TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Software- und Multimediatechnik, Lehrstuhl für Softwaretechnologie

Minor Thesis

Runtime Model driven ROS with RAGConnect

Edgar Solf

Born on: 16th August 2001 in Erfurt
Matriculation number: 4971025
Matriculation year: 2020

12th March 2025

Supervisor
Dr. Sebastian Götz

Supervising professor
Prof. Dr. rer. nat. Uwe Aßmann

Abstract

The fast evolution of robotics requires a new approach to managing robot systems, especially when dealing with changing environments and requirements. Traditional approaches to robotic software development, particularly in the context of the Robot Operating System (ROS), often rely on static code that requires manual modification to adapt to new circumstances. This thesis presents a model-driven solution aimed at addressing these challenges through the integration of runtime models with -based robotic applications with the use of JastAdd and RAGConnect.

The proposed approach uses Model-Driven Development (MDD) principles and Reference Attribute Grammers (RAGs) to define and adapt robotic behaviors in real time. Using JastAdd, a tool for aspect-oriented compiler construction, the system defines Domain Specific Languages (DSLs) as RAGs that interact with a runtime model, enabling robots to change their behavior based on changing conditions. The integration of RAGConnect, a method for establishing causal connections between runtime models and external systems, allows for communication between the ROS-based system and external environments using the Message Queuing Telemetry Transport (MQTT) protocol.

This solution is demonstrated through a case study involving robots and a dynamic web application. The approach is shown to be modular, scalable and extensible. The evaluation of the system, which includes performance tests with different numbers of robots and the introduction of new entities, drones, shows the effectiveness of the proposed model-driven framework.

In conclusion, the thesis provides a framework for building adaptive, scalable and maintainable robotic systems. The findings suggest that using runtime models and dynamic web interfaces can simplify the development process and improve the agility of robotic applications in real-time environments.

Contents

1 Introduction

The development of applications with Robot Operating System (ROS) has seen significant advancements in recent years. As robotics systems grow in size and complexity, managing them efficiently becomes a challenge. Traditional approaches often involve manually coding every aspect of a system, which can be time-consuming and error-prone. This paper explores a model-driven approach that simplifies the development process while keeping flexibility and adaptability.

Model-Driven Development (MDD) offers a structured methodology to define robotic behaviors using high-level models rather than low-level code. By leveraging runtime models, a robotic system can dynamically adjust to new requirements and environments without manual low level adjustments. This adaptability is crucial in scenarios where rapid changes occur, such as industrial automation or cobotics.

ROS is a widely adopted framework for robotic software development. It provides essential tools and communication protocols that enable modular design and allows components to be reused in different systems. However, integrating MDD principles with ROS remains a challenge, as traditional ROS applications often rely on static implementations that limit their flexibility.

This paper introduces an approach that combines runtime models with ROS using RAGConnect, a tool that allows connections between models and the execution environments. By including Reference Attribute Grammers (RAGs), the system can be modified easily without low-level modifications. Additionally, the Message Queuing Telemetry Transport (MQTT) protocol is utilized to communicate between ROS and RAGConnect.

The proposed solution is evaluated through a case study involving autonomous robots operating in a simulated environment. The study demonstrates how runtime models enhance scalability, extensibility and maintainability in ROS-based applications. By minimizing the need for manual code adjustments, this approach enables more agile development cycles and reduces the complexity of managing robotic systems.

This paper is structured as follows: Chapter 2 offers background information on ROS, MDD and runtime models. Chapter 3 deals with the state of the art in model-driven robotics. Chapter 4 explains the concept behind the proposed approach, especially its architecture and components. Chapter 5 describes the implementation process, including the integration of RAGConnect and MQTT. Chapter 6 evaluates the approach through experiments

and case studies. Finally, Chapter 7 concludes the paper and names future research directions.

2 Background

2.1 ROS2

2.1.1 Introduction to Robot Operating System (ROS)

ROS is an open-source middleware framework for developing robotic applications. Originally started in 2007 by Willow Garage, ROS has become a standard for robotic software development in research and academia due to its modular architecture and community-driven development [11]. The framework provides various libraries and tools for handling robot functionalities such as navigation, perception and control, enabling researchers to develop complex applications without building every component from scratch.

ROS 1, however, was designed primarily for academic use in controlled environments, which introduced certain limitations. Its architecture lacked robust support for real-time performance, security and distributed network configurations—features, which were increasingly required in modern robotics. These limitations led to the development of ROS 2 [1].

The core concepts of the ROS and ROS 2 middleware include Nodes, Topics, Services and Actions, which collectively define its communication patterns [8].

- **Nodes** are modular components designed to perform specific tasks, such as controlling a robot's wheels. Nodes interact with each other using Topics, Services and Actions, forming the entities within the ROS ecosystem.
- **Topics** represent the most commonly used communication pattern. They enable asynchronous message passing between nodes. A node can publish messages to a Topic, making the information available to other nodes that subscribe to that Topic.
- **Services** facilitate synchronous communication. In this pattern, a service client sends a request to a service server, which processes the request and returns a response, ensuring a direct interaction between the two nodes.
- **Actions** are designed for asynchronous, goal-oriented communication, often used for long-running tasks. In this pattern, an action client sends a goal to an action server. The server processes the goal, provides periodic feedback and sends a result when

the task is completed.

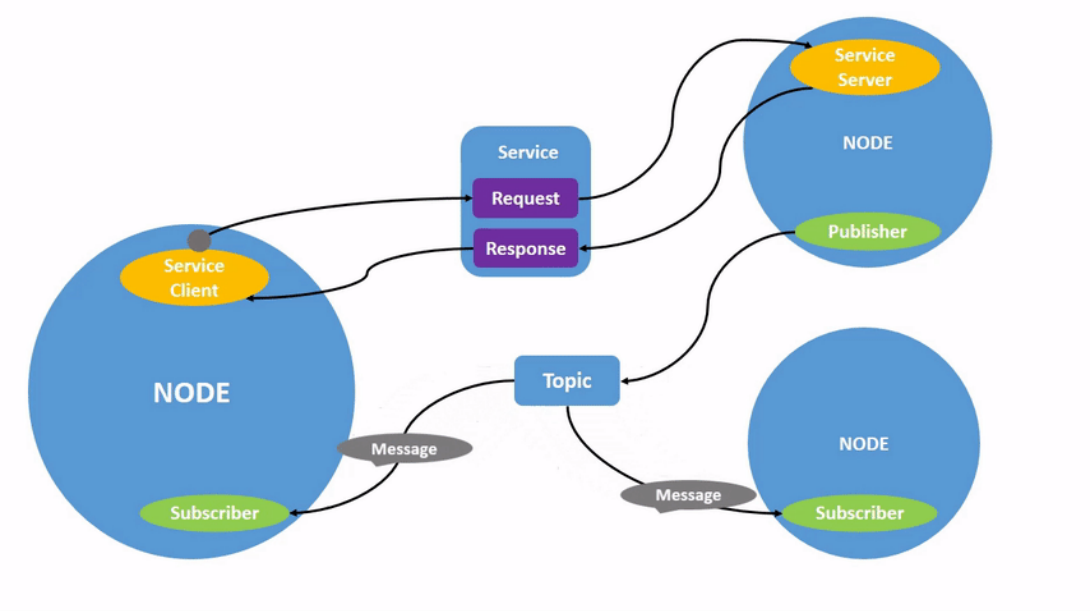

Figure 2.1: Interaction of Nodes, Topics and Services in ROS 2 [16].

2.2 Model-Driven Development

MDD is a software engineering approach that emphasizes the use of high-level models to represent system specifications, which are then systematically transformed into executable code. This approach aims to enhance productivity, maintainability and portability by abstracting complex system details into more comprehensible models.

MDD focuses on creating abstract representations of domain models of a particular application domain. By using models as primary artifacts in the development process, MDD displays automation in code generation and system validation. This paradigm shift addresses challenges in traditional software development, such as managing complexity and ensuring consistency across various system components.

The abstraction and automation offered by MDD make it especially valuable for managing complex systems, improving collaboration among developers and ensuring adaptability to changing requirements. By emphasizing models over manual coding, MDD enhances system transparency and reduces development effort, especially in large-scale projects.

This structured approach positions MDD as a good practice in modern software engineering, offering a bridge for the gap between system design and implementation.

2.3 Runtime models

Runtime models play a critical role in the development of self-adaptive systems, acting as a dynamic representation of the system and its environment during execution. These models are foundational to the Models@run.time paradigm, where a system's state, structure

and goals are continuously mirrored, enabling real-time monitoring and adaptation. Unlike static models used only for design or analysis, runtime models evolve together with the system, allowing them to react to contextual changes effectively. This adaptivity is essential in complex environments such as autonomous robotics or Internet of Things (IoT) ecosystems, where new conditions must be addressed dynamically [2].

A key feature of runtime models is their capability to enable causal connections, ensuring a bidirectional link between the system's model and its real-world representation. Changes in the system or its environment are propagated to the runtime model, which, in turn, drives adaptations or responses back to the system. This feedback loop, supported by mechanisms like incremental evaluation and dependency tracking, ensures efficiency and precision. Modern implementations use advanced technologies such as relational reference attribute grammars to enhance modularity, consistency and performance. These grammars allow runtime models to represent complex relations and dependencies, supporting complex computations natively [14].

Runtime models are more than representations—they are enablers of decision-making and automation in self-adaptive systems. Their formalism ensures that changes are not only detected but also processed efficiently, avoiding computational overhead and redundant recalculations. In the broader context, they bridge the gap between theoretical models and practical system execution, providing a robust foundation for systems to evolve in real-time while meeting high standards of reliability and scalability. This makes runtime models indispensable for the future of adaptive and autonomous computing [3].

2.3.1 ROS with runtime models

Runtime models are highly advantageous in the context of ROS 2 and the IoT, where the ability to manage distributed, heterogeneous and adaptive systems is critical. ROS 2, with its decentralized communication architecture and real-time capabilities, aligns well with IoT ecosystems that often involve diverse devices, sensors and actuators operating under varying conditions. Runtime models provide a unified and dynamic representation of this distributed system, enabling efficient monitoring and adaptation.

In IoT applications, runtime models can bridge the gap between physical devices and their digital counterparts. For instance, in a smart home scenario, a runtime model can maintain the current state of interconnected devices like thermostats, lights and security cameras. Integrated with ROS 2, these models can react to changes like a sensor detecting motion or a user adjusting the target temperature by propagating updates across the system and triggering coordinated actions in real-time.

Additionally, runtime models enhance the scalability of IoT systems. By capturing high-level abstractions of device states and interactions, they reduce the complexity of managing a growing number of devices. In ROS 2, this translates to better control of distributed nodes, ensuring seamless communication and coordination even in large-scale IoT deployments. Moreover, runtime models enable predictive and proactive adaptation, allowing IoT systems to anticipate changes, optimize resource usage and maintain performance under dynamic conditions.

Incorporating runtime models into ROS 2-powered IoT systems provides a robust foundation for creating adaptive, efficient and scalable solutions, meeting the demands of modern connected environments.

2.4 MQTT

MQTT is a lightweight, publish-subscribe messaging protocol designed for machine-to-machine communication, particularly in environments with limited resources and unreliable networks. Developed in 1999 by Andy Stanford-Clark of IBM and Arlen Nipper of Eurotech [9], MQTT was initially created to monitor oil pipelines over satellite connections, where bandwidth and battery power were sparse [15].

The protocol operates on top of the TCP/IP stack and employs a client-server architecture, where devices communicate through a central broker. Clients can publish messages to specific topics or subscribe to topics to receive messages, facilitating a decoupled communication model that enhances scalability and flexibility [9].

One of MQTT's defining features is its minimal overhead, making it ideal for IoT applications involving devices with limited processing capabilities and network bandwidth.

2.4.1 ROS with MQTT

In the context of ROS 2, integrating MQTT enables seamless communication between ROS 2 nodes and external devices or systems that utilize MQTT. This integration is done via bridging mechanisms that translate messages between ROS 2 topics and MQTT topics, allowing for interoperability between different communication protocols.

One such bridging solution is the mqtt_client package, which provides a ROS 2 node that enables connected ROS-based devices or robots to exchange ROS messages via an MQTT broker using the MQTT protocol. This works generically for arbitrary ROS message types [10].

Another solution is the mqtt_bridge package, which provides a functionality to bridge between ROS and MQTT in bidirectional. It uses ROS message as its protocol, where messages from ROS are serialized to JSON (or MessagePack) for MQTT and messages from MQTT are deserialized for ROS topics. This allows for seamless communication between ROS and MQTT-enabled devices [5]. Unfortunately the mqtt_package currently is only available for ROS 1.

Despite the advantage of JSON serialization the mqtt_bridge package won't be used in this project, because of the limitation to ROS 1. Instead the mqtt_client package will be used and the missing JSON serialization feature is compensated through a custom build ROS message to JSON serializer.

2.5 Reference Attribute Grammars

RAGs extend traditional Attribute Grammers (AGs) by incorporating mechanisms for referencing other nodes in a syntax tree, thus facilitating the modeling of non-local dependencies in a structured and declarative manner. AGs, first introduced by Knuth in 1968, provide a formalism for associating computations, known as attributes, with the nodes of a context-free grammar's parse tree. Attributes are categorized into synthesized at-

tributes, which are computed from child nodes to parent nodes and inherited attributes, which propagate from parent nodes to child nodes [7].

However, AGs have limitations in expressing non-local relationships, such as variable bindings, scope resolution or cross-references between entities in complex program analyses. To address these limitations, RAGs augment AGs with reference attributes, enabling nodes in a parse tree to hold references to other nodes within the same tree or even in external data structures [6].

2.5.1 Key Concepts of RAGs

- **Reference Attributes:** These are specialized attributes that do not merely compute scalar or composite values but instead establish links between nodes. For example, in a semantic analysis of a programming language, a reference attribute might link an identifier node to its corresponding declaration node.
- **Graph-Like Structure:** While traditional AGs operate strictly on tree structures, RAGs effectively transform the parse tree into a graph by introducing edges that represent the references. This graph-like structure facilitates the representation of more sophisticated language constructs and relationships.
- **Declarative Semantics:** RAGs retain the declarative nature of AGs by allowing the definition of attributes and their computations in a modular and reusable fashion. Reference attributes are typically specified through declarative rules that define how references are established and updated.
- **Applications:** RAGs are particularly useful in domains requiring the representation of complex relationships, such as:
  - **Language Analysis:** Resolving variable declarations and type checking.
  - **Modeling Systems:** Representing interdependencies in software or hardware systems.
  - **Graph-Based Computations:** Addressing scenarios where graph traversal and analysis are integral.

2.5.2 Advantages of RAGs

Expressive Power: By enabling non-local dependencies, RAGs significantly expand the expressive capabilities of traditional AGs. Modularity: Reference attributes allow for modular definitions of complex semantic rules, enhancing maintainability and readability. Formal Framework: RAGs offer a formal, well-defined approach to semantic specification, which is beneficial for automated tool generation [6].

2.5.3 Challenges and Limitations

Despite their advantages, RAGs introduce complexities, including:

- **Implementation Overhead:** Supporting graph structures and references necessitates more sophisticated algorithms and data structures, increasing implementation complexity.
- **Performance Considerations:** Non-local dependencies may require additional computations, potentially impacting performance in large-scale applications.

2.6 JastAdd

JastAdd is an aspect-oriented compiler construction system. Developed using Java, JastAdd uses an object-oriented Abstract Syntax Tree (AST) representation, supporting complex semantic analysis through a combination of imperative programming in Java and declarative specifications using RAGs. This approach allows attributes to represent references to other AST nodes, enabling connections across the tree structure.

Unlike conventional systems, JastAdd modularizes the compiler's behavior into distinct "aspects" that are woven into classes through aspect-oriented programming. This modularization supports the integration of fields into classes alongside methods, allowing seamless aspect composition within the object-oriented design of Java.

The JastAdd system uses a recursive evaluation strategy, as attribute evaluation is managed by caching computed values and detecting cyclic dependencies. Additionally, the system separates generated AST classes from handwritten modules, allowing for modifications and updates without disrupting the underlying codebase.

2.7 RAGConnect

RAGConnect is a transformative approach in the realm of self-adaptive systems, addressing a key gap in the integration of runtime models with external systems. Building upon the foundation of RAGs, RAGConnect enables seamless, efficient connections between a system's internal representation and its environment. This is pivotal in ensuring real-time adaptability, as it allows models to receive and propagate changes incrementally without manual dependency tracking or conversion into intermediate data structures. The approach supports a declarative specification of causal connections, which enhances modularity and reduces complexity during system development [14].

One of the standout features of RAGConnect is its ability to manage complex data flows through incremental evaluation mechanisms inherent in RAGs. By leveraging these mechanisms, the system can react dynamically to changes in external inputs while maintaining high performance. The use of a Domain Specific Language (DSL) simplifies the process of defining connections and mappings between runtime models and external systems, ensuring that even sophisticated interactions can be expressed concisely. This capability is further exemplified in applications such as industrial robotics, where the coordination of multiple entities demands precise, efficient and real-time synchronization [14].

RAGConnect represents a significant advancement in model-driven engineering for self-adaptive systems. It provides a robust framework for integrating models with diverse external systems. This approach enhances the flexibility, scalability and efficiency of adaptive

systems, making it a critical tool in the development of modern cyber-physical systems [14].

3 State of the art

3.1 ROS-Mobile

ROS-Mobile is an Android app that addresses the growing use of mobile devices and the lack of a tool that supports ROS on them. It was developed in 2020 by Nils Rottmann, Nico Studt and others at the University of Lübeck [12].

Built with the Model-View-ViewModel architecture, ROS-Mobile ensures modularity and ease of maintenance. It supports Android 5.0 and higher, requiring only a wireless connection to the ROS master. Users can configure nodes and access control and visualization tools directly via the app.

While ROS-Mobile accelerates robot testing and development, it focuses on basic control and lacks runtime models for advanced functionality. Its adaptability makes it suitable for diverse applications, including household robots, rescue robotics and healthcare navigation systems. However, enhancing abstraction and runtime adaptability could further expand its use cases.

3.2 LaRAAB

In his study, Adrian Scholze evaluates the application of Models@run.time to enhance the development of robotic applications by integrating adaptive graphical user interfaces with complex robotic control systems. The usual approach, relying on frameworks like ROS and simulators such as Gazebo or Webots, often lacks generality, modularity and adaptability. Scholze introduces a model-driven software architecture called "LaRAAB" (Laufzeitmodell-getriebene Robotik-Anwendung mit Adaptiver Benutzeroberfläche) to bridge these limitations [13].

The LaRAAB framework incorporates a runtime model that synchronizes the system state with a webapp, using a Monitor-Analyze-Plan-Execute while having a shared Knowledge (MAPE-K) loop for dynamic interaction between a mobile interface and robotic systems. Generators derived from runtime metamodels automate significant portions of interface creation and system logic, enabling easier scaling and domain-specific adaptations.

By focusing on runtime modeling and adaptive user interfaces, the work addresses existing challenges in usability and reuse in robotic applications. However, the prototype's reliance on a fixed set of tools and limited testing in multi-robot scenarios highlights its experimental stage. The paper leaves open further reducing of manual coding and configuration, which will be addressed in this paper.

3.3 Generating Causal Connections for Model-driven ROS-based Applications

In his research, Jiabin Dai explores the enhancement of model-driven development for ROS-based robotic applications, focusing on bridging runtime models and ROS through automated tools and abstractions. Traditional development in the ROS ecosystem often requires developers to operate in the solution space, dealing with intricate message structures and system intricacies. To address these challenges, Dai introduces a comprehensive framework emphasizing the integration of Models@run.time and DSLs [4].

The proposed framework includes a runtime model connected to ROS using a MAPE-K loop structure, encapsulated within a component-based architecture. Central to the innovation is a DSL implemented with the textX library, allowing developers to program in the problem space while abstracting complex backend details. This DSL is complemented by a code generator that translates high-level specifications into executable Python code, facilitating synchronization between the runtime model and ROS.

Dai evaluates the framework with practical case studies, such as a warehouse scenario involving robots and drones simulated in Webots. These experiments demonstrate the feasibility and adaptability of the proposed approach but also highlight areas for improvement, including scalability and integration with varied robotic platforms. Especially the hard limitation for "if-clauses" and other constructs need to be solved before any production ready systems are deployed. Future research aims to refine the DSL and expand its applicability to multi-robot systems and broader automation tasks [4].

4 Concept

In this section, the fundamental considerations regarding the concept and structure of the system are introduced. First, the problem encountered when developing a project with ROS is described, along with the solution proposed by Adrian Scholze. A new, extended solution is then presented, aiming to overcome some of the limitations of Scholze's approach. The decision to switch from PyEcore to JastAdd is explained, highlighting how this change improves flexibility and maintainability. An overview of the software architecture is provided, with a particular focus on the interactions between components such as the web application, the runtime model and the connection to ROS.

4.1 Problem

When developing a project with ROS traditionally every function of every component of the system has to be specifically implemented with a programming language like C++ or Python. Whenever changes want to be made, the whole codebase needs to be altered, which often leads to high developing costs and slows the production process, which is deadly in agile environments.

Adrian Scholze proposed a solution which used PyEcore to generate the python classes for the DSL structure. This is a step in the right direction, because it leads to fewer required adjustments when the DSL is altered. But in his approach each class that requires additional functionality (almost every class that is not a simple data class) still needs a manually implemented implementation class. The same is true for the main function which includes a MAPE-K loop that needs to be specifically written for this exact use case. Adding to that the connection to ROS is done via an hardcoded class which again can only handle the communication for this specific use case. And finally the web app doesn't allow specification for a different action nor for multiple actions. This paper tries to overcome these disadvantages and proposes a solution that keeps the core ideas of Adrian Scholzes approach.

4.2 Proposed Solution

Instead of using PyEcore the proposed solution switches to JastAdd and thereby has Java as the main programming language.JastAdd allows for defining the DSL as a RAG, which takes over this responsibility from PyEcore. In addition to that it introduces the option for defining aspects which can include functions or attribute methods that are only evaluated again once the input parameters have changed. The hard coded definition of the specific runtime model can be extracted from the main class and put into an initialization attribute method that can be defined via an aspect. With JastAdd it is also possible to take every other domain specific functionality out of the main class and make it suitable for different use cases in many DSLs.

What JastAdd still can't solve, is the connection between the runtime model and ROS. That is where RAGConnect can bridge the gap. RAGConnect offers a way of connecting an attribute with a MQTT topic. This can be used to have the MQTT topic as an intermediate between the runtime model and ROS. The connection between MQTT and ROS can be closed with an extra ROS node that handles this specifically by synchronizing MQTT and ROS topics.

To make the generation of the web app obsolete, the web app contains no domain specific functionality and instead adjusts itself to the received data. This makes it possible to even run the web app once and not even having to restart after changes in the DSL have been made. The available actions are contained in the data and are then displayed accordingly.

4.3 General Structure

The following structure is shown in Figure 4.1. The web app will be the front end of the system and will allow the user to observe the current status and set new goals. Through web sockets the web app will be up-to-date and communicate with the controller which runs inside the runtime model application. The controller itself will read the AST to send to the web app, but will also execute the goals received from the web app by manipulating the AST. RAGConnect will ensure that all changes in the AST nodes that are exposed for output will trigger a MQTT message that gets converted to a ROS message which reaches the actual ROS Application. On the other side every message in specific ROS topics will trigger the ROS MQTT Handler to convert it to a MQTT message. RAGConnect will read these messages and update the AST nodes exposed for input, which might trigger a new computation of the AST nodes exposed for output and starts the next cycle. This interaction forms the MAPE-K loop and will be further described in Chapter 4.4.

4.4 MAPE-K Loop

Figure 4.2 depicts the MAPE-K loop in the system. It consists of the interaction of RAG-Connect and the AST with the managed component being a ROS 2 application. The ROS 2 application has at least one sensor that acts as a publisher and informs the monitor via ROS (and MQTT) topics of current measurements. The monitor, a RAGConnect component, checks the received measurements and if a change is detected, forwards it to the

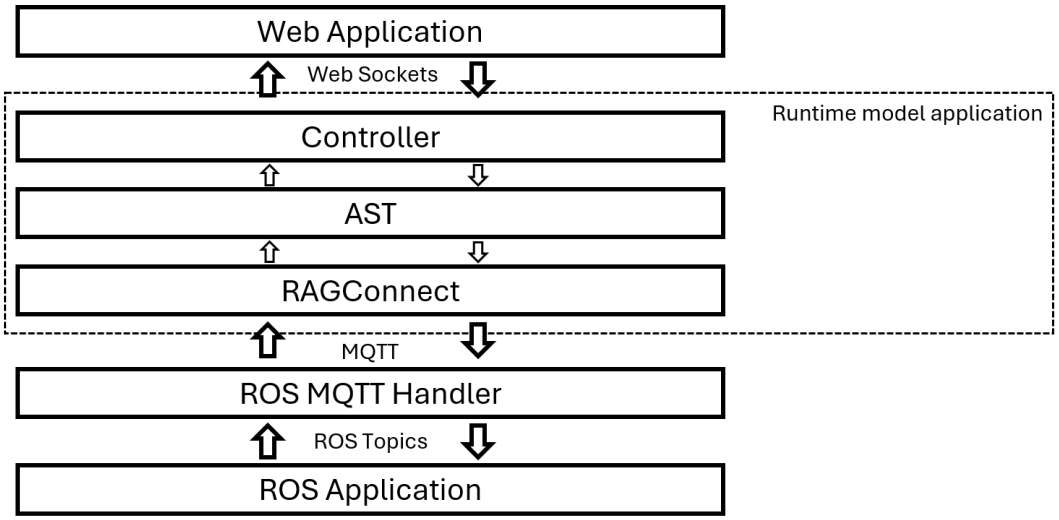

Figure 4.1: Structure of the components and their communication

analyzing phase by rewriting a value in the AST which invalidates all synthesized or inherited attributes that depend on it. By this, nodes are reevaluated that are part of the actions required to reach the specified goal, which makes the analyzing phase flow seemingly into the planing phase. At the end of this reevaluation all depending nodes have new values including RAGConnect output attributes, which trigger a MQTT message on change. This trigger build in RAGConnect makes up the execution part of the MAPE-K loop. The sent MQTT message will then reach an actuator, another ROS 2 Node, via MQTT and ROS topics. And all this happens with all necessary information stored in the AST which provides the knowledge base for the loop.

4.5 Generation

While the goal is to leave as little programming effort to the developer, there is a need to ensure that the developer still has control over everything in his ROS application. Figure 4.3 shows the generation dependencies and the required steps of the developer. As the web app dynamically adapts to the received data and actions, there is no need for any direct specification by the developer. Its structure is dynamically generated by the structure of the AST, which itself is defined by the user. This means that the user still has full control over the generation of the webapp. At the other end, the ROS Application with actuator and sensor nodes, needs to be manually defined in its entirety. While this leaves still a big gap to a fully generated system, the ROS nodes are too complex and flexible, some even needing entire URDF or SDF definitions. This could be a topic for further future research.

The part that needs generating is going to be the runtime model application. With JastAdd generating code from ".relast", ".jadd" and ".jrag" files and RAGConnect inserting the connection code from ".connect" files no further parsers are required for generation. The main structure will be defined with a RAG in the ".relast" file. With this JastAdd will generate the main class files, missing any functionality. Through ".jadd" and ".jrag" files new aspects can be added that introduce new methods (".jadd" files) or attributes (".jrag" files). The

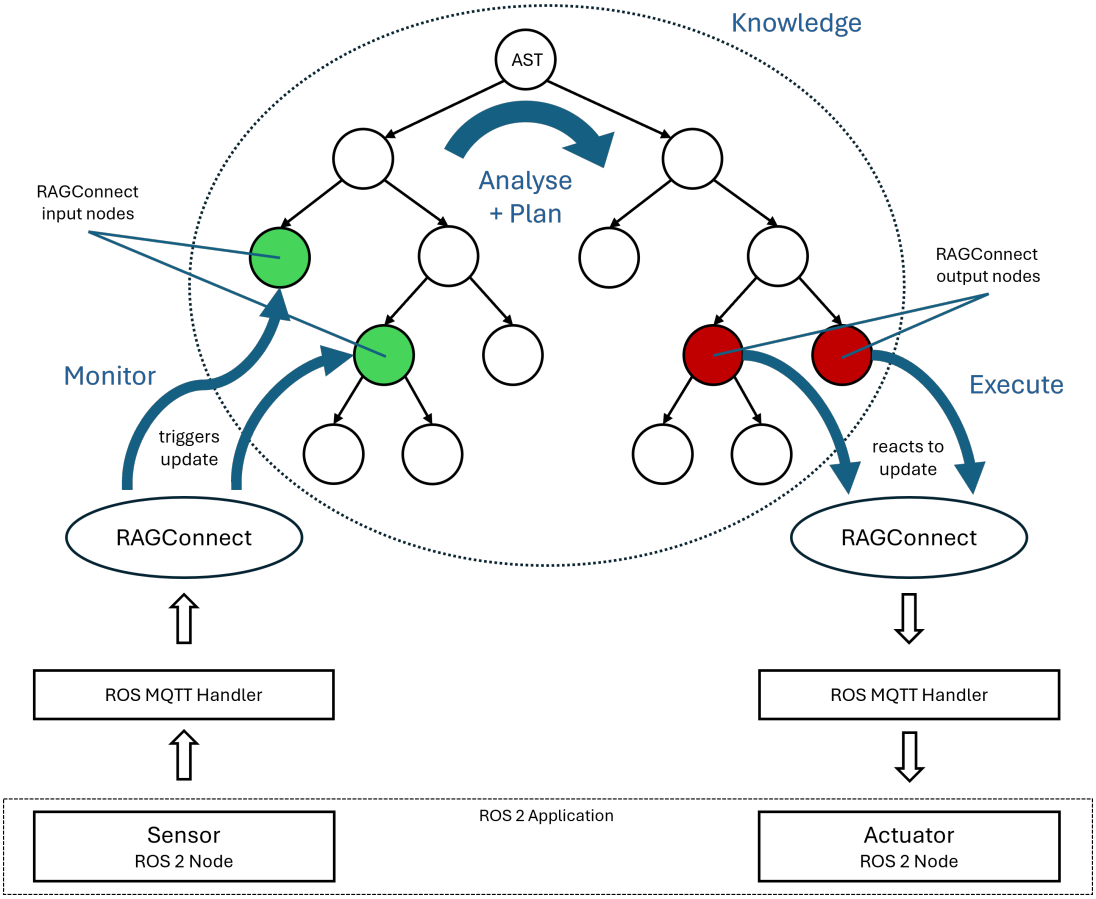

Figure 4.2: MAPE-K Loop

".connect" files will add the methods for defining connections between specific attributes of AST nodes and MQTT topics. Also it will allow for defining a mapping for converting the different data types between the runtime model and ROS.

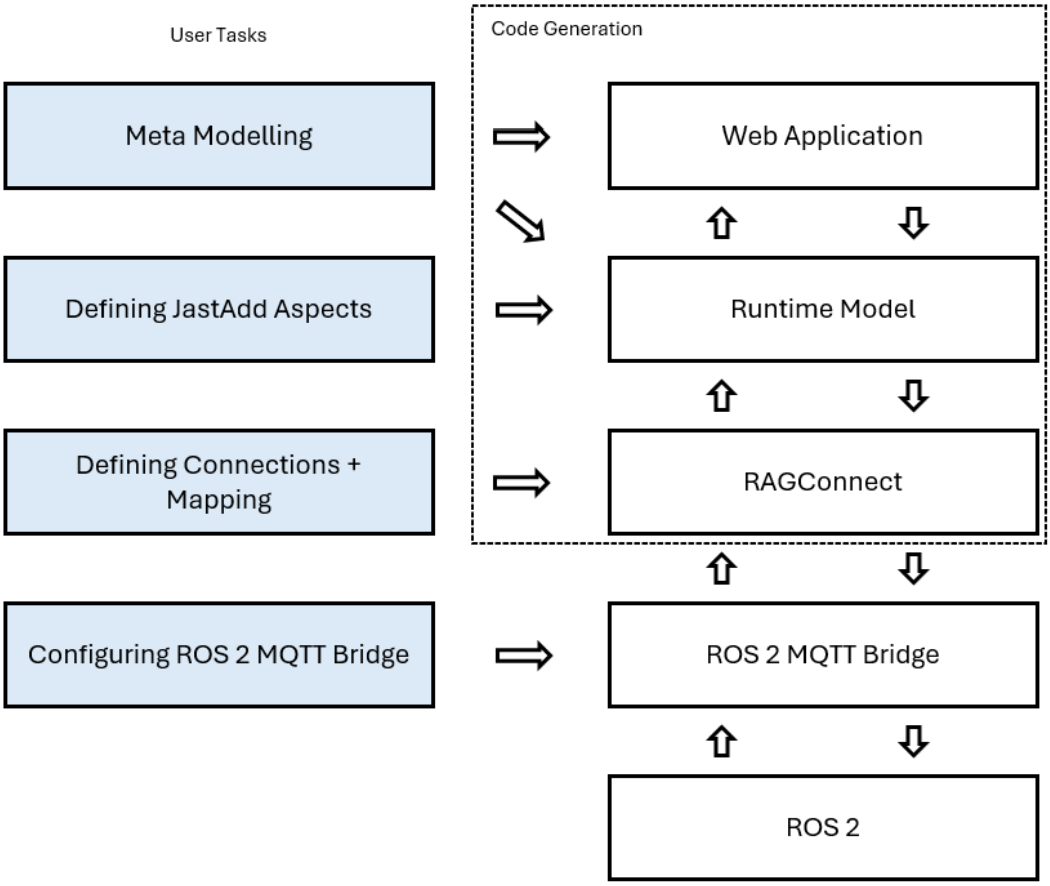

Figure 4.3: User tasks and generation of the component

5 Implementation

In this section, the fundamental considerations regarding the concept and structure of the system are introduced. First, the environment, where the implementation took place, is explained. After that the user interface and structure of the web application are illuminated. Following that the runtime model application is laid out in multiple sections, where all aspects of implementing an own application with the proposed framework are shown. Finally the required configuration of the ROS 2 MQTT Client is explained.

5.1 Environment

The basis of the implementation is the system developed by Sebastian Götz called "RuMROS" which stands for "Runtime Model driven ROS". It includes the component LaRAAB developed by Adrian Scholze presented in Chapter 3.2 that was replaced by the proposed solution.

RuMROS comes with a basic scenario of robots, areas and states. Each robot has a position and a state. In the state "driving" it is moving and in the state "waiting" it sits idle. A robot can also have a target area that he will drive to, provided he has the state "driving". While LaRAAB only supported a single robot, this solution targets to support multiple robots.

RuMROS also provides a gazebo simulated 3D environment that allows for full simulation of the scenario without realizing the robots in a physical form. A screenshot of the environment simulated with gazebo harmonic is shown in Figure 5.1.

Later, in the evaluation of this approach, the extensibility will be demonstrated by evolving this scenario. This will be done by adding flying drones as new entities next to the robots.

5.2 Web application

The web app consists of a python flask application that hosts the web page and acts as an proxy between the web page and the runtime model application. The main transfor-

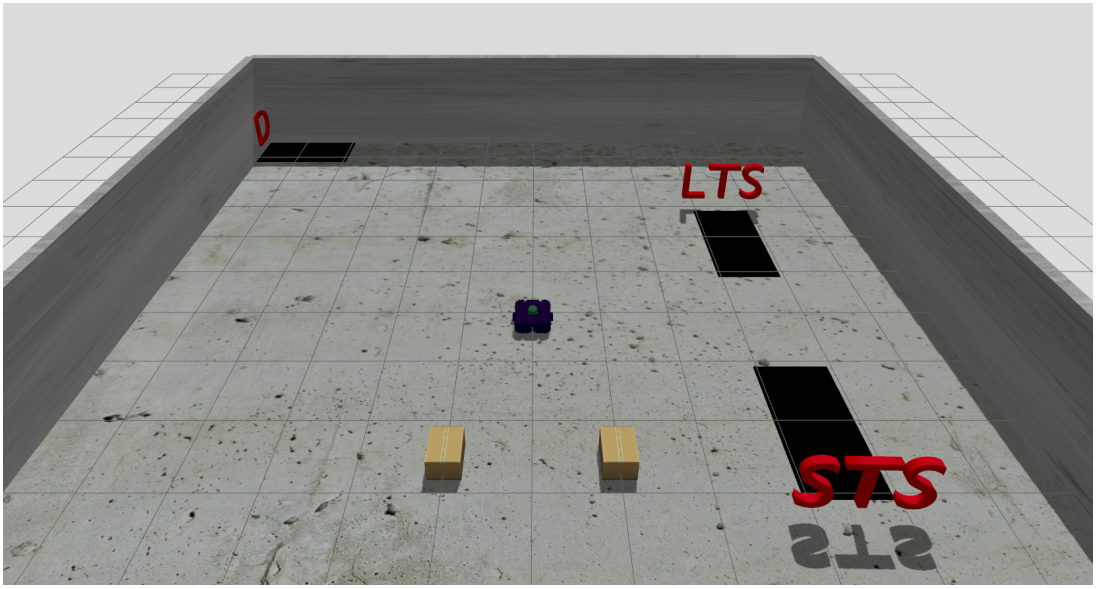

Figure 5.1: Environment of the RuMROS scenario

mations and generation of the dashboard is done by the javascript embedded inside the hosted HTML.

After receiving the first connection with a web client the flask application will try to connect to the runtime model application and open with a hello message. The runtime model application will then answer and follow up every every second through the web sockets with the current AST serialized as a JSON. The web app takes the received JSON and first separates the required child "actions" as this is where the available predefined actions are located (The definition of actions is explained in Chapter 5.3.3). Afterwards it processes the remaining JSON putting replacing references with actual values and grouping the JSON into multiple tables. Each class of children of the root node (excluding actions) will receive a separate table. Any further encapsulation will be dissolved and shown in the same table. So if "Robot" is a child of the root node it will receive its own table, but the containing node "Position" (with attributes x,y,z) will be restructured and displayed in the "Robot" table. Because of that the header of the tables can have multiple layers to show the original structure of these attributes (Figure 5.2). Following that, the actions are processed and for each action a panel is created. After one second the next JSON is received and the process starts again. While the data tables are cleared and overwritten, the action panels are only updated to allow the user to input text or select values without the process being overwritten every second, which would make user interaction impossible. A screenshot of a dynamically generated web app layout is shown in Figure 5.2.

5.3 Runtime model application

Multiple components make up the runtime model application. First of all there is the hard-coded main class that contains the web controller and computes the initial runtime model. Then there are some data classes for the action result or the connections via RAGConnect.

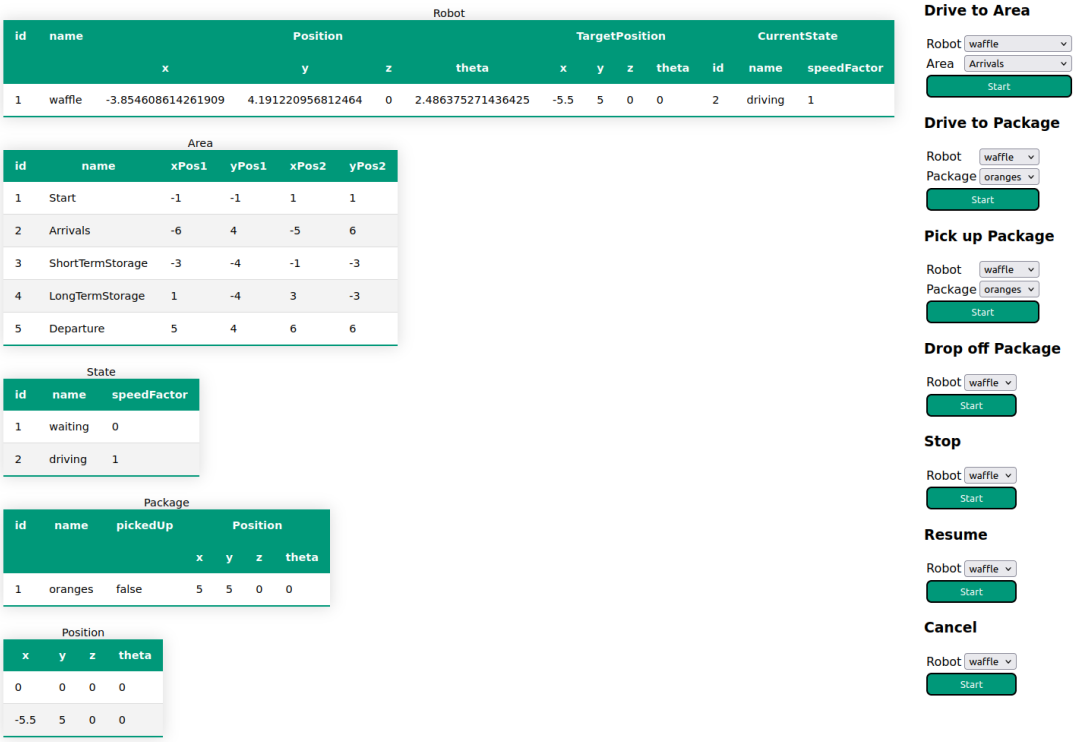

Figure 5.2: Screenshot of the web app in a simple scenario

5.3.1 Meta modeling

Defining the DSL for a system is handled in the Model.relast file which comes with some predefined elements. The RAG stripped down to the required attributes is shown in Listing 5.1. These required attributes have to be kept so actions are correctly included in the packages received by the web app. When the user is allowed to add as many basic attributes, composite attributes or reference attributes to the root node "Model". Everything that is included in this structure will be displayed by the web app with each of the immediate children of the root node being displayed in a separate table.

```
1  Model ::=
2      Action*;
3  Action ::=
4      <id:String>
5      <label:String>
6      Input*
7      Result;
8  Input ::=
9      <id:String>
10     <label:String>
11     <type:String>;
12 Result ::=
13     <type:ActionResultType>
14     <message:String>
15     <displayMillis:int>;
```

Listing 5.1: RAG defining the model structure with required attributes

A RAG definition for the running example is shown in Listing 5.2. The required root attribute "Model" has the attributes "Robot*", "Area*", "State*" and "Position*" in addition to the mandatory "Action*" attribute. The "*" trailing each attribute name allows each model to have multiple instances of these attributes. Following that are the 3 mandatory attribute definitions of "Action", "Input" and "Result". These need to be kept the same as in Listing 5.1. Then "Robot" and its relational attributes are defined. After that "Position" and "Velocity2D", that were used before in the definition of "Robot", are defined. Finally the specifications for the other 2 entities "Area" and "State" are set.

```
1
2  Model ::=
3      Action*
4      Robot*
5      Area*
6      State*
7      Position*;
8  Action [...];
9  Input [...];
10 Result [...];
11 Robot ::=
12     <id:int>
13     <name:String>
14     Position
15     /Velocity2D/;
16 rel Robot.CurrentState -> State;
17 rel Robot.TargetPosition? -> Position;
18 Position ::=
19     <x:Double>
20     <y:Double>
21     <z:Double>
22     <theta:Double>;
23 Velocity2D ::=
24      <speed:Double>
25      <rotationSpeed:Double>;
26 Area ::=
27     <id:int>
28     <name:String>
29     <xPos1:Double>
30     <yPos1:Double>
31     <xPos2:Double>
32     <yPos2:Double>;
33 State ::=
34     <id:int>
35     <name:String>
36     <speedFactor:Double>
37     <angleTolerance:Double>;
```

Listing 5.2: RAG defining the model structure for the running example

5.3.2 Implementing application logic

The application logic is defined with classic JastAdd attributes. The evaluation of these synthesized or inherited attributes can solve complex calculations. Almost every classic Java function can be converted into either a synthesized or an inherited attribute. In the running example 5 new aspects were added. These were "RobotImpl.jrag", "AreaImpl.jrag", "PositionImpl.jrag", "RobotInArea.jrag" and "GetState.jrag". Each implements a different aspect.

This part will take the most time like in other development approaches, but the extend may vary.

In the running example every attribute more or less supports the calculated attribute "Velocity2D" of robot. Every time its dependencies change, so every time the current state, target position or position of the robot change, this attribute is re-evaluated and because this attribute is configured as a "send" attribute in RAGConnect, a new message is sent via MQTT.

5.3.3 Initialization and action definition

To allow for an unrestricted but still easy initialization a synthesized attribute "init" was determined in the model class. This attribute is computed by the main class once when the application is launched and the result is used as the initial runtime model. The function for this synthesized attribute can be defined by the user via an aspect. An example for this is shown in Listing 5.3. This function needs to return the model that is initialized in the body.

```
aspect Init {
    syn Model Model.init() {
        // Initialisation
        Area a = new Area(1, "Start", -1.0, -1.0, 1.0, 1.0);
        Area b = new Area(2, "Arrivals", -6.0, 4.0, -5.0, 6.0);
        Area c = new Area(3, "ShortTermStorage", -3.0, -4.0,
            -1.0, -3.0);
        Area d = new Area(4, "LongTermStorage", 1.0, -4.0, 3.0,
            -3.0);
        Area e = new Area(5, "Departure", 5.0, 4.0, 6.0, 6.0);
        this.addArea(a);
        this.addArea(b);
        this.addArea(c);
        this.addArea(d);
        this.addArea(e);

        State waiting = new State(1, "waiting", 0.0);
        State driving = new State(2, "driving", 1.0);
        this.addState(waiting);
        this.addState(driving);

        Position startPosition1 = new Position(0.0, 1.0, 0.0,
            0.0);

        Robot robot1 = new Robot(1, "waffle", new
            RobotPositionReference(startPosition1));
        robot1.setCurrentState(stateWaiting);
        this.addRobot(robot1);


        //Actions
        [...]

        return this;
    }
}
```

Listing 5.3: JastAdd aspect for initializing the runtime model

The definition of actions is also done in this init function as the actions are part of the model. Listing 5.4 provides an example for this. First an input object is created for each input field. It is necessary to set an id (identifier in the backend), a Label (visible in the web app) and a type. The type needs to correspond to the full class path of the generated class of an attribute in the RAG or be a primitive like "java.lang.String", "java.lang.Integer" or "java.lang.Double". When a class of the AST is specified a selection box with an option for each instance of this class in the AST will be shown in the webapp, allowing the user to select the node it wants to run the action on. Otherwise if a primitive is specified, a text field, that allows the user to enter any text, only integer numbers or numbers with a decimal, is added to the form of this action. Following that, these inputs are combined in a JastAddList. Following that the actual action gets defined. It requires 4 parameters. First is an id, that will also be the name of the callback function that will be called on the model when the user runs the action. Second is again a label that is displayed in the webapp and the third parameter is the defined JastAddList of inputs. The fourth parameter is the starting result which will most certainly always be "null".

```
1  Input robotInput = new Input("robot", "Robot",
2                               "de.tudresden.inf.st.rumros.
                                   runtimemodel.Robot");
3  Input areaInput = new Input("area", "Area",
4                              "de.tudresden.inf.st.rumros.
                                  runtimemodel.Area");
5
6  JastAddList<Input> driveToAreaInputs = new JastAddList<>();
7  driveToAreaInputs.add(robotInput);
8  driveToAreaInputs.add(areaInput);
9
10 Action driveToArea = new Action("driveToArea", "Drive to Area",
11                                 driveToAreaInputs, null);
12 this.addAction(driveToArea);
```

Listing 5.4: JastAdd aspect for defining the actions

5.3.4 Defining the call functions for the actions

After the actions where defined, callback functions, that are executed when a user runs an action in the webapp, need to be implemented. This is done by adding a new ".jadd" aspect and adding method to the "Action" class (example shown in Listing 5.5). The name of the method has to be exactly the id of the action defined in the init aspect. This method has to have all the parameters defined as inputs in the init aspect in the same order with an additional parameter "model" at the first position. The "model" parameter is very useful when having to access other nodes in the AST that are not part of the inputs. The example shown in Listing 5.5 depicts such an use case.

The function interacts with the AST in some way, setting new goals, tweaking parameters, etc. Shown Listing 5.5 is a transition into a new state and the setting of a new target position.

Each action callback function needs to return an Result object. This Result will be shown to the user in the webapp, notifying the user if the execution of the action was successful or some problems were encountered. For example one can imagine a non successful result when an area can't be reached. In such use cases the result can be set dynamically. The different ActionResultTypes are: "EMPTY", "SUCCESS", "ERROR" and "TECHNICAL_ERROR". With "TECHNICAL_ERROR" being reserved for problems found by the framework, like the callback function not being defined for this action. Additionally a text message and a display time in milliseconds can be set.

With this setup only minor restrictions are made to the extend of the actions. With the "model" parameter full control over the AST is secured. Even user input in text fields can be evaluated and taken into account for goal setting.

```
1  aspect Actions {
2
3      Result Action.driveToArea(Model model, Robot robot, Area area
           ) {
4          robot.setCurrentState(robot.getStateWithName("driving"));
5          robot.setTargetPosition(model.getCenterPosition(area));
6
7          Result result = new Result(ActionResultType.SUCCESS, "
               Driving␣to␣area␣'" + area.getname() + "'", 2000);
8          return result;
9      }
10 }
```

Listing 5.5: JastAdd aspect for defining the action callback functions

5.3.5 Connecting the runtime models attributes with RAGConnect

Closing the gap between the runtime model attributes and MQTT is done via RAGConnect. First the ".connect" file is defined like specified in [14]. The configuration for the running example is shown in Listing 5.6 The hard part is the mapping because ROS sends and expects the data in the MQTT topics to be a serialized as a byte stream and the runtime model has the data as Java objects. This transformation can't be done native in Java in a clean way so an alternative had to be introduced. A helper class ROS2SerializeUtils was specifically written for this purpose and might has to be extended. ROS2SerializeUtils calls a function written in C++ that uses the ROS 2 packages to serialize and deserialize the Java objects. As shared data format between Java and C++ JSON was chosen. Unfortunately this means when introducing a new ROS 2 message type two new functions for serializing / deserializing need to be added to the C++ helper class. All the existing C++ code is included in the GitLab repository. After this the mapping in the ".connect" can look like shown in Listing 5.6.

```
receive Robot.Position using OdomToPosition;
send Robot.Velocity2D using Velocity2DToTwist;

OdomToPosition maps byte[] b to Position {:
    com.fasterxml.jackson.databind.ObjectMapper objectMapper
        = new com.fasterxml.jackson.databind.ObjectMapper();

    String json = ROS2SerializeUtils.toOdometryJson(b);

    com.fasterxml.jackson.databind.JsonNode jsonNode
        = objectMapper.readTree(json);

    double xPos = jsonNode.get("position").get("x").asDouble();
    double yPos = jsonNode.get("position").get("y").asDouble();
    double zPos = jsonNode.get("position").get("z").asDouble();

    double qx = jsonNode.get("orientation").get("x").asDouble();
    double qy = jsonNode.get("orientation").get("y").asDouble();
    double qz = jsonNode.get("orientation").get("z").asDouble();
    double qw = jsonNode.get("orientation").get("w").asDouble();

    double yaw = Math.atan2(2.0 * (qy * qz + qw * qx),
                            qw * qw - qx * qx - qy * qy + qz * qz
                               );
    double pitch = Math.asin(-2.0 * (qx * qz - qw * qy));
    double roll = Math.atan2(2.0 * (qx * qy + qw * qz),
                             qw * qw + qx * qx - qy * qy - qz *
                                qz);

    double theta = roll;

    return new Position(xPos, yPos, zPos, theta);
:}

/* Velocity2DToTwist mapping */
[...]
```

Listing 5.6: Example of a RAGConnect mapping using the ROS2SerializeUtils helper class

5.3.6 Connecting the RAGConnect attributes with specific MQTT topics

At this point the only thing that is missing is the assignment of a MQTT topic to each input / output. While RAGConnect itself offers easy connectability to MQTT topics by injecting a function "connect...()" for each attribute specified in the ".connect" file. This could be done inside the function of the synthesized init attribute, but this can be exhausting when having multiple robots and non-optimal for separation of concerns reasons. While it still can be done in that way, a configuration file "mqtt-connections.json" is introduced which offers an easier way and allows for mapping all instances of the same class with one definition. For each class that needs to be connected an entry is expected. In this entry multiple inputs and outputs can be defined. The MQTT topics can be parameterized with any attribute of the class that is available via a get method. This parameterization allows for topics like "mqtt://localhost/robot_$id/odom".

Additionally a property "condition" can also be added next to "class", that allows this connection rule to only apply for specific nodes of the AST. An example for such a condition is "$Parent.name == ëxample", which would make a rule only apply if the attribute "name" of the parent node in the AST has the value "example".

A complete example is shown in Listing 5.7.

```
[
  {
    "class": "Robot",
    "inputs": [
      {
        "attribute": "Position",
        "topic": "mqtt://localhost/robot_${id}/odom"
      }
    ],
    "outputs": [
      {
        "attribute": "Velocity",
        "topic": "mqtt://localhost/robot_${id}/cmd_vel"
      }
    ]
  }
]
```

Listing 5.7: Example of the introduced configuration file for connecting the RAGConnect attributes to MQTT topics

5.4 ROS 2 MQTT client

With the runtime model application being fully functional the gap between it and ROS needs to be closed. As mentioned before, this is done via the mqtt_client package. A separate ROS 2 node is launched that subscribes to a configured list of MQTT topics and forwards received messages to a configured ROS topic. This process also works the same in the other direction from ROS to MQTT. The mqtt_client node can be launched with some configuration file as a parameter. The configuration used for the running example is shown in Listing 5.8

```
/**/*:
  ros__parameters:
    broker:
      host: localhost
      port: 1883
    bridge:
      ros2mqtt:
        ros_topics:
          - /robot_1/odom
        /robot_1/odom:
          mqtt_topic: robot_1/odom

      mqtt2ros:
        mqtt_topics:
          - robot_1/cmd_vel
        robot_1/cmd_vel:
          ros_topic: /robot_1/cmd_vel
          ros_type: geometry_msgs/msg/Twist
```

Listing 5.8: ROS 2 MQTT client configuration file

6 Evaluation

This section evaluates the outcomes of the study, using the implementation of the sample scenario detailed in Chapter 5.1 as a case study. The scenario involves robots moving between predefined areas based on target inputs from a web application. The analysis focuses on the modularity, scalability and extensibility of the framework.

6.1 Modularity

The framework is structured as four distinct applications that completely independently and don't even need to be launched in a specific order. These applications are connected via ROS messages, MQTT messages and UDP messages containing JSON. With this design a high degree of modularity is achieved which is essential for flexible component substitution.

6.2 Scalability

When assessing scalability a current limitation is the maximum package size of UDP messages. As splitting the data of the serialized runtime model JSON into multiple UDP packages and merging them again in the webapp is not implemented yet, as the framework currently is a practical proof of concept prototype and not a framework meant for production software.

A test was conducted by running the sample system with 2, 10 and then 100 robots and comparing reaction times. The reaction time is the time from the pressing of the action button to the message being received by the ROS 2 node. The reaction times, shown in Figure 6.1, show that there is no significant difference in the reaction times when comparing systems with 2, 10 and 100 robots. The amount of robots has no influence on the reaction time and that other variables like other programs and background services can lead to a system of 100 robots reacting faster than a system of 2 robots.

Another test was conducted, comparing the interval between the triggered MAPE-K loop

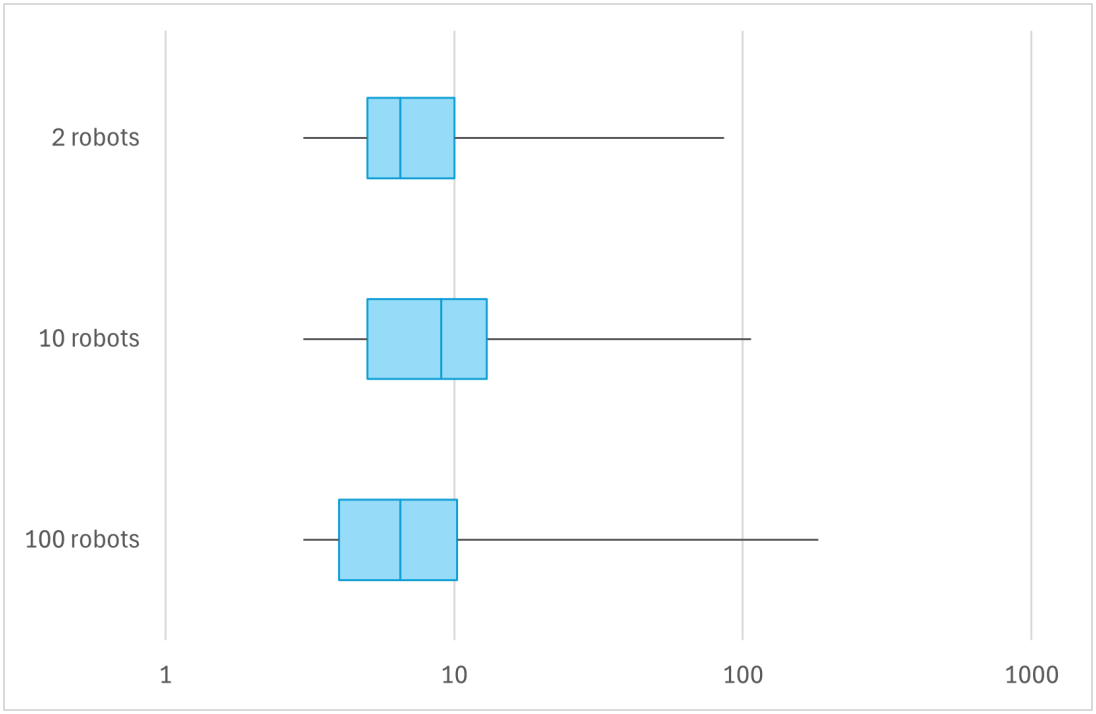

Figure 6.1: Reaction time of the system (time from the pressing of the action button to the message being received by the ROS 2 node)

iterations. Instead of the calculating position with odometry like before, the system calculated the position with slam_toolbox a ROS 2 package for Simultaneous localization and mapping (SLAM). This was done to get more precise positions but requires more computation power, which was also beneficial for this test to better show the influence of the count of robots. Shown in Figure 6.2, the number of robots does have an influence on the system, as the movement of more robots has to be simulated by gazebo and also more positions are calculated simultaneously with slam_toolbox.

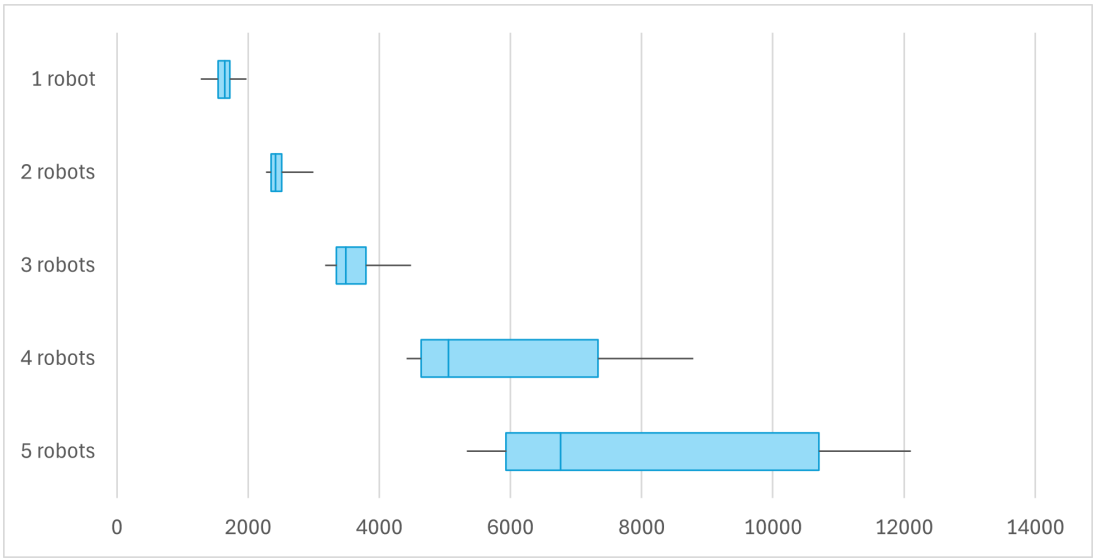

Figure 6.2: Interval between (triggered) MAPE-K loop iterations in ms

6.3 Extensibility

The extensibility is tested by adding drones to the original scenario. The first thing is to add a new child of model "Drone" to the RAG in the ".relast" file. Because the support of multiple drones is targeted, "Drone*" is added to the model. Then it is defined what a drone is. The structure from the robot definition can be copied and the drone gets an id, name, position and velocity. The positionobject originally defined for robots can also be reused . However what can't be reused is the velocity definition. A new Velocity3D class is neccesary here, because a new vertical velocity is required for the drones lift. Equivalent to robots, drones also need two reference attributes CurrentState and TargetPosition where the existing definitions of State and Position can be completely reused. The additions are shown in Listing 6.1.

```
1  Model ::=
2      [...]
3      Drone*;
4
5  Drone ::=
6      <id:int>
7      <name:String>
8      Position
9      /Velocity3D/;
10
11 rel Drone.CurrentState -> State;
12 rel Drone.TargetPosition? -> Position;
13
14 Velocity3D ::=
15     <horizontalSpeed:Double>
16     <verticalSpeed:Double>
17     <rotationSpeed:Double>;
```

Listing 6.1: Extensions to the RAG adding drones

With the DSL being extended, the next step is to extend the application logic, specifically add an Implementation for the logic of the drone. For this a new aspect "DroneImpl.jrag", shown in Listing 6.2, is added. It contains the implementation of the "Velocity3D" attribute that was just defined as part of the drone, but as a computed non-terminal. The other attributes are helper attributes to make the "Velocity3D" calculation easier to follow. The implementation is shortened in Listing 6.2 for readability reasons, but the full code can be viewed in the GitLab repository. The "GetState" aspect is also modified to include a function to get all possible states from every drone instance.

```
aspect DroneImpl {
    syn boolean Drone.getTargetReached() = this.
       getHorizontalDistanceToTarget() <= 0.2 && Math.abs(this.
       getVerticalError()) <= 0.1;

    syn Velocity3D Drone.getVelocity3D() {
        /* Calculate Velocity2D with CurrentState, TargetPosition
            and Position */
        [...]
        return velocity;
    }

    syn double Drone.getHorizontalDistanceToTarget() = Math.abs(
        Math.sqrt(
            Math.pow(this.getTargetPosition().getPosition().getx
                () - this.getDronePositionReference().getPosition
                ().getx(), 2) +
            Math.pow(this.getTargetPosition().getPosition().gety
                () - this.getDronePositionReference().getPosition
                ().gety(), 2)
        )
    );

    syn double Drone.getVerticalError() = this.getTargetPosition
       ().getPosition().getz() - this.getDronePositionReference
       ().getPosition().getz();

    syn double Drone.getHeadingError() {
        /* Calculate heading error with TargetPosition and
           Position */
        [...]
        return headingError;
    }

}
```

Listing 6.2: Aspect for the implementation of the drones logic

Then the next thing to do is to initialize the runtime model with a single or multiple drones. For simplicity a single drone will be defined and added. As described in the Chapter 5.3.3 the aspect that adds the Model.init() function needs to be extended. In the same function an action for flying the drone to a zone is defined. This all can be done in very few lines as shown in Listing 6.3.

```
aspect Init {
    syn Model Model.init() {
        [...]
        Position startPosition3
            = new Position(0.0, 0.0, 0.0, 0.0);
        Drone drone1
            = new Drone(1, "drone1", startPosition3);
        drone1.setCurrentState(waiting);
        this.addDrone(drone1);
        [...]
        Input droneInput = new Input("drone", "Drone", "de.
           tudresden.inf.st.rumros.runtimemodel.Drone");
        JastAddList<Input> areaDroneInputs = new JastAddList<>();
        areaDroneInputs.add(droneInput);
        areaDroneInputs.add(areaInput);

        Action flyToArea = new Action("flyToArea", "Fly to Area",
            areaDroneInputs, null);
        this.addAction(flyToArea);

        return this;
    }
}
```

Listing 6.3: Extensions to the Init ascpect adding drones

Next a call function for the action that was just defined will be added to the actions aspect, which is very similar to the call function of the robots "driveToArea" action. The extended code is shown in Listing 6.4.

```
aspect Actions {
    [...]
    Result Action.flyToArea(Model model, Drone drone, Area area)
       {
        drone.setCurrentState(robot.getStateWithName("driving"));
        drone.setTargetPosition(model.getCenterPosition(area));

        Result result = new Result(ActionResultType.SUCCESS,
            "Flying to area '" + area.getname() + "'", 2000);
        return result;
    }
}
```

Listing 6.4: Extensions to the Actions ascpect adding drones

After this is done, the connection to ROS 2 via RAGConnect and MQTT can be made. The position needs to be declared as received and the velocity as sent. A mapping is needed for both connections to convert the byte stream ROS sends and expects which RAGConnect will forward via acMQTT. The mapping is straightforward the most time consuming part would be extending the ROS2SerializeUtils and the loaded C++ lib when a new ROS 2 message type is connected due to there currently being no serialization library for arbitrary ROS 2 messages to JSON. In this case all serialization methods already exist as they were also

needed for the robot implementation. The final extension to the RAGConnect configuration is shown in Listing 6.5.

```
[...]
receive Drone.Position using OdomToPosition;
send Drone.Velocity3D using Velocity3DToTwist;

OdomToPosition maps byte[] b to Position {:
    com.fasterxml.jackson.databind.ObjectMapper objectMapper =
        new com.fasterxml.jackson.databind.ObjectMapper();
    String json = ROS2SerializeUtils.toOdometryJson(b);
    com.fasterxml.jackson.databind.JsonNode jsonNode =
        objectMapper.readTree(json);
    double xPos = jsonNode.get("position").get("x").asDouble();
    double yPos = jsonNode.get("position").get("y").asDouble();
    double zPos = jsonNode.get("position").get("z").asDouble();
    double qx = jsonNode.get("orientation").get("x").asDouble();
    double qy = jsonNode.get("orientation").get("y").asDouble();
    double qz = jsonNode.get("orientation").get("z").asDouble();
    double qw = jsonNode.get("orientation").get("w").asDouble();
    double yaw = Math.atan2(2.0 * (qy * qz + qw * qx), qw * qw -
        qx * qx - qy * qy + qz * qz);
    double pitch = Math.asin(-2.0 * (qx * qz - qw * qy));
    double roll = Math.atan2(2.0 * (qx * qy + qw * qz), qw * qw +
         qx * qx - qy * qy - qz * qz);
    double theta = roll;
    return new Position(xPos, yPos, zPos, theta);
:}
Velocity3DToTwist maps Velocity3D v to byte[] {:
    com.fasterxml.jackson.databind.ObjectMapper objectMapper =
        new com.fasterxml.jackson.databind.ObjectMapper();
    double horizontalSpeed = v.gethorizontalSpeed();
    double verticalSpeed = v.getverticalSpeed();
    double rotationSpeed = v.getrotationSpeed();
    com.fasterxml.jackson.databind.node.ObjectNode linearJson =
        objectMapper.createObjectNode();
    linearJson.put("x", horizontalSpeed);
    linearJson.put("y", 0.0);
    linearJson.put("z", verticalSpeed);
    com.fasterxml.jackson.databind.node.ObjectNode angularJson =
        objectMapper.createObjectNode();
    angularJson.put("x", 0.0);
    angularJson.put("y", 0.0);
    angularJson.put("z", rotationSpeed);
    com.fasterxml.jackson.databind.node.ObjectNode twistJson =
        objectMapper.createObjectNode();
    twistJson.set("linear", linearJson);
    twistJson.set("angular", angularJson);
    String json = objectMapper.writerWithDefaultPrettyPrinter().
        writeValueAsString(twistJson);
    return ROS2SerializeUtils.fromTwistJson(json);
:}
```

Listing 6.5: Extensions to the RAGConnect configuration adding drones

Currently the attributes are registered by RAGConnect but no specific MQTT topics are assigned to each attribute. This needs to be done in the mqtt-connections.json configuration file as described in Chapter 5.3.6. This was done exactly the same way for robots, so the parameters only need to be copied and renamed as shown in Listing 6.6.

```
[
  [...],
  {
    "class": "Drone",
    "inputs": [
      {
        "attribute": "Position",
        "topic": "mqtt://localhost/drone_${id}/odom"
      }
    ],
    "outputs": [
      {
        "attribute": "Velocity3D",
        "topic": "mqtt://localhost/drone_${id}/cmd_vel"
      }
    ]
  }
]
```

Listing 6.6: Extension of the ragconnect mqtt configuration file

To forward the new MQTT topics to ROS 2 topics the configuration file for the ROS 2 MQTT client needs to be changed too. This step can become obsolete if the topics always have the same path in MQTT and ROS 2 and a generator is build that takes in the current mqtt-connections.json file to generate the configuration file for the client. In the current state of the framework this step has to be done manually and involves the additions shown in Listing 6.7.

```
/**/*:
  ros__parameters:
    broker:
      host: localhost
      port: 1883
    bridge:
      ros2mqtt:
        ros_topics:
          - /drone_1/odom
            [...]
        /drone_1/odom:
          mqtt_topic: drone_1/odom

      mqtt2ros:
        mqtt_topics:
          - drone_1/cmd_vel
            [...]
        drone_1/cmd_vel:
          ros_topic: /drone_1/cmd_vel
          ros_type: geometry_msgs/msg/Twist
```

Listing 6.7: Extension of the ROS 2 MQTT client configuration file

The last missing part is the Velocity3D attribute of the drone, which doesn't yet have a computation function. A new aspect DroneImpl.jrag is added, that defines the getVelocity3D() method and some helper methods. The focus here is on the getVelocity3D() method shown in Listing 6.8 as the helper methods are self explanatory. A non zero is computed when the drone has a target and hasn't reached it yet. If it is right above the target, with some tolerance, it will descend. If it isn't yet at the target and hasn't got enough altitude it will ascend further. If all this isn't the case it will either move in the direction of the target or rotate in the direction of the target if its current heading doesn't point at the target.

```
syn Velocity3D Drone.getVelocity3D() {
    if (this.getTargetPosition() == null || this.getTargetReached
       ()) {
        return new Velocity3D(0.0, 0.0, 0.0);
    }

    double horizontalSpeed = 0.0;
    double verticalSpeed = 0.0;
    double rotationSpeed = 0.0;

    if(this.getHorizontalDistanceToTarget() <= 0.2) {
        return new Velocity3D(0.0, -1.0, 0.0);
    }

    if(this.getPosition().getz() < this.getTargetPosition().getz
       () + 1) {
        return new Velocity3D(0.0, 1.0, 0.0);
    }

    final double ANGLE_TOLERANCE = 0.3;
    final double MAX_SPEED = 1.0;
    final double GAIN = 0.2;
    final double ANGLE_GAIN = 0.5;
    final double headingError = this.getHeadingError();
    final double distanceToTarget = this.
       getHorizontalDistanceToTarget();

    if (Math.abs(headingError) > ANGLE_TOLERANCE) {
        rotationSpeed = ANGLE_GAIN * headingError * this.
           getCurrentState().getspeedFactor();
        if (Math.abs(rotationSpeed) > MAX_SPEED) {
            if(rotationSpeed < 0) {
                rotationSpeed = MAX_SPEED * -1.0;
            } else {
                rotationSpeed = MAX_SPEED;
            }
        }
    } else {
        horizontalSpeed = GAIN * distanceToTarget * this.
           getCurrentState().getspeedFactor();
        horizontalSpeed = horizontalSpeed < MAX_SPEED ?
           horizontalSpeed : MAX_SPEED;
    }

    return new Velocity3D(horizontalSpeed, verticalSpeed,
       rotationSpeed);
}
```

Listing 6.8: New aspect for implementing the computation of the drones required velocity

The end result is a new extended system with robots and drones. The addition of a completely new entity, drones, next to robots would normally require many modifications on both ends on the system. With the web app being fully dynamic all changes in the DSL are therefore automatically visible in the web app, without a separate modification. This leaves more time for the developer to focus on the actual functionality of the drone.

Often the changes are even smaller, when only updating or adding internal parts of the system. A change in the movement of the robot can be done in a single change of one of its aspects.

7 Conclusion

This thesis presented an approach to integrating runtime models with ROS-based robotic applications using RAGConnect. The proposed solution enhances modularity, scalability and extensibility while reducing the need for manually adapting code when system requirements evolve. By leveraging JastAdd for defining the DSL as a RAG and RAGConnect for establishing dynamic causal connections, this approach enables automated and incremental adaptation of robotic applications in real time.

A key achievement of this work is the introduction of a fully dynamic web-based user interface that adapts to changes in the runtime model. Unlike traditional approaches that require manually updating the user interface when extending the system, this implementation allows new elements and actions to be reflected automatically. This not only simplifies the development process but also makes the system more flexible and adaptable to various use cases.

The evaluation demonstrated that the approach maintains high responsiveness, even when scaling the system to include multiple robots. The reaction times and iteration speeds of the MAPE-K loop showed that increasing the number of entities does not significantly degrade system performance. Furthermore, the extensibility of the approach was validated by successfully introducing drones as a new entity type with minimal changes to the core system. This highlighted the benefits of the modular architecture and the clear separation of concerns achieved through RAGConnect and JastAdd.

Despite these successes, some limitations remain. The dependency on ROS2SerializeUtils for message serialization introduces a maintenance challenge when working with new ROS message types. Future research should explore automated serialization techniques to further reduce the need for manual adjustments. Additionally the webapp could offer more customizability which can be necessary for more complex scenarios.

Overall, this work contributes to the field of model-driven engineering for robotic applications by demonstrating a practical and scalable framework for runtime adaptation. The findings indicate that leveraging runtime models with incremental evaluation and causal connections can significantly improve the flexibility and maintainability of ROS-based systems. Future research directions include expanding the framework to support more complex multi-robot scenarios and further automating the model generation process to streamline integration with additional robotic platforms.

List of Acronyms

MDD Model-Driven Development

ROS Robot Operating System

AST Abstract Syntax Tree

AG Attribute Grammer

RAG Reference Attribute Grammer

MQTT Message Queuing Telemetry Transport

IoT Internet of Things

DSL Domain Specific Language

MAPE-K Monitor-Analyze-Plan-Execute while having a shared Knowledge

SLAM Simultaneous localization and mapping

Bibliography

[1] Dirk Thomas et al. *State of ROS 2 (Demos and the technology behind)*. Tech. rep. ROSCon 2015, Oct. 2015. URL: https://roscon.ros.org/2015/presentations/state-of-ros2.pdf.

[2] Nelly Bencomo et al., eds. *Models@run.time: Foundations, Applications, and Roadmaps*. 1st ed. Lecture Notes in Computer Science. Softcover ISBN: 978-3-319-08914-0, eBook ISBN: 978-3-319-08915-7. Cham, Switzerland: Springer Cham, 2014, pp. X, 319. ISBN: 978-3-319-08914-0. DOI: https://doi.org/10.1007/978-3-319-08915-7.

[3] Betty Cheng et al. *Using Models at Runtime to Address Assurance for Self-Adaptive Systems*. 2015. arXiv: 1505.00903 [cs.SE]. URL: https://arxiv.org/abs/1505.00903.

[4] Jiabin Dai. *Generating Causal Connections for Model-driven ROS-based Applications*. Dec. 2023.

[5] Junya Hayashi. *mqtt_bridge*. Oct. 2016. URL: https://github.com/groove-x/mqtt_bridge.

[6] Görel Hedin. "Reference Attributed Grammars". English. In: *Informatica* 24.3 (2000), pp. 301–317. ISSN: 0868-4952.

[7] Donald E. Knuth. "Semantics of context-free languages". In: *Mathematical systems theory* 2.2 (June 1968), pp. 127–145. ISSN: 1433-0490. DOI: 10.1007/BF01692511. URL: https://doi.org/10.1007/BF01692511.

[8] Steven Macenski et al. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (May 2022). ISSN: 2470-9476. DOI: 10.1126/scirobotics.abm6074. URL: http://dx.doi.org/10.1126/scirobotics.abm6074.

[9] Biswajeeban Mishra and Attila Kertesz. "The Use of MQTT in M2M and IoT Systems: A Survey". In: *IEEE Access* 8 (2020), pp. 201071–201086. DOI: 10.1109/ACCESS.2020.3035849.

[10] *mqtt_client*. June 2022. URL: https://github.com/ika-rwth-aachen/mqtt_client.

[11] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software* 3 (Jan. 2009).

[12] Nils Rottmann et al. *ROS-Mobile: An Android application for the Robot Operating System*. 2020. arXiv: 2011.02781 [cs.RO]. URL: https://arxiv.org/abs/2011.02781.

[13] Adrian Scholze. *Models@run.time in Mobilen Anwendungen für Robotik-Simulationen*. June 2023.

[14] René Schöne et al. “Incremental causal connection for self-adaptive systems based on relational reference attribute grammars”. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*. MODELS ’22. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, pp. 1–12. ISBN: 9781450394666. DOI: 10.1145/3550355.3552460. URL: https://doi.org/10.1145/3550355.3552460.

[15] Konglong Tang et al. “Design and Implementation of Push Notification System Based on the MQTT Protocol”. In: *Proceedings of 2013 International Conference on Information Science and Computer Applications*. Atlantis Press, Oct. 2013, pp. 116–119. ISBN: 978-90786-77-85-7. DOI: 10.2991/isca-13.2013.20. URL: https://doi.org/10.2991/isca-13.2013.20.

[16] *Understanding nodes*. URL: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html.