A Domain-Specific Language for the Development of Heterogeneous Multi-Robot Systems

Daniel Steen Losvik

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Bergen University College

Department of Informatics,
University of Bergen

June 2019

Western Norway
University of
Applied Sciences

Abstract

Robots are becoming more advanced each year and will increasingly become a bigger part of our lives. This thesis explores how model-driven software engineering can be used in the development of heterogeneous multi-robot systems where you have different robots with different capabilities. Multiple robots can achieve more complex tasks that are impossible to achieve for a single robot alone. This thesis proposes a framework where simple actions are used as building blocks to define larger tasks that require multiple robots with different capabilities to achieve. The thesis explores how task distribution can be performed in such a system and how the robot operating system can be utilized. The thesis also explores how a user interface can be used to define multiple different missions for a team of heterogeneous robots without the need for redeployment on each robot.

Acknowledgements

I would like to thank my supervisor Prof. Adrian Rutle for his continuous support throughout my work. His guidance and feedback have been invaluable. I would also like to thank Prof. Rogardt Heldal for his teachings on research in software engineering. Finally, I would like to thank my family for their support throughout my study.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Introduction

1.1 Background

As robots are becoming more complex and able to perform more complex tasks they will increasingly become a bigger part of our lives. In the recent past robots have mainly been used for repetitive tasks in manufacturing like building cars and electronic components on fully automated production lines. These are relatively easy tasks as the robots are placed in factories where no external forces can affect them, hence they have no need for sensors measuring their surroundings or the need for adaptive planning to deal with a dynamic environment. Still, these robots have increased the efficiency of the work and the quality of the products and have had a big economic impact. Today robots are becoming increasingly more used to complete everyday tasks and assist humans, like house cleaning robots. These types of robots are referred to as service robots and are more complex than industrial robots as they have to deal with dynamic environments.

As robots are becoming more advanced it becomes more relevant to do research on cooperative multi-robot systems. Applications which involves multiple robots working together. Figure 1.1 shows the yearly robot soccer world cup. Here you have multiple robots with different roles working together to achieve a common goal.

Figure 1.1: Robocup. The Robot Soccer World Cup. Source [1].

Multi-robot systems have a number of advantages over single-robot systems. Multiple robots can complete a more complex task which might be impossible using a single robot. Multiple robots can also often complete tasks faster than a single robot depending on the task that is being done. For example, search and exploration tasks can be completed faster as the robots can work in parallel. Using multiple robots are also more reliable in the way that if a robot failure occurs another robot can replace the failed one. Multi-robot systems introduce multiple additional challenges that need to be solved but have the potential to increase automation and the efficiency of work in many sectors.

This thesis explores how model-driven software engineering can be applied to simplify the development of heterogeneous multi-robot systems. In model-driven software engineering, models are used in the development process to define the system and then the code is derived from the models. Model-driven software engineering raises the level of abstraction and is well suited to deal with the complexity of heterogeneous multi-robot systems.

The thesis proposes a framework for developing heterogeneous multi-robot systems. The framework is made up of 4 components. A domain-specific language used to model both the robots and tasks, a task allocation module used to distribute the tasks amongst the robots, the robot operating system(ROS) used for

communication between the robots and advanced navigation, and a web interface used to create missions for teams of robots.

1.2 Challenges

Heterogeneous multi-robot systems introduce two big challenges over single-robot systems. The first is hardware heterogeneity between the robots. As different robots often are built using different hardware(sensors, actuators, microcontrollers) they need to be programmed in different ways. If you are developing a system comprised of multiple different types of robots that have different hardware, you need to write a program for each robot independently using different tools, libraries, and frameworks. This is highly inefficient and requires the developer to do a lot of research.

The second challenge is the task distribution between the robots. If one has multiple different robots with different capabilities and multiple different tasks, how should the tasks be distributed amongst the robots? This problem is referred to as multi-robot task allocation or MRTA [2]. This is an optimization problem that has been studied a lot as it can be reduced to many similar problems outside robotics, like multiprocessor scheduling.

1.3 Motivation

The goal of the thesis is to explore and acquire new knowledge on how model-driven software engineering can be applied to simplify the development of heterogeneous multi-robot systems. As model-driven software engineering is often used to raise the level of abstraction of complex systems, it is a good candidate to deal with the complexity of heterogeneous multi-robot systems. The motivation of the thesis is driven by the benefits of heterogeneous multi-robot systems. In hetero-

geneous multi-robot systems, the robots might have different capabilities and can work together to perform more complex tasks. This allows a bigger variety of tasks to be achieved than before.

1.4 Research Questions

The main focus of this thesis is the application of model-driven software engineering in the development of heterogeneous multi-robot systems and how task distribution can be performed in such a system. The research questions are chosen accordingly.

Main research question of the thesis:
How can model-driven software engineering be used to simplify the development of heterogeneous multi-robot systems?

Sub-questions:
How is model-driven software engineering applied to the development of heterogeneous multi-robot systems in today's research?

How can efficient and appropriate task allocation be achieved in different heterogeneous multi-robot systems?

1.5 Method

The research is split into two parts. First, a systematic literature review is conducted. This gives us an overview over existing research and related work. Design science research is then used to gain new knowledge through the design and evaluation of an artifact.

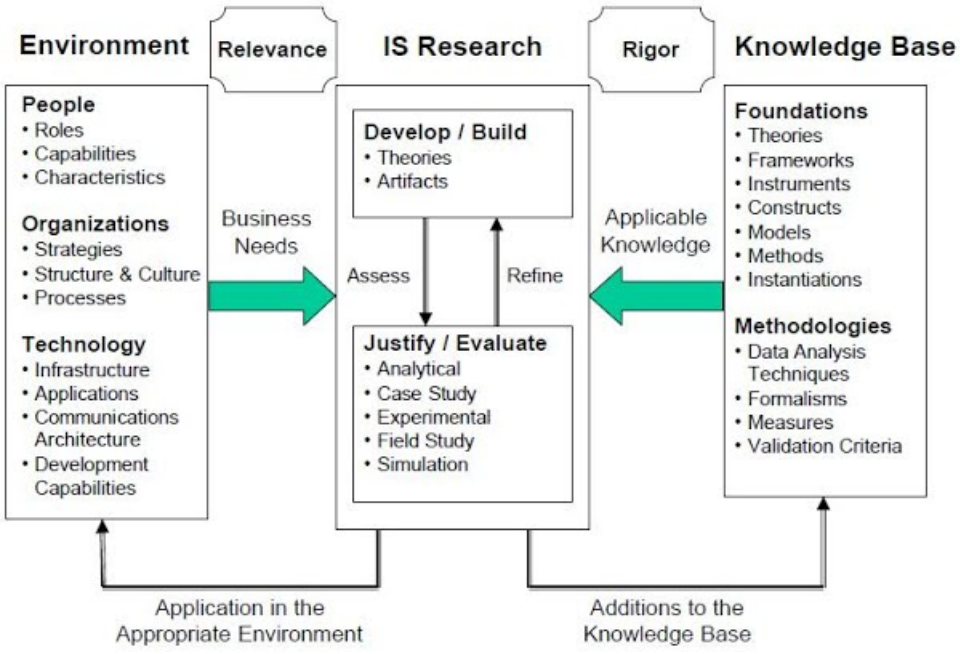


Figure 1.2: The Design Science Research Process. Source [3].

Design science is a research method often used in computer science. In design science, the research revolves around the design and evaluation of a so-called artifact[3]. The artifact can be an algorithm, framework, model, method, etc. The research is conducted in iterations by designing the artifact, implementing it, evaluating it, and then redesigning it and so on.

Figure 1.2 illustrates the process and is referred to as the three cycles of design science research [4]. The research is done in cycles of developing, building, justifying and evaluating. Evaluation of the artifact may lead to new knowledge which can be added to the knowledge base of the field. Field evaluation of the artifact in the appropriate environment is used to justify the relevance of the artifact in real-world applications.

In this thesis, the artifact is the proposed framework. The framework defines a process for how to design and implement a heterogeneous multi-robot system. Evaluation of the framework is done using simulation software with the goal of

obtaining new knowledge on how model-driven software engineering techniques can be used to design and implement heterogeneous multi-robot systems.

1.6 Thesis Outline

The rest of the thesis is structured as follows: In ***chapter 2*** a literature review is conducted to present related work. ***Chapter 3*** provides some background in the concepts and technologies used in the thesis. ***Chapter 4*** presents the developed framework and all of its components. In ***chapter 5*** the solution is evaluated. In ***chapter 6*** the solution is discussed, and in ***chapter 7*** the thesis is concluded.

Chapter 2

Systematic Literature Review

2.1 Introduction

In order to get an overview over existing work on the application of applying model-driven software engineering methods in the development of heterogeneous multi-robot systems, a systematic literature review(SLR) is conducted. The goal of a SLR is to identify and evaluate all available research on a specific topic in a systematic way using a well-defined methodology [5].

The method used will be based on Kitchenhams guidelines for conducting SLRs in software engineering [5]. Here Kitchenham derives research methods from other fields like medicine and adapts them to reflect the specific problems of software engineering research. Since SLRs are often quite big and involve multiple researchers, a light version will be used for this master thesis, proposed by [6]. Being a single researcher also increases the chance of biased results, which is a threat to validity. Conducting a literature review in a systematic way with a defined protocol reduces this threat.

2.2 Previous Work

The SLR will be based on a previous systematic mapping study conducted in 2015 [7]. The mapping study is on the topic of model-driven software engineering for mobile robot systems, which is a superset of the topic for this thesis: model-driven software engineering for heterogeneous multi-robot systems. In the study, they collected an initial set of 1681 papers from various digital libraries and by using forward snowballing. They then applied inclusion and exclusion criteria to exclude papers not related to software engineering or mobile robot systems. The result was 69 papers which were then classified into categories based on the type of publication, year released, type of research(evaluation research, solution proposal, opinion paper, etc.), and type of contribution(tool, method, model, etc.).

(mobile OR drive OR cruise* OR rover OR ground OR *water* OR aer*
OR fly* OR sail*)*
AND
(unmanned OR self OR autonomous OR robot OR vehicle*)*
AND
*(MDE OR MDD OR MDA OR MDSD OR meta model OR metamodel OR dsl OR
domain specific OR dsml OR model-driven OR model driven)*

Figure 2.1: Syntax of the Search String from the Mapping Study.

The conclusion of the mapping study was that model-driven software engineering methods are an increasing trend to use in mobile robot systems. Most research solutions are focused on the development of domain-specific modelling languages supported by tools that are mostly built ad-hoc. Fewer solutions are based on UML and Eclipse-based tools. They also concluded that there are few solutions that are validated through real-world projects which reflects that research on this topic is still young. They also concluded that there is a weak presence of studies on mobile multi-robot systems and that more research should be invested on teams of mobile robots.

Since this was a broad mapping study we can be confident that it has captured most relevant papers on model-driven software engineering for mobile robot-systems which also includes papers on model-driven software engineering for multi-robot systems. The SLR of this thesis will be conducted on the papers from the mapping study which is concerned with model-driven software engineering in multi-robot systems, not only heterogeneous systems as the research here is very limited. The previous mapping study was conducted in 2015 which means that it only contains papers released from 2000 - 2015. To identify papers released after 2015 a short database search will also be conducted.

2.3 The Need for a Review

The previously conducted mapping study concluded that the trend of using model-driven software engineering in the development of robot systems is increasing. This means that the use of model-driven software engineering is also increasing in the development of multi-robot systems and is probably going to continue to increase because of all the advantages of using model-driven software engineering in complex systems.

A short database search is conducted to check if there already exists a SLR on the topic of model-driven software engineering in multi-robot systems. The search string shown in figure 2.2 is used to perform an automatic search in the electronic database *"Ieee Xplore"* and index library *"Scopus"*. The result shows that there does not exist a SLR on the topic.

("literature Review" OR "mapping study" OR "Survey" OR "SLR" OR "SMS")
AND
("robot")
AND
("multi" OR "team" OR "swarm")
AND
("model-driven" OR "domain-specific" OR "DSL" OR "MDA" OR "MDE" OR "MDD" OR "MDSE")

Figure 2.2: Syntax of the Search String used to find Similar Literature Reviews.

2.4 Review Protocol

In this section we define the review protocol. This includes review questions, database search strategy, study selection criteria, and data extraction strategy.

2.4.1 Review Questions

The overall review questions of the SLR are as follows:

How are MDSE methods and techniques applied in the development of multi-robot systems?

What tools and frameworks exist to support the use of MDSE methods and techniques to develop multi-robot systems?

To answer the questions some sub-questions are defined which focus on what tools and frameworks are used, what type of solutions are developed, and how model-driven software engineering techniques are applied(in the form of a model, DSL, graphical tool, etc.).

RQ1: *What types of tools and frameworks exist which support the development of multi-robot systems using MDSE methods?*

RQ2: *How are MDSE methods applied (In the form of a model, a DSL, etc)?*

RQ3: *What type of problem does MDSE solutions solve (behavior, communication, task allocation, etc)?*

2.4.2 Database Search

To identify papers released after 2015 a short database search is conducted through the digital databases *"Ieee Xplore"* and *"scopus"*. The search string shown in figure 2.3 is defined to collect relevant papers. As this is a light review with only one researcher the search string defined is more narrow than the one used by the mapping study. The search is limited to only conference papers and articles written in English.

("robot")
AND
("multi" OR "team" OR "swarm")
AND
("model-driven" OR "domain-specific" OR "DSL" OR "MDA" OR "MDE" OR "MDD" OR "MDSE")

Figure 2.3: Search String used to find Additional Papers Published after 2015.

2.4.3 Selection Criteria

To be able to select relevant papers from the previous mapping study and the database search a series of inclusion and exclusion criteria are defined. These

criteria are applied to the title and abstract of each paper and should be defined to capture studies focusing on the use of model-driven software engineering in the development of multi-robot systems. The criteria are defined as follows:

Inclusion criteria:

- Studies proposing MDSE methods or techniques that can be applied in the development of multi-robot systems.
- Studies applying or evaluating MDSE methods or techniques used to develop multi-robot systems.

Exclusion criteria:

- Studies not concerned with MDSE or multi-robot systems.
- Studies not concerned with software development (i.e., studies on robotic hardware or mechanics).
- Not peer review studies.
- Studies published before 2015 (for the papers from the database search).

2.4.4 Data Extraction

The data extracted from each study should provide the necessary information which can be used to answer the defined review questions.

Data extracted:

- Type of MDSE solution developed (model, DSL, graphical tool, etc.).

- Focus area(behavior, communication, task allocation, etc.).
- Tools used to develop the solution.
- Evaluation method used.

2.5 Results

From the mapping study, 8 papers were selected from the initial set of 69 using the selection criteria. From the database search, 4 additional papers were selected from a total of 31 produced by the search. The papers from the database search capture additional studies published after the broader mapping study from 2015. A total of 12 studies were selected [see table 2.1].

2.6 Discussion

On the topic of applying model-driven software engineering in the development of mobile multi-robot systems we can see from table 2.2 that most studies focus on the use of MDSE to create tools and DSLs that can be used to describe the behavior of the robots on an abstract level while low-level platform-dependent code is partly generated. This lets users create an application for a team of robots easier and more efficiently.

In [S7], [S8], [S9] and [S10] DSLs with graphical tools was developed which can be used to specify the behavior of the robots using finite state machines and statecharts. While in [S4] and [S5] they showed how finite state machines can be used to model a swarm of heterogeneous robots using RoseRT. Finite state machines and statecharts are popular to use to model robot behavior as they are good for capturing real-time requirements.

Id	Author	Date	Name
[S1]	D. D. Ruscio, I. Malavolta, & P. Pelliccione	2014	A Family of Domain-Specific Languages for Specifying Civilian Missions of Multi-Robot Systems
[S2]	F. Ciccozzi, D. D. Ruscio, I. Malavolta, & P. Pelliccione	2016	Adopting MDE for Specifying and Executing Civilian Missions of Mobile Multi-Robot Systems
[S3]	S. Dragule, B. Mayers, & P. Pelliccione	2017	A Generated Property Specification Language for Resilient Multirobot Missions
[S4]	D. Quellet, S. N. Givigi, & A. J. G, Beaulieu	2011	Control of swarms of autonomous robots using Model Driven Development - A state-based approach
[S5]	A. J. G, Beaulieu, S. N. Givigi, D. Quellet., & J. T. Turner	2018	Model-Driven Development Architectures to Solve Complex Autonomous Robotics Problems
[S6]	C. Pinciroli & G. Beltrame	2016	Buzz: An extensible programming language for heterogeneous swarm Robotics
[S7]	T. Amma, P. Baer, K. Baumgart, P. Burghardt, K. Geihs, J. Henze, S. Opfer, S. Niemczyk, R. Reichle, D. Saur	2009	Carpe noctem 2009
[S8]	H. Skubch, M. Wagner, R. Reichle., & K. Geihs	2011	A modelling language for cooperative plans in highly dynamic domains
[S9]	A. Paraschos, N. I. Spanoudakis, & M. G. Lagoudakis	2012	Model-driven behavior specification for robotic teams
[S10]	E. M. Martinez, A. F. Caballero, & J. M. G. Noheda	2012	Model-driven engineering techniques for the development of multi-agent systems
[S11]	P. A. Baer, R. Reichle, M. Zapf, T. Weise, & K. Geihs	2007	A generative approach to the development of autonomous robot software
[S12]	P. A. Baer, R. Reichle, & K. Geihs	2008	The spica development framework–model-driven software development for autonomous mobile robots

Table 2.1: SLR General Data.

Id	Type of solution	Focus area	Tool used	Evaluation
[S1]	DSL & UI	Behavior	EMF	Real-world application
[S2]	DSL & UI	Behavior	EMF	Real-world application
[S3]	DSL	Constraints	EMF	Real-world application
[S4]	Method	Behavior	RoosRT	Simulation
[S5]	Method	Behavior	RoosRT	Simulation
[S6]	DSL	Behavior	EBNF	Simulation
[S7]	Graphical Tool	Behavior	EMF	Real-world application
[S8]	Graphical Tool	Behavior	EMF	Real-world application
[S9]	Graphical Tool	Behavior	EMF	Real-world application
[S10]	Graphical Tool	Behavior	EMF	Real-world application
[S11]	DSL	Communication	EBNF	Real-world application
[S12]	DSL	Communication	EBNF	Real-world application

Table 2.2: SLR Review Data.

In [S1] and [S2] they developed multiple DSLs together with a user interface which lets the user specify the task for each robot on a map. The user interface was designed for aerial vehicles, however. In [S6] on the other hand, they developed a DLS which lets you specify the behavior of a swarm of robots using a textual language and not state machines. In [S9] and [S10] they showed how agent-based model-driven software tools could be applied to specify the behavior of a multi-robot team. In [S3] they developed a DSL that can be used to define task constraints. A constraint can, for example, be that a robot can only perform a certain task if another robot is at a certain position.

Very few studies involve the use of MDSE to solve problems like communication, task allocation and coordination between the team of robots. These problems are often solved using specific algorithms and are hidden from the user. [S11] and [S12] were the only studies that proposed DSLs which lets you model the communication infrastructure of the robot team. The DSLs can be used to define the messages and protocols the robots use for communication on a platform-independent level. While in most other studies like [S1] and [S8] the communication is achieved using well-known middlewares like ROS and is not modeled by the user.

However, most of the proposed solutions assume that low-level control functions for each robot are provided. This is necessary because of the high amount of different hardware and software libraries that are possible if any types of mobile robots are going to be supported. To complete a specific task the robot might need a function like *"Moveforward(m/s)"* which is implemented in very different ways depending on the wheel configuration and what microcontroller the robot use. The idea in most of the studies is that low-level control functions are defined independently by robot specialists and then the mission of the robot team can be specified by a non-specialist using the proposed language or tool.

Most of the proposed solutions are developed using the Eclipse Modelling Framework. This is an advantage as different solutions developed using the same framework are often easier to integrate and reuse.

2.7 Conclusion

Most studies on the topic of applying MDSE to develop mobile multi-robot systems are concerned with the ability to specify the mission or behavior of a team of robots on a high level using finite state machines, statecharts or in some cases a textual language. As there is a high amount of different hardware and software libraries used on different robots the proposed solutions often assume low-level control functions are provided for each robot before a mission can be specified.

There are few studies concerned with applying MDSE to model other features of a multi-robot system like communication infrastructure, coordination and task allocation. These features are often implemented using specific algorithms and middlewares in the different solutions and are not modeled by the user. A good amount of the proposed solutions are developed using the Eclipse Modelling Framework. This indicates that EMF is starting to become mainstream in the application of applying MDSE techniques to software development.

For future work, more research should be done to figure out how a mission for a heterogeneous team of robots can be specified on a high level without the need of a robot specialist implementing the low control functions for each different robot. As most studies focus on the ability to define a mission using state machines and domain-specific languages, there is also needed more research on how missions can be specified at runtime by non-programmers using maps or user interfaces.

In this thesis, we propose a textual DSL for modelling the behavior of the robots by using simple actions as building blocks to define tasks that involves multiple robots. The user can specify a mission for a team of robots through a web interface at runtime. As opposed to other solutions our solution allows the user to define different missions without the need for redesigning the model, regenerate code and redeploy the generated code to all the robots. The user can also add new tasks and change the definition of existing tasks without the need for redeployment.

Chapter 3

Theoretical Background

3.1 Overview

This chapter provides some background in the concepts and technologies used in the thesis. Beginning with an introduction to model-driven software engineering and MDSE in robotics. Then moving on to the multi-robot task allocation problem. Finally, an overview of the robot operating system is provided.

3.2 Model-Driven Software Engineering

3.2.1 What is Model-Driven Software Engineering?

Model-driven software engineering is a software development paradigm that focuses on the use of models in the software development process. In model-driven software engineering, you often develop a high-level model of the application instead of writing low-level code, the code is then derived from the model using model-to-model transformation and code generation. The use of models allows developers to work at a higher level of abstraction, thus reducing complexity and improving the software quality [8].

Model-driven software engineering is based on the separation of the system functionality being developed and the implementation of such a system for one specific platform, i.e., to clearly separate the analysis from the implementation details. Thus raising the level of abstraction and allowing the use of concepts closer to the problem domain [9].

The concept of raising the level of abstraction to reduce complexity is nothing new. Assembly can be seen as an abstraction over machine code, while high-level programming languages can be seen as an abstraction over assembly again which hides low-level machine-specific instruction by introducing higher-level abstractions such as variables that are translated into machine code by the compiler [8].

Figure 3.1 illustrates common elements used in model-driven software engineering. At the top level is the metalanguage used to define the domain-specific language. This could be Ecore, extended backus-naur form, or any language capable of defining another language. The domain-specific language can then be used to define a model of the application that contains the functional requirements. Transformation rules can be specified in a transformation language like ATL or M2M. The model can then be transformed into another model or into application code. And the application code usually uses a framework that conforms to an architecture.

These components can be implemented by developers that have different roles as described in [10]. You may want a domain expert and language engineer to develop the language, transformation, and platform expert to implement the transformation rules, and an application developer to develop and test the model.

3.2.2 Model-Driven Architecture

Model-driven architecture is a software development approach initiated by the OMG (The Object Management Group) [11]. Model-driven architecture is often referred to as a subset of model-driven software engineering. It provides a set of guidelines for defining models and the transformations between them.

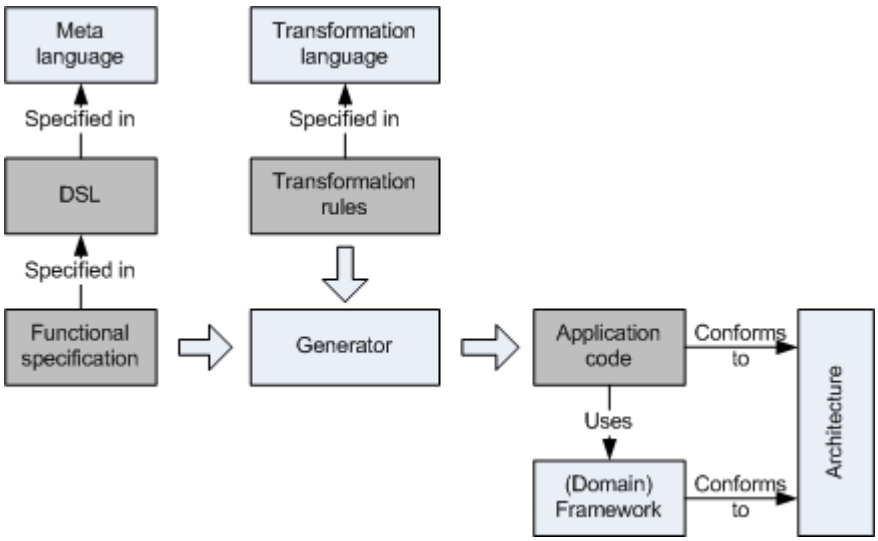


Figure 3.1: The Model-Driven Software Engineering Process. Source [10].

Figure 3.2 illustrates the different layers of models in model-driven architecture. As explained in [11] the models can be separated into 3 layers. The computing independent model (CIM) defines the system specifications on the highest level. It defines all the system functions without any technical specifications. The platform-independent model (PIM) defines common platform-independent concepts. The platform-specific model (PSM) uses the PIM and platform details to generate the final source code. A single PIM is often used to create multiple PSM‘s. Model to model transformations are used to convert the models down the layers while model to text transformation is used to generate the final code from the PSM.

3.2.3 Pros and Cons of Using MDSE

Model-driven software engineering can have many advantages over traditional coding depending on the complexity of the system being developed. In [12] they discuss some of the benefits with model-driven software engineering. It can in-

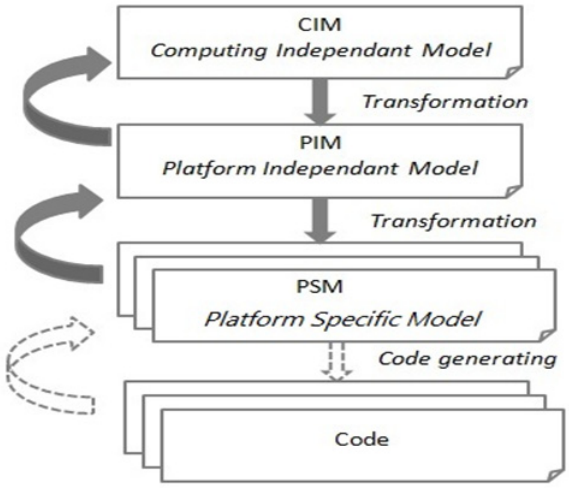


Figure 3.2: Levels of Abstraction in Model-Driven Architecture. Source [11].

crease the productivity of the developers as the use of models and code generation can lead to faster development. It can increased reusability over different platforms as you often develop platform-independent models. It can increase the quality of solutions since the use of models forces you to focus on the design of the system and not implementation details. It can improve the communication between developers as they can reason about high-level concepts instead of low-level code. And it can improve communication with stakeholders as models hide implementation details and are closer to the problem domain.

In [13] they conducted a twelve-month long empirical study to investigate whether all the claimed benefits of model-driven software engineering is true or not. They concluded that the use of model-driven software engineering does have a positive effect on productivity and maintainability.

In [8] some of the problems of model-driven software engineering are discussed. It can increase redundancy as there are multiple representations of the same artifact(generated files, generated documents, generated models, etc.) at different levels of abstraction. If these are manually created then duplicate work is required. Also the more levels of abstraction and the more models you have, the

more complex model relations you get. Changes in one model could propagate and lead to unexpected changes in all related artifacts. Raising the level of abstraction also may lead to oversimplification and may hide important implementation details from the developer.

3.2.4 MDSE in Robotics

One of the main problems with software development in robotics is due to the hardware heterogeneity of different robots. Different robots are often built from different types of sensors, actuators, and microcontrollers. All these components can also be put together in many different ways. As a consequence different robots need different code to be able to perform the same functions. This makes code reuse difficult and one often has to start from scratch when developing a new robotic system.

In [14] and [15] they conducted studies on research trends related to software architecture in robotic systems. Both studies concluded that model-driven architecture was one of the most popular and promising architecture to apply when developing software for robotics systems. Since robotic systems often are quite complex, raising the level of abstraction can be very beneficial. It also increases reusability by introducing platform-independent models. By separating hardware-specific and hardware-independent specifications we can create more reusable robotic components.

3.2.5 Domain-Specific Languages for Robots

One of the core concepts in model-driven software engineering is that of a domain-specific language or DSL. A DSL is a software development language specialized for developing applications in a particular domain, for example robot applications, or even more specific like robot perception. The language should make it easier

to develop an application for that particular domain by defining abstractions and notations relevant to that specific domain.

As described in [16] a DSL usually contains only a restricted set of notations and abstractions as compared to a general-purpose language or a general-purpose modelling language like UML. This allows a DSL to highlight domain concepts in the language itself, while in a general-purpose language the domain concepts have to be implemented in the code and are more hidden. A DSL for robotic systems should, therefore, highlight concepts and problems specific to the development of robot applications.

There already exist many DSLs for robot development. In [16] they conducted a survey on 41 DSLs. The DSLs usually only deals with a very specify function like perception or control, which is the essence of a DSL as opposed to a general-purpose language. There also exists larger model-driven toolchains like RobotML [17], BRICS [18] and Smartsoft [19] which contains multiple DSLs to be used together when developing the robotic system.

3.2.6 How are the DSLs Developed?

The survey conducted on 41 different domain-specific languages for robotic systems [16] showed that most of the domain-specific languages were developed using the Eclipse Modeling Framework (EMF).

The Eclipse Modeling Framework provides a toolchain for the development of domain-specific languages and metamodels. It provides a large set of tools and code generation facilities to support metamodelling. The metalanguage used in EMF is called Ecore and is based on the MOF (Meta Object Facility) metalanguage defined by the Object Management Group.

After EMF most domain-specific languages in the survey were developed by creating a custom toolchain or by using a general-purpose language. A domain-specific language developed using a general-purpose language is called an internal domain-specific language and is developed by extending the syntax of the host

language with domain-specific notations and abstractions.

3.3 Multi-Robot Task Allocation

3.3.1 The Multi-Robot Task Allocation Problem

One of the challenges in a multi-robot system is the problem of task distribution. Given a set of robots and a set of tasks, how do we decide which robot should perform which task? The multi-robot task allocation (MRTA) problem is about how a set of tasks should be distributed between a set of robots and is an optimization problem. As the number of heterogeneous robots and different types of tasks increases the problem becomes even more complex since heterogeneous robots may have different capabilities and different strengths and weaknesses.

The problem of efficient task allocation has been an active research topic for many years as the different variations of the problem can be reduced to many similar problems, like multiprocessor scheduling [2]. In this section we will look at some of the different variations of the MRTA problem, cost functions, and solution architectures.

3.3.2 Problem Variations

In [2] they proved a taxonomy of MART problems and the different ways of classifying them. They explain how the problems can be classified along 3 main axes as shown in figure 3.3.

Single-task robots (ST) vs multi-task robots (MT): ST means that each robot can only perform a single task at the time, while ST means that there are some robots with the ability to perform multiple tasks simultaneously.

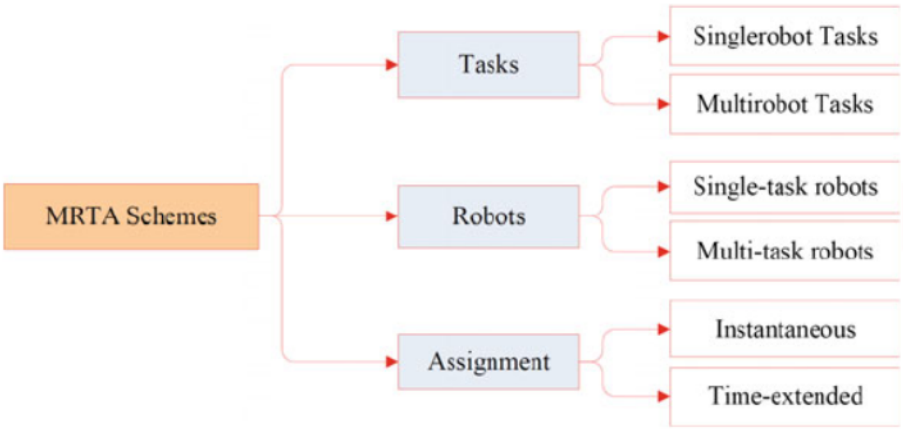


Figure 3.3: Multi-Robot Task Allocation Schemes. Source [20].

.

Single-robot tasks (SR) vs multi-robot tasks (MR): SR means that each task requires exactly one robot to achieved it, while MR means that there are some tasks which require multiple robots.

Instantaneous assignment (IA) vs time-extended assignment (TA): IA means that only Instantaneous allocation of tasks is allowed, while TA means that planning for future tasks are allowed. For example, if there are more tasks than robots you might want to create a schedule for each robot.

A specific MRTA problem is then referred to as for example ST-SR-IA. This gives us 8 types of problems that require different approaches to solve. ST-SR-IA is the simplest variation of the problems as you only have one-to-one relations between the robots and the tasks with no future planning. As explained in [2] there exist algorithms that are able to find the optimal solution to this type of problem, like the Hungarian method [21]. While an instance of the ST-SR-IA problem can be solved efficiently the remaining problem variations are all NP-hard and there exist only approximation algorithms that find a sub-optimal task distribution [2].

Besides these 3 axes, there might be additional factors that change the problem. One thing that is done a lot of research on in multi-robot systems is task constraints [22]. In [23] they list some of the most commonly used task constraints. For example, there might be tasks that need to be completed before a set of other tasks can be started. Or there might be tasks that need to be completed before a specific deadline. This again increases the complexity of the problem and would require the task allocation algorithm to be much more advanced when planning.

3.3.3 The ST-MR-IA Problem

The main focus of the thesis is on multi-robot systems where different robots with different capabilities have to cooperate. This means that the problem can be seen as an instance of the ST-MR-IA problem. As they explain in [2] when you have tasks that require multiple robots working together to complete, the problem becomes significantly more difficult. This problem is often referred to as a coalition formation problem where a coalition is a group of robots. The problem of dividing a set of robots into coalitions for each task such that the cost of performing the tasks is minimized can be reduced to the maximum utility set partitioning problem. The problem is NP-hard as finding the optimal solution would require going over all possible task-collision pairs to find the solution with minimum cost [2].

They also mention in [2] that there has been a lot of studies on this type of problem because its application in solving crew scheduling problems for airlines, so there exist many good heuristics algorithms which can find suboptimal solutions in a reasonable amount of time, like [24] and [25]. These, however, are not directly applicable to MRTA problems as they are designed to solve crew scheduling problems and are slightly different.

3.3.4 Cost Function

The MRTA problem can be looked at as an optimization problem since we are trying to find the best distribution of tasks between the robots to optimize the performance of the whole mission. Depending on the mission you may want to optimize different performance metrics. If you have a time-critical mission you might want to minimize the time of completing the slowest task so the whole mission is completed as early as possible. If the robots are using fuel you might want to minimize the sum of distances traveled by each robot to minimize fuel usage. In [23] they refer to these as optimization objectives and list some of the most commonly used ones.

What to optimize is expressed through what is called a cost function. For each robot-task pair, a number is calculated based on what the cost is to perform the task by the robot. The cost can be calculated by the distance between the robot and the task or by how long time the robot needs to complete the task or some other available variable based on what you want to optimize.

A cost function is necessary to be able to optimize, but what is called a quality function is also common to use. For each robot-task pair, a number is then calculated based on how well the robot is able to perform the task. This could be based on the speed of the robot or the sensor accuracy or any available variable or combination of variables.

We can then define the utility of the robot as the quality minus cost. The robot-task pair with the highest utility is then paired. In [2] they define the utility as shown in figure 3.4, where U is the utility value of the robot-task pair, Q is the quality and C is the cost. Although there are multiple ways of defining the utility this is one of the most used commonly used ones.

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

Figure 3.4: Utility Function. Source [2].

3.3.5 Solution Architectures

Multi-robot architectures can be classified into two main types as they explain in [20].

Centralized architecture is when there is a single robot that acts as the central unit. This robot runs the task allocation algorithm and allocates the tasks to the other robots. The advantage of a centralized system is that it is easier and cheaper to implement. The disadvantage of a centralized system is that it has a single point of failure. It is also less scalable and robust than a distributed system.

Decentralized architecture is when there is no central unit, but each robot communicates with other robots directly to collect data from each other, and each robot decides on their own how to proceed based on the collected data. This type of system is much more robust and flexible as it has no single point of failure. The processing of data is more distributed which makes the system more scalable as well. The disadvantage is that it is more difficult and expensive to implement a decentralized system. Each robot also has less information about the whole system, as this requires each robot to message all other robots.

Amongst the most popular types of multi-robot task allocation algorithms are the auction and market-based algorithms [26]. Here each robot calculates a bid on each task based on its utility function and then the central unit acts as the auctioneer which allocates each task to the highest bidder. Figure 3.5 illustrates an auction-based architecture.

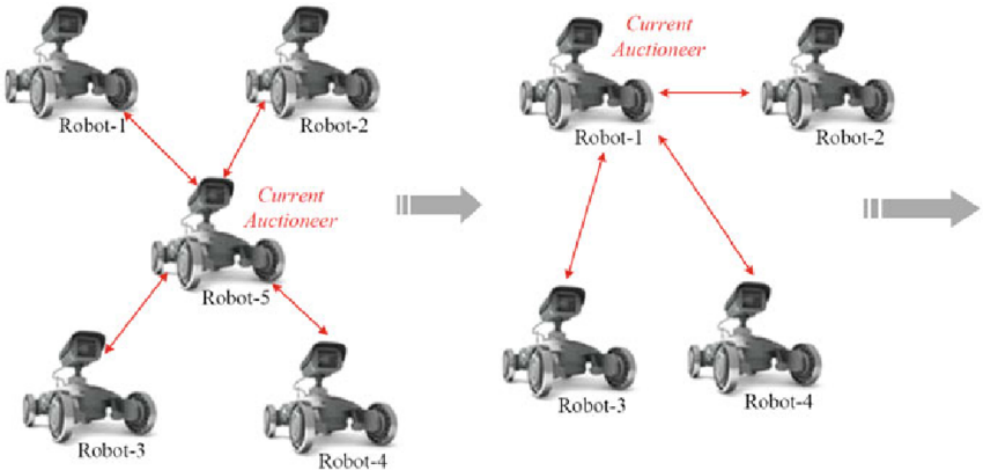


Figure 3.5: Auction-based Architecture. Source [20].

3.4 The Robot Operating System

3.4.1 What is ROS?

ROS (Robot Operating System) is an open-source robotics middleware. It consists of a collection of tools and frameworks for the development of robots. It is not an operating system but provides services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers [27].

ROS has grown into one of the most popular middleware for robotics and it does now contain a huge amount of packages implementing functions commonly used in robotics like navigation algorithms. This allows developers to focus on the logic of their application, while ROS provides finished implementations of algorithms and other commonly used functionality.

3.4.2 ROS Architecture

A ROS application is made up of nodes. A node is a process that performs computations and communicates with other nodes. A ROS node usually either process sensor data from a single sensor, runs sensor data through an algorithm or controls an actuator. The node structure makes the system loosely coupled and promotes component-based development. The nodes can be running on the same computer or on multiple different computers. ROS uses a publish-subscribe architecture for communication between all the nodes. Nodes can publish data on topics or subscribe on topics to receive data from other nodes. A node called ROS-master acts as the broker which takes care of the coordination of the massages between all the nodes.

A node is implemented by creating an executable and use the ROS client library to create and subscribe to topics. ROS has tools to launch all nodes at once which starts up the application and starts the sensor processing and actuator control. A ROS application usually consists of a combination of nodes provided from ROS repositories and nodes made by the developer to deal with the sensors and actuators of the specific robot used.

Figure 3.6 illustrates a common ROS application called the navigation stack [28]. Given a goal coordinate, the application will navigate your robot to the goal while avoiding collision with obstacles. The blue nodes are platform-specific and need to be implemented by the developer. One node reads and publishes laser data, another publishes odometry data (i.e., position and orientation), another publishes sensor transforms which is the position of the sensors on the robot which may change over time, and one node subscribes on movement messages and drives the robot.

The white nodes are provided by ROS. In this case there is a node called *"move_base"* which is again made up of 5 nodes. This node subscribes on all the topics that publish sensor data. The node can at any time be given a goal coordinate and will then run all the sensor data through multiple algorithms to plan a path to the goal and then publish movement messages that the base controller

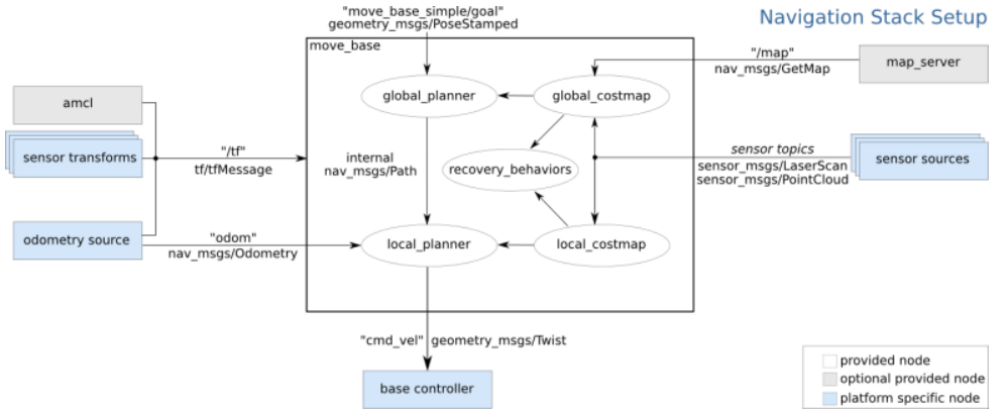


Figure 3.6: Example ROS Application. The Navigation Stack. Source [28].

node can pick up to drive the robot correctly based on its wheel configuration.

3.4.3 Pros and Cons

Besides the operating system like services for heterogeneous computer clusters provided by ROS, some additional advantages of ROS include its increase in code reusability as modules are separated into different packages. ROS also provides a large library of already implemented packages. Another advantage of ROS is that it is language-independent [27]. Most of the libraries are implemented using c++, python or lisp, but additional languages can be used. You can even use multiple nodes written in different languages in the same application. This way you can, for example, use python for object recognition as python has good support for machine learning, and use c++ for driver controls. One of the disadvantages of ROS is that it only runs on Unix-based platforms [27]. Many microcontrollers run their own operating system and therefore cannot use ROS. ROS is also considered difficult to learn.

Besides ROS, there are many other middlewares which all have their own advantages and disadvantages. Alternative middlewares include Player/stage [29], Orocos [30], Miro [31] and many others. A full list of robotic middlewares can be

found in [32] where they conducted a survey on the most popular robotic middlewares.

Chapter 4

A Framework for Heterogeneous Multi-Robot Systems

4.1 Framework Architecture

From the literature review, we saw some of the related work in applying model-driven software engineering in the development of multi-robot systems. In this thesis, we propose a framework where simple actions are used as building blocks to define more complex tasks performed by multiple robots. The framework is made up of 4 components. A domain-specific language used to model both the robots and tasks, a task allocation module used to distribute the tasks amongst the robots, the robot operating system(ROS) used for communication between the robots and advanced navigation, and a web interface used to create missions for multiple teams of robots.

The core part of the framework is the Task Definition Language (TDL) which allows the user to define all the necessary elements of a multi-robot system and also define the tasks which will be executed by the robots. The language is used to define a set of composite tasks which may require multiple robots to perform. The composite tasks are made up of sub-tasks that are performed by a single robot. The sub-tasks are made up of a sequence of simple actions. This way

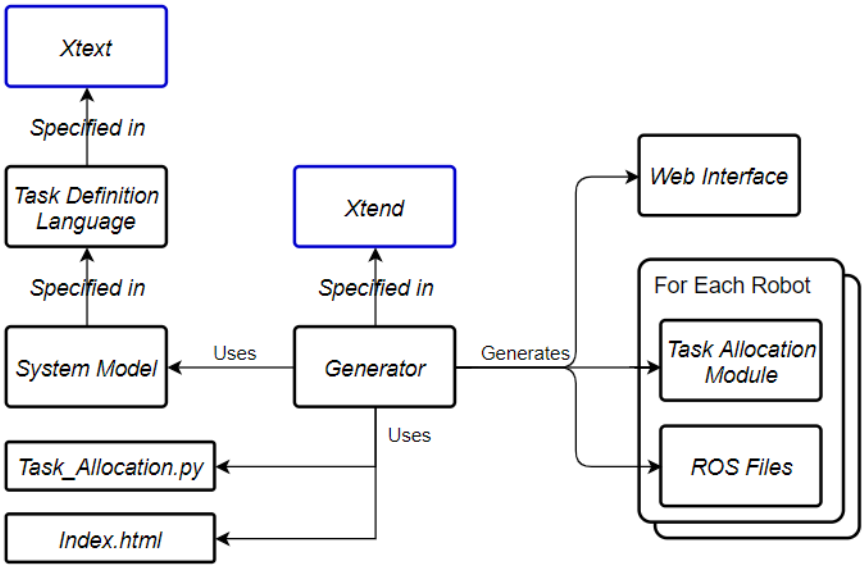


Figure 4.1: Framework Architecture. Blue Boxes are Existing Technologies.

complex tasks performed by multiple robots can be defined using simple actions as building blocks. A robot engineer adds a robot to the framework by providing an implementation of each simple action the robot is able to perform together with any necessary ROS nodes. After all the tasks are defined and the robots are added the model of the system is used to partially generate all of the components. This includes the ROS files and task allocation module for each robot, and a web interface that is used to connect to the robots and define a mission from the set of available tasks.

Figure 4.1 provides an overview of the framework and its components. At the top level is the task definition language which is defined in Xtext. Xtext is a language used to create other languages by defining syntax and semantics. The task definition language lets the user define a model of the system and all its components at a higher level of abstraction. A code generator written in Xtend is used to generate files for each robot based on the provided model. The generator uses the model together with a partially finished web interface and task allocation module

to generate the final web interface and task allocation module and all necessary ROS files for each robot. The web interface is used to define and start a mission consisting of multiple tasks and robots. ROS takes care of the communication between the robots while the task allocation module is responsible for distributing the task appropriately.

4.2 The Task Definition Language

The task definition language is the core component of the framework. The task definition language is a domain-specific language used to design and implement a heterogeneous multi-robot system. It allows you to add any custom-built robots capable of running ROS by providing an implementation of each "Simple Action" that the robot is able to perform. The language is also used to create a model that defines each task that a team of robots might need to perform by using the "Simple Actions" as building blocks. In this section, we present the design of the language and all its elements, how the language is implemented, and how the language is used.

4.2.1 How is the Language Developed?

The Task Definition Language is implemented using Xtext. Xtext is a framework for developing programming languages and domain-specific languages [33]. It provides a language that can be used to specify the semantics and syntax of another language. This way we can develop our own domain-specific language for the development of heterogeneous multi-robot systems.

Xtext is not only used to develop a domain-specific language but after the language is created Xtext generates the language components. This includes a parser that is used to check that the input text conforms to the semantics and syntax of the language. It provides the user with feedback if he misspells or has

the wrong structure on the input model. Xtext also generates all the java classes for the object model. This lets us easily iterate through all the elements of the input model which is necessary when generating code from the model. Xtext also generates the editor where we can use the language, and all the additional components which belong with a programming language.

Xtext is part of the Eclipse Modelling Framework(EMF). EMF is a modelling framework and code generation facility for building tools and other applications based on a structured data model [34]. From the literature review, we saw that most of the other solutions had used EMF to developed their domain-specific language. This is good as it makes different solutions more integratable and more standardized.

The Task Definition Language is defined by creating a Xtext model using Xtext's grammar language. The Xtext model is transferred to an Ecore model. Ecore is eclipse's meta metamodel and is one of the core parts of the Eclipse Modelling Framework used to describe different models. Figure 4.2 helps to illustrate the different layers of models defined in model-driven architecture. Ecore is at the top layer and is a M3-model which defines itself. The M3-model is the language used to create M2-models. The Task Definition Language is a M2-model and is a language used to create M1-models. M1-models represent real-world objects(M0).

The Task Definition Language is a domain-specific language which means it is used to create M1-models for a specific domain. In our case the domain is heterogeneous multi-robot systems. This means the M1-models represents a heterogeneous multi-robot system(i.e., the robots and the tasks performed by the robots).

Now that we have seen how the Task Definition Language is made we can go on to looking at the language itself and its elements and how it is supposed to be used.

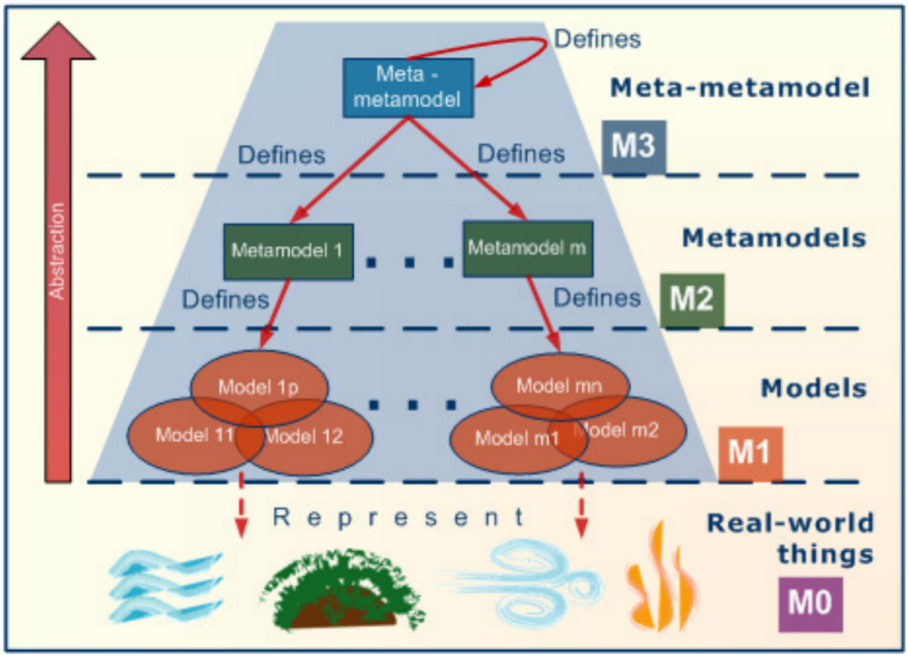


Figure 4.2: Modeling Spaces. Source [35].

4.2.2 The Elements of the Language

Figure 4.3 shows the metamodel of the Task Definition Language [A larger version can be found in Appendix A]. The language can be divided into two parts. One part for modelling the available robots, and one part for modelling the required tasks. The metamodel defines all the elements of the language. These elements should cover all the necessary concepts to enable the user of the language to develop a fully functional heterogeneous multi-robot system. In the next section, each element of the robot model and task model will be explained in further detail.

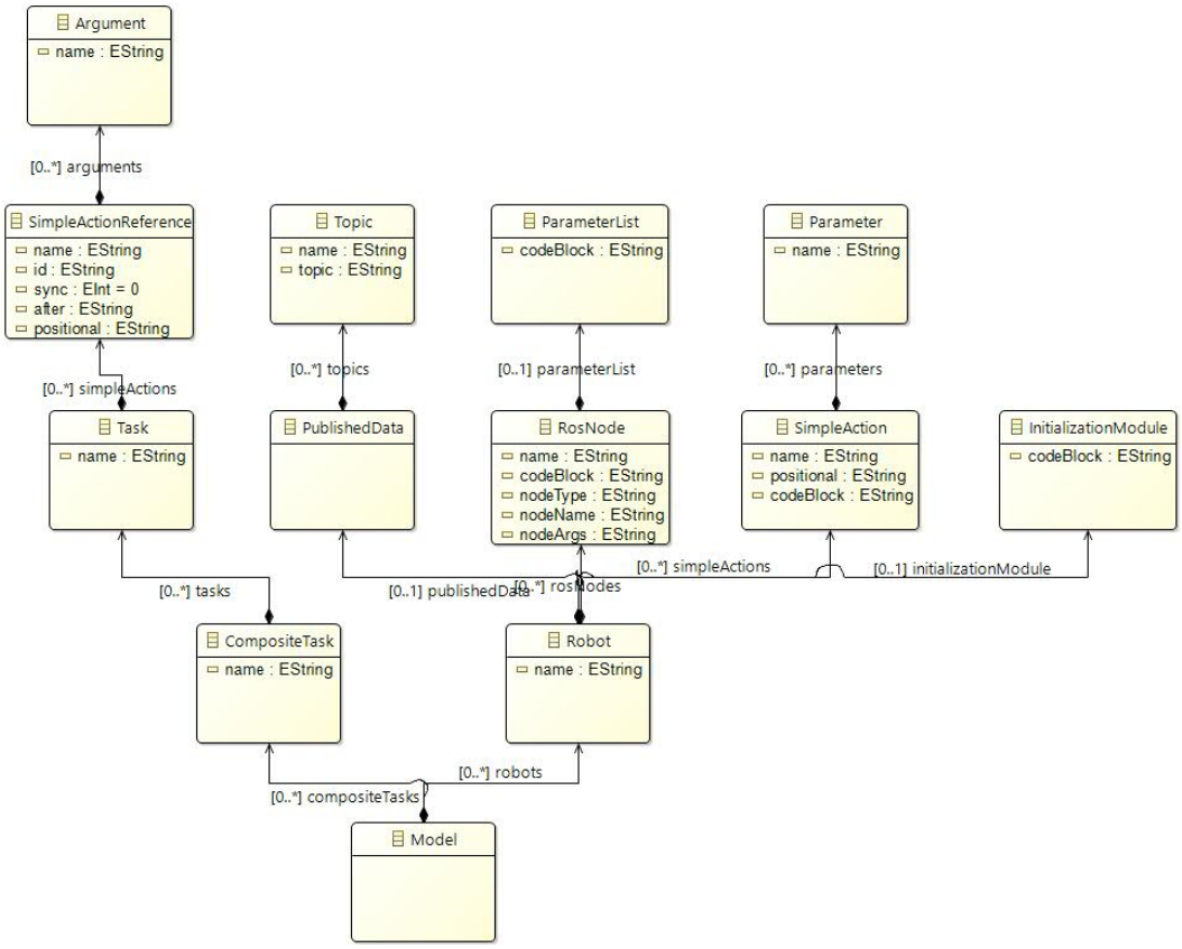


Figure 4.3: Task Definition Language Meta Model.

4.2.3 Robot Model

The part of the language for modelling the robots is mostly platform-specific as different robots need different code to perform the same action since they are built from different hardware and have different configurations. Each robot has a collection of “Simple Actions”. This is an implementation of a simple action that the robot is able to perform. The simple actions should be as generic as possible to support a large set of tasks. Each robot also has a collection of "ROS Nodes". This is an implementation of a process that continuously runs on the robot either processing sensor data or controlling actuators. A simple action can use multiple ROS nodes to achieve the desired action. Each robot also has a list of "Published Data". This is data that is shared between the robots and that is sent to the web interface where it can be displayed to the user. Lastly, each robot also has an initialization module containing imports and global variables.

Simple Action

A Simple Action is an implementation of an action the robot is capable of performing. This part of the model is hardware dependent. The element contains a python code block provided by the user which will be executed as part of a bigger task. Listing 4.1 illustrates an example "Simple Action" called "moveForward". The illustration shows how the action is implemented on a differential wheeled raspberry pi based robot. But this implementation might look completely different on another robot. This is the reason why the code has to be provided by the user and why it is so difficult to make abstract models which the code can be derived from or to make high-level libraries that can be used for all types of robots.

The core idea of the simple action element is to limit the user to only provide the code that the robot needs to perform the action, but not provide any application logic or any information about what task the robot will perform. Thus a simple action is task-independent and can be used across multiple different tasks. A set

of actions can also act as a library for a specific type of robot which can be used by others that have a similar robot.

```
simpleAction moveForward(time):
    pi.set_PWM_dutycycle(leftMotorPmwPin, 150)
    pi.set_PWM_dutycycle(rightMotorPmwPin, 150)
    _time.sleep(time * 0.001)
    pi.set_PWM_dutycycle(leftMotorPmwPin, 0)
    pi.set_PWM_dutycycle(rightMotorPmwPin, 0)
```

Listing 4.1: A Simple Action called moveForward.

ROS Node

A ROS node can be thought of as a background process that is continuously running on the robot. A ROS node can either read sensor data, control actuators or perform calculations on sensor data using complicated algorithms. The ROS nodes are typically publishing data that can be accessed from a simple action block, or a simple action can call a ROS node to help to execute the action. There are two types of ROS nodes.

The first type is ROS nodes that interface with hardware such as sensors and actuators. These are hardware-dependent and must be provided by the user. There might however exist drivers for a particular robot-hardware pair that can be used. The second type is ROS nodes performing calculation on sensor data but does not interact directly with any hardware. These are provided by ROS and often contain complicated state-of-the-art algorithms to perform for example path planning or environment mapping from various sensor data.

A user-provided ROS node contains a python code block that will be launched and executed as a process on the robot. As these nodes often contain much low-level code no example is provided here. However, the implementations of these

nodes are necessary as the framework is built on top of ROS and needs to follow the ROS architecture. Using ROS allows us to easier implement quite complicated functions for our robots. Like navigation and arm manipulation which are required to perform common actions. Sample files can be found in the Github repository of the project under multi-robot-simulation/robots [36].

Parameter List

Each ROS node may have a list of parameters that describes the robot's attributes. This could be for example the robots radius or the maximum forward velocity of the robot. These attributes are mainly used by ROS nodes to optimize the robot's behavior. By knowing, let's say, the radius of a robot a node running a path planning algorithm can calculate a more precise path around objects. A robot typically has one parameter list for each node. This is because some parameters are used to describe how the node should run also. For example, the minimum distance away from objects that the robot should be before reacting can be configured through the parameter list. The path planning node will then adjust the algorithm it is running.

Parameters are paramount when modelling robots as different robots often have many different attributes that affect their behavior [see figure 4.4]. For example, the distance between the wheels of a robot is part of deciding what velocity each wheel must have to achieve a specific angular velocity for the whole robot. Parameters are used a lot in ROS to configure algorithms and different processes, and a robot usually has tens or even hundreds of parameters.

Right now parameters are used only by ROS, but may also be used by the task allocation module in the future. You could, for example, have task requirements when defining the tasks in the system. A task may only be doable for robots with high enough velocity or a small enough radius. And the task allocation module could make sure to only allocate tasks to robots that meet the requirements.

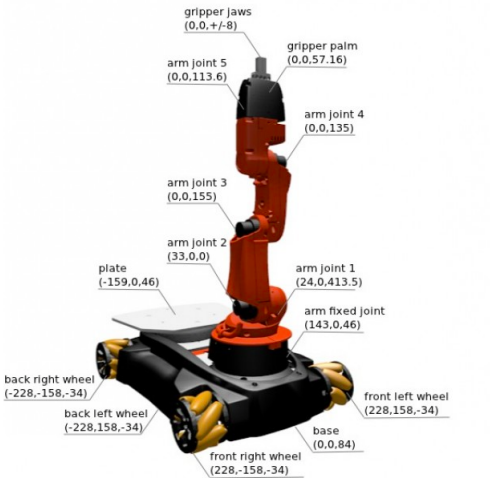


Figure 4.4: A Robot has Many Unique Attributes. Kuka Youbot. Source [37].

Published Data

A robot can publish data over a ROS topic which will be sent to the web interface where it can be displayed to the user. It can also be used as a shared data pool between the robots. Right now the published data element is mainly used to send the position of the robots to the web interface so that the user can track the robots.

Initialization Module

The Initialization Module is just a module that will be executed at robot startup. This is provided by the user and is just a python code block. It is where the user can define imports and global variables accessible by all simple actions. Here the user can also perform other setup functions if necessary.

4.2.4 Task Model

The Task Model represents the tasks that the robots can perform. The idea of the task model is to be able to define a multi-robot task without providing any information about what type of robots are going to performing the task. This means that the task model is robot independent which makes it highly reusable. It is also much easier to develop then the robot model as it does not require any skills in robot engineering. The core idea is to use simple actions as building blocks to define bigger tasks that involve multiple robots. The implementation detail of an action is hidden. The user can then define which actions a task is composed of and if there are any dependencies between actions across robots. The task model contains the following elements.

Composite Task

A composite task is a set of tasks performed by a team of robots working together. Each task that the composite task is made up of is performed by exactly one robot. A composite task has a position provided by the user through the web interface. A composite task can, for example, be to play ball at a football field or to paint a house at a specific location.

Task

A task is defined as a sequence of simple actions that will be performed by one robot. A task can, for example, be to perform a penalty kick. By using simple actions as building blocks to define a task we can reuse the actions to define many different tasks.

Simple Action Reference

A simple action reference is a reference to a simple action used when defining a task. A simple action reference can be passed arguments. The user can also define dependencies between actions across robots. Let's say multiple robots needs to lift a heavy object together you can specify that the lifting action must be synchronized by using the "sync" keyword. Or if there is an action that must occur after another action you can specify that by using the "after" keyword.

Listing 4.2 illustrates an example composite task. The composite task is called "do a penalty shoot" and is made up of two tasks called "defend goal" and "shoot ball". The tasks are again made up of a sequence of simple actions. If the user wants a team of robots to perform this composite task he just chooses a location and the task allocation module takes care of distributing each task to the most appropriate robot that has an implementation of each simple action.

```
compositeTask do_a_penalty_shoot(lat,lng):

  task defend_goal():
    goTo(lat,lng):
    locate(args=("Goal")):
    moveTo(args=("Goal")):
    faceObject(args=("ball"), id="defender_ready"):

  task shot_ball():
    goTo(lat,lng):
    Locate(args=("Ball")):
    moveTo(args=("Ball")):
    aimAt(args=("Ball", "Goal")):
    kick(args=("ball"), after("defender_ready")):
```

Listing 4.2: An Example Composite Task.

4.2.5 The Generator

The generator is the part of the framework that takes all the information from the model and merges it with partly finished code files to create all the finished code files for the system. This includes all the files for each robot (i.e., the task allocation module and the ROS nodes) and the index file for the web interface [see figure 4.1].

The generator is implemented using Xtend. Xtend is a programming language that translates into Java source code. It is the language that is often used when writing code generators in the Eclipse Modelling Framework and it works well together with Xtext to generate application code from a model. Xtend allows us

to easily iterate over all the elements in the model and extract the user implementation of the simple actions and ROS nodes, and integrate the code with the task allocation module and web interface. The development, code generation, and deployment process are as follows:

Development Process:

1. The user adds the Task Definition Language plugin to Eclipse and creates a new Eclipse project.

2. He then creates a .tdl(Task Definition Language) file for each robot where he provides the implementation of each simple action that the robot is able to perform. He can also provide implementations of ROS nodes and define parameters.

3. He then creates a .tdl file containing the definition of all the tasks that the robots may need to perform.

4. The generator then goes through each robot file. For each robot, the generator extracts the simple actions and merges them with the task allocation module. A python file is created for each ROS node, and a ROS .yaml file containing the parameters. A ROS launch file is also generated which starts the application. All of the generated files for each robot must then be deployed to the appropriate robot.

5. The generator then goes through the file containing the task definitions and merges them with the .html file. This way all the available tasks and their definitions are available through the web interface.

6. When the generated .html file is opened it will connect to the robots through a server and the user can see the connected robots and start to define missions.

New robots can be added to the system at any time by creating additional .tld files without having to restart the other robots. This can be important for long missions where you might want to add new robots without stopping the whole mission.

4.3 The Task Allocation Module

4.3.1 System Architecture

The second component of our framework is the Task Allocation Module. The Task Allocation Module is responsible for distributing the tasks between the robots. Our solution uses an auction-based architecture where each robot bids on tasks based on a cost function. One of the robots is chosen to be the auctioneer and is using a task allocation algorithm to decide which robot should perform which task based on the bids. Figure 4.5 illustrates the architecture of the system. The list of steps performed when the user sends a mission through the web interface are as follows:

List of steps:

1. Auctioneer is chosen: One of the robots is chosen to have the role as the auctioneer.
2. Tasks are received from user: A mission / set of tasks are sent to the auctioneer from the user using a web interface.
3. Auction begins: The tasks are distributed to all other robots.
4. Bidding begins: Each robot goes through all the tasks to check whether they are able to perform the task and then calculates a bid on it using a cost function.

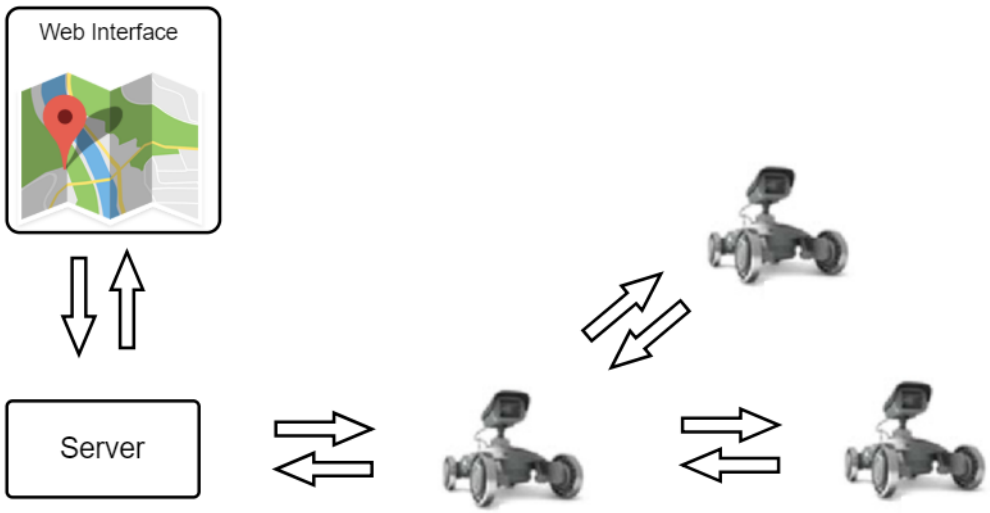


Figure 4.5: System Architecture.

5. Auction finished: After some time has passed the auctioneer distributes the tasks to available robots(i.e., robots not currently performing any task).

6. Task Execution Starts: Each robot starts executing its allocated task.

The reason for using an auction-based architecture is due to their popularity [26]. Auction-based solutions have been getting a lot of attention because of their advantages. In [23] they list some of the advantages of an auction-based solution. Simplicity is one of them as the idea behind most auction-based protocols is simple and intuitive. Another is fault tolerance as the auctioneer can keep track of sold tasks and their execution status he has the ability to reallocate a task if the robot performing it fails. As we are using an auction-based solution we have a centralized algorithm as it is the auctioneer who ultimately decides which robot will perform which tasks based on all the received bids.

4.3.2 The Task Allocation Algorithm

The problem of multi-robot task allocation was discussed back in chapter 3.3. As we discussed the problem can be classified in multiple ways depending on the system, how the tasks are defined, how the robots are performing the tasks, what assumptions are made, and what task constraints are used. What algorithm we need to able to optimize a specific type of system depends on how the system is defined and what assumptions are made. We can say that the problem space is large as the solution depends on many variables. This makes it difficult to design an algorithm that fits for all types of multi-robot systems. It also makes it more difficult to reuse algorithms designed in other solutions.

This is why we have decided to design our own algorithm which fits our system. We identify our specific instance of the problem by defining the system and assumptions about the system. We use the 3 main axes from [2] and some additional axis from [38], like demand and available resources.

System Definitions

Task

There is a set T = t1, t2, ..., ti of tasks. A task is performed by a single robot.

Composite-Task

There is a set C = c1, c2, ..., ci of composite-tasks. A composite-task is a task performed by a team of robots. Each composite-task contains a set of tasks and has a geographical position.

Robot

There is a set R = r1, r2, ..., ri of robots. Each robot has a set A = a1, a2, ..., ai of actions representing their capabilities to perform certain tasks.

Coalition

There is a set F = f1, f2, ..., fi of coalitions. A coalition is a team of robots working together on one composite task.

System Assumptions

Single-task Robots

A robot can only perform one task at the time and can only be part of one coalition at the time.

Multi-robot Tasks

A composite task may require multiple robots to achieve it.

Instantaneous Assignment

There may be more tasks than robots, but the tasks will only be distributed as long as there are robots without tasks. No future planning will be made.

Unit Demand

A task is required to be executed exactly one time and not repeated.

Limited Resources

There may be more tasks than robots. This means that we can't complete all tasks in the first iteration of task distribution since we don't plan for future tasks. So we have to choose the best subset of tasks which minimizes the cost.

Variable Profit

The profit is the revenue minus the cost of performing a composite task by a coalition. We assume each composite task gives equal revenue(i.e., each composite task has equal priority) but coalitions have different costs of performing different composite tasks.

Central Decision Making
We assume that there is a central unit(the auctioneer) which is able to collect information from all the robots in the system and distribute tasks based on this information.

Based on these definitions and assumptions we can design an algorithm that fits our system. The goal is to create a coalition for each composite task which minimizes the cost of performing all the composite tasks.

We design a centralized greedy coalition formation algorithm that chooses the highest bidder in each iteration. In our case, each composite task is made up of a set of sub-tasks which will be performed by a single robot. In each iteration of the algorithm, the coalition-task pair with minimum cost will be chosen. The cost of coalition-task pair is calculated by taking the sum of the highest bid on each sub-task. After the best coalition-task pair is found the task is marked as sold and the chosen robots are marked as unavailable. Then the second-best coalition-task pair is calculated and so on until no more tasks can be allocated.

The algorithm uses instantaneous assignment as opposed to time-extended assignment which means that tasks are assigned to robots only when the robots are available. If a robot is in the middle of a task while new tasks arrive from the user the robot will not be allocated any of the new tasks before it has finished the current task. An algorithm supporting time-extended assignment would plan the execution of future tasks by creating a schedule for each robot if there are more tasks than there are robots available. The pseudo-code for the algorithm is provided below. The list of steps taken by the algorithm is as follows.

Algorithm Steps:

1. While there are more unallocated doable composite-tasks do the following.

2. For each composite-task do the following.

3. For each sub-task in the composite-task pick the robot with the highest bid on the sub-task that is available. If the sum of bids of performing all the sub-tasks in the composite task is higher than the current highest composite-task bid mark this composite-task as new best.

4. Allocated the group of robots that bidded highest on the best composite-task to the appropriate sub-tasks that each of them bidded on and mark the composite-task as sold. If there are no more doable composite-tasks mark "more doable tasks" as false.

Algorithm 1 Task Allocation Algorithm

```
1:  procedure DISTRIBUTE_TASKS(Mission_table)
2:      more doable tasks mdt ← True
3:      while mdt == True do
4:          best composite task bct ← None
5:          best composite task value bctv ← None
6:          for all ct ∈ composite_tasks(Mission_table) do
7:              if ct.is_Sold == False then
8:                  composite task is doable ctid ← True
9:                  composite task value ctv ← 0
10:                 for all st ∈ sub_tasks(ct) do
11:                     highest bidder hbr ← None
12:                     highest bid hb ← None
13:                     for all bid ∈ bids(st) do
14:                         if bid.value > hb then
15:                             if bid.robot.is_available == True then
16:                                 hbr ← bid.robot
17:                                 hb ← bid.value
18:                             end if
19:                         end if
20:                     end for
21:                     if hb ≠ None then
22:                         st.robot ← hbr
23:                         st.robot.is_available ← False
```

```
                        ctv ← ctv + hb
                    else
                        ctid ← False
                    end if
                end for
                if (ctid == True) and (ctv > bctv or bct == None) then
                    bct ← ct
                    bctv ← ctv
                end if
                for all st ∈ sub_tasks(ct) do
                    st.robot ← None
                    st.robot.is_available ← True
                end for
            end if
        end for
        if bct ≠ None then
            bct.is_sold ← True
            for all st ∈ sub_tasks(bct) do
                highest bidder hbr ← None
                highest bid hb ← None
                for all bid ∈ bids(st) do
                    if bid.value > hb then
                        if bid.robot.is_available == True then
                            hbr ← bid.robot
                            hb ← bid.value
                        end if
                    end if
                end for
                st.robot ← hbr
                st.robot.is_available ← False
            end for
        else
            mdt ← False
        end if
    end while
end procedure
```

4.3.3 The Cost Function

As the task allocation algorithm is designed to optimize the utility of the system we need to define utility. As we discussed back in chapter 3.3 utility can be defined in multiple ways. The utility can be reward minus cost where each task has a reward associated with completing the task and each robot has a cost of performing the task. If the system is to support task prioritization then reward can be used where the tasks with higher prioritization give a higher reward. Another way of defining utility can be fitness minus cost where each robot has a fitness for how well the robot can perform the task.

In our solution, we have chosen to only use the negative value of the cost of performing the task. The cost is defined as the distance between the gps coordinates(in latlng) of the robot and the task. A large distance between the robot and the task will evaluate to a large negative value which is a low bid. If a robot is not able to perform the task at all because it lacks the required actions it will not send in a bid at all to the auctioneer.

$$Bid = \begin{cases} -cost & \text{If robot is able to perform the task} \\ 0 & \text{Else} \end{cases}$$

In our solution, we have chosen to use distance to calculate the cost, but there are many other ways it can be defined. It can be the distance to the task, time used to complete the task, resource usage, etc. How to define cost often depends on the task and robots used which makes it difficult to design one solution which works well for all types of multi-robot systems. However, we have chosen to use distance in our solution. Assuming all the robots have approximately equal speed it also minimizes the time of the mission which is often the goal in most situations. It does, however, require that each robot has a gps and knows its own location.

There are also multiple ways we can minimize the cost. We can minimize the sum of the cost over all the robots. This would minimize the total distance

traveled by all robots if the distance is used as cost. We can minimize the cost of the worst robot which leads to the solution with minimum timespan. We can minimize the average cost per task which leads to each task being completed approximately equally fast. In [23] they refer to the different ways of optimizing as the optimization objective and they explain the pros and cons with each in more detail. In our solution we have chosen to minimize the total sum of cost over all robots(minSum).

4.4 ROS Setup

4.4.1 Why use ROS?

The Framework is built on top of The Robot Operating System (ROS). It utilizes the publish-subscribe pattern implemented by ROS which can be used to send messages between processes(nodes) running on different machines over a network. It also provides the user with many libraries and advanced robotics algorithms that can be used to implement advanced robot behavior. In this chapter, we explain how the nodes in the system are implemented and launched and how communication between them is achieved. We also explain how ROS libraries and algorithms can be used to implement different actions.

4.4.2 A Publish-Subscribe Pattern

ROS implements a publish-subscribe service. This fits our auction-based architecture where we have one auctioneer and multiple bidders sending messages in a one-to-many fashion. In ROS you often have many nodes running on each robot which sends messages between each other over different topics. This makes the different components of the system loosely coupled and results in a component-based architecture.

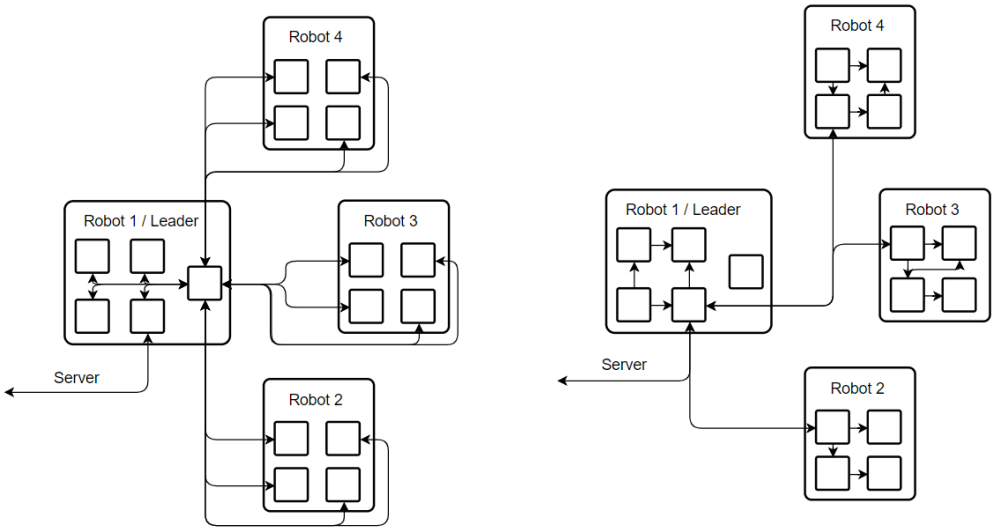
When using a publish-subscribe pattern we need a broker to organize the topics and messages between publishers and subscribers. In ROS the broker is referred to as the ROS master. The ROS master runs as a ROS node and is the first node which needs to be launched as all other nodes won't start running before they have registered with the master.

As described on the official ROS page the ROS Master provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer [39]. This means that all the messages between the different publishers and subscribers do not go through the master, but directly between the nodes in a peer-to-peer fashion. The nodes only connect to the master to get the addresses of all the publishers for the topics they are subscribing on.

In our system the first robot that connects to the server will be told to startup a new ROS master node. The server will store the ip address of the robot running the master. When a new robot connects to the system it will get the ip address of the robot running the master node from the server. The robot can then set the "ROS_MASTER_URI" variable which is all that is required by ROS before the nodes on the robot can start communicating with the master node and find all the other nodes in the network.

Figure 4.6 illustrates the nodes on each robot in the system at and after startup. Once a robot has the ip address of the robot running the master node all the nodes on the robot will launch and connect to the master node where they will get the address of all the nodes publishing data on topics they are interested in. After that, the nodes will only send messages to each other peer-to-peer and not through the master node.

In ROS, standard names are often used on topics. A node that is reading laser data from a laser scanner publishes this data on a topic called /scan. Other nodes that use laser data to let's say build a map of the robot's environment expect laser



(a) Nodes at Startup

(b) Nodes after Startup

Figure 4.6: ROS Nodes at and after Startup.

data to be published on /scan. These nodes are often developed by others and are available through the ROS package repository. So to avoid name crashing of topics the framework takes care of remapping the topics using namespaces for each robot. This is done by the generator when generating the launcher file. This way we avoid nodes picking up sensor data from other robots and nodes will only communicate with other nodes on the same robot.

The only node not running in a namespace is the node containing the task allocation module. This node contains the simple action implementations and the bidder module which contains all the logic to bid on tasks and start executing actions. The leader robot will additionally run a sub-module called the auctioneer module. Messages will be sent between the auctioneer and all the bidder nodes running on other robots. The messages sent between the auctioneer and bidders are in the form of json objects containing mission information, task bids, which robot won which task, task execution status and so on.

4.4.3 Launching The ROS Nodes

A launch file is what is commonly used in ROS to startup all the nodes in a ROS application. Our generator creates a launch file for each robot that needs to be deployed on the robot together with all the nodes. Figure 4.7 illustrates a shorted down launch file. Here three nodes are launched. One called "odometry_source" which is a node typically written by the user that publishes the position and orientation of the robot based on sensors like wheel encoders. This node is different for each robot based on the robots wheel configuration which is why it has to be provided by the user in the robot model.

The other node in the figure is called "move_base" and is available through the ROS package repository. This node contains multiple advanced algorithms used for navigation. The node works by subscribing to odometry and laser data from other nodes, running the data through path planning algorithms and publishing movement commands. This way the node can be used to navigate the robot to a goal location.

The last node in the figure is the task allocation module which contains logic for task distribution, bidding, and running actions. This node is created by the generator. It is not launched in a namespace so the node can communicate with all other nodes running the task allocation module on other robots. Nodes with the package name "multi_robot_simulation" are the ones provided by the user or created by the generator while other nodes are provided through existing ROS packages.

Only a part of a launch file is shown here. Typically there are many nodes running on one robot. Nodes that read laser data, odometry data, path planning nodes, nodes controlling the motors and different actuators and so on. Depending on the complexity of the robot and the number of sensors and actuators a robot may run tens of nodes.

The Launch file does not only contain information about namespaces, name, and location of nodes, but also the location of parameters. The parameters for the robot provided by the user is stored in yaml files by the generator. Using yaml files

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
    <group ns="youbot">

        <node pkg="multi-robot-simulation" type="odometry_source.py" name="odometry_source">
                <remap from="/tf" to="/youbot/tf" />
        </node>
        <node pkg="move_base" type="move_base" name="move_base_node">
                <remap from="/tf" to="/youbot/tf" />
                <rosparam file="$(find multi-robot-simulation)/src/youbot/move_base_node.yaml" />
        </node>

    </group>

    <node pkg="multi-robot-simulation" type="task_allocation_module.py" name="task_allocation_module">
    </node>

</launch>
```

Figure 4.7: Example Launch File Created by the Generator.

is the standard way of storing parameters in ROS. When the launch file is executed ROS fetches the parameters and stores them on what is called the parameter server which is managed by the ROS master. Here all the parameters can be dynamically accessed by all the nodes in the system and used to optimize the behavior of the robot.

4.4.4 Using ROS Stacks and Algorithms

ROS is not only used for the communication between the robots and to achieve a component-based architecture, but also to provide the user with tools and libraries which can be used to implement the robot's actions. ROS provides tools for visualization, simulation, debugging, and monitoring of topics and messages sent between nodes.

ROS also provides multiple stacks(collection of packages) that the user can use to implement the different actions. Some example stacks are the navigation stack and Movelt. The navigation stack consists of many nodes working together to achieve navigation. The stack can be called like a service from a simple action to move the robot to a goal position. This way the stack can be used to implement

a “goTo” action that moves the robot to a location on the map or to an object. Similarly, MoveIt is a stack used for arm manipulation and can move a robot arm to a goal location and can also be invoked like a service. This stack can be used to implement actions where the robot interacts with objects.

To sum up, ROS is used to achieve a publish-subscribe based communication system between the robots that fit our auction-based architecture. It also provides the user with tools, libraries, and algorithms to help implement different actions. The implementation of advanced generic actions is the key to achieve reusable components that can be used as building blocks to define big complex tasks. Actions like “move forward” and “raise arm” are too simple and rarely used to define commonly required tasks. While actions like “move to ball” and “defend goal” are too specific and cannot be reused easily. The framework works best when we have generic and relatively advanced actions available like “move to (object)” which can be reused to define many different tasks. The framework is built on top of ROS as ROS libraries and stacks make it easier to implement these actions. The framework could have been built on top of other popular middlewares such as Orocos [30] or Miro [31], but ROS was chosen as it is currently one of the most popular ones.

4.5 The Web Interface

4.5.1 Why Use a Web Interface?

The web interface is the last component of the framework. The web interface allows the user to define a mission from the set of available tasks. The web interface can also be used to monitor the mission and the robots. From the conducted literature review it seems like there are few solutions using a web interface. [S1] and [S2] were the only other solutions using a web interface, and the solutions were mainly targeting aerial vehicles. Most other solutions let the user define a

graphical model of the system often in the form of a state machine representing the different states the robots can be in and the conditions for when the robots should transitions from one state to another. The state can be “chase ball” or “defend goal” and are usually provided by the user similar to what we have defined as simple actions. However in most other solutions if you want the robots to perform another activity then they are currently doing you have to redesign the model, generate new application code and deploy the new code on the robots. The robots also have to be placed at the right location like a football field before starting the system.

In our proposed solution we try to avoid the need for redeployment by using the web interface. Definitions of the tasks are stored in the web interface. The robots do not possess any application logic only the implementation of reusable actions. The definition of a task is sent to the robot from the web interface at runtime when the user wants the task performed. The task allocation module checks whether the robot can perform the task based on the task definition and what actions the robot has available, and it takes care of executing the actions.

This way the robots can perform one activity like playing ball one day and another activity the next day without any redeployment. New tasks can be added and old tasks redefined without any redeployment or without even restarting the system. The web interface allows you to control what tasks or activities the robots should perform at a given time amongst all the tasks they are able to perform. By using a web interface with a map the solution also works better for systems where you have multiple teams of robots over different locations.

The problem however with using maps for ground robots is that it is difficult to navigate robots outdoor over large distances. Our solution requires the robots to have a gps and a “go to” action which lets the robot navigate to a global position on the map. This is required or else it is not possible to define different tasks at different locations, and a cost function based on distance is not possible.

4.5.2 Leaflet

The web interface provides the user with a satellite map over the robot's location. The map is loaded using Leaflet, an open-source javascript library for interactive maps [40]. Leaflet lets us load maps from different providers. In our case, we use a satellite map from Esri. Leaflet maps are also highly customizable and allow us to place markers at different locations. Figure 4.8 shows a screenshot of the web interface.

At the right side, you have all the defined composite tasks from the task model. After you have placed some tasks at different locations you can start the mission. The tasks are sent to the server and then to the robots and the bidding process starts. The task allocation module takes care of distributing the tasks only to robots that have all the required actions to perform the task. The user only decides what task should be performed and where, not what sub-tasks and actions they are made up of as this is defined in the task model.

An example composite task could be to play ball at a football field. Another could be to paint a house at some location. Two teams of robots would be created and they would start navigating to the target location. Each robot in the teams would then start performing their sub-tasks. The sub-task might be catch ball, defend goal, measure house, paint wall, etc. All based on how the composite tasks are defined.

The web interface is partly based on a previous project where a multi-robot mission planner was designed [41]. The mission planner could be used on vehicles running an autopilot software called Ardupilot. And the vehicle could mainly be used to fly from one point to another and to search an area. In that solution, the vehicles are restricted to those who can run the autopilot software. And only three types of tasks were supported. While in our solution there is more focus on the ability to add any type of robot and define any type of task.

Figure 4.8: The Web Interface.

Chapter 5

Evaluation

5.1 Evaluation Method

To evaluate the developed framework we use simulation software. Performing simulation is one of the most common methods to use for evaluation in robotics as it can be quite expensive and time consuming to use real robots. Using simulation software greatly reduces the time and cost of testing the solution and its viability. While the use of simulation has many benefits it has some drawbacks also. Simulation software may not capture all external factors which can affect the behavior of the robots in the real world. Therefore there is no guarantee that the robots will act exactly the same in the real world as in the simulator. In our case, the communication between the robots is difficult to validate using simulation since the simulator runs on a single computer. To deal with this the framework will additionally be tested between multiple distributed machines.

Simulation software is used to perform functional testing and to create a scenario that demonstrates the usage of the framework. The simulation software that is used is called Gazebo. Gazebo is very popular to use together with ROS. In [42] they performed a survey on different simulation software for robotics. The survey showed that Gazebo is the most popular simulator. It can be integrated with multiple different physics engines but default uses the Open Dynamic Engine (ODE).

It also supports multi-robot simulations and the use of ROS messages and services to control the simulated robots.

5.2 Simulation Setup

To quickly set up a simulation environment we use Gazebos model repository. This gives us access to ready to use models of various robots and objects, saving us a lot of time defining the physical and visual properties of the robots. A screenshot of the simulated world can be seen in 5.1. It contains 4 robots(one turtlebot, one pioneer 3at, and two youbots) and a football field with some objects. The robots have simulated sensors like laser scanners which makes them able to detect and navigate around objects in the simulated world like they would in the real world.

Gazebo allows us to control the robots directly by using the ROS messages received from the different ROS nodes running in the system. This allows us to implement the control of the robots much faster than on a real system. The developed framework is also heavily dependant on gps data as this is used to distribute tasks and to show the position of the robots in the web interface. To solve this the coordinates of the simulator are mapped to real-world gps coordinates by mapping the center of the football field in the simulator to the center of a football field in the real world showed in figure 5.1.

5.3 Scenario

To demonstrate the usage and utility of the framework a scenario is created using the simulated environment. Let's say we want two of the robots to entertain the humans on the side of the field and two robots to play ball and do a penalty shot. The process is as follows. We start by creating a new Eclipse project where we add the tdl plugin found in the Github repository of the project [36]. We then create

Figure 5.1: Gazebo Simulator Setup with 4 Robots.

a file with the extension tdl for each of the robots. In these files, each action that the robots are able to perform is implemented together with any necessary ROS nodes and parameters. These files can get quite big depending on what actions are implemented so they are not shown here. A full version of each robot file can be found at the Github repository under multi-robot-simulation/robots [36]. We implement the following actions for each robot:

- goTo(lat, lng)
- moveForward(time)
- moveBackwards(time)
- turnLeft(degrees)(time)
- turnRight(degrees)
- moveToGoal()
- moveToBall()

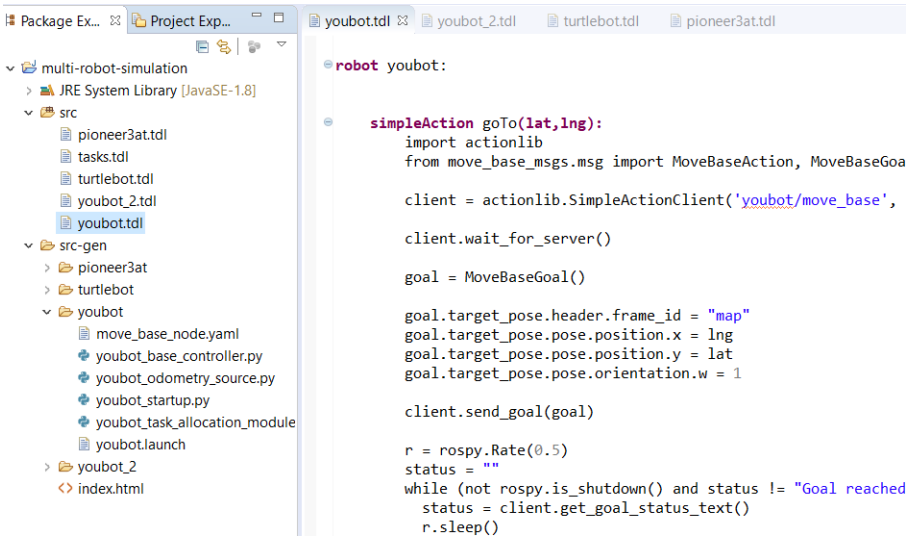


Figure 5.2: Generated Files.

- kickBall()

Some shortcuts are used when implementing the actions. Like the action "moveToBall" teleports the robot to the ball as an actual implementation would require object detection and fine-tuned navigation. Shortcuts are used to quickly get a set of sample actions we can use for demonstration and testing. After the actions are implemented the generator will create a folder for each robot containing all the generated files [see figure 5.2]. On a real system, these files would need to be deployed on the robots, but when we use a simulator we can just move them into a ROS project. Together with the generated files is a launch file used to startup each robot. This will start up the ROS nodes and the task allocation module which again will connect to the server and then the ROS master and start to listen for tasks to bid on.

After the robots have been added we can define the tasks we want the robots to perform. We create a file called tasks.tdl [see figure 5.3]. Here we define the tasks by using simple actions. After the file is saved an index file is generated. This file opens the web interface and connects to the server. Figure 5.4 shows a

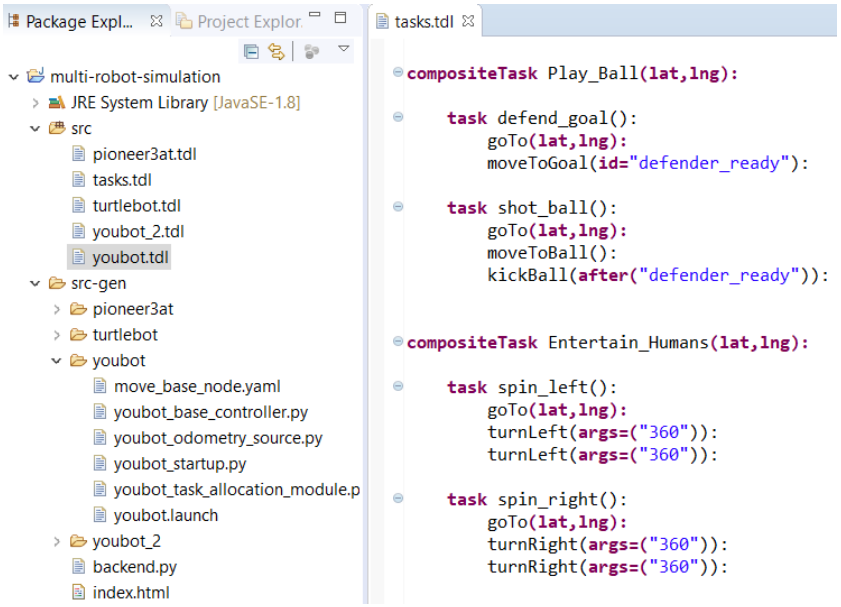


Figure 5.3: The Task Definitions.

screenshot of the simulator with the web interface on the left side. Through the web interface, we can define a mission by placing tasks on the different locations on the map. The robots will divide the tasks amongst themselves and form groups and then start executing their actions.

The simulator is mainly used to perform functional testing of the framework and its components. This lets us validate that the framework meets all of its functional requirements and that it works as intended. For example that the correct actions are performed as defined in the task model, that robots are only allocated tasks that they have the ability to perform, that a composite task is distributed to the closest group of robots, and so on.

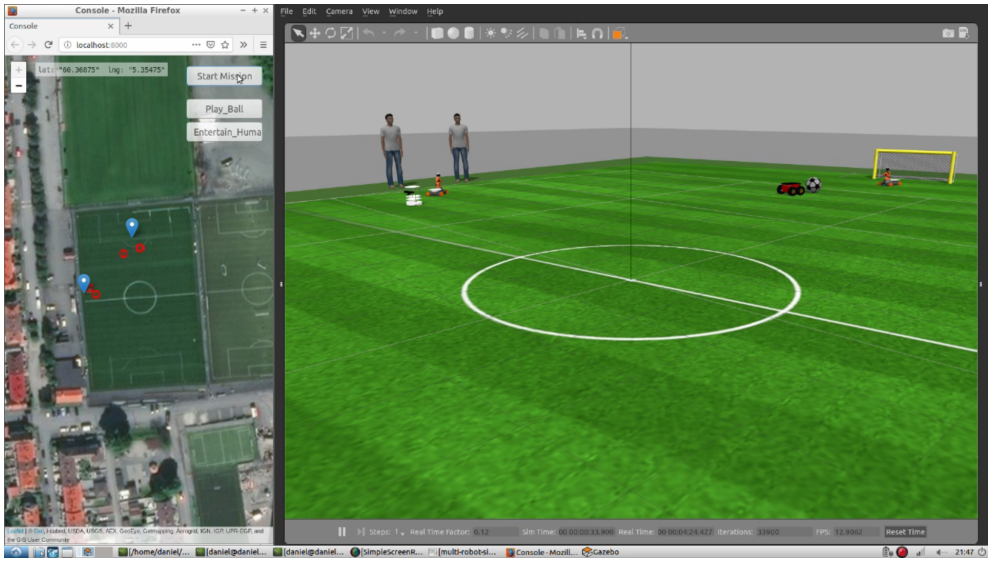


Figure 5.4: Two Teams of Robots Performing Different Tasks.

5.4 Testing on Distributed Machines

In addition to the functional testing with the simulator, the framework is also tested on real distributed machines. This allows us to test that the communication works as intended and that it works on a real system and not only in the simulator. One robot and two computers are used for the testing. The robot used is a raspberry pi based rover running ROS while the computers are using virtual machines to run ROS. We add the robot and computers by creating a tdl file with some actions. The rover has some moving action while the computers have actions that prints text to the console. The rover and computers are also publishing fake gps data as they don't have a GPS. The setup is shown in figure 5.5. In our case the laptop to the left is running both the server and web interface and is also the team leader, but either of them could have been the leader. We can then perform testing and validate that the framework works on a real distributed system.

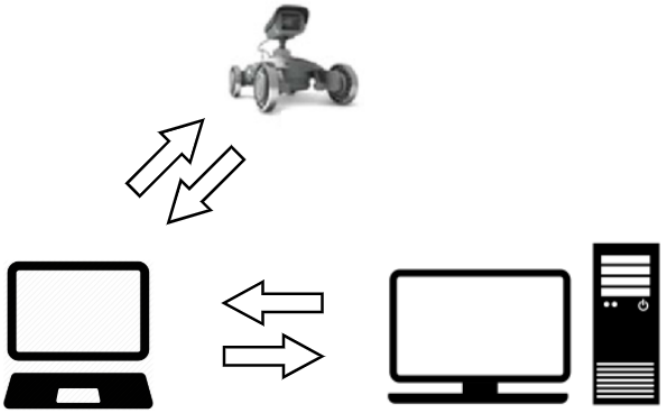


Figure 5.5: Real System Setup.

5.5 Additional Evaluation

So far we have mainly performed functional testing of the framework using simulation software. This includes testing that the developed language and its generator produces the correct code and that the task allocation module and the communication between the robots works as intended, and that the framework meets all of its functional requirements. What has not been evaluated is the usability of the language. One of the main purposes of a domain-specific language is that it should simplify the development of a particular application in a particular domain. Since the framework takes care of the communication setup and the task allocation between the robots and also provides the user with a ready to use web interface, the design and implementation of a specific multi-robot application should be a bit faster and easier when using the framework rather than doing it from scratch. This is however not validated.

The usability of the language can be evaluated by having test subjects first implement a particular multi-robot application without using the language and then by using the language, and then comparing the time or lines of codes used. The problem, however, is that while the language is designed to simplify the development process it is still quite difficult and time-consuming to implement the actions

for the robots and develop a proper multi-robot application. This requires that the test subjects need to know robotics and preferably ROS. This makes it difficult to find participants and also it would be very time consuming for the participant to test it properly as it is quite complicated to develop any type of multi-robot application. The evaluation of the usability of the language is left for future work.

Another possibility for future evaluation could be to test the system with a higher number of robots. For now, the simulator is using very few robots as each robot takes a lot of computer resources. Future evaluation could include testing the system using cloud services to be able to run many more robots at once and to test how the system performs with many teams of robots.

Chapter 6

Discussion

In this thesis, we have looked at how model-driven software engineering can be applied in the development of heterogeneous multi-robot systems. In such a system you may have different robots with different capabilities able to perform different tasks. By working together the robots are able to perform more complex tasks than a single robot can perform alone. In such a system there are additional challenges that need to be solved. Like how tasks should be distributed amongst the robots and the fact that different robots often are built from different hardware which means that they need different code to perform the same action.

To explore how model-driven software engineering can be applied in the development of heterogeneous multi-robot systems the following research questions were formulated.

Main research question of the thesis:
How can model-driven software engineering be used to simplify the development of heterogeneous multi-robot systems?

Sub-questions:
How is model-driven software engineering applied to the development of heterogeneous multi-robot systems in today's research?

How can efficient and appropriate task allocation be achieved in different heterogeneous multi-robot systems?

To answer the first sub-question a systematic literature review was conducted. The conclusion of the review was that most of the studies focused on the modelling of the behavior of the robots through the use of textual DSLs with graphical tools enabling the user to create finite state machines and statecharts describing the robot's behavior on a high-level [S7], [S8], [S9], [S10]. Low-level control code for each robot is often assumed to be provided by the user beforehand. There were also some solutions defining textual DSLs to specify the behavior of the robots [S6]. Only two studies were concerned with modelling the communication of the robots rather than the behavior [S11], [S12]. The communication is rarely addressed and are often taken care of by a middleware. [S1] and [S2] were the only studies proposing the use of a user interface with a map to define a mission for a team of robots. The user interface was designed for aerial vehicles, however.

In this thesis, we have proposed a framework for modelling the behavior of the robots by using simple actions as building blocks to define tasks that involves multiple robots. As opposed to other solutions our solution allows the user to define different missions without the need for redesigning the model, regenerate code and redeploy the new code on all the robots. The user can also add new tasks and change the definition of existing tasks without the need for redeployment. The proposed framework consisting of 4 components. A domain-specific language used to model both the robots and the tasks. A task allocation module used to distribute the tasks amongst the robots. The robot operating system used for communication between the robots and to achieve advanced navigation and robotic behavior. And a web interface used to create missions for teams of robots.

The developed domain-specific language called Task Definition Language allows the user to create a model of the robots and a model of the tasks that the robots are going to perform. The task model uses simple actions as reusable building blocks allowing the user to define many different tasks using the same

set of actions. While the task model is platform-independent and highly reusable, the robot model is platform-specific and can contain a lot of low-level control code which implements the different actions that a specific type of robot can perform. This can be difficult and time-consuming. Unfortunately, there is no way around this problem if we want the framework to support any type of custom-built robot. As different robots often need different code to perform the same action, the implementation of the actions must be provided by the user for each robot.

One of the main ideas of the thesis is that there exists a set of core actions that are often used in most physical tasks. Even if there exists a huge number of more random like actions like "moveForward", "turnAround", "raiseArm" and so on, these are too simple and rarely used to define useful tasks. On the other hand actions like "locate("object")", "moveTo("object")", "pickUp("object")" are actions that are used in a huge number of different tasks. So the theory is that if the user is able to implement a few of these core actions he has the possibility to reuse them to define a large number of different tasks.

This is however difficult to validate as these types of actions are difficult to implement. The ability to locate specific objects requires the robot to have a depth camera with image recognition. While the ability to move to an object or pick up an object can be implemented using ROS libraries, like the navigation stack or Movelt.

To answer the second sub-question we explored how task allocation can be achieved in such a system. As we saw in [2], [20] and [23] there are multiple variations of this problem. As we are mainly concerned with systems where you have tasks that require multiple robots to be achieved we can classify our problem as an ST-MR-IA problem, or a coalition formation problem. Unfortunately, this problem again has many variations based on how we define our system, what assumptions are made, if task constraints are used, and so on. The problem space is large as the solution depends on many variables. This makes it difficult to design an algorithm that fits for all types of multi-robot systems.

In this thesis, we have proposed an algorithm that fits our system. We have

designed an auction-based centralized greedy coalition formation algorithm that chooses the highest bidder in each iteration. In our solution, the bid on a specific task is based on the distance between the robot and the task.

We have also explored how the Robot Operating System can be utilized in the development of a heterogeneous multi-robot system. The developed framework uses the publish-subscribe pattern implemented by ROS to achieve the one-to-many communication between the auctioneer robot and the bidder robots. We have also shown how ROS can be used to implement a "goTo" action for the robots. ROS can also potentially be used to implement additional actions.

We have also shown how a web interface with a map can be used to define a mission at runtime. The use of a map to specify and monitor a mission is commonly used in drone systems. In [S1] and [S2] they also used a web interface to define missions for aerial vehicles. In this thesis, however, we propose the use of a web interface to define a mission for any type of robot. The use of a web interface allows us to define multiple different missions without having to redesign the model and redeploy the new generated code to the robots. This way the robots can perform one task one day and another task the next day without any redeployment. This is possible because the robots do not possess the definitions of the tasks, only the implementation of the actions they are able to perform. The definition of a task is sent to the robot when the user starts a new mission. New tasks can be added to the task model or old tasks can be modified. Then when the newly generated index file is opened it will connect to the server and the new tasks with the new definitions will be available.

This means that our system is good for long missions where you might need to add new robots or new tasks dynamically. By using a map our system is also good for when you have multiple teams of robots over different geological locations. The drawback, however, is that the robots need to be able to navigate outdoor over large distances which is difficult to achieve with today's robots.

We evaluated our proposed framework by using simulation software. We created a simple scenario where two teams of robots performed different tasks to

demonstrate the utility of the framework. However, the implemented actions were very simple. To really validate whether a sequence of simple actions can be used to perform complex tasks a much more complicated scenario with proper actions will need to be tested.

Chapter 7

Conclusion

In this thesis, we have explored how model-driven software engineering can be applied in the development of heterogeneous multi-robot systems. We have proposed a framework where simple actions are used as building blocks to define larger tasks that are performed by multiple robots. The idea is to use simple actions as reusable components to be able to define many different tasks.

The biggest problem in a heterogeneous multi-robot system is the robot heterogeneity. As different robots need different code to perform the same action the user must provide an implementation of each action for each robot. In the future, there might be general-purpose robots with high-level libraries which allows you to perform different actions. This might solve the problem of having to provide an implementation of each action for each robot. The ability to support any type of custom-built robots may not be necessary and is difficult to achieve.

In this thesis, we have also looked at how to perform task allocation in such a system and we have proposed an auction-based centralized greedy coalition formation algorithm that chooses the highest bidder in each iteration. Where the bid on a specific task is based on the distance between the robot and the task. We have also looked at how the robot operating system can be utilized in such a system for the communication between the robots and the implementation of different actions. We have also shown how a web interface can be used to define a mission

for a team of robots at runtime. By sending the task definitions to the robots at runtime we remove the need for redeployment each time we want the robots to perform a different activity.

For future work, more research should be done on how to easier be able to add new robots and their action implementations as this is the main problem of the framework. The language can also be extended to support nested composite tasks. Also, the possibility to generate tasks from a set of available actions can be explored.

The task allocation algorithm can also be improved to support additional features such as task constraints. The ability to define task requirements at runtime could also be useful. For example, the task of painting a house that is 3 meters tall requires a robot with a minimum of 3 meters reaching distance. The height of the house might not be known at design time so the user should have the ability to set requirements for a specific instance of the task through the web interface. The task should then only be allocated to a robot that meets the requirements.

The web interface can also be improved in many ways. The user should be able to have more control over the robots and receive more information about the mission status. The user should also have the ability to provide task inputs. For example, if the task is to paint a house the robots need to know which color they should use. Often in complex tasks, there are a lot of choices to make which cannot be specified beforehand. This information could be provided through the web interface.

To sum up, there is a lot of useful functionality which can be provided through a web interface in such a system that gives the user more control over the robots and their actions. In this thesis, we merely provide the bare bones of a proper web interface. There is also a lot of further research which can be done on how to apply model-driven software engineering in the development of heterogeneous multi-robot systems.

Bibliography

[1] "Robocup." https://www.sporttechie.com/robocup-robots-practicing-2050-human-vs-robot-soccer-tournament/. Accessed: 2019-05-22.

[2] B. P. Gerkey and M. J. Mataric, "A formal analysis and taxonomy of task allocation in multi-robot systems," *I. J. Robotics Res.*, vol. 23, pp. 939–954, 2004.

[3] A. Hevner, A. R, S. March, S. T, P. , J. Park, R. , and S. , "Design science in information systems research," *Management Information Systems Quarterly*, vol. 28, pp. 75–, 03 2004.

[4] A. Dahanayake and B. Thalheim, "Enriching conceptual modelling practices through design science," *Lecture Notes in Business Information Processing*, vol. 81, pp. 497–510, 01 2011.

[5] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[6] Z. Stapic, E. García López, A. García, C. Luis De, M. Ortega, and V. Strahonja, "Performing systematic literature review in software engineering," *In CECIIS 2012-23rd International Conference*, 2012.

[7] G. L. Casalaro and G. Cattivera, "Model-driven engineering for mobile robot systems: A systematic mapping study," Master's thesis, Malardalen University, Vasteras, Sweden, 2015.

[8] B. Hailpern and P. L. Tarr, "Model-driven development: The good, the bad, and the ugly," *IBM Systems Journal*, vol. 45, pp. 451–462, 2006.

[9] V. García Díaz, E. Núñez Valdez, J. Espada, B. Pelayo García-Bustelo, J. Cueva Lovelle, and C. Marín, "A brief introduction to model-driven engineering," vol. 18, pp. 127–142, 04 2014.

[10] J. D. HAAN, "Roles in model driven engineering," 2009.

[11] S. Roubi, M. Erramdani, and S. Mbarki, "A model driven approach for generating graphical user interface for mvc rich internet application," *Computer and Information Science*, vol. 9, 04 2016.

[12] R. Picek and V. Strahonja, "Model driven development-future or failure of software development?," *Proc. 18th Int'l Conf. Information and Intelligent Systems*, pp. 407–413, 01 2007.

[13] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," p. 471, 2011.

[14] C. Pons, R. Giandini, and G. Arévalo, "A systematic review of applying modern software engineering techniques to developing robotic systems," *Ingeniería e Investigación*, vol. 32, pp. 58–63, 04 2012.

[15] A. Ahmad and M. A. Babar, "Software architectures for robotics systems: A systematic mapping study," *Journal of Systems and Software*, vol. 122, pp. 16–39, 2016.

[16] A. Nordmann, N. Hochgeschwender, and S. Wrede, "A survey on domain-specific languages in robotics," in *Simulation, Modeling, and Programming for Autonomous Robots*, vol. 8810, 10 2014.

[17] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "Robotml, a domain-specific language to design, simulate and deploy robotic applications," in *SIMPAR*, 2012.

[18] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. K. Kraetzschmar, L. Gherardi, and D. Brugali, "The brics component model: a model-based development paradigm for complex robotics software systems," in *SAC*, 2013.

[19] C. Schlegel, T. Hassler, A. Lotz, and A. Steck, "Robotic software systems: From code-driven to model-driven designs," in *2009 International Conference on Advanced Robotics*, pp. 1–8, June 2009.

[20] A. M. Khamis, A. Hussein, and A. M. Elmogy, "Multi-robot task allocation: A review of the state-of-the-art," in *Advances in Social Media Analysis*, 2015.

[21] H. Kuhn, "The hungarian method for the assignment problem," *Naval Res. Logist. Quart.*, vol. 2, pp. 83–98, 01 1955.

[22] E. Nunes, M. McIntire, and M. Gini, "Decentralized allocation of tasks with temporal and precedence constraints to a team of robots," in *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pp. 197–202, Dec 2016.

[23] A. R. Mosteo and L. Montano, "A survey of multi-robot task allocation," 2010.

[24] K. Hoffman and M. Padberg, "Solving airline crew scheduling problems by branch-and-cut," *Management Science*, vol. 39, pp. 657–682, 06 1993.

[25] A. Atamturk, G. Nemhauser, and M. W. P. Savelsbergh, "A combined lagrangian, linear programming and implication heuristic for large-scale set partitioning problems," *Journal of Heuristics*, vol. 1, pp. 247–259, 1995.

[26] M. B. Dias, R. Zlot, N. Kalra, and A. Stentz, "Market-based multirobot coordination: A survey and analysis," *Proceedings of the IEEE*, vol. 94, pp. 1257–1270, July 2006.

[27] "What is ros?." http://wiki.ros.org/ROS/Introduction. Accessed: 2019-05-06.

[28] "Setup and configuration of the navigation stack on a robot." http://wiki.ros.org/navigation/Tutorials/RobotSetup. Accessed: 2019-05-06.

[29] R. B Rusu, A. Maldonado, M. Beetz, M. Kranz, L. Mösenlechner, P. Holleis, and A. Schmidt, "Player/stage as middleware for ubiquitous computing," *Proceedings of the 8th Annual Conference on Ubiquitous Computing (Ubicomp 2006), Orange County California, September 17-21, 2006*, 05 2019.

[30] H. Bruyninckx, "Open robot control software: the orocos project," vol. 3, pp. 2523 – 2528 vol.3, 02 2001.

[31] H. Utz, S. Sablatnög, S. Enderle, and G. K. Kraetzschmar, "Miro - middleware for mobile robot applications," *IEEE Trans. Robotics and Automation*, vol. 18, pp. 493–497, 2002.

[32] A. Y. Elkady and T. M. Sobh, "Robotics middleware: A comprehensive literature survey and attribute-based bibliography," *J. Robotics*, vol. 2012, pp. 959013:1–959013:15, 2012.

[33] "Language engineering for everyone!." https://www.eclipse.org/Xtext/. Accessed: 2019-05-06.

[34] "Eclipse modeling framework (emf)." https://www.eclipse.org/modeling/emf/. Accessed: 2019-05-06.

[35] D. Djuric, D. Gasevic, and V. Devedzic, "The tao of modeling spaces," *Journal of Object Technology*, vol. 5, pp. 125–147, 01 2006.

[36] "Task-definition-language." https://github.com/95danlos/Task-Definition-Language. Accessed: 2019-05-31.

[37] "Kuka youbot kinematics, dynamics and 3d model." http://www.youbot-store.com/developers/kuka-youbot-kinematics-dynamics-and-3d-model-81. Accessed: 2019-05-06.

[38] H. C. Lau, "Task allocation via multi-agent coalition formation: Taxonomy, algorithms and complexity," in *ICTAI*, 2003.

[39] "Ros master." http://wiki.ros.org/Master. Accessed: 2019-05-06.

[40] "Leaflet." https://leafletjs.com/. Accessed: 2019-05-25.

[41] "Multi-robot-mission-planner." https://github.com/95danlos/Multi-Robot-Mission-Planner. Accessed: 2019-05-06.

[42] S. Ivaldi, V. Padois, and F. Nori, "Tools for dynamics simulation of robots: a survey based on user feedback," *CoRR*, vol. abs/1402.7050, 2014.

Primary Studies

[S1] D. D. Ruscio, I. Malavolta, and P. Pelliccione, "A family of domain-specific languages for specifying civilian missions of multi-robot systems," in *MORSE@STAF*, 2014.

[S2] F. Ciccozzi, D. D. 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.

[S3] S. Dragule, B. Meyers, and P. Pelliccione, "A generated property specification language for resilient multirobot missions," in *SERENE*, 2017.

[S4] D. Ouellet, S. N. Givigi, and A. J. G. Beaulieu, "Control of swarms of autonomous robots using model driven development - a state-based approach," *2011 IEEE International Systems Conference*, pp. 512–519, 2011.

[S5] A. J. G. Beaulieu, S. N. Givigi, D. Ouellet, and J. T. Turner, "Model-driven development architectures to solve complex autonomous robotics problems," *IEEE Systems Journal*, vol. 12, pp. 1404–1413, 2018.

[S6] C. Pinciroli and G. Beltrame, "Buzz: An extensible programming language for heterogeneous swarm robotics," pp. 3794–3800, 10 2016.

[S7] T. Amma, P. Baer, K. Baumgart, P. Burghardt, K. Geihs, J. Henze, S. Opfer, S. Niemczyk, R. Reichle, D. Saur, *et al.*, "Carpe noctem 2009,"

[S8] H. Skubch, M. Wagner, R. Reichle, and K. Geihs, “A modelling language for cooperative plans in highly dynamic domains,” *Mechatronics*, vol. 21, pp. 423–433, 03 2011.

[S9] A. Paraschos, N. I. Spanoudakis, and M. G. Lagoudakis, “Model-driven behavior specification for robotic teams,” in *AAMAS*, 2012.

[S10] E. M. Martinez, A. F. Caballero, and J. M. G. Noheda, “Model-driven engineering techniques for the development of multi-agent systems,” 2012.

[S11] P. A. Baer, R. Reichle, M. Zapf, T. Weise, and K. Geihs, “A generative approach to the development of autonomous robot software,” in *Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe’07)*, pp. 43–52, March 2007.

[S12] P. A. Baer, R. Reichle, and K. Geihs, “The spica development framework – model-driven software development for autonomous mobile robots,” in *Intelligent Autonomous Systems 10 – IAS-10*, pp. 211–220, jul 2008.

Appendices

Appendix A

Meta Model

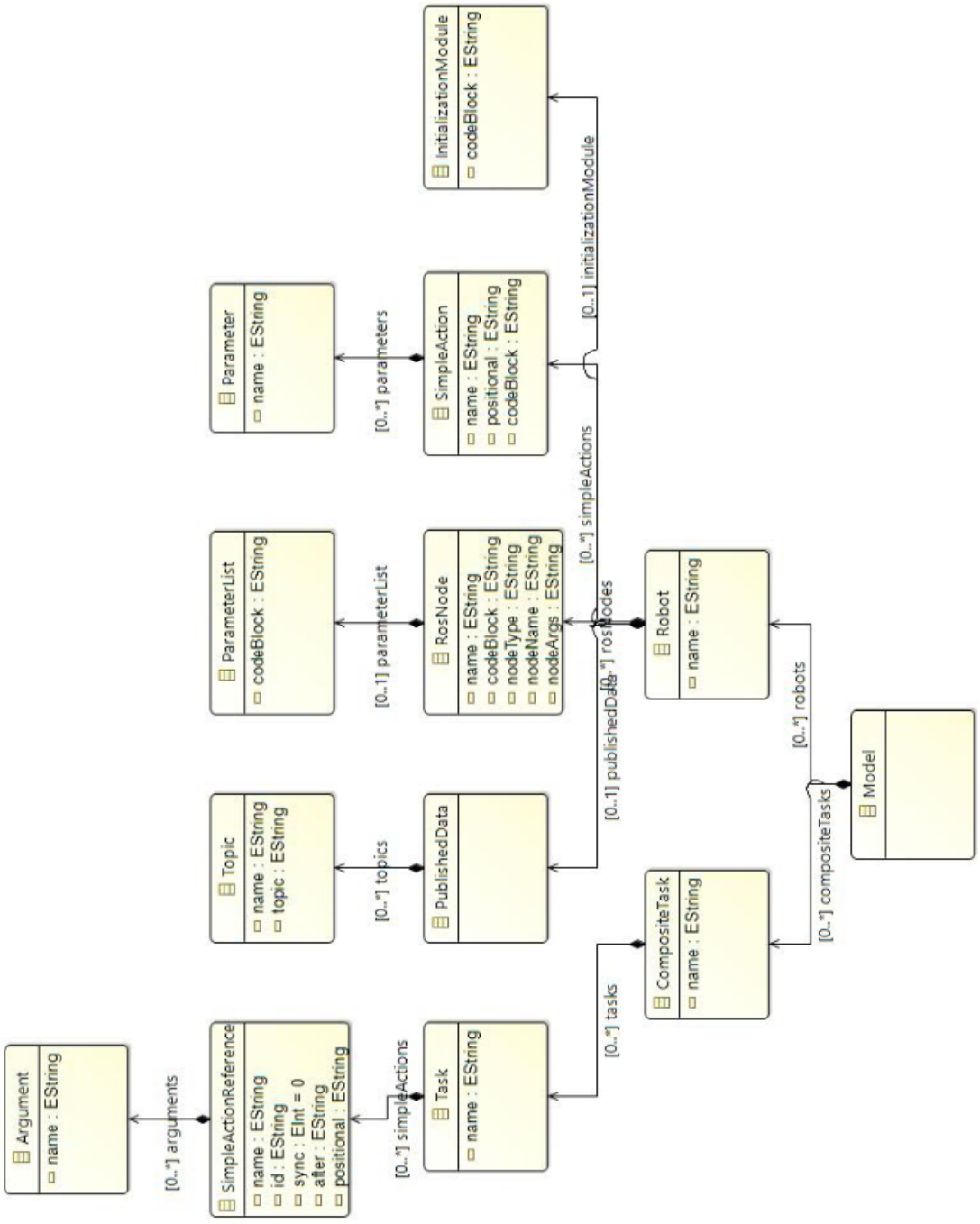


Figure A.1: Task Definition Language Meta Model Large.

Appendix B

Task Definition Language Grammar

```
grammar org.xtext.tdl.Tdl with org.eclipse.xtext.common.Terminals

generate tdl "http://www.xtext.org/tdl/Tdl"

Model:
        compositeTasks += CompositeTask*
        robots += Robot*
;

CompositeTask:
        "compositeTask" name=ID "(lat,lng):"
        BEGIN
                tasks += Task*
        END
;

Task:
        "task" name=ID "():"
        BEGIN
                simpleActions += SimpleActionReference*
        END
;

SimpleActionReference:
        name=ID
        (
                "(" ( ("id="id=STRING",")? ("sync("sync=INT")"?","?)? ("
                    after("after=STRING")"?","?)? ("args=("arguments+=
```

```
                    Argument*")"?","?)? ) ( "):" | ")):" )
        |
                positional="(lat,lng):"
        |
                "():"
        )
;

Argument:
        name=STRING ","?
;

Robot:
        "robot" name=ID ":"
        BEGIN
                publishedData = PublishedData?
                initializationModule = InitializationModule?
                simpleActions += SimpleAction*
                rosNodes += RosNode*
        END
;

InitializationModule:
        "setup:"
        BEGIN
                codeBlock = CodeBlock
        END
;

SimpleAction:
        "simpleAction" name=ID ( "("(parameters+=Parameter*)"):" |
            positional="(lat,lng):" | "():" )
        BEGIN
                codeBlock = CodeBlock
        END

;

Parameter:
        name=ID ","?
;

RosNode:
        (
        "rosNode" name=ID "():"
        BEGIN
```

```
                codeBlock = CodeBlock
        END     )
        |
        ("rosNode" name=ID "(" nodeType=STRING "," nodeName=STRING ("," 
            nodeArgs=STRING)? "):"
        (
        BEGIN
                parameterList = ParameterList
        END
        )?
        )
;

ParameterList:
        "Parameters:"
        BEGIN
                codeBlock = CodeBlock
        END
;

PublishedData:
        "PublishedData:"
        BEGIN
                topics += Topic*
        END
;

Topic:
        name = ID ":" topic = STRING
;

CodeBlock hidden():
    (ID | WS | INT | STRING | ML_COMMENT | SL_COMMENT | ANY_OTHER | "(" | ")
        " | "):" | "()" | ":" | "'" | "," | "))" | CodeBlock_2 )*
;

CodeBlock_2 hidden():
        BEGIN
                CodeBlock
        END
;

terminal BEGIN: 'synthetic:BEGIN';
terminal END: 'synthetic:END';
```

Listing B.1: Task Definition Language Grammar.

Appendix C

User Manual

Here we will explain how to setup the project in Eclipse, how to setup the simulator, how to deploy on real robots, and how to setup the project for further development. The project can be found at https://github.com/95danlos/Task-Definition-Language.

Setup on simulator requires Linux with ROS installed. Setup on real robots requires Linux with ROS installed on each robot and each robot must have a gps.

```
git clone https://github.com/95danlos/Task-Definition-Language.git
```

C.1 Eclipse Setup

Install Xtext and Xtend:

In the Eclipse menu bar click –> help –> Eclipse Marketplace and search for Xtext and Xtend.

Add the Task-Definition-Language Plugin:

In the Eclipse menu bar click –> help –> install new software –> add –> local –>

select the plugin folder in the cloned project, if the plugin is not listed uncheck "Group item by category" –> install the plugin and restart Eclipse.

Create a new project:

In Eclipse click –> new project –> general –> project.

Create a new folder call it src, here you can add a file for each robot and one file describing the tasks.

To add a new robot create a new file under src with the extension .tdl. Click yes when asked to convert to Xtext project.

A robot file should contain an implementation of each simple action that the robot is able to perform, and can also contain implementations of ROS nodes and parameters. See the ROS tutorials for information on how to write ROS nodes http://wiki.ros.org/ROS/Tutorials.

Create a new .tdl file and define each task that the robots should perform.

Example files for task definitions and simulated robots can be found under multi-robot-simulation/robots.

C.2 Simulator Setup

Setup on simulator requires ROS. Follow the ROS setup tutorial at http://wiki.ros.org/ROS/Installation.

Install the Gazebo simulator by following the steps at http://gazebosim.org/tutorials?tut=ros_installing.

Move the gazebo models from Task-Definition-Language/multi-robot-simulation/gazebo-models over to ~/.gazebo/models.

Create a catkin workspace by following the steps at http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment.

Create a new ROS package called multi-robot-simulation by following the steps at http://wiki.ros.org/ROS/Tutorials/CreatingPackage.

Build the package by following the steps at http://wiki.ros.org/ROS/Tutorials/BuildingPackages.

Copy the files from the folder called multi-robot-simulation in the cloned project over to the newly created package.

Make the src files executable:

```
chmod +x -R ~/catkin_ws/src/multi-robot-simulation/src
```

Install python websocket client:

```
pip install websocket_client
```

Start the server:

```
python ~/catkin_ws/src/multi-robot-simulation/server.py
```

Launch the simulation:

```
roslaunch multi-robot-simulation multi-robot-simulation.launch
```

Open the index file in ~/catkin_ws/src/multi-robot-simulation.

Figure C.1: Gazebo Simulator Setup.

C.3 Setup on Real Robots

Setup on real robots requires all of the robots to have ROS installed. Follow the ROS setup tutorial at http://wiki.ros.org/ROS/Installation.

Create a catkin workspace: http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment.

Create a new ROS package called multi-robot-simulation: http://wiki.ros.org/ROS/Tutorials/CreatingPackage.

Build the package: http://wiki.ros.org/ROS/Tutorials/BuildingPackages.

Add a .tld file for each robot and one for the tasks as described under the Eclipse Setup section above.

Change the SERVER_IP_ADDRESS variable in each generated setup file and in each generated task allocation module file under src-gen to the ip address of the machine that is going to run the server.

Move the src-gen folders for each robot to
~/catkin_ws/src/multi-robot-simulation/src on the appropriate robot.

Run the startup files on each robot, then start the server and open the index file.

C.4 Maintainer Setup

Import to Eclipse:

In Eclipse click –> File –> Import –> Existing Projects into Workspace –> Browse –> select the Task-Definition-Language-Project folder.

Install Xtext and Xtend:

In the Eclipse menu bar click –> help –> Eclipse Marketplace and search for Xtext and Xtend.

The project org.xtext.tdl contains six files used for development found under src:

Tdl.xtext contains the grammar for the task definition language written in Xtext.

TdlGenerator.xtend contains the generator which is used to take information from the robot and task files to create the index file for the web interface, and the task allocation module, launch file, startup file, and ROS files for each robot.

HelperMethods.java contains methods used by the generator to format generated files.

index.html contains the web interface and task definitions.

server.py connects the web interface with the robots and the robots with each other.

task_allocation_module.py is used to distribute tasks amongst the robots and execute actions.