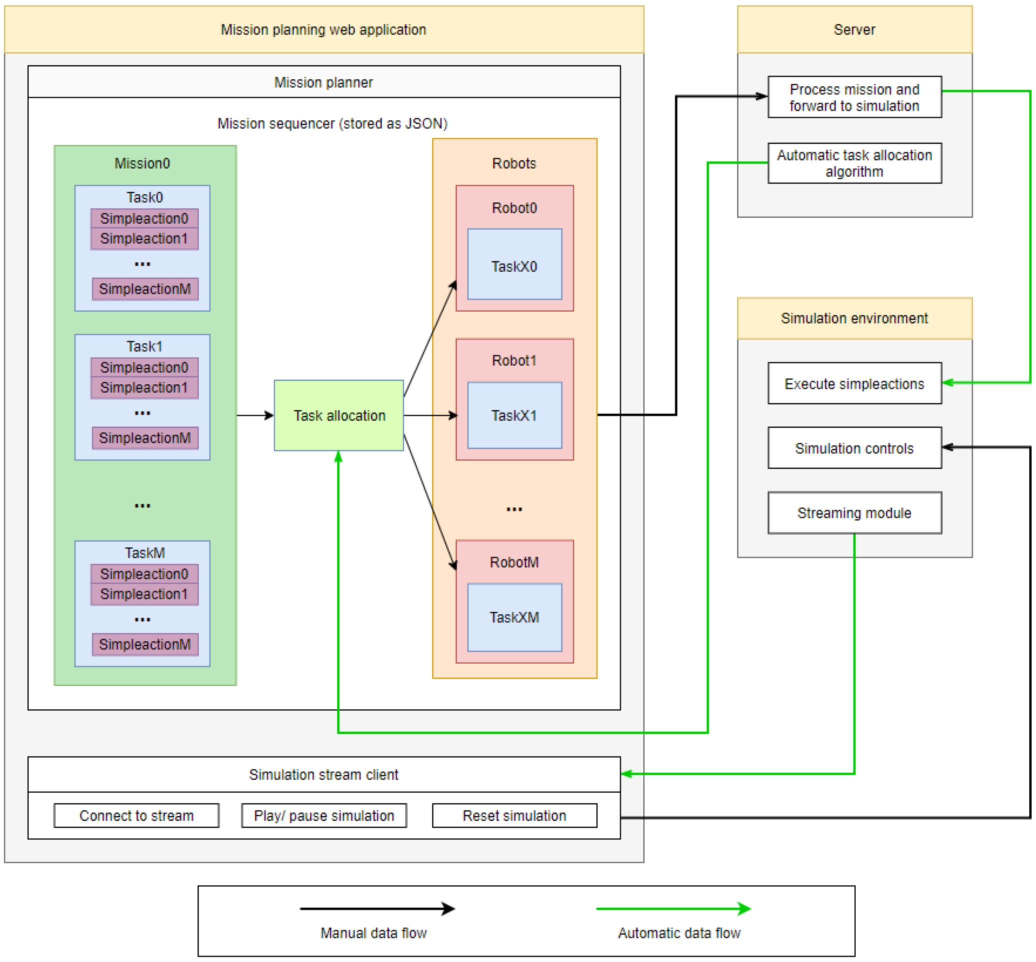


Figure 4.5: Mission planner web application architecture model

The mission planning web application is broken down into two parts, the **mission planner** and the **simulation stream client**. The mission sequencer is the graphical depiction of how a mission is going to be executed. The missions (green) contain a sequence of some tasks (blue) and each task contain a sequence of some simpleactions (purple). These sequences are created by the user in order to define how a mission will be executed.

When the mission has been created, the tasks can be allocated to robots using an automatic allocation algorithm that runs in the server, so when the user asks for automatic allocation, the mission is sent back and forth to the server. This allocation can then be checked and modified by the user if so desired before being sent. Each task needs to be distributed to a robot, and when all tasks are allocated, the user can start executing the mission. The mission will then be sent to the server, which processes the mission and forwards each task to their respective robots. These robots will then execute their tasks one simpleaction at a time in the simulated environment, in order to execute the mission as a whole.

Figure 4.6 shows the design of the mission planner in the web interface. This design was developed by applying the low-code principles and similar work discussed in Sections 2.2.5 and 3.6 with the system architecture that we have proposed (Figures 4.3, 4.4 and 4.5). Our design reflects the relations between mission, task, and simpleaction in an easy to understand low-code format. We have the missions on the top, while the task and simpleactions are below. The mission selection act as tabs, selecting a mission will show the task within that mission. The same goes for tasks, showing the simpleactions within that task. Each task in this case has already been distributed to a robot.

The simulation stream is a continuous stream directly from the simulated environment that can be viewed in the web application. This simulation stream can be controlled from the web application, making it possible for users to avoid having to interact with the simulation software when executing missions, they can simply plan missions in the mission planner and execute them directly from the browser.

Figure 4.7 shows the design of the simulation stream. This includes a player that is created by Webots, which the user can use to control the simulation. The stream also allows for changing the viewpoint in the same way as from in the simulating tool by dragging the screen around.

More about the details and implementation of both Figure 4.6 and Figure 4.7 will be discussed further in Section 4.9. The concurrency of robot

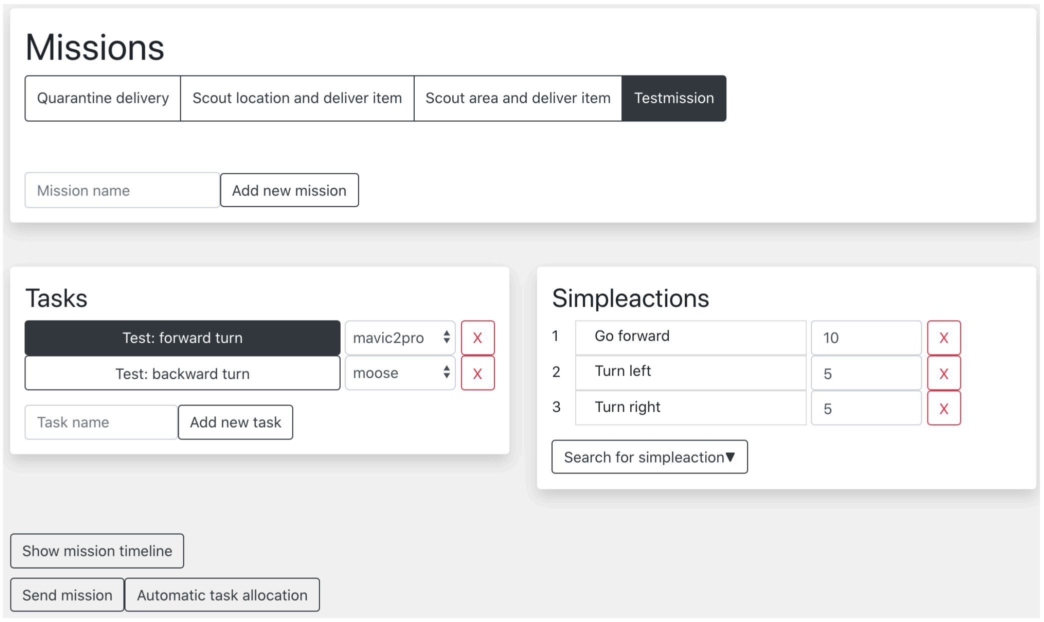


Figure 4.6: Design of the mission planner

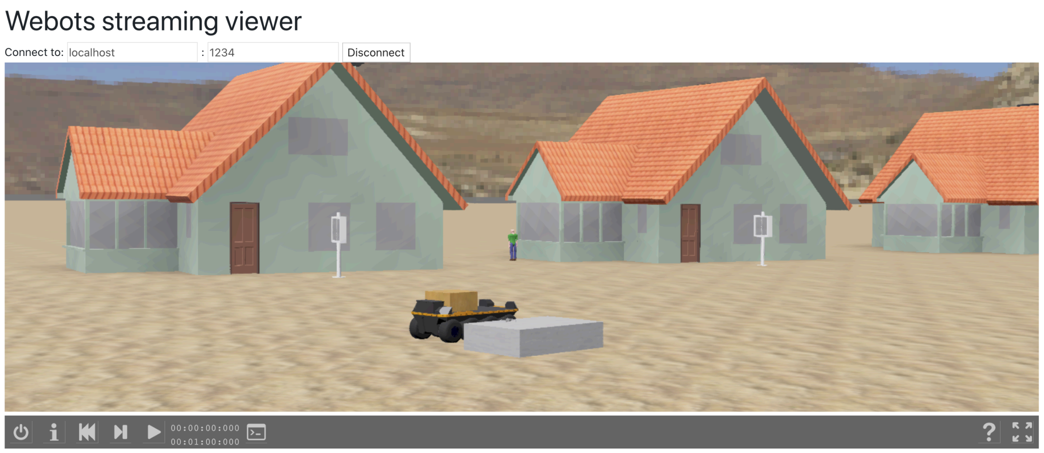


Figure 4.7: Design of the simulation stream client

execution will be further discussed in Section 4.7.

4.6 Simpleaction structure

Simpleactions will be one of the core components of the mission planning system. The name "simple actions" was a term that was used by D. Losvik in TDL [40] to define the lowest level of programming that needed to be done for robots, and in this thesis we simply put "simple actions" together to one word "simpleactions", to have a single name for the construct of the actions or functions that are executed by robots at the lowest level. The benefits and utility of the mission planning system grow in parallel with the number of robots with corresponding simpleactions that are available. This is why the system must support a wide range of robots and different development methods for simpleactions. The simpleactions can also be classified into different categories of simpleaction, where some are more specific for some mission, while others are more generic. All simpleactions will show up in the web interface as can be seen in Figure 4.6 and the relations and flow can be seen in Figure 2 and Figure 4.4.

Generic simpleactions Generic simpleactions are simpleaction that implies that they can easily be shared between different robots without the need for little (if any) additional implementation. Implementing an action like *go forward* would be a unique function to each robot, depending on the robots' set of motors, size, sensors, etc. But, when the function for going forward is implemented, one would only need to point to which function is to be executed when the simpleaction *go forward* is called from the higher abstraction levels (for levels, see section 4.4). Many simpleactions can, therefore, be shared between robots, as long as the correct simpleaction gets matched with the corresponding function for the robots.

The easiest way to define simpleactions is by looking at the given framework, and deduce which functions that are already exist and export them to WiRoM to be used as a simpleaction. The chosen framework for this thesis is the Webots *controller module*. This module contains a lot of prefabricated classes and functions for controlling the robots within the simulation, with a short route to a real-world robot implementation. This framework gives very easy access to most of the *nodes*, *actuators* and *sensors* available in the different robots. This process usually consists of importing different *modules* and calling their functions.

A weakness of the Webots *controller module* is that it does not include any API or algorithms for doing *path finding, flight stabilization, collision avoidance* etc. The framework has some innate algorithms for functionality such as *object recognition* for the cameras, but other than such exceptions, it is required to implement these algorithms manually. Luckily there exist a lot of samples, demos, and tutorials for Webots which use a lot of algorithms that can be easily be reproduced to create the desired simpleactions.

Specific simpleactions Specific simpleactions would consist of simpleactions which are too specific to be used generically, having to be applied to either a very unique robot or a specific assignment in a mission. A specific simpleaction can easily be as big as an entire *task* within a mission. Or in other words, it is when a problem is so complex and unique that using predefined simpleactions in a sequence won't be sufficient for solving the given problem, and the need for more specific implementation is needed.

One specific type of robot that would require a lot of different types of specific simpleactions would be robot arms. Robot arms would have to adjust a lot of variables for picking up different items. A simpleaction like *Pick up item* would in itself be very complex to program because one would need to move the arm to the right position, and then pinch the item with enough force to be able to pick it up. But then you consider that not all items can be picked up equally. As mentioned in the earlier example, a cup with liquid cannot be picked up in the same way as a bottle of liquid. An apple might be crushed by the force required to hold a steel ball. For this reason, one would need a lot of specific programming to make the arm work. One might be able to create simpleactions like *Pick up cup*, which would be specific for cup and could be reused with for the same type of robot arms, but may be limited for other robot arms because of different arm lengths, joints, etc.

The gray area Then there are also robot actions that are in the "gray-area" of the simpleactions, which are algorithms (like the ones mentioned above), communication, and synchronization of the robots. These implementations are often reusable, but some might be more specific and not as easily reused. They are not simpleactions themselves, because they won't be exposed to the mission planning system directly, and therefore lie a level below the simpleactions. They are still a part of (or used by) the simpleactions, which means that in some frameworks one would also need to implement some algorithms, communication, and synchronization. Different frameworks contain different functionality, meaning that there might be some framework where none of this is needed and one could simply implement simpleactions to be used.

Simpleaction structure conclusion We know that there are multiple different ways to develop simpleactions in WiRoM. Most robots would require some initial programming to set up the use of simpleactions, but some require more than others depending on their uniqueness and complexity, combined with the amount of API/functions and simpleactions available to the developer. Some robots or missions may require specific simpleactions which are more "hard-coded" to autonomously solve a specific problem, which is to complex for any task sequenced by generic simpleactions. There are **no** specific simpleactions developed for the missions that are part of WiRoM in this thesis, which means that all simpleactions that are mentioned can be considered as generic simpleactions unless something else is mentioned.

4.7 Robot synchronization

Robot synchronization is very important when working with multi-robot setups. We have seen that missions are defined as sequences of tasks, which again consist of sequences of simpleactions (Figures 4.5 and 4.3). It is therefore required to make it clear when and how the different simpleactions will be executed for each respective robot.

The robots running in Webots will run similarly to what they do in real life. Physical robots have their own respective hardware, and if two robots are given two separate tasks at the same time, they will execute them in parallel. Each robot runs a separate controller that is programmed by a developer. We want the simpleactions to be executed in a sequence, and the easiest way to do so is to let the robots have a queue. Each robot executes it's respective queue asynchronously by taking one and one simpleaction out of the queue. This means that each simpleaction in a queue has to wait for the previous simpleaction of that same queue to finish. Some simpleaction takes little-to-no time to execute, e.g. simpleactions that enable sensors, while others take more time, e.g. simpleactions that perform a movement. The ordering of the simpleactions matters a lot when creating missions. There is no use in enabling the sensors for a scouting mission after all the movement is done.

Robots can execute tasks in parallel, and this means that if we want to use two robots to solve a mission, and one robot depends on the actions of the other, then we need to program that robot to wait for some signal from the other in order to work. We need to have some communication between the different robots, such that they can synchronize with each other when a mission requires them to do so. It is, therefore, useful to implemented some simpleactions that actively wait and notify, such that this kind of synchronization is possible. There are many ways to do so, and using simulations would allow us to use things like the internet connection in order to provide a communication platform for any robot.

Showing the synchronization and execution of robots can be difficult with a low-code setup, without having to add a lot of details and complexity to the web interface (Figure 4.6). It will be useful to have some external graph or timeline that can be brought forward by the user to show the mission execution and synchronization between robots in a mission.

Now that we have discussed the design and architecture of the system, we can move on to the resulting system and the implementation of it.

4.8 Folder structure

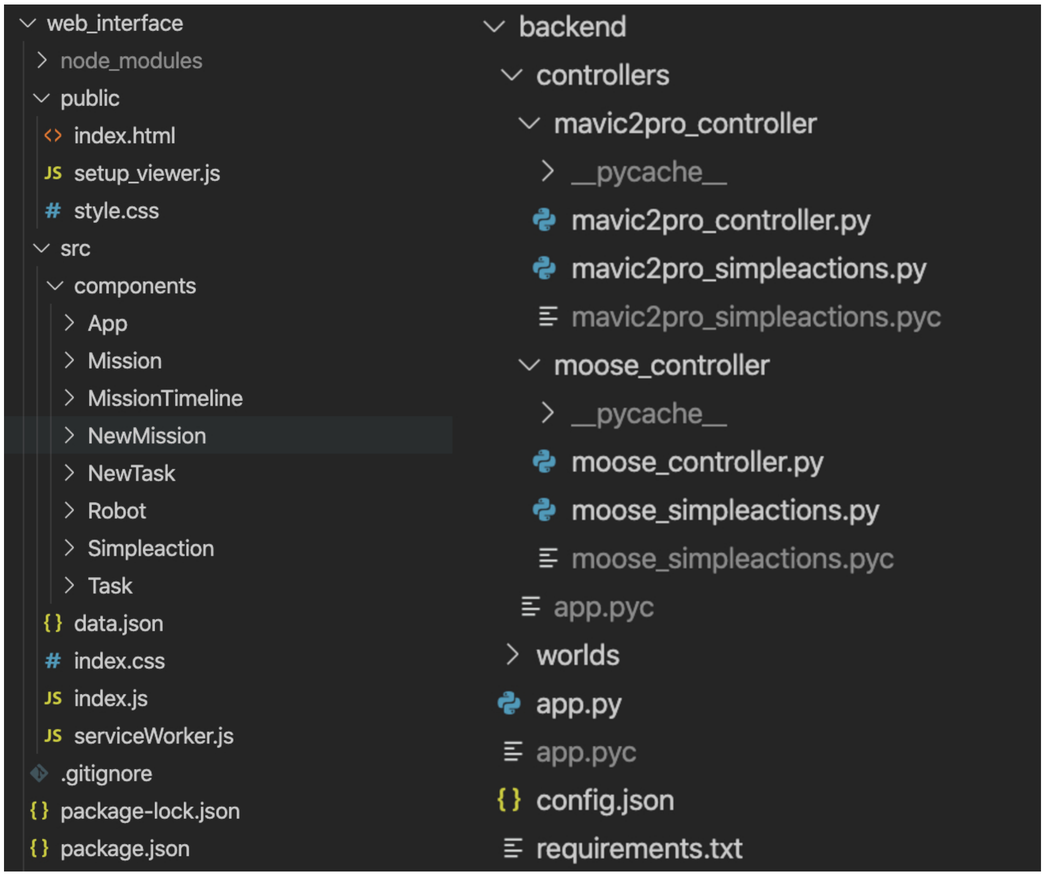
We start out by presenting the folder structure to show what the codebase looks like and how the entire project is structured.

Figure 4.8 shows the folder structure of the WiRoM project, which can be found at the WiRoM GitHub repository **WiRoMgithub** The left part is the web interface, while the right is server and simpleactions.

4.8.1 Frontend folders

The outer folder for the frontend is the *web_interface* folder. A standard React.js boilerplate called *create react app* [2] was used to initialize the project. In the web interface folders we find the *node_modules* folder, and the files *package-lock.json* and *package.json*, these are always present in any project that uses the Node Package Manager (NPM) and is used to store the packages that are used in the project. The folder *public* contain standard React files

[2] `https://github.com/facebook/create-react-app`



```
web_interface
  node_modules
  public
    index.html
    setup_viewer.js
    style.css
  src
    components
      App
      Mission
      MissionTimeline
      NewMission
      NewTask
      Robot
      Simpleaction
      Task
    data.json
    index.css
    index.js
    serviceWorker.js
  .gitignore
  package-lock.json
  package.json
```

```
backend
  controllers
    mavic2pro_controller
      __pycache__
      mavic2pro_controller.py
      mavic2pro_simpleactions.py
      mavic2pro_simpleactions.pyc
    moose_controller
      __pycache__
      moose_controller.py
      moose_simpleactions.py
      moose_simpleactions.pyc
    app.pyc
  worlds
  app.py
  app.pyc
  config.json
  requirements.txt
```

Figure 4.8: Screenshot of the folder structure for the project

like *index.html* used to show the website, this was added with the boilerplate as well as *index.js, index.css* and *serviceWorker.js.*

The folder *src* and the file *data.json* are the parts that contain the code for the system and the data used by it, as well as metadata about robots and simpleactions. React is divided into components, so each component that has been created has its own folder in *src/components* containing the .js and .css files for that component. Data.json is the file that stores all the data used in by the web interface. This includes information about the robots, missions, tasks, and simpleactions (this will be discussed further in Section 4.9). This information lets the user know what kind of robots are available and what types of simpleactions are implemented for them.

4.8.2 Backend folders

The outer folder for the backend is simply called *backend.* This folder contains a lot of information used by Webots, like the *worlds* folder and *robotname_controller.py* files. These files are used to define the simulation world and the controller that is connected to each robot. The files *robotname_simpleactions.py* contain the simpleactions for the respective robot, and these files are initialized from the controller files. *App.py* is the server of the system, and the file *config.json* contains the configuration specifications for how the server act on the information it receives (task allocation, task forwarding to robots). This also contains metadata about the robots and simpleactions, to know which robot can do which simpleactions (and the quality/cost). *Requirements.txt* contains a list of packages that are needed in order to make the system work. All files and folders that are grey are cache folders used by Python.

When new robots are added to the simulation or new simpleactions are added to any robot, one would also need to change the metadata in the *data.json* and *config.json* files, such that they are up to date with the current state of the robots and simpleactions. Changing up the robots or simpleactions (without changing names) will fix itself automatically and won't need to be updated in said files. Other parts of the system like missions, tasks, and mission environments will add new parts to them automatically. Having support for the same when adding robots or simpleactions is also a feature that can be added to the system for future work, to improve on the user-friendliness of adding new robots and simpleactions.

Changes to the web interface will be done mostly through the different components (in each component folder), changes to the server will happen in *app.py* and changes to simpleactions/robots are in the *robotname_simpleactions.py*) files. Some changes to the system would require making changes to the metadata in *data.json* and *config.json* files. These files contain the state of the system for the frontend and the backend, and it needs to be up to date with the robots and simpleactions in the environment.

Changes to the system that would require editing *data.json* and *config.json* include:

- Adding new simpleactions to robots (in their *robotname_simpleactions.py*)
- Adding new robots
- Changing a simpleaction such that the quality or cost is changed for a robot solving that simpleaction

Other changes to the systems source code (or changes to missions and tasks that can be made through the web interface) happen automatically and won't need any additional documenting to work. It is ideal for the system to avoid users having to edit *data.json* and *config.json* when adding new simpleaction and robots, by having some function read the scripts automatically or let users fill information into some form in the web interface. This has not been implemented in the system but is up for future work.

4.9 Web interface

The system is implemented as a **web application**, meaning one of its main components is the **web interface** that runs in the browser. Missions are planned in the web interface (at the highest level) by the user before it is sent for processing at the lower levels. The web interface communicates directly with a server, which in turn communicates with the simulation, as shown in the system architecture model in Figure 4.4.

4.9.1 Navigating the mission planner

Figure 4.9 shows a screenshot of the implemented web interface in its entirety. We have already looked at the mission planner design a bit in Figure 4.6 in Section 4.4, but now we go more in-depth on how it actually works.

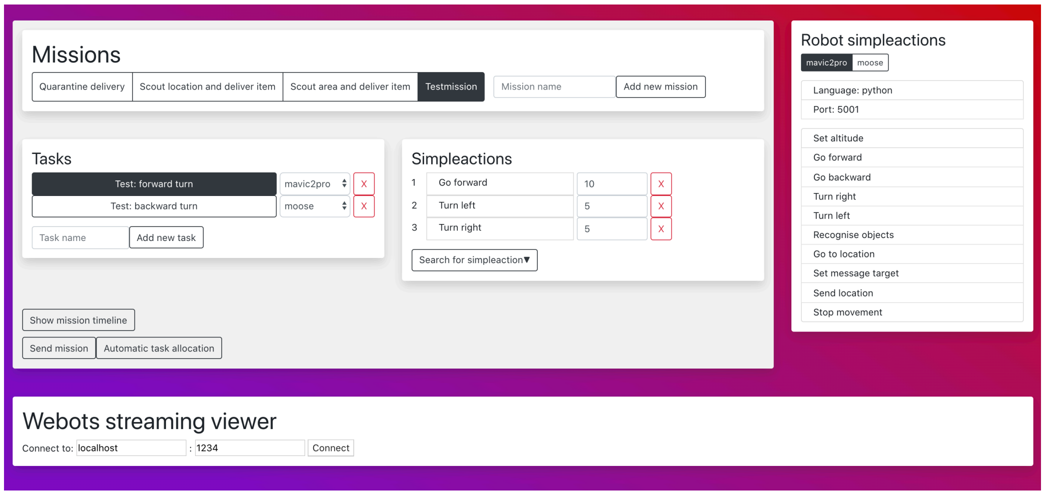


Figure 4.9: Screenshot of web interface

Missions are selected at the top, and **tasks** and **simpleactions** are below. Each task shows its name, then the name of the robot that will execute it, and next to the name of the robot there is a *delete*-button to remove the task if so desired. The simpleactions have numbers, which will indicate the *sequence of simpleaction execution*, next to the number there is the name of the simpleactions and the *arguments* it takes, followed by a *delete*-button for the simpleaction. Hovering over the argument box will show *information* on what the simpleaction wants in that field

The tasks and simpleactions are manipulated by **drag-and-drop**, changing the order of the simpleactions and tasks. This way it makes it easy and intuitive for users to change around the sequence of tasks and simpleactions. Adding a new mission can be done at the top by typing in the name of the wanted mission, and pressing **add new mission**. This adds a new mission with no tasks or simpleactions. Then adding new tasks is done in the same way, typing in a name and pressing **add new task**. This creates a new task with no simpleactions and no robot assigned to it.

The user can add new simpleactions either by searching for it using the **search for simpleactions** button or dragging and dropping a simpleaction from the *Robot simpleactions* list on the right. The *Robot simpleactions* list shows information for each robot in available, and the user can select a robot from this list, and then simply drag and drop simpleactions from this list to the simpleactions box to add them to the task. After selecting a simpleaction for the task, one can add appropriate arguments to it through the form in the simpleaction (some are disabled if they take no arguments).

The button **show mission timeline** will display the mission timeline for the selected mission. The mission timeline will be discussed further below.

The button **Automatic task allocation** will provide an allocation of robots to the current mission. The task allocation will be discussed further below.

At the bottom of the page, we find the *simulation stream*, and by pressing the button **connect**, it tries to look for a Webots instance streaming to *localhost:1234*, and if then show the stream if it finds it. Pressing **send mission** will send the currently selected mission to the simulation, and one can press play from the simulation window (see Figure 4.7) to execute the mission.

Further instructions and videos using the system can be found in the readme in the WiRoM GitHub repository [74]

4.9.2 Implementation

This web interface is developed using React.js, as was decided in Section 4.1. React.js is based on creating reusable components, such that one can add multiple cases of the same component without having to rewrite a lot of HTML code. React.js also uses states to handle the data in the system. Each component is written in JavaScript and returns HTML to show in the browser, and each component has a state that controls all the data within that component. The initial state for all of the components is based on data that is stored in the *data.json* file as seen in Section 4.8, and this data is brought in to the states when the webserver is deployed. This data will be stored in a database if the system is ever deployed.

React bootstrap was used to create the interface itself, in order to be efficient when developing the design of the web interface (see Figure 4.9). This can be improved upon a lot if we had some proper designer look at the application, but will be good enough for this prototype of the system. A lot of different libraries and packages were used to create the web interface, but going into the details of this is unnecessary and provides minimal content to this section. The main ones were *ReactSortable* for the drag-and-drop effect and *Dropdown* for dropdown buttons.

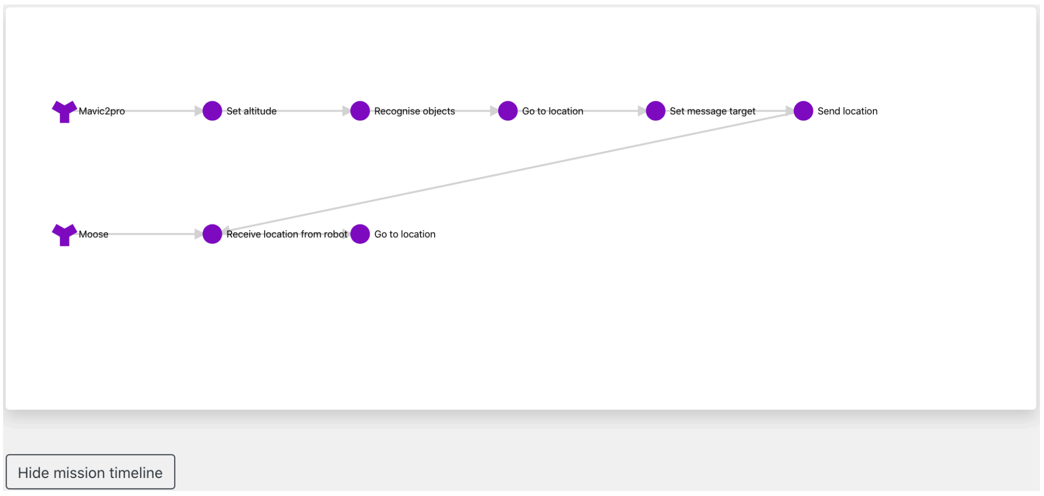


Figure 4.10: Mission timeline for a scouting mission

4.9.3 Mission timeline

As discussed in Section 4.7, the mission planner (Figure 4.6) might not provide good insight into how and where the sequencing of the simpleactions and the synchronization of the robots happen in the simulation. Providing a more detailed and visual representation of the sequence and synchronization would give the user a more clear view of how the mission will be executed.

Figure 4.10 shows the mission timeline that represents the mission execution and robot synchronization in the system, which can be seen from the web interface.

Every arrow in the graph represents a synchronization, meaning that a simpleaction can only start executing after all simpleactions pointing to it are finished. The arrows do not represent time, so the length of the arrows does not matter. The actual time it takes to execute each level (and the entire sequence) is unique to each robot. The first simpleaction of all robots will start at the same time, but the second simpleaction of each robot will (most likely) start at completely different times.

The mission timeline is a directed graph that is created using a library for React called *react-d3-graph*. This graph provides a representation of the order of the simpleactions in each task. The graph also shows which simpleaction that is notifying a waiting robot, to let it move on with its execution.

4.10 Server implementation

The server acts as the central part of the system, communicating with the robots in the simulation environment and the web interface (as seen in Figures 4.4 and 4.5). The server is a Python server, and most of its job is running *Flask*. Flask is an HTTP request handler that is used in Python, and it opens up a port for communication via HTTP, such that it can communicate with other entities. This Flask instance is what is used to communicate with the robots and web interface. The server uses the *config.json* file (Section 4.8) to get information about the robots and simpleactions that exist in the system. This information is then used to parse the missions and forward them using HTTP. The information is also used in the automatic task allocation algorithm, which is running on the server when called via HTTP from the web interface (Figure 4.9), which will be further discussed later.

4.11 Simpleaction implementation

We have already looked at the simpleaction structure in Section 4.6. Simpleactions are implemented as functions that manipulate global variables in a script that runs on each robot. This means that if we want a robot to *go forward* it triggers a function that runs on the robot that changes some variable called e.g. *forward_speed* to 10 (what 10 means depends on the robot and the developer would need to figure this out to set an appropriate speed). Each robot also has a main loop, which runs in a separate thread while the robot is running in the simulation. This main loop processes the global variables when they are changed by the simpleactions. For instance, the main loop would be responsible for applying the updated global variable *forward_speed* to the corresponding motors of the robot, such that it moves forward. The simpleactions are received and started via HTTP through separate Flask instances. This Flask instance and the main loop run in separate threads, but in the same process, meaning they are in the same scope and have access to the same global variables. This is why the main loop can run while the Flask instance changes a variable, and this variable then gets read by the main loop afterward.

The implementation of the robot synchronization as mentioned in Section 4.7 is done by utilizing the same Flask instances that were mentioned earlier. Flask also allows robots to communicate with each other, as long as we know their ports that are provided through Flask. All the ports of the robots are stored in the config.json file (see Section 4.8) and can be used by any robot in the system. The communication has also been created as simpleactions, meaning that the user needs to define in the mission when they want robots to communicate.

```
def moose_main():
...
    while robot.step(timestep) != -1:
        if navigate:
            navigate_to_location()
        for motor in left_motors:
            motor.setVelocity(left_speed)
        for motor in right_motors:
            motor.setVelocity(right_speed)
...

def go_forward(duration):
    global left_speed
    global right_speed
    left_speed = 7.0
    right_speed = 7.0
    if duration is not 0:
        time.sleep(duration)
        left_speed = 0
        right_speed = 0
...

@app.route('/location', methods=['POST'])
def receive_location():
    global location
    msg = request.get_json()
    location.append(msg['location'])
    return "Received location", 200
...
```

Listing 1: Code snippet from Moose simpleaction

Listing 1 show two simpleactions that are implemented for the robot called *Moose*. The first simpleaction show a *go_forward(duration)* function, which will go forward for some duration. It sets the global variables for the left and right motors to seven, then waits for the duration using *time.sleep(duration)*, before it resets the variables to 0. This way, we have made the robot go forward for *duration* amount of seconds.

The second simpleaction in Listing 1 is a communication simpleaction called *receive_location()*. This simpleaction is running in the Flask instance of the robot and catches an HTTP Post request, which triggers the function. This simpleactions lets the robot receive a message containing some location and then stores this location in its own global variable *location*. This location can then used to navigate using for instance a *go to location* simpleaction.

At the top of Listing 1 we see a snippet from the main loop, which continuously manipulates the speed of each motor for the wheels based on the global variables *left_speed* and *right_speed*.

4.12 Model transformation

Having everything stored in JSON-format makes it simple to use HTTP to send messages to the server via the API that is exposed from the server. The server received these messages and processes the missions before sending them to the robots (see Figure 4.4).

Listing 2 shows a code snippet from some of the JSON that is stored in *data.json*. This data includes information on the robots, which port they run on, which simpleactions the robots have and the language it is implemented in. The file also contains information about the mission, the tasks within it, the robot allocated to that task, and the simpleactions in that task.

```
{
    "robots" : {
        "mavic2pro": {
            "language" : "python",
            "port" : "5001",
            "simpleactions": [
                {
                    "name":"set_altitude",
                    "numArgs": 1,
                    "type":"move"
                }, # simpleactions ...
            ]
        }, # robots ...
    },

    "missions": {
        "Scout location and deliver item": {
            "tasks": [
                {
                    "name":"Scout location",
                    "id": 0,
                    "robot":"mavic2pro",
                    "simpleactions":[
                        {
                            "name":"set_altitude",
                            "args":"1",
                            "id":0
                        }, # simpleactions ...
                    ]
                }, # tasks ...
            ]
        }, # missions ...
    }
}
```

Listing 2: Code snippet from mission planning data stored in JSON format

The JSON object can be seen as a *model* of the data which is used by the system. The data that is shown in Listing 2 is used by the web interface to create the GUI, and contain more information than the server actually needs, and in a format that is hard to process. In order to process missions efficiently, we need some way to transform this model to code that can be executed by the robots.

One of the advantages of this project was using MDSE core concepts, and model transformation is one concept that can be used to raise the abstraction levels. The low-code model that is created by the mission planning system, shown in Listing 2, can be considered as a PIM, which we can transform to a PSM and further down to runnable code/ function calls, much like the concepts we discussed in Section 2.2.3. Listing 3 shows a sample of the Listing 2 transformed into an object where we only see the mission that the server will receive. Missions are sequences of tasks, which again are sequences of simpleactions, and these need to be structured in a way that can be processed easily by the server. The transformation from the model in Listing 2 to the model in Listing 3 is created by a simple method that loops through all the tasks in a mission and saves the information needed for each robot to execute the mission. The object is structured this way because we only want to send the information that is necessary for the server to parse it and generate the desired code, hence we do not care about the mission name or the task names in this object, only the robot configuration needed to parse the mission and simpleactions that the robot will solve in the mission.

```
"currentMission":{
    "mavic2pro":{
        "language" : "python",
        "port" : "5001",
        "simpleactions":[
            {"name":"set_altitude",
             "args":"1", "id":0},
            {"name":"recognise_objects",
             "args":"", "id":1},
            {"name":"go_to_location",
             "args":"[388, -365]", "id":2},
            {"name":"set_message_target",
             "args":"'moose'", "id":3},
            {"name":"send_location",
             "args":"", "id":4}
        ]
    },
    "moose":{
        "language" : "python",
        "port" : "5002",
        "simpleactions":[
            {"name":"receive_location_from_robot",
            "args":"", "id":0},
            {"name":"go_to_location",
                "args":"[]", "id":1}
          ]
     }
}
```

Listing 3: Code snippet of a scouting mission, in the format that is used by the server

The code generation itself is not a very advanced process. Each robot has a file *robotname_simpleactions.py* (Section 4.8), and each respective robot is running a Flask instance in said script, as mentioned in Section 4.11. When the Python server receives the JSON file as depicted in Listing 3, it packs out the simpleactions, and generate strings that resemble function calls. These strings are then sent via HTTP to their respective robot. The Flask instances running on each robot will receive these strings and add all the strings to a queue. This queue is then executed using the *eval()* function. This function accepts strings and executes them as Python code within the context that the function is running. This means that *eval("go_forward(10)")*, will run then function *go_forward(10)* as if it was written normally in the script.

Using function like *eval()* is generally recognized as *bad practice*. The reason why it is used here is that we want the robots to be able to receive missions dynamically, i.e. receive new missions at run-time, without having to redeploy. The code that runs within the simulation is not easily called from outside of the simulated context. A possible solution that wouldn't use eval() would be to have the robots share some global variables stored outside the simulation context, which can then be manipulated from the outside. The problem with this is that the robots must read the variable constantly for it to be interactive, making it a costly procedure, and it would also require a lot of time to produce a good solution (because of concurrency). *Eval* presents a simple and low-cost solution to the problem, even though it might be *bad practice*. Creating a better solution to this problem has the potential to be solved by some future work.

4.13 Multi-robot task allocation

Having performed the theoretical foundation for the multi-robot task allocation problem (MRTA) provided a proper label for our MRTA problem (Section 2.3.2). This MRTA problem was labeled as an XD [ST-SR-IA] problem, meaning the problem was a single-task robot executing single-robot tasks, allocated instantaneously via a centralized controller and with cross-schedule dependencies.

Algorithm Our approach wanted to use a centralized controller. This part was also discussed in Section 4.5 and naturally selected to be the Python server (Section 4.10) of the application.

The algorithm that will be used is a simple auction-based algorithm, much like the ones we have seen earlier. It calculates the bid for each robot based on its quality and cost for each simpleaction in a task. If the robot cannot perform a given task (e.g. it is lacking some of the simpleactions) it will be given a score of 0 (see Figure 2.1). The cross-site dependencies will also factor into the bid that each robot has for each task (see Section 2.3.2). The robot with the highest bid for each task (between 0 and 1) will be allocated to that task.

Algorithm 1 shows the pseudo-code for the algorithm used to solve this problem. The first loop is the first part of the algorithm. This starts by looping through all the tasks and then the robots, and if a robot can execute a simpleaction in a task we calculate the utility for this simpleaction and multiply it with the existing bid that the robot will give for that task. The function *calculateUtility()* subtracts the quality with the cost that the given robot has for the given simpleaction. The function *calculateUtility()* also calculate and factor in the eventual cross dependency constraints. If the robot cannot execute a simpleaction then the robot cannot execute any task containing that simpleaction, and the bid is instantly set to 0.

After this, the bid is added to the list of bids, under its task and robot. This is performed for each task, resulting in a list of all bids for all tasks by all robots. The second part is the function *allocateTasksToHighestBidders()*. This function is simply sorting the robot within the tasks based on the highest bidder, and then update the list of tasks with the new robot allocations and then return that list.

This is a prototype, the quality and cost estimates are only rough estimates done by us, and the algorithm is not optimal for solving this problem. The algorithm is called via a simple HTTP call to a *python* server, so changing or replacing the algorithm is as simple as routing the HTTP call to some other algorithm and make sure it returns the proper response.

Algorithm 1 Task allocation algorithm

```
function TaskAllocation(tasks, robots)
    $bids \leftarrow None$
    for $task \in tasks$ do
        for $robot \in robots$ do
            $bids.tasks \leftarrow task$
            $bids.tasks.robots \leftarrow robot$
            $bid \leftarrow 1$
            for $sa \in task.simpleactions$ do
                if $sa \in robot.simpleactions$ then
                    $utility \leftarrow calculateUtility(robot.simpleactions.sa)$
                    $bid \leftarrow bid * utility$
                else
                    $bid \leftarrow 0$
                end if
            end for
            $bids.tasks.robots.bid \leftarrow bid$
        end for
    end for

    $tasks \leftarrow allocateTasksToHighestBidders(tasks, bids)$
    return tasks
end function
```

Chapter 5

Users, cases and workflow

5.1 System cases and user roles

It is already mentioned throughout the thesis that the system should provide a high-level solution while being operable at all the different levels of abstraction (see Section 4.5 and 4.4). We, therefore, want to define some different use cases of the system, and different types of users who want to use the system. The system should be able to be used in different cases by users of different levels. It is also desired for the system to be as user friendly as possible when operating within all of the cases.

Four different cases provide the main use cases of the system. These four are the following:

Case 1 **Mission planning:** this case would require a user to perform mission planning in the web interface using all the available robots, simpleactions, and simulation setups. This would include creating new missions and manipulating existing ones.

Case 2 **Extending the mission setup:** a user would in this case be able to change the setup by adding or manipulating simpleactions, robots, and the simulation environment. This would require the user to program simpleactions for either existing robots or new robots, as well as change the configurations of the system for it to match the setup. A user with knowledge of the simulation tool could also use this to extend the simulation environment to something that they want to use for planning some specific mission.

Case 3 **Extending WiRoM:** this case would require more programming experience from the user. Extending WiRoM is in itself a multi-case, meaning there are many different ways the system can be extended. The user might be the domain-expert or researcher within some field, that would want to extend the functionality of the mission planning system. This could include adding things like new task allocation algorithms, machine learning, optimization, etc. or simply making the system better with adding new features.

Case 4 **Changing the low-level development technology:** this case would require some knowledge about programming and simulation tools from the user. The system should be able to let the user select a simulation tool and language (or framework) for programming the robots at the lowest level. Users should be able to use whatever technology they are comfortable with, and it should be simple to integrate with the existing client/server setup. The system should be able to send missions regardless of language, and let the user have some predefined API or easy configuration for adding things like simpleactions and robots. Doing this would however lose some of the innate benefits and features of using the current simulation tool but should be possible to do nonetheless without losing any functionality from a mission planning standpoint.

Naturally, the different cases would require different amounts of experience for the users. Defining the different users and possible cases are useful to see the level of user-friendliness the system needs for the different cases, and how the different users might think to interact with the system. Below is Figure 5.1 which is an illustration showing the different cases for each type of user and the relations between them in the form of a use case diagram. Users of the system can be categorized into three roles based on domain and programming experience:

- **Non-developer:** the non-developer is a user with little-to-no experience with programming or robotics.
- **Intermediate developer:** the intermediate developer is a user with at least some experience with programming and technologies in the software development domain. Such experience would include working with source code from other developers, as well as use technologies like

git and command line for running software. These types of users would include students in the later years of their studies, software developers, and some more advanced hobby developers.

- **Domain-expert developer:** the domain-expert developer is not only a user with programming and technology experience but also with some expertise within a relevant domain. The scope of their domain expertise may decide which of the parts of the system they would work within since it would depend on the type of experience the user has.

Table 5.1 show the proposed distribution of the different types of users for the different cases:

	Case 1	Case 2	Case 3	Case 4
Non-developer	x			
Intermediate-developer	x	x		
Domain-expert developer	x	x	x	x

Table 5.1: Distribution of cases to users of the system

This distribution is based upon the experience required to operate within the different use cases of the system.

Figure 5.1 is a use case diagram of the relation between users and cases in the system. This use case diagram also shows some of the different operations within the cases that each of the users should be able to perform.

Non-developers with little software development experience will work within the highest abstraction levels of the system. The abstraction level of the system were shown in the Figure 4.4 in Chapter 4. Non-developers will work with the mission planner itself, manipulating the missions, running them, and viewing the execution directly from the browser.

Intermediate developers will work at the lower abstraction levels (Figure 4.4), within the scope that the system is already in. This means using the existing technologies and solutions to extend the missions by manipulating robots, simpleactions, or the simulation environment.

Domain-expert developers will be able to work at all the different levels shown in Figure 4.4, while also working outside of the scope that the system is currently in. This means replacing the lower level parts of the application (as discussed in Section 4.5). The focus of their domain expertise would

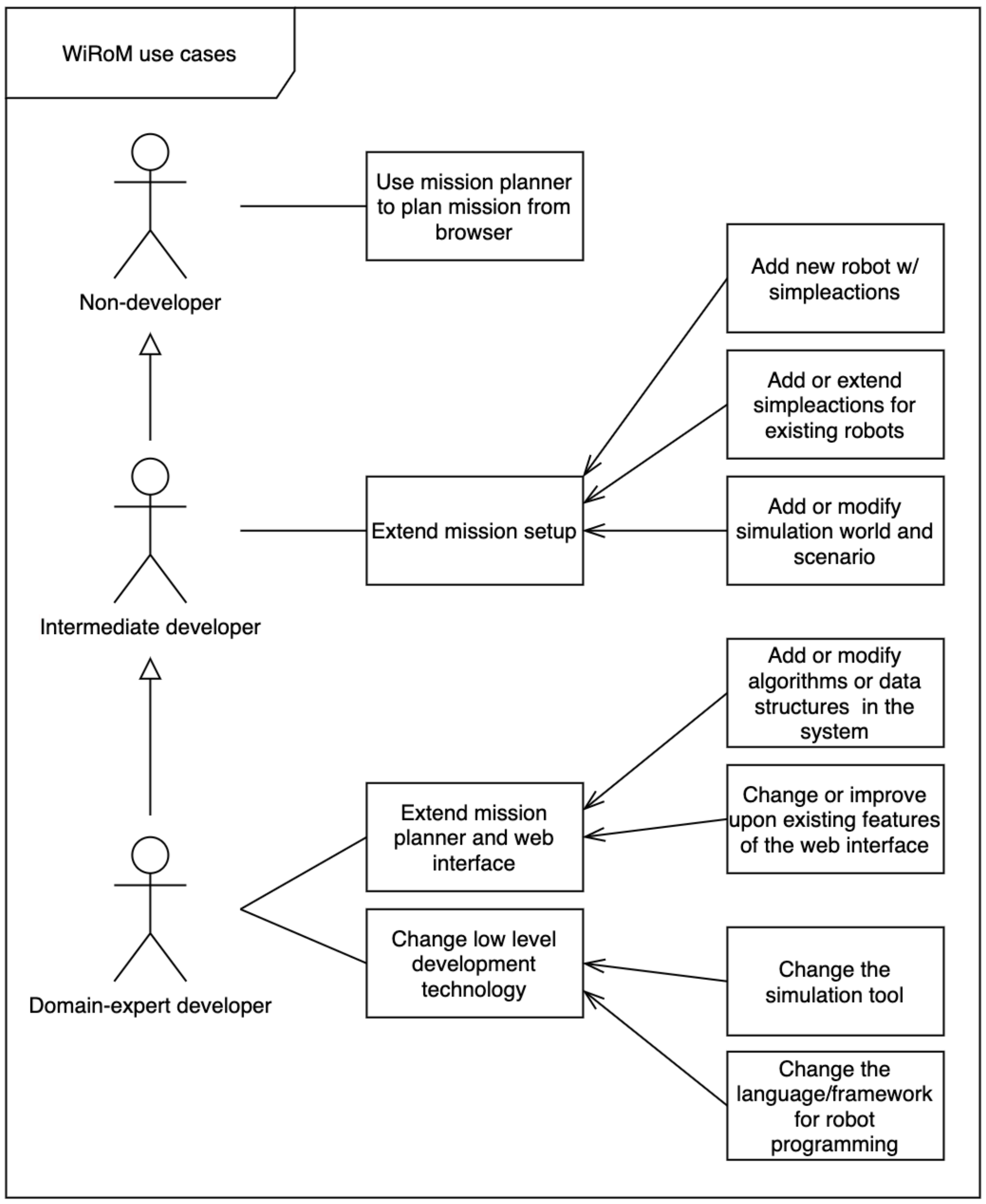

WiRoM use cases
Non-developer
Intermediate developer
Domain-expert developer
Use mission planner to plan mission from browser
Add new robot w/ simpleactions
Add or extend simpleactions for existing robots
Extend mission setup
Add or modify simulation world and scenario
Add or modify algorithms or data structures in the system
Extend mission planner and web interface
Change or improve upon existing features of the web interface
Change low level development technology
Change the simulation tool
Change the language/framework for robot programming

Figure 5.1: Use case diagram showing the different cases for each type of user and the relations between the users

define where this user would actually apply their expertise, domain experts within task allocation might want to change the task allocation algorithm in the server, while robot hardware experts might want to swap the simulation software for real-world components.

5.2 Workflow

The workflow of a system is generally defined as the pattern of activity for one user of that system. WiRoM has different user roles, so the workflow will be different based on which type of user is operating the system, and which cases the user is working with (e.g. one could be an expert user, but only work with the mission planning case). This section will discuss the different workflows for different cases and users.

5.2.1 Setup

One of the main purposes of creating the system as a web application was that the system could be deployed online such that mission planning could be performed by any user without having to do any setup. The project of this thesis did **not** prioritize to actually deploy the system, but rather provide a functional prototype. This means that the non-developer users are not assumed to be able to perform the setup of the source code that is required for the system to run as it is, because it uses several technologies like *command line* and *git*. Any non-developer would be recommended to have someone with experience with such technologies perform the setup for them.

Setting up the system from source is still required for performing all other cases, so the intermediate developers and domain-expert developers are the main target for performing this type of setup, and should be able to perform the task with ease.

We present a quick summary of the setup that is needed to set up the system. The readme for WiRoM found on GitHub [74] contains the full installation instructions for setting up and running the system.

To run the system locally, one needs to clone the repository from git. After the repository is cloned, one can install all the needed packages using *npm install* and *pip install*, and then simply start the application by running the client and server. Starting the client from the terminal is done by the command *npm start* and the server is *flask run*. After using these commands, a window should pop up in the browser containing the user interface of the application. This interface is used to plan missions, all communication between the browser and the simulation (aside from the streaming) goes via the server.

Webots can be downloaded and installed from `cyberbotics.com`, and you can either run the software normally or in streaming mode. It is required to open Webots in *streaming mode* in order to view the simulation directly from the browser. Opening in streaming mode requires the user to open Webots from the command line using the argument *–stream*. After opening the first time, one would need to import the simulation world file that comes with the repository, which will fit the default missions that come with the mission planner. After this is done, and the user had the client and server running, the user can start to use the system. The details of the design and how to use the software comes later in this chapter.

5.2.2 Mission planning

Mission planning is considered to be a case for all user roles as we have seen in Table 5.1. To plan missions we first want to connect to the simulation stream, as shown in 4.9. Now we have established the connection between Webots and the web interface as we showed in Section 4.5.

When this is done we can use the low-code web interface to manipulate existing missions or create new actions. As discussed in Section 4.9, the web interface is implemented in such a way that a user can drag and drop simpleactions and tasks around to plan a mission. This case also allows for using automatic task allocation in combination with manual allocation to distribute robots to the tasks and viewing the mission timeline. When we want to execute a mission, we can send the mission and start the simulation. A mission can be altered during run-time, making it possible to start the simulation once and send multiple missions, or stop a mission during execution and send a new one, without having to redeploy the robot or reset the simulation. See Section 4.9 for more information on the web interface.

5.2.3 Extending the mission setup

This part requires some development experience, so this is not for the non-developer (Table 5.1). Users would be in the *server* and *robot programming* *language* levels of the system during this case, as we showed in the architectural model in Figure 4.4. To extend the mission setup, one needs to install the system from source (as described in the setup Section 5.2.1). There are a few different ways one can extend the mission setup:

- Modify existing simpleactions: this is a fairly simple process, that can be done by modifying the *robotname_simpleactions.py* file that exists for each robot (see Section 4.8). Section 4.11 describe how different simpleactions are implemented.

- Adding new simpleactions: this is the same as the process above, but now one would also have to add that simpleaction to the client of the system, such that it can be used from the mission planner. This was also discussed in Section 4.8. The metadata about the new simpleaction would be needed to be added to the *data.json* file, under *robots:"robotname":simpleactions*. In order to make the automatic task allocation work with the new simpleaction one would also need to add the simpleaction metadata to the *config.json* file in the server. This will require the developer to add an estimated quality and cost in order for the algorithm to calculate the robot's performance of that simpleaction. A function that adds these variables to their files automatically would be desired as an improvement for future work.

- Modify/add simulation environments: environment manipulation will be done directly from Webots, and the documentation on their website[1] contains a lot of information on how to do this. Nothing is needed to be modified anywhere else for this to work.

- Adding new types of robots to the system: This can be done in Webots by simply clicking the + button in the top left and then selecting: *Proto nodes (Webots project)* → *robots*, and select some robot that hasn't got any simpleaction implemented for it. One would have to create a new file with all the new simpleactions for that robot and add their data to the required files (as mentioned above). A new robot also requires a file that runs within the simulated environment, that simply initializes the *robot_simpleactions.py* file (see Section 4.8). The existing implementations of robots and simpleactions should provide a good template to

[1]`https://www.cyberbotics.com/doc/guide/index`

how the different parts of a new robot should look like. Future improvements to this would be to have this done automatically when adding a new robot and new simpleactions, such that the user wouldn't have to focus on editing all the *data* and *config* files for everything to work.

5.2.4 Extending the mission planner and change low-level technology

This part is designed for users with more domain-expertise because it requires some knowledge about how the mission planner can be extended, and how other low-level technology works and can be applied to the existing system (Table 5.1). Some of these processes would also be time-consuming, even for someone with a lot of domain expertise. In this case, users would act on all of the levels in which were shown in the architectural model in Figure 4.4 (including *middleware, etc.* if so desired). Users here would be able to manipulate the system in its entirety in order to improve the system or perform research. These are the different processes that fit within these two cases:

- Extending web interface: This would require knowledge about front-end technologies, specifically *React.js* in this case. This is the place where there is the most room for improvements in the system. This can be done by modifying the files in the *robot-mission-planner* folder of the project (see Section 4.8). The design of the system is due for some improvements, and a lot of functionality that lets the user add data about simpleactions and robots without having to locate different types of files can be added to the system. Also, there are a lot of possibilities to receive data from the simulation and use this for showing more data on the screen or maybe use it for planning missions.

- Extending underlying algorithms: this part includes both existing algorithms and possible extensions for future work. Right now the main underlying algorithm is the MRTA algorithm (Section 4.13). Changing this to some other algorithm is as simple as letting the new algorithm be triggered by an HTTP request and send the new allocation as the response (this can be done by any language, it does not need to exist in the same server, one simply needs to specify where to send it). Other parts that can extend the mission planner is having automatic mission validation, automatic mission optimization, use machine learning to plan missions or improve the algorithms for the robots (pathfinding, object recognition, collision avoidance, etc.)

- Changing the language for programming robots: the system uses the Webots controller framework, which is specifically created for the Webots simulation tool. Using any other framework with Webots is not recommended, but the language that uses said framework can be changed to any of the supported languages for the framework (see Section 4.1). Taking the current setup, one can convert everything from the simpleactions files to equivalent code in another language, and simply add those new files to the system, and specify the new language in the metadata (*config.json*, see Section 4.8)for that robot. As is, the check for language is not used (because the only language used is Python), but everything is ready for making the server distribute to other languages.

- Changing the simulation tool: The server communicates with the robots via HTTP (see Section 4.10). This means that the server does not really care how the robots are programmed and what tool it uses, because it can still communicate with it. To make some new setup compatible one would, therefore, need the robots to receive instructions from the server via HTTP, and then call individual simpleactions in a similar manner to what is done with the current setup (see Section 4.11 for specific implementation practices). If adding the new simulation tool with new simpleaction files in another language is done correctly, then the rest of the process is as simple as adding new robots and simpleactions to the existing system (i.e. adding the metadata to the *config.json* and *data.json* files as shown in Section 4.8). This would obviously require some domain expertise to find a reason to swap the simulation tool and then program the robots such that they will fit the system. The simulation tool could also be replaced with real robots, but this would obviously require a lot of work on programming these robots, and create the simpleactions such that they fit the system web interface.

These are the different proposed use cases of WiRoM. The system is strongest in the hands of domain experts who can manipulate and use the system in all the different cases.

Chapter 6

Evaluation and Assessment

6.1 Research strategy

The means of evaluation is referred to as the **research strategy**. This is mainly a three-part strategy, which in itself conforms to the overall research methodology in this thesis, which is a case study, as discussed in Section 2.5. M. Shaw wrote an article called *What makes good research in software engineering?* [75] in 2002 where she had researched different types of research strategies was used by software engineers when submitting papers to *International Conference on Software Engineering* (ICSE), showing the most common strategies, and which types of research strategies that were most likely to get accepted. Mary Shaw proposes that there are three components of a research strategy, and those components are: type of research question, type of results, and type of validation.

One type of research question is *Method or means of development*, which Mary Shaw defined as the types of questions with the nature of "*How can we do/create (or automate doing) X? What is a better way to do/create X?*". This research question definition fits well with the research questions presented in this thesis in Section 1.4. WiRoM has been developed as a prototype, to see if the new approach works. This type of result is very obviously a *Procedure or technique* which Mary Shaw defined as a "*New or better way to do some task, such as design, implementation, measurement, evaluation, selection from alternatives*".

In [75] M. Shaw show that the most common types of research strategies when dealing with *method or means of development* question are to provide *procedure or technique* results and validate with *example*. It is also common to have similar research strategies, but with *experience* or *analysis* as validation. M. Shaw mentions that using persuasion as validation was never good for anything other than *feasibility* type questions because only relying on displaying something that works is not valid enough. This thesis is conducted as a case study, and we will gather quantitative data about the system to evaluate the results from two different sources, as mentioned in Section 2.5. The user testing will validate the system through *experience* while the mission will validate the system by *example*. Both types of validation combined with the given questions and results are considered by Mary Shaw as good strategies, they are common and a high percentage of papers using such strategies are accepted.

This means that we combine the validations in this thesis, and have the following research strategy: Method or means of development → procedure or technique → example and experience.

6.2 Data collection

There are several ways to collect data from WiRoM. The key is to find the right data that can be used to answer the research questions asked in the thesis. Two different methods of data gathering have already been mentioned shortly several times throughout the thesis, and that is *mission scenario example* and *user testing*

These two methods of collecting data will provide a solid foundation for evaluating the system and answer the research questions. It is required to do a bit of work to decide what kind of missions that can be used on the system, and how to go forward with user testing the system and interviewing the users afterward.

6.3 Mission scenario example

There are infinitely many possible missions that can be developed using the system, since one can always add another simpleaction to a task with any arbitrary argument, in addition to being able to extend the simpleactions and robots of the system. It's therefore important to focus on some simulated scenarios that can be easily translated to a real-life scenario, a scenario that has some practical usability. The chosen scenario will also have many different ways to reach its goal, so trying to get a simple and efficient solution to the scenario is also important.

The initial mission scenario for this evaluation was an exploration mission. Much of the literature that was reviewed (see Sections 2.4.1 and 3.5) focused on such missions because they have clear connections to real-world scenarios. These types of missions are also easily compatible with heterogeneous multi-robot setups, where different robots have different purposes in the exploration mission. E.g. a small but nimble robot can scout an area, while a bigger robot can transport or fetch materials since it can carry more weight. A separate robot might be present to load/unload items from the bigger robot etc.

An implementation of this scenario was provided using a drone and a rover. The drone is a *Mavic 2 Pro* drone, having a camera that can be used for object recognition. The rover is a *Moose*, an unmanned vehicle with a big loading space on the top. The goal of this mission was to deliver/fetch supplies or other items to humans out in the field. A simple version was developed, and it was working fine with the current setup. But when the lock-down happened due to Covid-19, there suddenly was a real-world example at hand that we could solve using WiRoM.

6.3.1 The quarantine delivery mission

This mission would be a different take on the same *exploratory missions* scenario. When the lock-down started, people were recommended to stay at home, and people who were sick or came home from travel was quarantined or isolated and couldn't go outside of their homes. Robots could be used to drive around with necessary supplies to people in quarantine, such that they wouldn't have to be in contact with any people to get their groceries, medicine, and other supplies. A drone could fly around a neighborhood and look for some signal that would notify the drone that there were people in quarantine in that household that needed supplies (e.g. a flag that was put out). The drone would then send a message to a rover (or set of rovers) that would drive around with supplies. This way, the rovers would avoid having to drive to every house, only to the ones that needed it. The goal of this mission is to have autonomous delivery of supplies to people in quarantine, without knowing who needs it beforehand. Several news articles and posts have already been published on such solutions that have been deployed in the real world, for instance, [76], [77] and [78], just to mention a few. By showing that WiRoM could plan such a mission we give a good indicator of the practicality of the system.

If WiRoM was poorly designed or developed, then creating such a mission from the existing set of simpleactions would have been difficult, luckily that was not the case. There were some minor changes needed in the code of the simpleactions, but no new simpleactions had to be added to solve a version of this mission. Changing the simulation world to conform to the new mission was also a part of creating the *quarantine delivery*-mission because the current setup would not let us successfully test the mission in a compatible environment. Converting from the standard exploration mission to the *quarantine delivery*-mission was an easy assignment, one could simply use the mission planner to plan this new mission after doing some simple tweaks. The tasks were automatically allocated using the MRTA algorithm to the best fitting robots. This mission supports the execution of one route, for one drone-rover pair in the prototype.

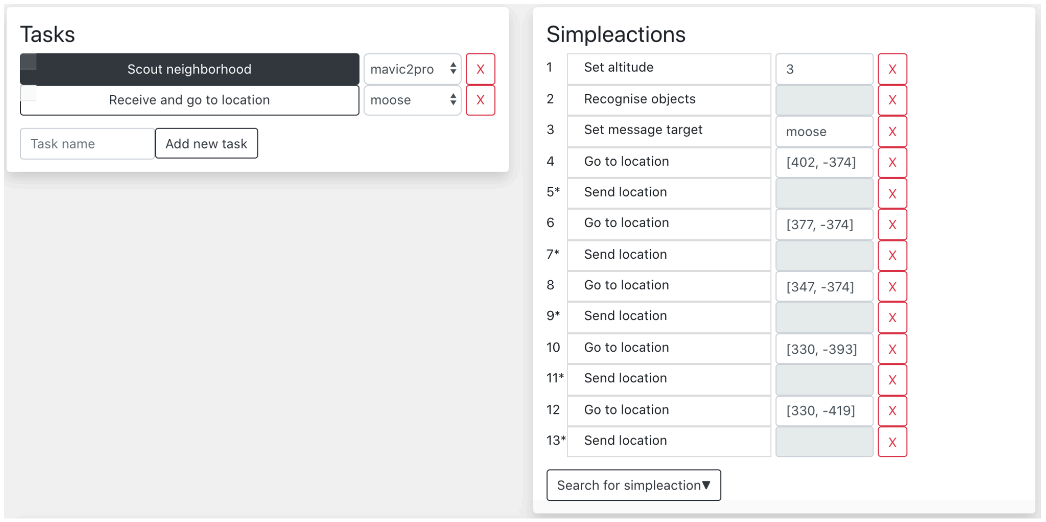
A video recording of the *quarantine delivery*-mission can be viewed on YouTube[1]. The description contains more information about what you see and timestamps for events in the mission. Screenshots from the *quarantine delivery*-mission that is used in the video can be seen in Figures 6.1 and 6.2. The mission executes as follows:

1) The drone starts by setting its altitude such that it avoids obstacles, then enable object recognition to see flags, and set the message target to *moose* such that it knows where to send the messages. The moose starts by actively waiting for its first location.

2) The drone will try to fly to the first given locations, and when reaching the location, it will send that location to the moose if there is a flag there. When the rover receives a location it will go to that location and actively wait for the next one.

3) The drone will keep flying to the next location looking for flags, and the rover will follow if there are flags at the location. If the moose receives a while it was driving, it will put that location in a queue, and skip the next active wait. Send location will be called after reaching each location, even if there are no flags (if there are no flags, the message will not contain any new location). This will repeat for each given location.

[1]`https://www.youtube.com/watch?v=TE_qN2Zqp8E`



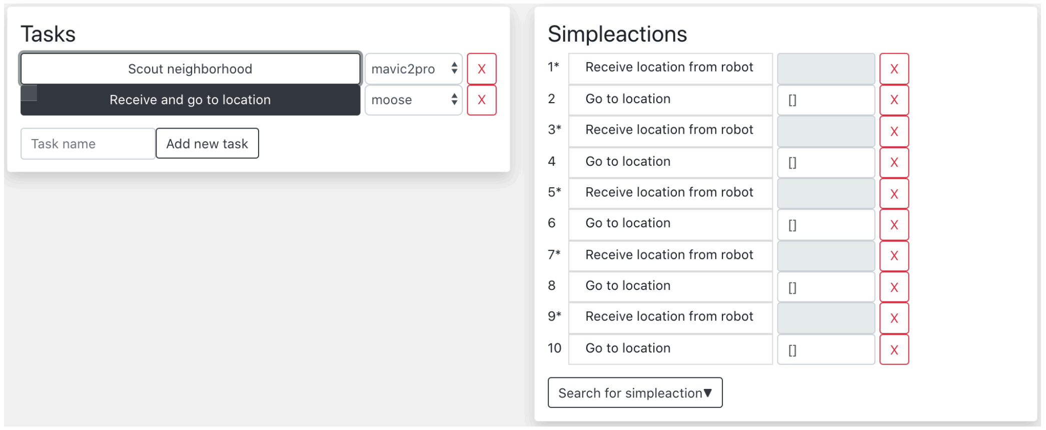
Tasks

Scout neighborhood	mavic2pro
Receive and go to location	moose

Simpleactions

1	Set altitude	3
2	Recognise objects	
3	Set message target	moose
4	Go to location	[402, -374]
5*	Send location	
6	Go to location	[377, -374]
7*	Send location	
8	Go to location	[347, -374]
9*	Send location	
10	Go to location	[330, -393]
11*	Send location	
12	Go to location	[330, -419]
13*	Send location	

Figure 6.1: Screenshot from *quarantine delivery mission* for the *Scout neighborhood* task allocated to the *mavic2pro*



Tasks

Scout neighborhood	mavic2pro
Receive and go to location	moose

Simpleactions

1*	Receive location from robot	
2	Go to location	[]
3*	Receive location from robot	
4	Go to location	[]
5*	Receive location from robot	
6	Go to location	[]
7*	Receive location from robot	
8	Go to location	[]
9*	Receive location from robot	
10	Go to location	[]

Figure 6.2: Screenshot from *quarantine delivery mission* for the *Receive and go to location* task allocated to the *moose*

Some assumptions and requirements are needed for the mission to work. There need to be a *rover* and a *drone* (with a camera) present in the simulation (in this case the *moose* and the *mavic2pro*) and a simulation environment that emulates a neighborhood. The mission also assumes that the user who plans the mission knows the **exact** coordinates of the location, i.e. where the drone needs to look for flags (i.e. where the drone has to fly to look for flags, in real life this would be addresses or street names). Flags being detectable are defined in the simulation environment, not by the simpleactions, so this being a flag is arbitrary, just to show the example. The loading and unloading of supplies are assumed to be done by humans in this scenario and is therefore not a part of the mission (the rover drives around with a supply box but this is not interacted with inside the simulation).

6.3.2 Quarantine delivery expectations versus results

The description of the problem WiRoM is supposed to solve in this thesis is a good indicator of the expectations from our mission scenario example: heterogeneous multi-robot mission planning. The expected results were to be able to plan missions using heterogeneous multi-robot setups in WiRoM. The *quarantine delivery*-mission is such a mission fulfills our expectations and also has clear parallels to a real-life scenario that would be useful today.

This does not however mean that the system can execute any thinkable mission possible. Going from a more generic *exploratory*-mission to the *quarantine delivery*-mission was a fairly easy process, but this could be because they require roughly the same simpleactions to be executed. This is not to take away from the fact that the *quarantine delivery*-mission is some of the more complex missions that can be executed by the robots that we have used in this project.

6.4 User testing

Disclaimer Collecting user data for this research will be somewhat restricted because of the current state of the world. Covid-19 has restricted the ability to perform physical user testing. This greatly affected the number of users that tested the system, because it would require more from the users. Many adjustments had to be made, which resulted in the entire process of user testing to be very time-consuming. In retrospect, this could have been avoided had we put less emphasis on functionality, and rather prioritize to deploy the system as a web application from the beginning, but predicting such events is impossible, and we have been making the most out of what we had available during these times.

We wanted to have users with different backgrounds and different amounts of experience test the system for the different cases (as shown in Section 5.1). User testing the system will provide data that can be used to answer our research question in Section 1.4. This data was intended to be collected by interviewing and observing users as they test the system, performing a set of given instructions, without interference from the observer (Section 2.5).

We wanted to do this by approaching different users that were physically available but this had to be changed because of lock-down, narrowing down the possibilities of physical testing. The observation part will fall away because we cannot observe the user's struggles and realizations in the same way as with physical testing. The interview will also be slightly altered, we decided to use a questionnaire instead of a normal interview, making it simpler for people to answer the same questions over the web. This would also simplify the process of categorizing the answers and compare the different results.

6.4.1 Questionnaire

Some research was performed to create the questionnaire. We wanted to find some template or sample questionnaire with questions about a similar topic that could be used to create the questionnaire for this evaluation. The problem was that there were no templates or samples that fit the criteria for this evaluation. Therefore, we focused on finding some guides, or a set of rules that was pretty standard for creating questionnaires. Two good articles were located, one from the Harvard University program on survey research [79] and one from NHS England [80].

Both articles mention a few common concepts for writing a questionnaire:

- Being short and concise with the questions, otherwise, the users might be swayed away from answering properly (if at all)
- Ordering of questions is important to make the questionnaire flow nicely
- Be specific with the questions, to get the answers you desire

- Avoid double-barreled questions, i.e. one question asking about two different things
- Use clear language, such that the user is not confused or misinterpret the questions.

When writing a questionnaire there are different ways to create questions and options, deciding how the users should provide feedback to the questions. The main different questions are:

- Open-ended questions, where the users will have to type some answer
- Close-ended questions, where that the user will have some options available to choose from when answering

We want to use close-ended questions when collecting data to keep the answers consistent. This also makes it easier to read the answers and put them up against each other, but we need some strategy to do so.

Close-ended questions also have a lot of different types of styles one can use. One of those styles is to use a *Likert scale*. Likert scale is when each question is written as a statement, and the user answer based on how much they agree with that statement *(strongly agree, agree, neither agree nor disagree, disagree, strongly disagree)*. We wanted to use this type of question in our questionnaire, but with slight alterations.

Neither agree nor disagree would provide very little information from the users based on the statements because it might not be a proper answer to some of the questions. E.g. answering *Neither agree nor disagree* to the statement *I had few to no complications with X*) does not provide much of an answer. Additionally, some experiences that users have with the system might not fall within the Likert scale at all, depending on the situation. Because of this, it was decided to remove the *neither agree nor disagree* option and add an *other* option instead, letting users type out some answer to the question that they see fit more than the Likert scale (so they could still answer *neither agree nor disagree*, but they would have to type it out in the *other* option).

The articles mention some different things about the types of answers that can come in. [79] mentions that having *agree* or *disagree* options for questions that ask about positive experiences might make the answer bias towards answering agree. This is important to note if all of the users seem to tip towards agreeing with most of the statements. [80] mentions that it can be useful to provide some questions at the beginning about the user that is being questioned, to have some data on which type of user provides which types of answers.

The questions themselves can be seen in the next subsection together with the results from the questions (to avoid repeating the questions). This questionnaire was sent out together with the setup and test instructions [74] [81] to a group of users.

6.4.2 Results

A few questions were asked at the beginning about the users' experience, the time they spent on the instructions, as well as spent on installation (if it was needed). This was to gather some info about the user that may have affected the results.

- How much experience would you say you have with programming? (No experience, Novice, Intermediate, Advanced, Expert)
- How much time would you say that you spent on completing the given instructions? (Fill in the answer)
- If you had to install the software before testing it, how much time would you say that you spent on this process? (Fill in the answer)

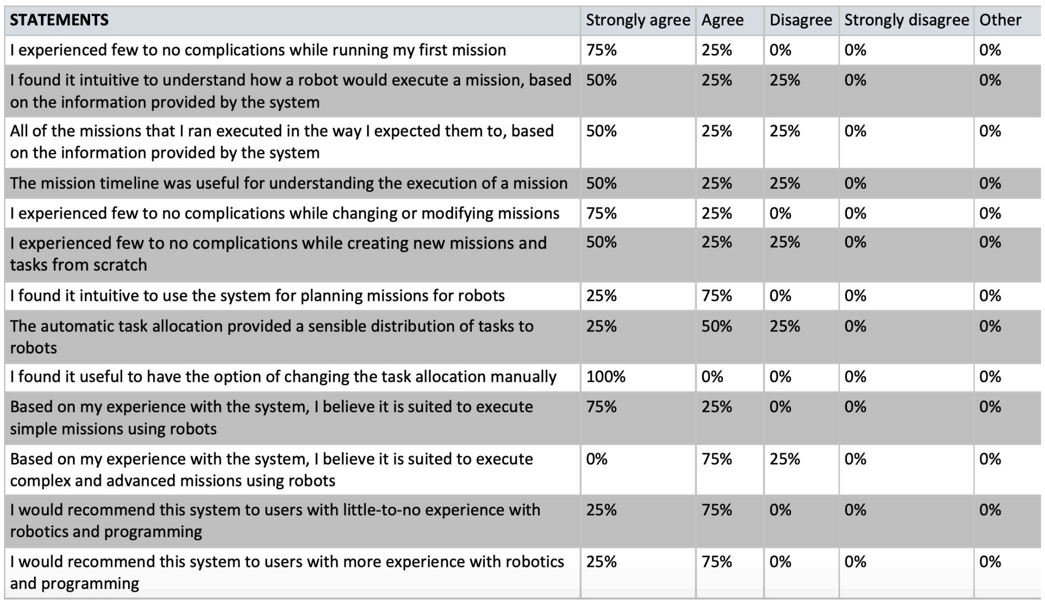
The most common users were advanced programmers, with one user having no experience. All except one user also spent 30 minutes on their instructions. One person noted their installation time, to also be 30 minutes.

Table 6.3 show the questions that were asked, as well as the results. No more than four users were able to perform the user testing of the system, and these were the answers that were given.

6.4.3 User testing expectations versus results

Some initial surprises such as no *strongly disagree* answers nor *other* answers were provided. One additional note was also added to the end of one of the answers, which was asking for more pictures in the instructions of the system.

It was expected for the users to experience *some* complications when running such a system for the first time, because of either bad instructions,



STATEMENTS	Strongly agree	Agree	Disagree	Strongly disagree	Other
I experienced few to no complications while running my first mission	75%	25%	0%	0%	0%
I found it intuitive to understand how a robot would execute a mission, based on the information provided by the system	50%	25%	25%	0%	0%
All of the missions that I ran executed in the way I expected them to, based on the information provided by the system	50%	25%	25%	0%	0%
The mission timeline was useful for understanding the execution of a mission	50%	25%	25%	0%	0%
I experienced few to no complications while changing or modifying missions	75%	25%	0%	0%	0%
I experienced few to no complications while creating new missions and tasks from scratch	50%	25%	25%	0%	0%
I found it intuitive to use the system for planning missions for robots	25%	75%	0%	0%	0%
The automatic task allocation provided a sensible distribution of tasks to robots	25%	50%	25%	0%	0%
I found it useful to have the option of changing the task allocation manually	100%	0%	0%	0%	0%
Based on my experience with the system, I believe it is suited to execute simple missions using robots	75%	25%	0%	0%	0%
Based on my experience with the system, I believe it is suited to execute complex and advanced missions using robots	0%	75%	25%	0%	0%
I would recommend this system to users with little-to-no experience with robotics and programming	25%	75%	0%	0%	0%
I would recommend this system to users with more experience with robotics and programming	25%	75%	0%	0%	0%

Figure 6.3: Table containing results from user testing the system

system faults, or user errors, but not a lot of complications. The points of interest for complications were expected with running missions or with changing and modifying missions (as reflected in the questions). The results were mostly as expected, maybe even slightly better, and everyone made it through the instructions.

The system was expected to be intuitive for users (especially when they were given proper instructions), because of its low-code design. The results showed that users found the web interface more intuitive than counter-intuitive. The web interface might not be overly intuitive in the beginning, but it seems like people quickly learned how to make use of the system. The lack of problems while running and modifying missions also indicate that the system was intuitive. Even though the web interface is designed and developed as a low-code platform, the actual design has not been done by a proper designer, so this might affect the initial intuitive aspects.

The automatic task allocation was expected to perform well, and the manual task allocation was expected to be useful. The automatic task allocation seemed to satisfy most users, but then as expected, the manual option was very useful for all users, implying that the users who were dissatisfied with their automatic allocation were happily making the desired corrections.

The system was generally accepted as a system for simple missions rather than advanced missions. It was expected for it to be accepted for both types of missions, but this shows that the design and low-code approach might not be the best fit for more advanced missions, because it does not provide the necessary tools needed to perform more advanced missions. Some mission configurations might go missing, or it is necessary to go to the lower abstraction levels of the system to change things to perform the desired missions. It is important to note that it is not specified in the questionnaire what a *simple* or *advanced* mission was, which might be an invalidating factor for these results. However, the initial user impression of the system and their view on what is *simple* and *advanced* will still be useful for providing perspective on the capabilities of the system, especially considering the differences in the amount of experience.

Finally, it was expected for most users to want to recommend the system for both inexperienced and experienced users. The results reflected this nicely, most people would recommend it to both. This shows that after performing all of the instructions, the users generally found it simple and user-friendly to use the system, and would recommend it as a system for planning robot missions.

6.5 Threats to validity

When discussing the validity of our results, two different types of validity are considered: Internal validity and external validity [82]. Threats to internal validity include faults in the research method or data collection. Threats to external validity are considered as faults that prevent the system from being valid in practice.

6.5.1 Internal validity

Showing that one mission (or one type of mission) fulfill our requirements does not automatically mean that all missions will fulfill the same requirements. Only a few mission was planned using WiRoM that had some parallel to practical real-world scenarios, and the evaluation is focusing on the most relevant mission and use this to argue the validity of the system. Having one such mission does not mean that any thinkable mission, or even any type of mission, can be planned using WiRoM, but it shows that it works for this scenario (and similar scenarios). To increase the internal validity of mission scenarios we would set up an experiment to develop new missions that were considered practical. The diversity and sample size of missions would be increased, to increase the validity of mission execution in WiRoM.

The users that test the system are not randomly selected, but rather a selected group of people. This could lead to bias towards choosing good options for the statements because most of the users have personal relationships with us, and would want us to get good results rather than just being honest about their opinions. Additionally, using a Likert scale with positive statements can skew any user to give more 'agree'-answers (as discussed in Section 6.4.1). To increase internal validity we would have randomized the users for testing the system, as well as applying a control group, comparing the usage of WiRoM to non-usage of WiRoM for mission planning.

6.5.2 External validity

The missions planned by the same people who designed and developed the system, which means that the mission might not reflect what a user without the same knowledge about the system might be able to achieve. The *exploration*-type of missions was the main type of missions that was intended for the system, so the simpleactions have been created with this type of mission in mind, and might not be easily transferable. Having an experiment where external users would plan missions and try to solve different practical problems would be a good way to increase the validity of the missions.

The mission scenario (and the system as a whole) only use two different robots, which is not ideal when testing a heterogeneous multi-robot setup. Validity could be improved by adding more robots to the system an running them through the same missions and task allocation.

The sample size of users testing the system is very low, and the quality of testing is also affected, because of the lack of physical observation. Having a low-sample size makes it difficult to assess how valid the system might be for more than just the selected people that tested the system. Increasing external validity would be done by increasing the sample size of people that tested the system. All cases (Section 5.1) would be tested to provide valid results for the entirety of the system.

6.6 Discussion

The research questions (Section 1.4) for this thesis are all *how-questions*, asking how well some concepts affected our results. To answer these questions, we need to assess whether we got to our wanted results using the proposed methods and means of development, and to what extent our results cover the topics in the questions.

The research questions that were for this thesis were the following:

Main research question

RQ1 *How well do Model-Driven Software Engineering core concepts combined with robot programming enhance the practicality and usability of a high-level heterogeneous multi-robot mission planning system?*

Sub-questions

RQ2 *How well do the concepts and principles of low-code platforms enhance the user-friendliness of the system*

RQ3 *How efficiently are tasks allocated to multiple heterogeneous robots within the system?*

To answer the main research question (**RQ1**), we must assess the **practicality** and **usability** of WiRoM. There are two main perspectives when describing the practicality of the system. The main perspective is the practicality of the tool as a mission-solving tool. The *quarantine delivery*-mission is a practical mission, as we have seen in Section 6.3.1, there already exists similar robot missions that are being applied to the real world right now, and the need for good solutions to this problem in the future is clear. It also solves a mission (or type of mission) that is similar to missions that are proposed by related work like TDL, ERGO, and FLYAQ, as well as proposed missions that we reviewed in DARS (see Sections 3 and 2.4.1). The *quarantine delivery*-mission provides a good example that the system can be practical for planning missions in a simulated environment.

Practicality of the system can also be seen as the practicality the system has as a research/learning tool. The results from the questionnaire show that adding/modifying missions and running missions in the simulation were not considered a complicated process for both experienced and inexperienced users. The benefit of this is that the system will provide a platform for easy mission planning as well as mission execution in simulated environments, letting domain-expert users create some set of missions that they can use to test their research topics, without having to learn much about robotics or mission planning.

Based on our results from both the *quarantine-delivery*-mission and user testing, the system can be seen as practical. We would expect WiRoM to be most practical as a research tool because of said benefits, but we cannot validate these expectations because we haven't had any domain expert users try to perform research using the system.

The **usability** of the system can be seen as a factor of how well the system can be used by all of the different users and cases that are proposed as use cases for the system. There are some different perspectives when trying to assess whether the system is usable, or at least to what extent. It does not only have to be able to work, but it has to work well for the different use cases. As mentioned, because of current restrictions, the data that was collected through the questionnaire only could only focus on the mission planning case of the system, but with various types of users, which means that this is where we can evaluate the usability of the system. From the results that were collected, all users who tried managed to get quickly through the instructions, and were able to make use of the system, regardless of programming experience. The system was also generally recommended for other users and considered as a system that worked well with simple missions. The sample size is not large, but the data collected does point towards the system being used for the mission-planning case. We can assess the system as usable for mission-planning, but because of a lack of data on other use cases, we cannot assess it for the other use cases of the system.

The **MDSE core concepts** brings the **practicality** and **usability** of WiRoM to a higher level, and it all happens autonomously. We can argue that practicality and usability have been enhanced because the system has raised the abstraction levels and therefore reduced the complexity of heterogeneous multi-robot mission planning while maintaining the complexity of mission execution at the lower levels. When using MDSE core concepts in combination with robot programming allowed us to create a prototype for this system, such that it could be tested.

Answering the first sub-question (**RQ2**), we need assess the **user friendliness** of the system. Low-code principles are considered a driving factor for increasing the user- friendliness of systems. We have already seen other low-code platforms that have been created as such, to provide programming for users with little experience (see Sections 2.2.5 and 3.6). The intuitiveness of the system from the user testing results show that the system was leaning towards being *intuitive*, and the fact that all users got through the instructions with a maximum time of 40 minutes regardless of programming experience are some clear indications of user-friendliness. Our results also told us that most users would recommend the system to users with different amounts of programming experience, and additions like the mission timeline and means of allocating tasks are assisting the users to understand the system and were accepted as good contributions.

We need to remember that the **user-friendliness** could propose a threat to the complexity of the system. This is always a risk when abstracting a system (because reducing complexity is one of the purposes of abstraction), but the key challenge when abstracting is to keep as much functionality as possible. The results from the questionnaire generally showed that the system was accepted as a system for simple missions rather than advanced missions, and this is one of the possible drawbacks that we try to avoid when using this type of low-code design. The *quarantine delivery* mission is considered a complex mission, which shows that we have done a good job of abstracting without loosing too much complexity. Based on these results, the system can be considered as user-friendly, with definitive room for improvements from a design standpoint.

Higher abstraction levels are considered as synonymous with user-friendliness, as we have seen earlier in Section 2.2 because it removes complexity from the system. A high-level low-code platform for mission planning is assessed as user-friendly based on the results from user testing the system, meaning it successfully enhanced the user-friendliness of the system, while maintaining relatively complex mission execution.

Answering the second sub-question (**RQ3**), we need to assess the how **task were allocated efficiently** in the system. Having good MRTA algorithms that run automatically is the most efficient approach to allocating tasks to robots (see Section 2.3), but one problem is to get correct results when having heterogeneous robots. WiRoM, therefore, provides manual allocation such that the users can make up for any faults in the task allocation algorithm. The results from the questionnaire show that automatic task allocation seemed to work well for most, providing good task allocations for their missions. These tasks were allocated mostly according to the expectations, and if they were not, then the users would use manual allocation to get their desired allocation.

The literature tells us that MRTA algorithms are efficient for the MRTA problem (see Section 2.3), as well as some related work like TDL which used MRTA algorithms 3.2. Since we know that MRTA algorithms are efficient, we provided an implementation of one such algorithm in WiRoM, to show that MRTA algorithms can be used to allocate tasks efficiently in the system. The results from our data indicate that the implementation works as intended, and manual allocation was available as a solution to the flaws of the MRTA algorithm. These results are less prone to threats to their validity because very few examples and experiences are needed to show that a type of algorithm works. We can use the literature to show that being able to implement any MRTA algorithm is an efficient solution because the extent of the efficiency can be increased by optimizing the MRTA algorithm while mentioning that this could be the topic for an entire thesis in itself.

Discussion conclusion Based on the information we have gathered, we can argue that WiRoM is showing *indications* of being practical, usable, user-friendly, and provide good task allocation efficiency. We choose to say that we have indications of such because the threats to the validity make it difficult to realistically assess all the results as valid. This does not mean that the results are invalid, but rather that there needs to be more data collected and research done on the system to further confirm the indications that we have seen throughout this thesis.

Chapter 7

Conclusion

This thesis presented a solution to the heterogeneous multi-robot mission planning problem. **WiRoM** was developed as a prototype to solve this problem. The system was designed and developed using MDSE core concepts, low-code principles, and multi-robot task allocation, providing solutions to the challenges and questions presented in this thesis.

We have used studied MDSE and located the core concepts that were useful for the development of WiRoM. The core concepts included system abstraction, model-driven architecture, code generation, and model transformation. These core concepts acted as the foundation for how the different parts of WiRoM were to be developed. By having such a foundation and combining it with robot programming (using the Webots controller framework in Python), we achieved a system that was considered as practical and usable, based on the data gathered in this thesis.

We started by looking into the limited literature that surrounds low-code platforms and proposed a low-code design for the web interface of WiRoM from what we learned. The data collected in this thesis showed that the users that tested the system found the web interface user-friendly, implying that the low-code platform web interface, combined with other assisting tools from the system, was a driving factor for user-friendliness.

By studying MRTA problems and algorithms, we located an algorithm for solving the XD [SR-ST-IA] problem that was presented in WiRoM and implemented it in the system to show that such an efficient solution could be used. The data gathered confirmed that both the automatic allocation and the manual allocation worked as intended. The extent of efficiency of the MRTA algorithm provided in this thesis is not optimized, but this shows that it's possible to increase the efficiency of the automatic task allocation in WiRoM by simply optimizing or replacing the existing algorithm.

WiRoM comes with some useful propositions to the heterogeneous multi-robot mission planning problem, with the possibility of further improvements both to the system and the research. The robotics and mission planning domain is rapidly evolving and having new solutions like the one developed in this thesis provide some perspective on the future of research in this domain.

Chapter 8

Further Work

There are several different parts of the system that can be optimized or utilized for further work. A high-level system was to allow domain-expert users to easily plan missions in a simulated environment and use this to research within the robotics domain. Possible research topics that can be performed using the system include optimizing and testing robotic algorithms (collision avoidance, path planning, task allocation, etc.), apply machine learning, and create automatic validation and verification of missions. These are some of the possible topics that could make use of having such a system as the base for performing research.

The system is a prototype, which means that it is not a finished product that is can be applied commercially. Almost all of the different parts of the source code for WiRoM can be optimized for future usage. Everything is created such that it was functional, just to show that the approach works, but this means there are lots of room for further improvements. For instance, the web interface can be improved (greatly), information about robots could be utilized better and low-level programming is not optimal because it was not the main focus of the project. This is just to mention a few things that would be next on the priority list.

The system could also be deployed properly as a web application, making it closer to a commercial prototype, and making it easier to distribute. This would require hosting the web interface, the Python server, and a Webots instance on separate servers, making the entire system available over the internet, straight from the browser without having to do any installation. This would also require things like a database for the system to store data. This project has been created to be able to be deployed in the future, meaning it runs locally in the exact way that it would if it was deployed.

If the system itself is not worked on in the future, we hope that the WiRoM prototype can be used as inspiration for future work on the heterogeneous multi-robot mission planning problem.

Bibliography

[1] Oxford Dictionary, *Robot definition.* [Online]. Available: `https://www.lexico.com/en/definition/robot` (visited on 01/20/2020).

[2] R. Yin and SAGE., *Case Study Research: Design and Methods*, ser. Applied Social Research Methods. SAGE Publications, 2003, ISBN: 9780761925521. [Online]. Available: `https://books.google.no/books?id=BWea%5C_9ZGQMwC`.

[3] Oxford Dictionary, *Robotics definition.* [Online]. Available: `https://www.lexico.com/definition/robotics` (visited on 01/20/2020).

[4] M. Ben-Ari and F. Mondada, "Robots and their applications," in *Elements of Robotics*. Cham: Springer International Publishing, 2018, pp. 1–20, ISBN: 978-3-319-62533-1. DOI: `10.1007/978-3-319-62533-1_1`. [Online]. Available: `https://doi.org/10.1007/978-3-319-62533-1_1`.

[5] Oxford Dictionary, *Task definition.* [Online]. Available: `https://www.lexico.com/definition/task` (visited on 01/20/2020).

[6] ——, *Mission definition.* [Online]. Available: `https://www.lexico.com/definition/mission` (visited on 01/20/2020).

[7] L. Zlajpah, "Simulation in robotics," *Mathematics and Computers in Simulation*, vol. 79, no. 4, pp. 879–897, 2008, 5th Vienna International Conference on Mathematical Modelling/Workshop on Scientific Computing in Electronic Engineering of the 2006 International Conference on Computational Science/Structural Dynamical Systems: Computational Aspects, ISSN: 0378-4754. DOI: `https://doi.org/10.1016/j.matcom.2008.02.017`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0378475408001183`.

[8] R. Sargent, "Verification and validation of simulation models," vol. 37, Jan. 2011, pp. 166–183. DOI: 10.1109/WSC.2010.5679166.

[9] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice: Second Edition*, 2nd. Morgan & Claypool Publishers, 2017, ISBN: 1627057080.

[10] Object Management Group, "Omg unified modeling language® (omg uml) version 2.5.1," Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/PDF (visited on 05/24/2020).

[11] ——, "Object management group model driven architecture (mda) mda guide rev. 2.0," Jun. 2014. [Online]. Available: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01 (visited on 05/24/2020).

[12] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A survey on domain-specific languages in robotics," in *Simulation, Modeling, and Programming for Autonomous Robots*, D. Brugali, J. F. Broenink, T. Kroeger, and B. A. MacDonald, Eds., Cham: Springer International Publishing, 2014, pp. 195–206.

[13] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd. Addison-Wesley Professional, 2009, ISBN: 0321331885.

[14] Mendix, *Mendix home page.* [Online]. Available: www.mendix.com (visited on 05/19/2020).

[15] J. den Haan, *Low-code principle number 1: Model-driven development, the most important concept in low-code*, Jan. 2020. [Online]. Available: https://www.mendix.com/blog/low-code-principle-1-model-driven-development/ (visited on 05/19/2020).

[16] J. Cabot, *Low-code platforms, the new buzzword*, Sep. 2016. [Online]. Available: https://modeling-languages.com/low-code-platforms-new-buzzword/ (visited on 05/19/2020).

[17] Mendix, *Mendix low code guide*, Mar. 2020. [Online]. Available: https://www.mendix.com/low-code-guide/ (visited on 05/19/2020).

[18] D. Brugali, "Model-driven software engineering in robotics: Models are designed to use the relevant things, thereby reducing the complexity and cost in the field of robotics," *IEEE Robotics Automation Magazine*, vol. 22, no. 3, pp. 155–166, Sep. 2015, ISSN: 1558-223X. DOI: 10.1109/MRA.2015.2452201.

[19] A. Khamis, A. Hussein, and A. Elmogy, “Multi-robot task allocation: A review of the state-of-the-art,” in. May 2015, vol. 604, pp. 31–51, ISBN: 978-3-319-18299-5. DOI: 10.1007/978-3-319-18299-5_2.

[20] B. P. Gerkey and M. J. Mataric, “A formal analysis and taxonomy of task allocation in multi-robot systems,” *The International Journal of Robotics Research*, vol. 23, no. 9, pp. 939–954, 2004. DOI: 10.1177/0278364904045564. eprint: https://doi.org/10.1177/0278364904045564. [Online]. Available: https://doi.org/10.1177/0278364904045564.

[21] G. A. Korsah, A. Stentz, and M. B. Dias, “A comprehensive taxonomy for multi-robot task allocation,” *The International Journal of Robotics Research*, vol. 32, no. 12, pp. 1495–1512, 2013. DOI: 10.1177/0278364913496484. eprint: https://doi.org/10.1177/0278364913496484. [Online]. Available: https://doi.org/10.1177/0278364913496484.

[22] R. Zlot, A. Stentz, C. Dias, M. Veloso, and T. Balch, “An auction-based approach to complex task allocation for multirobot teams thesis committee,” PhD thesis, Dec. 2006.

[23] H. W. Kuhn, “The hungarian method for the assignment problem,” in *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*, M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 29–47, ISBN: 978-3-540-68279-0. DOI: 10.1007/978-3-540-68279-0_2. [Online]. Available: https://doi.org/10.1007/978-3-540-68279-0_2.

[24] H. Asama, T. Fukuda, T. Arai, and I. Endo, *Distributed Autonomous Robotic Systems*. Jan. 1994, ISBN: 978-4-431-68277-6. DOI: 10.1007/978-4-431-68275-2.

[25] *Call for papers dars 2020*. [Online]. Available: https://www.swarm-systems.com/dars-swarm2020/cfp (visited on 01/23/2020).

[26] V. Govindarajan, S. Bhattacharya, and V. Kumar, “Human-robot collaborative topological exploration for search and rescue applications,” in *Distributed Autonomous Robotic Systems*, N.-Y. Chong and Y.-J. Cho, Eds., Tokyo: Springer Japan, 2016, pp. 17–32.

[27] D. Hougen, M. Erickson, P. Rybski, S. Stoeter, M. Gini, and N. Papanikolopoulos, "Autonomous mobile robots and distributed exploratory missions," Nov. 2000. DOI: `10.1007/978-4-431-67919-6_21`.

[28] D. Martinez and A. Halme, "Marsim, a simulation of the marsubots fleet using netlogo," in. Jan. 2016, pp. 79–89, ISBN: 978-4-431-55877-4. DOI: `10.1007/978-4-431-55879-8_6`.

[29] A. Marjovi and L. Marques, "Multi-robot topological exploration using olfactory cues," in. Jan. 2013, vol. 83, pp. 47–60, ISBN: 978-3-642-32722-3. DOI: `10.1007/978-3-642-32723-0_4`.

[30] K.-C. Ma, Z. Ma, L. Liu, and G. S. Sukhatme, "Multi-robot informative and adaptive planning for persistent environmental monitoring," in *Distributed Autonomous Robotic Systems: The 13th International Symposium*, R. Groß, A. Kolling, S. Berman, E. Frazzoli, A. Martinoli, F. Matsuno, and M. Gauci, Eds. Cham: Springer International Publishing, 2018, pp. 285–298.

[31] J. Guerrero and G. Oliver, "Multi-robot task allocation method for heterogeneous tasks with priorities," in *Distributed Autonomous Robotic Systems 6*, R. Alami, R. Chatila, and H. Asama, Eds., Tokyo: Springer Japan, 2007, pp. 181–190.

[32] S. Sariel and T. Balch, "A distributed multi-robot cooperation framework for real time task achievement," in. Jun. 2007, pp. 187–196. DOI: `10.1007/4-431-35881-1_19`.

[33] S. Alili, R. Alami, and V. Montreuil, "A task planner for an autonomous social robot," in *Distributed Autonomous Robotic Systems 8*, H. Asama, H. Kurokawa, J. Ota, and K. Sekiyama, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 335–344.

[34] J. D. Bjerknes and A. F. T. Winfield, "On fault tolerance and scalability of swarm robotic systems," in *Distributed Autonomous Robotic Systems: The 10th International Symposium*, A. Martinoli, F. Mondada, N. Correll, G. Mermoud, M. Egerstedt, M. A. Hsieh, L. E. Parker, and K. Støy, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 431–444.

[35] D. Jeong and K. Lee, "Distributed communication and localization algorithms for homogeneous robotic swarm," in *Distributed Autonomous Robotic Systems*, N.-Y. Chong and Y.-J. Cho, Eds., Tokyo: Springer Japan, 2016, pp. 405–418.

[36] G. Vorobyev, A. Vardy, and W. Banzhaf, "Supervised learning in robotic swarms: From training samples to emergent behavior," in *Distributed Autonomous Robotic Systems*, M. Ani Hsieh and G. Chirikjian, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 435–448.

[37] S. Demeyer, "Research methods in computer science," Sep. 2011, p. 600. DOI: `10.1109/ICSM.2011.6080841`.

[38] J. Creswell, *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications, 2014, ISBN: 9781452226095. [Online]. Available: `https://books.google.no/books?id=PViMtOnJ1LcC`.

[39] B. Kitchenham, O. [Brereton], D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering – a systematic literature review," *Information and Software Technology*, vol. 51, no. 1, pp. 7–15, 2009, Special Section - Most Cited Articles in 2002 and Regular Research Papers, ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2008.09.009`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0950584908001390`.

[40] D. S. Losvik and A. Rutle, "A domain-specific language for the development of heterogeneous multi-robot systems," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2019, pp. 549–558.

[41] D. S. Losvik, *Task definition langauge repo.* [Online]. Available: `https://github.com/95danlos/Task-Definition-Language` (visited on 01/20/2020).

[42] S. Garcia, P. Pelliccione, C. Menghi, T. Berger, and T. Bures, "High-level mission specification for multiple robots," in *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2019, Athens, Greece: Association for Computing Machinery, 2019, pp. 127–140, ISBN: 9781450369817. DOI: `10.1145/3357766.3359535`. [Online]. Available: `https://doi.org/10.1145/3357766.3359535`.

[43] D. Bozhinoski, D. D. Ruscio, I. Malavolta, P. Pelliccione, and M. Tivoli, "Flyaq: Enabling non-expert users to specify and generate missions of autonomous multicopters," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 801–806.

[44] D. Swaib, B. Meyers, and P. Pelliccione, "A generated property specification language for resilient multirobot missions," Aug. 2017, pp. 45–61, ISBN: 978-3-319-65947-3. DOI: 10.1007/978-3-319-65948-0_4.

[45] F. Ciccozzi, D. Di Ruscio, I. Malavolta, and P. Pelliccione, "Adopting mde for specifying and executing civilian missions of mobile multi-robot systems," *IEEE Access*, vol. 4, pp. 6451–6466, 2016.

[46] D. D. Ruscio, I. Malavolta, P. Pelliccione, and M. Tivoli, "Automatic generation of detailed flight plans from high-level mission descriptions," in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '16, Saint-malo, France: Association for Computing Machinery, 2016, pp. 45–55, ISBN: 9781450343213. DOI: 10.1145/2976767.2976794. [Online]. Available: https://doi.org/10.1145/2976767.2976794.

[47] D. Bozhinoski, I. Malavolta, A. Bucchiarone, and A. Marconi, "Sustainable safety in mobile multi-robot systems via collective adaptation," in *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems*, 2015, pp. 172–173.

[48] D. Di Ruscio, I. Malavolta, and P. Pelliccione, "A family of domain-specific languages for specifying civilian missions of multi-robot systems," vol. 1319, pp. 16–29, Jan. 2014.

[49] S. Gerasimou, N. Matragkas, and R. Calinescu, "Towards systematic engineering of collaborative heterogeneous robotic systems," in *2019 IEEE/ACM 2nd International Workshop on Robotics Software Engineering (RoSE)*, 2019, pp. 25–28.

[50] ERGO, *Ergo homepage.* [Online]. Available: https://www.h2020-ergo.eu/ (visited on 11/13/2019).

[51] J. Ocón, J. M. Delfa, A. Medina, D. Lachat, R. Marc, M. Woods, I. Wallace, A. Coles, A. J. Coles, D. Long, T. Keller, M. Helmert, and S. Bensalem, "Ergo: A framework for the development of autonomous robots," in *ICRA 2017*, 2017.

[52] M. Perrotin, ESA/ESTEC, and TEC-SWE, *What is taste?* 2017. [Online]. Available: `https://download.tuxfamily.org/taste/misc/what_is_taste.pdf` (visited on 01/28/2020).

[53] *Taste home page*. [Online]. Available: `taste.tools` (visited on 01/28/2020).

[54] I.-E. Dragomir, *The ergo framework presentation*. [Online]. Available: `https://sites.google.com/site/modevva/program` (visited on 11/13/2019).

[55] M. Tisi, J.-M. Mottu, D. S. Kolovos, J. De Lara, E. M. Guerra, D. Di Ruscio, A. Pierantonio, and M. Wimmer, "Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms," in *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019)*, ser. CEUR Workshop Proceedings (CEUR-WS.org), Eindhoven, Netherlands, Jul. 2019. [Online]. Available: `https://hal.archives-ouvertes.fr/hal-02363416`.

[56] NESTLab, *Buzz home page*. [Online]. Available: `https://the.swarming.buzz/` (visited on 01/24/2020).

[57] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "Robotml, a domain-specific language to design, simulate and deploy robotic applications," in *Simulation, Modeling, and Programming for Autonomous Robots*, I. Noda, N. Ando, D. Brugali, and J. J. Kuffner, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 149–160, ISBN: 978-3-642-34327-8.

[58] Cyberbotics Ltd., *Robotbenchmark*. [Online]. Available: `robotbenchmark.net` (visited on 05/24/2020).

[59] A. Hussein and A. Khamis, "Market-based approach to multi-robot task allocation," in *2013 International Conference on Individual and Collective Behaviors in Robotics (ICBR)*, 2013, pp. 69–74.

[60] C. Menghi, C. Tsigkanos, P. Pelliccione, C. Ghezzi, and T. Berger, *Specification patterns for robotic missions*, 2019. arXiv: `1901.02077 [cs.SE]`.

[61] A. Nordmann, N. Hochgeschwender, D. L. Wigand, and S. Wrede, "A Survey on Domain-Specific Modeling and Languages in Robotics," *Journal of Software Engineering in Robotics (JOSER)*, vol. 7, no. 1, pp. 75–99, 2016. [Online]. Available: `https://corlab.github.io/dslzoo/index.html` (visited on 05/20/2020).

[62] R. Waldron, *Johnny-five github repository.* [Online]. Available: `https://github.com/rwaldron/johnny-five` (visited on 09/25/2019).

[63] ——, *Johnny five home page.* [Online]. Available: `johnny-five.io` (visited on 09/25/2019).

[64] thehybridgroup, *Cylon.js home page.* [Online]. Available: `https://cylonjs.com` (visited on 09/25/2019).

[65] Node.js foundation, *Node.js home page.* [Online]. Available: `https://nodejs.org/en/` (visited on 10/02/2019).

[66] Wikipedia, "Wiki listing robotic simulation tools," [Online]. Available: `https://en.wikipedia.org/wiki/Robotics_simulator` (visited on 09/26/2019).

[67] Jslee02, *Awesome robotics libraries.* [Online]. Available: `http://jslee02.github.io/awesome-robotics-libraries/` (visited on 09/26/2019).

[68] Cyberbotics Ltd., *Webots github repository.* [Online]. Available: `https://github.com/cyberbotics/webots` (visited on 05/05/2020).

[69] ——, *Webots home page.* [Online]. Available: `https://cyberbotics.com/` (visited on 06/02/2020).

[70] ——, *Webots robot repository.* [Online]. Available: `https://www.cyberbotics.com/doc/guide/robots?version=develop` (visited on 06/02/2020).

[71] ——, *Webots language support guide.* [Online]. Available: `https://cyberbotics.com/doc/guide/language-setup` (visited on 06/02/2020).

[72] D. Robinson, *Blog-post: The incredible growth of python*, Sep. 2017. [Online]. Available: `https://stackoverflow.blog/2017/09/06/incredible-growth-python//` (visited on 05/20/2020).

[73] E. Wohlgethan, "Supporting web development decisions by comparing three major javascript frameworks: Angular, react and vue. js," PhD thesis, Hochschule für Angewandte Wissenschaften Hamburg, 2018.

[74] J. M. Grutle, *Multi-robot-mission-planner*. [Online]. Available: https://github.com/joakimgrutle/WiRoM (visited on 06/03/2020).

[75] M. Shaw, "What makes good research in software engineering?" *STTT*, vol. 4, pp. 1–7, Oct. 2002. DOI: 10.1007/s10009-002-0083-4.

[76] E. Guizzo, "Robot vehicles make contactless deliveries amid coronavirus quarantine," [Online]. Available: https://spectrum.ieee.org/automaton/transportation/self-driving/robot-vehicles-make-contactless-deliveries-amid-coronavirus-quarantine (visited on 05/24/2020).

[77] C. Metz and E. Griffith, "A city locks down to fight coronavirus, but robot come and go," [Online]. Available: https://www.nytimes.com/2020/05/20/technology/delivery-robots-coronavirus-milton-keynes.html/ (visited on 05/24/2020).

[78] R. Baldwin, "Robot deliveries might end up being common, post-coronavirus pandemic," [Online]. Available: https://www.caranddriver.com/news/a32133440/robot-car-delivery-pandemic/ (visited on 05/24/2020).

[79] H. university Program on Survery Research, "Tip sheet on question wording," [Online]. Available: https://psr.iq.harvard.edu/files/psr/files/PSRQuestionnaireTipSheet_0.pdf (visited on 05/01/2020).

[80] NHS England, *Writing an effective questionnaire*. [Online]. Available: https://www.england.nhs.uk/wp-content/uploads/2018/01/bitesize-guide-writing-an-effective-questionnaire.pdf (visited on 05/01/2020).

[81] J. M. Grutle, *Mission planner test instructions*. [Online]. Available: https://github.com/joakimgrutle/WiRoM/blob/master/Mission%20planning%20instructions.pdf (visited on 06/03/2020).

[82] R. McDermott, "Internal and external validity," in *Cambridge Handbook of Experimental Political Science*, J. N. Druckman, D. P. Green, J. H. Kuklinski, and A. Lupia, Eds. Cambridge University Press, 2011, pp. 27–40. DOI: 10.1017/CBO9780511921452.003.