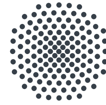

University of Stuttgart
Institute of Architecture of Application Systems (IAAS)
Chair of Service Computing

Master Thesis

Modeling and Timing Analysis of Micro-ROS Application on an Off-Road Vehicle Control Unit

Suraj Rao Bappanadu

Course of Study:	M.Sc. Information Technology Specialization Embedded Systems
Examiner:	Prof. Dr. Marco Aiello
Supervisors:	Dr. Jan Staschulat, Robert Bosch GmbH Dr. Ralph Lange, Robert Bosch GmbH Ing. Julian von Mendel, Bosch Rexroth AG
Commenced:	12.10.2021
Completed:	12.04.2022

Acknowledgement

The satisfaction and euphoria that accompany the successful completion of any work would be incomplete, without the mention of people who made it possible, whose constant guidance and encouragement crown all the efforts with success.

I would like to first thank University of Stuttgart, Robert Bosch Corporate Research, Renningen and Bosch Rexroth AG, Schwieberdingen for giving me the opportunity to pursue my thesis.

I would like to thank my supervisors, Dr. Ralph Lange , Dr. Jan Staschulat and Julian von Mendel, for their continuous valuable guidance, advice and persistent encouragement throughout the thesis work. I would like to also thank the colleagues and mentors from Robert Bosch Modeling Team, Mobile Electronics Team, Schwieberdingen and Inchron for their support and encouragement.

In particular, I would like to mention out Jan Staschulat and Ralph Lange for their constant support, feedback and motivation which helped me bring my work to a higher level. Their suggestions have not only helped me improve this thesis work but also my approach towards research work.

My sincere and grateful acknowledgement to Prof. Dr. Marco Aiello for examining my thesis work and reviewing all my steps during this process.

Last but not the least, I would like to thank my family and friends for their constant moral support and encouragement for the entire duration of my Master's studies.

Abstract

ROS is known to be the most popular middleware for the development of software in modern day robots. It's next version, ROS 2 is highly modular and offers flexibility by supporting on microprocessors running desktop operating systems. Micro-ROS puts the major ROS 2 features on microcontrollers, i.e., highly resource-constrained computing devices running specialized real-time operating systems. ROS 2 is also of great importance for other domains, including autonomous driving and the off-road sector. Accordingly, there is significant interest in bringing micro-ROS to typical automotive control units. These embedded platforms support AUTOSAR Classic OSEK-like operating system which is very different in many aspects when compared to the platforms supported by micro-ROS. Some of the aspects have already been addressed in a previous work. This thesis mainly focuses on mapping the micro-ROS execution scheme to AUTOSAR scheme and dynamic memory management of the micro-ROS stack. From the micro-ROS architecture perspective, to successfully port the stack on an AUTOSAR-based ECU, the middleware and other layers of the stack are also analysed and adapted using a standard approach to support tasks-like execution model instead of threads-like execution model. Additionally, the support for standard CAN protocol based on custom transport configuration with the hardware CAN on the BODAS ECU is introduced. Model-based development methods have proven their utility in automotive industry. Therefore, we also focus on describing the timing properties of the micro-ROS stack in a model-based approach. We develop a generic model which is independent of a specific modeling language. In the next step, we realize the generic model using the widely used AMALTHEA language and analyse how well the developed model predicts the timing behavior of micro-ROS tasks. Finally, the effectiveness of the approach regarding timing and modeling is demonstrated with a micro-ROS test application first on Linux and then on the off-road vehicle control unit BODAS RC18-12/40 by Bosch Rexroth.

Contents

1 Introduction **11**
- 1.1 Problem Statement 12
- 1.2 Methodology 14
- 1.3 Document Structure 15

2 Preliminaries **17**
- 2.1 Robot Operating System 17
- 2.2 Micro-ROS 19
  - 2.2.1 Micro-ROS Architecture and Features 20
- 2.3 Scheduling in Embedded Systems 22
  - 2.3.1 Scheduling under POSIX 22
  - 2.3.2 Fixed Periodic Preemptive Scheduling Scheme 23
- 2.4 BODAS RC40 Series Controller for Off-Highway Applications 24
  - 2.4.1 Hardware Specification 25
  - 2.4.2 Software Specification 26

3 State of the Art **28**
- 3.1 Micro-ROS/ROS 2 Execution Model 28
- 3.2 Modeling and Performance Analysis in Automotive 29
  - 3.2.1 Standards for Model Development 29
  - 3.2.2 Performance Analysis Simulation Tools 30
- 3.3 Modeling of ROS Processing Chains 31
- 3.4 Specific Research Questions 32

4 Mapping Execution Management to AUTOSAR **33**
- 4.1 Micro-ROS Stack Analysis 33
  - 4.1.1 Initialization Phase 34
  - 4.1.2 Operation Phase 34
- 4.2 Initialization Phase with State-Machine Model 36
- 4.3 Operation Phase with Periodic AUTOSAR Tasks 39
- 4.4 Implementation on BODAS RC18-12/40 40

4.4.1 Additional Layers: Support on BODAS 41
4.4.2 Micro XRCE-DDS over Custom Transport 43
4.4.3 Other Layers: RCLC, RCL, RMW 48

5 Performance Modeling and Analysis 50
5.1 Generic Performance Model 50
5.2 System Modeling with AMALTHEA 55
5.2.1 AMALTHEA Data Models 56
5.3 Generic Performance Model using AMALTHEA 58
5.3.1 Hardware Model 59
5.3.2 Generic micro-ROS Model on BODAS 60
5.3.3 Application Model 61
5.4 Performance Analysis using chronSUITE 61

6 Evaluation 63
6.1 Micro-ROS Test Application 63
6.2 Modeling of micro-ROS Test Application 65
6.3 Test Experiments 66
6.3.1 Initialization Phase 66
6.3.2 Operation Phase 67

7 Conclusion 78
7.1 Summary 78
7.2 Future Work 79

Bibliography 88

List of Figures

2.1 ROS 2 architecture . . . 18
2.2 Micro-ROS vs ROS 2 architecture . . . 19
2.3 Micro XRCE-DDS client-server architecture . . . 22
2.4 Fixed periodic preemptive scheduling . . . 24
2.5 Bosch Rexroth RC40 Series Controller . . . 25
2.6 BODAS RC18-12/40 software stack . . . 26

4.1 Execution Management in ROS 2 . . . 35
4.2 Comparison of Execution Management . . . 36
4.3 Flow Chart for entity request-response handling in Client . . . 37
4.4 Execution Management mapped to AUTOSAR task scheme . . . 39
4.5 Micro-ROS stack on BODAS RC18-12/40 . . . 40
4.6 Format of standard CAN frame . . . 45
4.7 CAN data transmission workflow . . . 46
4.8 CAN data reception workflow . . . 47

5.1 Abstract overview of models . . . 53
5.2 Generic Performance Model . . . 54
5.3 Overview of AMALTHEA system model . . . 55
5.4 AMALTHEA data models . . . 56
5.5 Generic Performance Model using AMALTHEA data models . . . 59
5.6 Hardware data model . . . 60
5.7 Software data model . . . 61

6.1 Micro-ROS test application on BODAS RC18-12/40 . . . 64
6.2 Control Function . . . 64
6.3 Micro-ROS test application model in AMALTHEA . . . 66
6.4 Micro-ROS Initialization Phase . . . 67
6.5 Response Time of Control Function using Simple Model . . . 69
6.6 Response Time of Control Function using Advanced Model . . . 71
6.7 BODAS-ESP32 setup . . . 71

6.8 Distributed latency of Control Function without micro-ROS stack on ESP32 . . . 73
6.9 Distributed latency of Control Function with micro-ROS stack on ESP32 . . . 73
6.10 Specified task chain . . . 74
6.11 Timing Model in chronSUITE . . . 75
6.12 Distributed E2E latency of a task chain . . . 76
6.13 Minimum E2E latency of a task chain . . . 76
6.14 Maximum E2E latency of a task chain . . . 77

List of Tables

1.1 Micro-ROS vs AUTOSAR Classic 13

2.1 Hardware Specifications of RC18-12/40 26

3.1 Comparison of model elements covered in three standards . . . 30

4.1 API calls of micro-ROS stack 38
4.2 Payload Header value description 45

6.1 Results based on Simple Model 68
6.2 Results based on Advanced Model 70
6.3 Pin connections between BODAS and ESP32 72
6.4 Measurement on ESP32 72
6.5 Measurement of E2E latency of a task chain 77

Acronyms

API Application Program Interface.
APP4MC Application Platform Project for Multi- and Many-Core Systems.
ASW Application Software.
AUTOSAR Automotive Open System Architecture.

BODAS Bosch Rexroth Digital Application Software.
BSW Base Software.

CAN Controller Area Network.
CAN FD Controller Area Network Flexible Data-Rate.
CPU Central Processing Unit.
CUBAS Common UBK Basic Software.

DDS Data Distribution Service.

ECU Electronic Control Unit.

FIFO First In First Out.

GPIO General Purpose Input Output.

HW Hardware.

I2C Inter-Integrated Circuit.
ISO International Organization for Standardization.
ISR Interrupt Service Routine.

LIN Local Interconnect Network.

MCU Microcontroller Unit.
MTU Maximum Transmission Unit.

OEM Original Equipment Manufacturer.

OS	Operating System.
OSEK	Open Systems and their Interfaces for the Electronics in Motor Vehicles.
POSIX	Portable Operating System Interface.
QoS	Quality of Service.
RAM	Random Access Memory.
RCL	ROS Client Library.
RMW	ROS Middleware.
ROS	Robot Operating System.
RR	Round Robin.
RTOS	Real-Time Operating System.
RTPS	Real-Time Publish-Subscribe.
SOA	Service-Oriented Architecture.
SPI	Serial Peripheral Interface.
SW	Software.
TCP	Transmission Control Protocol.
UART	Universal Asynchronous Receiver Transmitter.
UDP	User Datagram Protocol.
XRCE	Extremely Resource Constraint Environment.

Chapter 1

Introduction

The immense impact that robotics has on today's world has fascinated and teleported everyone into a world of wonderment imagining the extent of modernization that is growing beyond bounds. Considering such a scenario, it is proven that in the development of robots, both software and hardware has played a vital part. From land-based mobile robots, to quadrotor helicopters, to humanoids - the robotics community has made significant progress, and there is more efficient software and reliable hardware available than ever before [1]. One of the reasons for such advancement is the software framework used in development, i.e., Robot Operating System (ROS).

ROS is the most common used middleware in the development of complex, but modular systems in a distributed computing environment [2]. Its framework is used in programming complex robots because of its four main features: (1) its middleware supporting communication between multiple programs to exchange various types of data, (2) its support for tools to debug, monitor and visualize data, (3) its capabilities for navigation, path-planning algorithms, and (4) its development supported by shared and energetic community [3]. Its next version, Robot Operating System 2 (ROS 2) primarily focused on five objectives: (1) provide the developers a software platform for carrying their research and from prototyping to production, (2) support on different operating systems, (3) support multi-robot systems involving low/poor networks, (4) support on microcontrollers, and (5) support for real-time features. It uses industry standard, Data Distribution Service (DDS) as its middleware. In principle, it could achieve the above mentioned three objectives other than support on microcontrollers and real-time features [4]. Micro-ROS, a variant of ROS 2 is mainly tailored for resource-constraint devices like microcontrollers that feature a few hundreds of kilobytes of RAM only. Its stack is highly flexible and can be used with various Portable

Operating System Interface (POSIX)-compliant real-time operating systems (RTOS).

AUTOSAR (Automotive Open System Architecture) is a de-facto standard used in the basic software development for most of the applications in the automotive industry. With the objective of providing standardized software architecture for automotive electronic control units (ECU), it is being developed in partnership with various automotive manufacturers, suppliers and service providers [5]. The software components of AUTOSAR provide a layer that helps in linking the application code and the underneath ECU hardware. This allows the user for independent software development irrespective of the chosen embedded hardware, ensuring software reusability [6].

With the standard mentioned above like AUTOSAR, followed in the basic software development, the user now has the freedom to develop the application code also for advanced functionalities. As a result, there is a constant increase in the complexity of application development. Therefore it is necessary to analyse the application before the actual implementation. Hence, it is highly beneficial to have model-based systems and performance analysis in the early design phase.

The development of advanced automation functions with popular robotics framework combining the software components with automotive proven standards is now gaining popular also in the off-highway market. Therefore, all these features and functions of ROS 2/micro-ROS will draw our attention towards supporting such a stack also on the off-road vehicle control units running AUTOSAR Classic OSEK-like operating system, so that it would speed-up the development of new algorithms and advanced functions on such embedded platforms.

1.1 Problem Statement

The analysis of micro-ROS and AUTOSAR Classic framework at the conceptual level reveals the fundamental differences in many aspects that includes component model, operating system (OS), communication, safety, execution management and memory management as described in the Table 1.1.

Mapping the *component model* from micro-ROS to AUTOSAR Classic is well supported by AUTOSAR tooling. The key challenge in mapping the two *operating systems* is the scheduling scheme. This aspect will be addressed as

	micro-ROS	**AUTOSAR Classic**
Component model	Lightweight model	Strict meta-model
OS	POSIX	OSEK
Communication	Queues	Shared memory communication
Safety	Functional safety not administered	Functionally safety administered
Execution & real-time control	Event-driven approach, for real time little mechanisms available per component only	Static schedule, fixed time slices
Memory	Dynamic memory usage	Static memory usage

Table 1.1: Micro-ROS vs AUTOSAR Classic
[7]

a part of the execution management in this thesis. In a recent work [7], *communication* and *safety* aspects have been addressed, especially regarding freedom from interference with existing safety-critical software functions and the support for asynchronous session creation in the micro-ROS middleware via standard CAN wrapped under serial transport has been implemented. In this thesis, we mainly focus on the *execution management* and *memory management*.

The execution management in micro-ROS is based on an event-driven approach and assumes POSIX like operating system, which features the concept of threads and processes, where each thread has its own stack. Messages are processed sequentially and run into completion. Such concepts are not supported by AUTOSAR Classic with an underlying OSEK operating system. The execution management in automotive applications with such an architecture is based on fixed periodic preemptive scheduling scheme in which the functions are organised into periodic tasks running at different periods like 1 ms, 10 ms, 100 ms etc. The functions of each task are executed periodically in a sequential order. Each function runs to completion and stores all state for the next run in static global memory. Tasks with shorter period can preempt the other tasks but share the same stack. Therefore, the whole system uses one stack only. In AUTOSAR Classic, there is no concept of dynamic memory allocation, unlike in ROS 2/micro-ROS.

As a result, for any application designed using micro-ROS architecture on an AUTOSAR-based platform, the handling of execution management and memory management will be one of the biggest challenges. In addition, the middleware of micro-ROS with the DDS-based communication model has to be analysed and the concepts has to be developed and implemented to support its functionality with tasks instead of threads. Not only the middleware, but for the successful porting of micro-ROS software stack on an AUTOSAR-based platform, requires analysis and implementation adaptions in other layers of the software stack too to support such a functionality.

As explained in the last paragraph, for any application designed using such an architecture with the execution model mapped from event-based to fixed periodic preemptive scheduling scheme needs efficient analysis that would describe the characteristics of real-time critical tasks in terms of modeling. In automotive industry, applications developed using model-driven development is widely used for prototyping and products [8]. This also includes timing models that specify real-time properties of the application at the system level. Such timing models provide a strong foundation for problem analysis and enable the support at early design stage without the actual implementation of an application on an embedded device. As a result, automotive applications that mostly involve control functions running at different rates, directly depend on the model-based systems so that at the system level the reaction time of the processing path involving these functions are well within the boundary. Currently, there are no generic models that describe the micro-ROS architecture on an AUTOSAR-based ECU and the necessary modeling elements that would specify the characteristics of real-time critical micro-ROS tasks on such a platform.

Thus, the goal of the thesis work is mainly in terms of modeling and timing analysis of micro-ROS application on an AUTOSAR Classic-based ECU, with an example of BODAS RC40 Series Controller for mobile working machines.

1.2 Methodology

To begin, a systematic study of the complete micro-ROS framework and AUTOSAR Classic framework was conducted with respect to support the micro-ROS framework on an AUTOSAR Classic-based platform. The analysis proposed the refactoring of certain micro-ROS core APIs in every layer using a standard approach, so that the framework would adapt to AUTOSAR

periodic task scheme. Along with this, it also helped to develop the concept of mapping the event-based execution model to fixed periodic preemptive scheduling scheme with as minimal architectural changes to the overall micro-ROS stack, to ensure the long-term compatibility and maintainability. Then, for the successful porting of such a stack on these platforms, necessary memory handling functions, compatibility functions from POSIX and standard C library and standard CAN transport based on custom transport configuration were implemented. Firstly, the proof of concept developed was validated with prototypical implementation over Linux platform, in which the scheduling scheme was mapped similar to AUTOSAR-based scheduling. Then the concept developed was validated on the BODAS RC18-12/40 controller unit, which is an AUTOSAR-based ECU popular in off-highway applications. From the modeling perspective, the analysis of the state-of-the-art timing model approach in automotive industry was conducted. Along with this, the most popular standards for model development and simulation tools for performance analysis were investigated. Then the timing properties associated with each entity in every layer of micro-ROS stack was analysed, which helped to design the generic performance model that describe the micro-ROS based applications on AUTOSAR-based ECUs.

The effectiveness of the approach and implementation was demonstrated in a micro-ROS test application on a BODAS RC18-12/40 Series Controller with a Control Function at 1 kHz and a concurrent communication workload at 50 Hz with a microprocessor running ROS 2. The test application was also modeled using the designed generic performance model with one of the popular standards for model development in automotive industry. The goals to be achieved were:

1. to guarantee the latency of 1 ms and less than 0.5 ms jitter for the Control Function

2. to predict the latency by the timing model with a precision of ±20%.

1.3 Document Structure

The document is organized into several chapters as follows:

- *Chapter 2* gives the necessary background knowledge to understand the fundamental terms and concepts discussed in this thesis work.

- *Chapter 3* explains the state-of-the-art technologies and the existing relevant work related to this thesis. Further, based on the evaluation,

four specific research questions are formulated concerning the modeling and timing analysis of micro-ROS stack.

- *Chapter 4* mainly describes the analysis, concepts and implementation details for porting the micro-ROS Client framework on BODAS platform or any other embedded platform running AUTOSAR/OSEK like OS.
- *Chapter 5* deals with the analysis of timing properties of micro-ROS stack and the design and development of generic performance model using one of the popular modeling languages. And also explains how these developed models could be further used for performance analysis using one of the popular simulation tools.
- *Chapter 6* evaluates the effectiveness of the approach as explained in Chapter 4 and Chapter 5 by a micro-ROS test application implemented on BODAS platform. It also discusses the results obtained from timing and modeling perspective of the application.
- *Chapter 7* provides the conclusion and lists the possible future work.

Chapter 2

Preliminaries

This chapter provides an overview of various topics that help to understand the technical aspects of this thesis work. In Section 2.1, ROS and ROS 2 is explained along with its benefits and drawbacks. In Section 2.2, an introduction to micro-ROS along with its key features and architecture is given. Section 2.3 covers topics related to scheduling under POSIX, followed by the explaination of fixed periodic preemptive scheduling scheme. In Section 2.4, the vehicle control unit BODAS RC18-12/40 for off-highway applications is described including its hardware and software specifications.

2.1 Robot Operating System

In today's world, robots have decreased the human effort in numerous applications related to various domains across multiple industries. One of the main reasons for the progress is the methodology followed in the software development as compared to the earlier methods. Considering the frameworks which play a vital role in the software development, *ROS* is the most commonly used frameworks for service robotics. ROS, a meta-operating system [9] mainly provides services related to process management, package management and support for low-level device control including hardware abstraction. Its message types and formats are a de-facto standard for robot software and these standards are used in interfacing from sensors and actuators to fundamental robotic algorithms like motion planning, navigation, localization etc. The development is open-source and it also provides tools and libraries for visualization, simulation, data-recording and monitoring [9] [10]. Its modular design supports code reusability and implementation of advanced algorithms in robotics. With all these features, it is preferred over other frameworks such as Microsoft Robotics Developer Studio [11], Orocos [12], YARP [13] and is

also used in the development of industrial robots and agricultural robots. But it had its own certain limitations mainly in terms of real-time support, cross-platform support, supported only under Linux, mostly applications in academic research etc. All these drawbacks led to the development of next version, *ROS 2*.

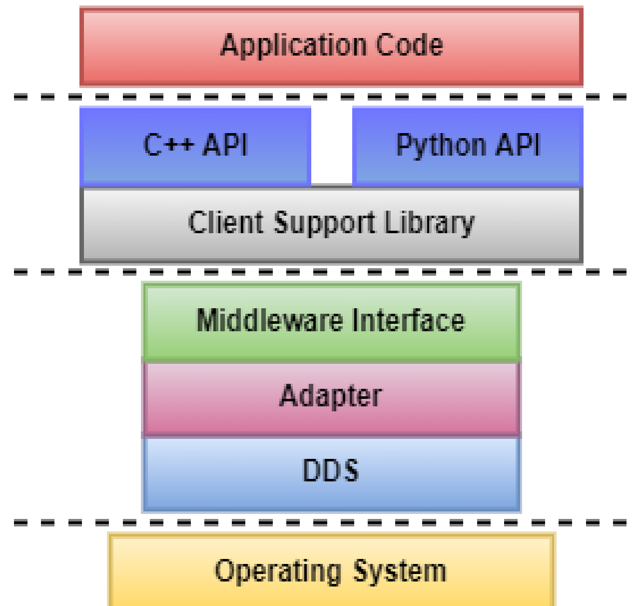

Figure 2.1: ROS 2 architecture

A lot of effort was put into the creation of ROS 2 to offer flexibility for cross-platform development including support for several operating systems such as Linux, Mac OS, Windows, POSIX compliant RTOSes and also with the goal of defining the layers of abstraction to ensure code reusability in a more advanced way compared to ROS [4]. Figure 2.1 depicts the multi-layered architecture of ROS 2. ROS 2 uses DDS as its middleware layer which is suitable for real-time embedded systems as DDS meets the requirements for fault-tolerance, safety and security [14]. It consists of Data-Centric Publish Subscribe model that ensures data distribution and data transport between processes to be reliable [15]. It comes with QoS mechanisms which helps in determining the behaviour of its service [16]. To ensure code reusability, ROS 2 abstracts the DDS middleware from application code by an additional layer on top of DDS, known as ROS Middleware Interface (RMW) [17]. ROS 2, being tested in the Continuous Integration (CI) environment also supports development of packages in C++ standard and Python [18]. With all these distinctive features and architecture support, ROS 2 still has two important limitations: (1) it lacks support on microcontrollers and (2) real-

time support. These drawbacks led to the development of *micro-ROS* , with the main goal of getting ROS 2 onto microcontrollers.

2.2 Micro-ROS

In most of the robotic applications today, microcontrollers are used. The main reasons are real-time support with hard and low latency, power saving with regular sleep periods, support for hardware peripherals like ADC, DAC, GPIO and communication protocols like UART, SPI, I2C, CAN and Ethernet [19][20]. As mentioned before, ROS 2 lacks support on these devices which has constraints mainly in terms of memory and CPU utilization, hence, micro-ROS [19] aims at bridging the gap between these resource-constraint embedded devices and larger processors with ROS framework. Figure 2.2 depicts the differences in micro-ROS and ROS 2 from the architecture perspective, which shows that the middleware is adapted in micro-ROS to support platforms with resource-constraint environment and client-library is updated to give real-time support features. Micro-ROS has been developed as a part of EU funded project OFERA (Open Framework for Embedded Robot Applications) [21] with the primary goal of extending robot operating system for microcontrollers, so that the set of features and tools developed for the ROS framework could also be used on these tiny embedded devices.

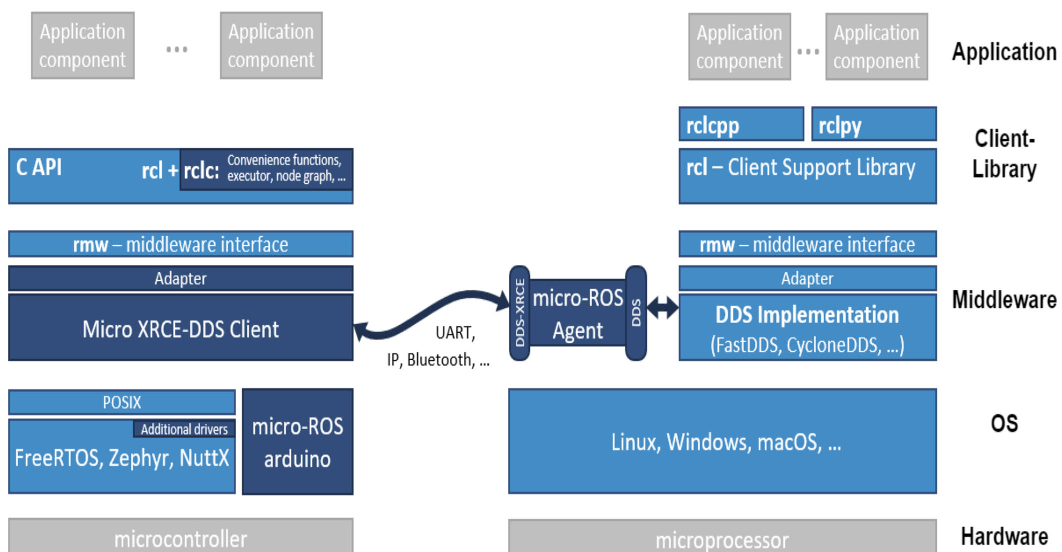

Figure 2.2: Micro-ROS vs ROS 2 architecture
[22]

2.2.1 Micro-ROS Architecture and Features

As shown in the Figure 2.2, micro-ROS stack follows the ROS 2 architecture i.e., most of the components in micro-ROS are from ROS 2. The components developed specifically for micro-ROS are its client-library (rclc), which is an extension to the existing ROS 2 client support library (rcl) and its middleware, XRCE-DDS (DDS For Extremely Resource-Constrained Environments) [23], which is optimized for low resource consumption embedded devices. Finally, at the very bottom lies the RTOS, which is assumed to be POSIX-compliant.

There are about seven key features offered by micro-ROS which make them as the first-choice for any robotic product based on microcontrollers [22].

- **Optimized client-library for ROS concepts on microcontrollers**
  The client-library of ROS 2 (rcl)[24] includes the concepts of nodes, publishers/subscriptions, topics, client/service, node graph, lifecycle, actions, parameters etc. Micro-ROS brings all these core concepts onto microcontrollers along with the set of extensions and convenience functions (rclc)[25] allowing implementation of common scheduling patterns from embedded systems. Together, rcl+rclc form a complete client-library in micro-ROS.

- **Flexible middleware, considering extremely resource-constraint environment**
  ROS 2 is implemented with DDS as it middleware. Micro-ROS comes with a new DDS for extremely resource-constraint environment, called *Micro XRCE-DDS*, implemented by eProsima, meeting all the necessary requirements for embedded systems [26]. Micro XRCE-DDS follows a client-server architecture, in which Micro XRCE-DDS Client [27], running on resource-constraint embedded devices connects to Micro XRCE-DDS Agent [28], running on larger processors. The Agent is responsible to route the information from clients to DDS world and vice versa. Figure 2.3 depicts the client-server architecture of Micro XRCE-DDS and shows how the Agent bridges the clients to DDS world and vice versa.

- **Indefectible integration with ROS 2**
  As explained in the last key feature, micro-ROS nodes running on microcontrollers seamlessly connects to the external ROS 2 system with the help of micro-ROS Agent running on that system. As a result, with the known ROS 2 tools [29] and APIs, micro-ROS nodes could be accessed just as normal ROS nodes.

- **Support on any POSIX-compliant RTOS**
  Micro-ROS application can be built and ported on any RTOS such as FreeRTOS, NuttX, Zephyr with POSIX interface. In ROS 2 package, RTOS-specific tools are integrated with a few generic setup scripts that can be run through command line. During the firmware creation step, application developers can choose the RTOS and these scripts will build the framework. It is very hard to notice the differences in the configuration and definition of executables between the various RTOS supported [30].

- **Non-restrictive license**
  The complete micro-ROS stack including middleware layer, client-library related packages and tools are under Apache License 2.0 [31], like ROS 2. The two exceptions are the benchmarking tool and RTOS layer. Benchmarking tool is included under GPL v3 License [32] and it does not have a effect on product license because generally it is only used during development and is not part of the product. But the developer needs to consider the license agreement of RTOS used in their product [33].

- **Shared development by supportive community**
  The ROS Embedded Working Group, which is a formal ROS 2 working group [34] constantly supports in the development of micro-ROS applications and creation of benchmarking tools that check the performance, CPU time consumption, memory-usage of the application and help in optimization of the application on the embedded hardware [35]. This community which is self-organised also provides tutorials from basics to advanced level with detailed explaination on the framework and tools [36]. In addition to this, it also provides quick support via Slack channel and GitHub. eProsima provides the commercial support for Micro XRCE-DDS middleware.

- **Supports interoperability**
  Apart from few extensions and functions in the micro-ROS stack in support for embedded platforms, it is highly flexible and modular with mature and stable software components: open-source RTOS, OMG standard lightweight middleware and ROS 2 client-library (rcl). Due to this, long-term maintainability and interoperability can be assured.

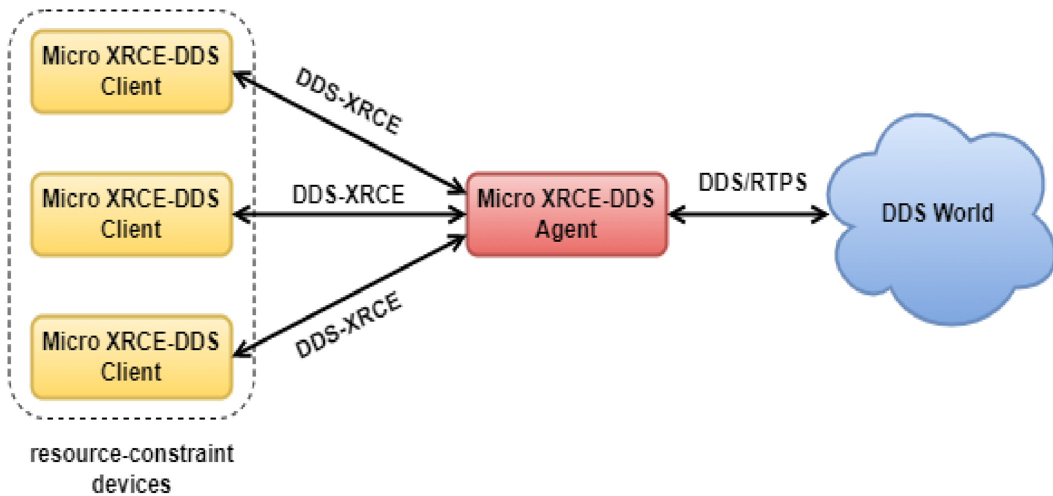

Figure 2.3: Micro XRCE-DDS client-server architecture
[37]

2.3 Scheduling in Embedded Systems

A combination of microprocessor, memory and several input/output peripheral devices built for a dedicated function or range of functions within a larger mechanical or electronic system is referred to as an embedded system [38]. In such a system, a *scheduler* is mainly responsible for determining the sequence and the duration time of tasks to run on the CPU [39]. It decides which tasks to be executed based on the scheduling algorithm used in implementing it. There are many scheduling algorithms in embedded systems, with each one having its own pros and cons. Therefore, its important to design the application based on the scheduling algorithm supported by the OS on the embedded platform to ensure the task's timing-properties are well within the specified boundary.

2.3.1 Scheduling under POSIX

POSIX, an open operating system interface standard has been developed primarily to promote interoperability and portability of application programs to support their compilation and execution without changing the source code across variants of Unix OSs [40]. In POSIX, each executing instance of an application program is referred to as a *process*, which owns various resources that includes address space. Within a process, multiple flow of executions is possible and these flow of executions are referred to as *POSIX threads* [39]. Thus, a process can also be referred to as a collection of one or more

threads. Each thread owns a separate stack, however the threads within a process share some resources like open file descriptors, global variables etc. The standard POSIX 1003.1b defines three scheduling policies:

- **SCHED_FIFO** [41]
  It schedules threads/processes according to first come first serve using a FIFO queue. There are different queues, with each queue assigned a priority level. Hence, the thread at the front of highest-priority FIFO queue will run into completion without preemption as there is no time-slicing among threads of equal priority.

- **SCHED_RR** [41]
  It is an extension of SCHED_FIFO with time-slicing among threads of equal priority i.e., each thread is allowed to run for a defined time quantum only.

- **SCHED_OTHERS** [41]
  Its implementation is usually system-specific. On Linux kernel source code, it is named as SCHED_NORMAL. Completely Fair Scheduler is the default time-sharing scheduler of the SCHED_NORMAL class which is considered for all threads with no special real-time execution constraints [41].

2.3.2 Fixed Periodic Preemptive Scheduling Scheme

Fixed periodic preemptive scheduling is popular and commonly used in real-time automotive systems. In this scheduling, fixed task periods are defined i.e., 1 ms, 10 ms, 100 ms etc and their priorities are defined according to the rate monotonic scheduling assignment i.e., task with shorter periods have higher priorities. As an example, consider three tasks defined as Task 1 with a period of 5 ms, Task 2 with a period of 10 ms and Task 3 with a period of 30 ms. As per priority assignment, Task 1 will have the highest priority, Task 2 a medium priority and Task 3 the lowest priority. Therefore, Task 1 can preempt Task 2 and Task 3, whereas Task 2 can only preempt Task 3. Figure 2.4 depicts the scheduling of these three tasks under fixed periodic preemptive scheduling scheme.

OSEK [42], an industrial standard API supports portability and reusability of software in automotive applications, so that pure software-solution can be obtained and run on any OSEK-compliant ECU. It abstracts away the development from the underlying hardware. However, there are two distinct

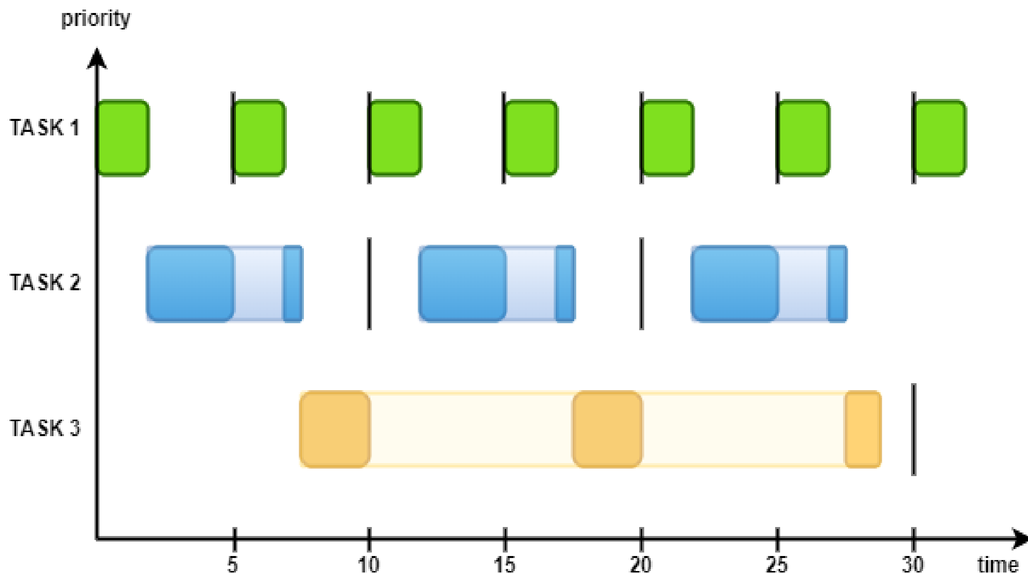

Figure 2.4: Fixed periodic preemptive scheduling

features that distinguishes OSEK kernel from other standards. The first feature is that all objects to be statically defined at compile time i.e., no support for either dynamic memory allocation or dynamic tasks creation [43]. The second feature is the Single Stack Architecture, where all the tasks and interrupts run on a single stack [44].

Regarding task management, OSEK kernel supports co-operative scheduling (not considered in this work) and preemptive scheduling. Under preemptive scheduling, not only tasks with fixed periods as explained in the last paragraph is supported, but also tasks with variable rate (not considered in this work) is supported. In automotive, an example for task with variable rate (non-periodic) is rpm-synchronous task for engine control.

2.4 BODAS RC40 Series Controller for Off-Highway Applications

The BODAS RC40 Series Controllers from Bosch Rexroth are automotive technology based specifically designed to enable flexibility in the development of controlling hydraulics in construction, forestry and agricultural machinery. These controllers satisfy the functional safety requirements of agricultural and forestry machinery listed according to ISO 25119 [45] up to Agricul-

tural Performance Level (AgPL) "d". This allows the customers to support the development of future autonomous applications and integration of new connectivity trends on these controllers [46]. They are available in three versions: Small series (RC5-6/40), Medium series(RC18-12/40) and Large series (RC27-18/40). For our experiments, we have chosen the RC18-12/40 Medium Series Controller.

2.4.1 Hardware Specification

RC18-12/40 ECU, the key component of the BODAS system, is a multi-core controller with 32-bit processor operating at 300 MHz clock frequency with hardware safety module (HSM). With robust and compact design, it is highly suitable for the development of flexible, secure and safety-related applications in mobile working machines [47]. Figure 2.5 depicts an image of one of the RC40 Series Controllers from Bosch Rexroth.

Figure 2.5: Bosch Rexroth RC40 Series Controller
[48]

Regarding the I/O details, a total of 58 inputs are provided, out of which 8 provide an option for connecting sensors to measure frequency signals via SENT protocol and the rest for reading analog/digital voltages or resistance. A total of 30 outputs are provided, out of which 14 are high-side (battery-switching) and 16 are low-side (ground-switching) [47]. Regarding the communication to the other devices, three independent standard CAN 2.0 / CAN FD interfaces are provided, out of which one of them can be used for diagnosis and flashing with BODAS service tool 4.x [49]. It also provides support for one LIN and one Ethernet channel [47].

Table 2.1 lists the hardware specifications of RC18-12/40 Series Controller.

Attribute	**Value**
Supply Voltage	12 V or 24 V nominal
Processor	Infineon TC389
Clock Frequency	300 MHz
Max. supply current	40 A
SRAM	1 MB
DFlash	128 kB
PFlash	10 MB
Temperature Range	−40°C to +85°C
Weight	930 g, ± 5 %

Table 2.1: Hardware Specifications of RC18-12/40
[47]

2.4.2 Software Specification

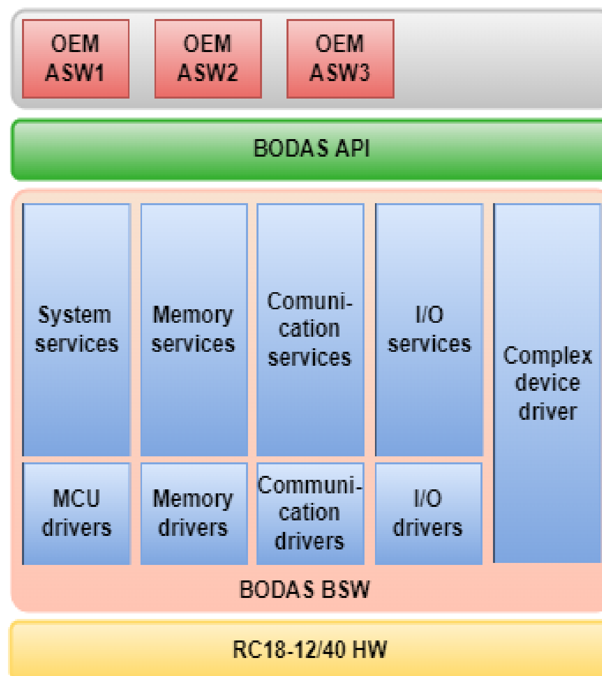

Figure 2.6: BODAS RC18-12/40 software stack
[50]

BODAS software stack mainly consists of three layers that includes microcontroller (HW) layer, base software (BSW) layer and application software (ASW) layer, similar to AUTOSAR software architecture. Figure 2.6 depicts the software stack on BODAS RC18-12/40 Series Controller.

In the stack, the lowest layer is hardware layer, i.e., RC18-12/40 ECU hardware in our case. Next, the BODAS BSW layer includes system services, memory services, communication services, I/O services along with the necessary device drivers. It is built upon AUTOSAR CUBAS (Common UBK Basic Software), which is the Bosch implementation of AUTOSAR for its base software development for active and passive systems. The BODAS ASW layer includes complete user code, which is application-specific developed by OEMs. Finally, the responsible layer that supports the ASW development to be independent of BSW is the BODAS API layer. It mainly creates an interface for the user code in ASW layer to access the functionalities in the BSW layer via standard C Language APIs.

Chapter 3

State of the Art

In this chapter, Section 3.1 describes the related work that has been carried out in handling the execution management of micro-ROS on different RTOS, with POSIX and non-POSIX compliant. Section 3.2 covers the related work being carried out in model-based design in automotive industry, followed by the modeling languages available for automotive applications and then the simulation tools available for model-based performance analysis. In Section 3.3, the related work being carried out in modeling of ROS processing chains is analysed. Finally, in Section 3.4, we derive four specific research objectives for this thesis work from the overall problem statement discussed in Section 1.1.

3.1 Micro-ROS/ROS 2 Execution Model

The executor concept introduced in ROS 2 to support the execution management lacked real-time control capabilities. As a result, the executor concept in micro-ROS which is almost similar to ROS 2 is refined and is more flexible in support of real-time guarantees. Most of the existing works explains the refinement of executor concept of micro-ROS and development of micro-ROS application with scheduling algorithms supported by RTOS which assumes POSIX-compliant. One such work [51], in which a micro-ROS application was successfully demonstrated with a reservation based scheduling of the NuttX RTOS on a STM32 series microcontroller with the executor concept of ROS 2 supporting real-time scheduling capabilities. In another work [52], a model of ROS application running on top of a resource reservation scheduler (SCHED_DEADLINE) in Linux was presented and validated, with the major goal of improving ROS 2 implementations from real-time perspective. Other works [53][54][55] have also contributed in support of real-time exten-

sion for the ROS architecture.

As mentioned in Section 2.2.1, a RTOS with POSIX-compliant is assumed in the standard approach to micro-ROS. In addition to Linux and Windows, it is supported by the RTOSes FreeRTOS, Zephyr, NuttX [56]. Recently, a micro-ROS example for Arm Mbed OS is also supported [57]. From the non-POSIX compliant perspective, micro-ROS is available for bare-metal applications. This is supported with the release of micro-ROS as a standalone library with header files for Arduino IDE [58][59]. However, none of these refinement could support the executor concept of micro-ROS on AUTOSAR/OSEK OS. Thus, no equivalent approaches mapping ROS 2 execution management which is event-based to fixed-periodic scheduling in automotive ECUs for real-time control has been proposed so far.

3.2 Modeling and Performance Analysis in Automotive

In the automotive industry, model-based design have already proven its worth across OEMs, suppliers and developers on different levels. In one such work [60], it is described that the application developed using such a design improve significantly according to performance, meeting timing constraints. In another work [61], it is shown how the entire system is developed using the software models, with the presentation of three case studies based on adaptive cruise control.

3.2.1 Standards for Model Development

The three commonly used standards for model development in automotive industry are: (1) ASAM MDX [62] (Association for Standardization of Automation and Measuring Systems Model Data Exchange Format) , (2) AUTOSAR [5] and (3) AMALTHEA [63]. In ASAM MDX, the description of software components of a single ECU with their interfaces and data elements is specified, without defining its own methodology. In AUTOSAR, the description includes software and hardware. But is available to its own members and partners. AMALTHEA, an open source is more accomplished supplement than AUTOSAR, supporting the modeling of multi-core systems [64]. A detailed comparison on the model elements, system models, methodology and reference implementations of each standard is available in the work [65] and conclusions are also drawn with advantages and disadvantages of every

standard. Table 3.1 shows the comparison of model elements covered by these three standards.

	ASAM MDX	**AUTOSAR**	**AMALTHEA**
Hardware Model	No	Yes	Yes
OS Model	No	Partly	Yes
Static Software Architecture	Yes	Yes	Yes
Dynamic Software Architecture	No	Partly	Yes
Mapping	No	Yes	Yes
Timing Requirements	Partly	Partly	Yes
Software Design Constraints	Partly	Partly	Yes

Table 3.1: Comparison of model elements covered in three standards [65]

3.2.2 Performance Analysis Simulation Tools

The model-based designs must be verified, if the design constraints particularly related to real-time requirements are satisfied. Different approaches could be followed to verify whether real-time behaviour of an application is satisfied. The commonly used approach in automotive applications is with the help of simulation tools, that determines the real-time properties such as end-to-end latency, worst-case response time, jitter, bus and processor utilization at the system level. Few of the popular simulation tools for timing and performance analysis in these applications are [66]:

- open source - Cheddar [67], MAST [68]
- proprietary - Timing Architect [69], SymTA/S [70], chronSUITE [71]

In one work [72], it is shown that using the SymTA/S tool, performance analysis could be done at the system level for complex embedded applications and properties like end-to-end latencies, deadlines, buffer over-underflows, transient overloads could be verified. In another work [73], an automotive application model is applied for real-time analysis to chronSUITE, SymTA/S

tools and two prototype university specific tools and a comparison is made based on real-time verification methods and conclusions are drawn with pros and cons of each tool. The simulation tools mentioned in the last paragraph could be used independent of modeling languages mentioned in Section 3.2.1 with a small configuration change during the setup.

3.3 Modeling of ROS Processing Chains

With most of the service robot applications being time-critical, real-time properties like reaction times, jitter, deadlines etc play a safety-critical role. However, these aspects are hidden in the component or framework implementations even in the most commonly used middlewares in robotics, like ROS , which abstract away the application development from the underlying embedded platform and execution management of the operating system on it. But still, on the one hand, with in-depth knowledge about the framework and the scheduling supported by the RTOS on the embedded device, real-time extension on such a architecture could be possible. But on the other hand, then few compromises had to be done with respect to the cost and time for the application development.

As a result of the problem mentioned above, in most of the software-intensive systems, model-based performance analysis has been used for many years during the early stages of development as they help to analyse the timing properties of the system. Such a effort is also seen with robotic software framework. One such work [74], in which model-driven tools implemented in SmartMDSD toolchain [75] were used to support the robotics development process and in the system-configuration step, tools such as SymTA/S was used for timing and performance analysis. The results from these analysis almost represented the real robot's run-time performance. In another work [76], an automatic latency manager for ROS 2 framework was proposed, which showed that it reduces the maximum observed latency under load compared to the default Linux scheduler. With experiments on mobile robot it was also demonstrated the feasibility of the approach and drawbacks related to platform limitations and real-time analysis in ROS 2 were identified. But, currently for applications with micro-ROS framework developed on any embedded platform, there are no generic model-based design that would help in analysis of real-time properties of micro-ROS task using appropriate simulation tools.

3.4 Specific Research Questions

From the problems explained in terms of timing analysis and modeling for micro-ROS in Section 3.1, Section 3.2 and Section 3.3, four specific research objectives are formulated which need to be explored and analysed in this thesis work. They are as follows:

[Q1.] How to map the event-based execution management of (micro-)ROS to fixed periodic preemptive scheduling with a single stack like in AUTOSAR?

[Q2.] Which modeling elements are necessary to specify the characteristics of real-time critical micro-ROS tasks including processing chains?

[Q3.] How to describe the timing characteristics of the micro-ROS stack and an application in such a model on a specific microcontroller? Which existing meta-models be reused?

[Q4.] How precisely does the developed model allow to predict computing latencies using an appropriate simulation tool?

Chapter 4

Mapping Execution Management to AUTOSAR

This chapter mainly addresses the first research question [Q1] mentioned in Section 3.4. In Section 4.1, the analysis of the micro-ROS Client stack and the problems involved in supporting the stack on AUTOSAR/OSEK like OS is explained. In Section 4.2, the concept for adaptation of the stack layer-by-layer to support the AUTOSAR/OSEK tasks-like implementation instead of threads is explained. In Section 4.3, the concept for mapping the event-based execution model of (micro-)ROS to fixed periodic preemptive scheduling scheme with a single stack like in AUTOSAR is explained. Section 4.4 covers the implementation details on BODAS RC18-12/40 hardware, i.e., the support for standard C library functions, mapping the missing POSIX functions based on BODAS API, the support for standard CAN interface based on custom transport configuration in Micro XRCE-DDS middleware, and finally, the changes in the stack layer-by-layer according to the concepts realized in earlier sections of the chapter.

4.1 Micro-ROS Stack Analysis

As explained in the Section 2.2.1, the micro-ROS Client stack, i.e., the stack on MCU usually assumes the underlying RTOS to be POSIX-compliant which supports the POSIX Thread model (pthread) in particular. The stack is comprised of Micro XRCE-DDS which is the default middleware, RMW_Microxrcedds which is ROS 2 middleware abstraction layer and RCL + RCLC layer which is the client-library. In general, the execution of any application with this stack on an embedded platform can be divided into two phases: (1) initialization phase, which handles the creation of core en-

tities of an application, (2) operation phase, which handles the execution management of an application.

4.1.1 Initialization Phase

The main participants of the ROS 2/micro-ROS ecosystem are *nodes* which exchange messages of various data types via actions, services, topics or parameters [77]. The communication between them is using the publish and subscribe paradigm. In this paradigm, the two main entities that communicate are referred to as publisher and subscriber. On one hand, a particular node is allowed to publish the message on a topic using datawriter associated to the publisher and on the other hand, a corresponding node subscribed to the same topic receives the corresponding message using datareader associated to the subscriber. In the micro-ROS Client stack, the middleware Micro XRCE-DDS is based on DDS communication model in which node, topic, publisher/subscriber and datawriter/datareader form the core entities. These entities has to be created in the Client with the support of the Agent. Therefore, the Client sends the creation request and waits for response from the Agent. In case of POSIX-compliant RTOS, each entity's request-response is handled by separate thread in a blocking function call. Such a concept is not supported on an AUTOSAR-OSEK like OS which is non-POSIX compliant. As a result, to design the micro-ROS application on such an OS which is non-POSIX compliant, the stack which includes DDS communication model must be adapted in each layer to support tasks-like periodic execution model instead of threads-like execution model.

In addition, the functions in each layer of the stack are dependent on certain standard C library functions and POSIX functions. Therefore, the support for missing standard C library functions and mapping POSIX functions based on the chosen platform, must be developed.

4.1.2 Operation Phase

Executor [78] is a new concept introduced in ROS 2 to support the execution management which follows an event-driven approach. The executor mainly coordinates the execution of all the callbacks. Figure 4.1 depicts the default executor concept in ROS 2 on a POSIX like OS with two callbacks. The data is received via the defined transport in the middleware and the middleware puts the data into DDS queue. The executor looks up the wait sets, which notifies it about the available messages from the DDS queue. It takes the

message and process the corresponding callback until completion in a sequential order [79]. In addition, it also supports the execution of timers, which are always prioritized over messages. Apart from this, it does not support any prioritization for the incoming callbacks. Moreover, it lacks the support for real-time control features by not controlling the execution order of callbacks, irrespective of the real-time support from the underlying OS scheduler [79]. Such a behaviour would affect any robotic application involving time-critical callbacks as there is a high probability that these callbacks could miss its deadline, which leads to failure in the predicted behaviour of the system.

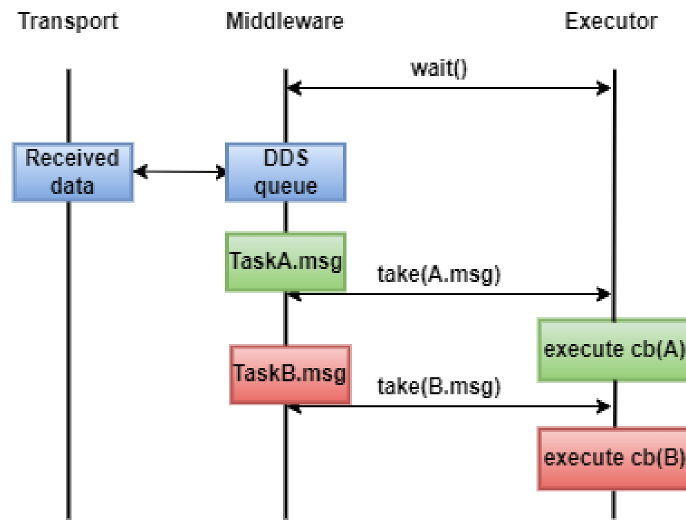

Figure 4.1: Execution Management in ROS 2

To address the drawbacks of ROS 2 executor as explained in the last paragraph, in micro-ROS, an extension of such an executor was developed, called *rclc Executor* [80] which mainly provides the support of following three new features: (1) triggered condition, to control the start for processing of any callback , (2) user-defined execution order, to control the processing order in which the callbacks has to be executed and (3) scheduling configuration, to control the prioritized processing of callbacks by using the scheduling parameters of the underlying OS [80].

The execution management in AUTOSAR Classic OSEK-like OS running on BODAS RC18-12/40 Series Controller uses fixed periodic preemptive scheduling scheme, in which functions are defined and assigned to tasks to run at certain periods and they can also be given priority. Figure 4.2 depicts the comparison of execution management in AUTOSAR and ROS 2. As an example, two periodic tasks (TASK A and TASK B) with different rates and

execution times as shown in the Figure 4.2 are considered. TASK A with shorter period and TASK B with longer period. Thus, as explained in the Section 2.3.2, TASK A has higher priority over TASK B. Firstly, scheduling as per AUTOSAR scheme is shown with preemptions at four different points and tasks meeting their deadlines. Below that, scheduling as per ROS 2 is shown, with the deadline missed for the second arrival of TASK A, because in ROS 2, Task A and Task B implemented as callbacks are executed until completion and then only the next message is processed. Therefore, such a ROS 2 executor concept cannot be applied on a BODAS RC18-12/40 Series Controller.

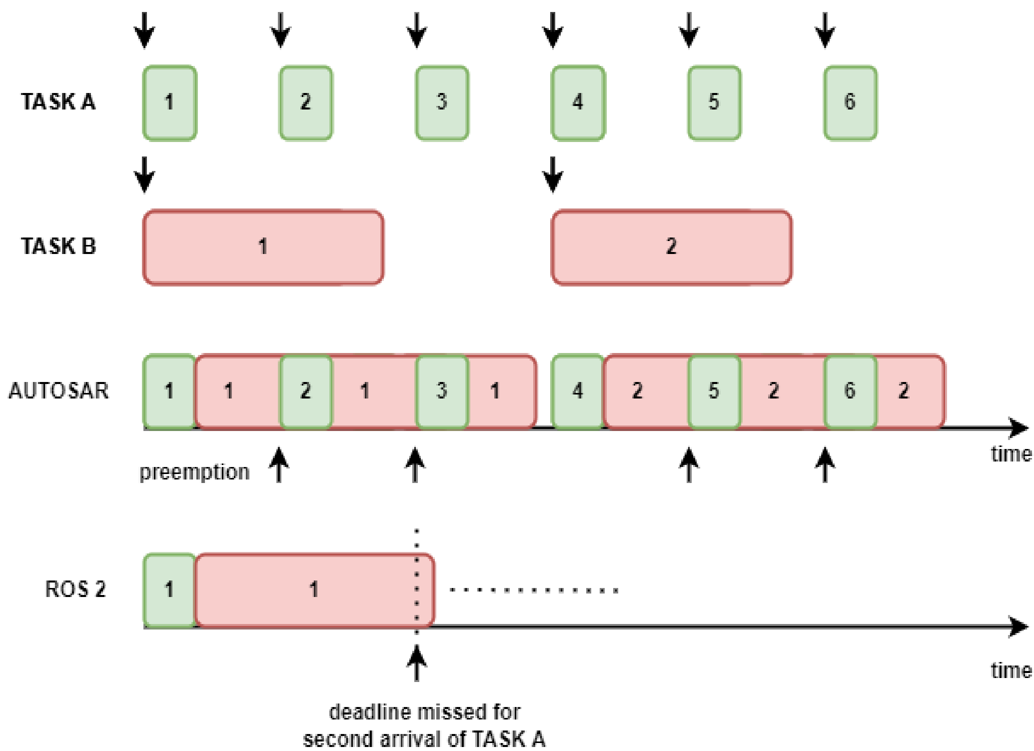

Figure 4.2: Comparison of Execution Management

4.2 Initialization Phase with State-Machine Model

Considering the DDS communication model on an AUTOSAR-based platform, the request-response for entities creation in a task can be handled with

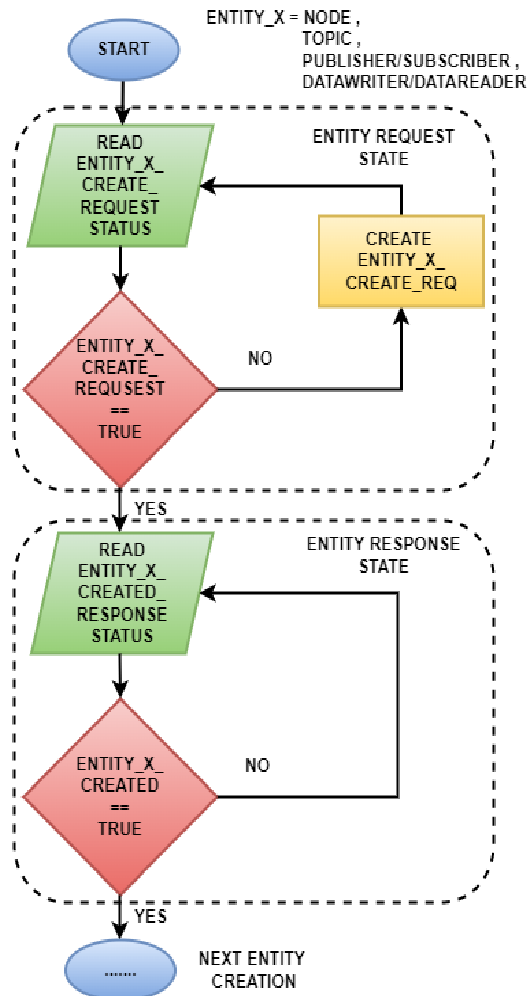

Figure 4.3: Flow Chart for entity request-response handling in Client

the help of a *State-Machine model* approach, mainly to avoid blocking function calls. In this approach, for each entity creation in the Client two states are considered. In the first state called the *entity request state*, the request for a particular entity creation is checked and sent from Client. In the second state called the *entity response state*, the corresponding response from the Agent is acknowledged and decided whether to move to next entity creation request state or not. Figure 4.3 depicts the handling of request-response mechanism for an entity in the middleware of micro-ROS Client. The transition between the states must be handled by updating the respective global variables in every task cycle. The entities creation must follow a step-by-step sequence as explained below:

1. On the Client side, a node creation request is created and sent in one cycle.
2. On the Agent side, the node creation request is accepted and accordingly response is sent.
3. In parallel, on the Client side, in the next cycle, a response for the node creation request is acknowledged. If there is successful response from the Agent, then it moves to next entity creation request state. If there is no response from the Agent, in the next iteration a response for the node creation request is waited for.
4. The above steps are repeated until all the entities i.e., node, topic, publisher/subscriber and datawriter/datareader are sequentially created.

To support such a mechanism in the micro-ROS middleware, subsequent API function calls in the above layers of micro-ROS Client software stack must be analysed and also adapted. Table 4.1 lists the API calls in each layer of the stack to be updated to support the state-machine model like approach discussed in the last paragraph.

	Node	**Topic**	**Publisher/ Subscriber**	**DataWriter/ DataReader**
RCLC	rclc_node_init_default rclc_node_init_with_options	-	rclc_publisher_init/ rclc_subscription_init	-
RCL	rcl_node_init	-	rcl_publisher_init/ rcl_subscription_init	-
RMW_ Microxrcedds	rmw_create_node, create_node	rmw_create_publisher/ rmw_create_subscription, create_topic	rmw_create_publisher/ rmw_create_subscription	rmw_create_publisher/, rmw_create_subscription
Micro XRCE-DDS	run_xrce_session	run_xrce_session	run_xrce_session	run_xrce_session

Table 4.1: API calls of micro-ROS stack

4.3 Operation Phase with Periodic AUTOSAR Tasks

The client libraries RCLC and RCL include the APIs involved in handling the execution management. The executor and callback functions which are event-based in micro-ROS have to be mapped to periodic AUTOSAR tasks. The executor function must be assigned to task executing at a period (frequency) which is equal or less (equal or high) than the least (highest) period (frequency) among the tasks to which callbacks are assigned. The assignment of callback functions to periodic tasks depends on two factors:

1. Incoming Message Interval (Frequency)
   The callback related to the incoming message must be assigned to a task executing at a same or lower period (higher frequency) than the incoming message interval (frequency), so that none of the messages are lost. For example, if the interval (frequency) of an incoming message is ’x’ seconds (Hz), then the callback must be assigned to a task executing at a period (frequency) atleast lower (higher) than ’x’ seconds (Hz).

2. Execution Time of Callback
   The callback must be assigned to a task executing at a higher period (lower frequency) than its execution time, so that it has enough time for computation. For example, if the execution time of the callback is ’y’ seconds (Hz), then the callback must be assigned to a task executing at a period (frequency) atleast higher (lower) than ’y’ seconds (Hz).

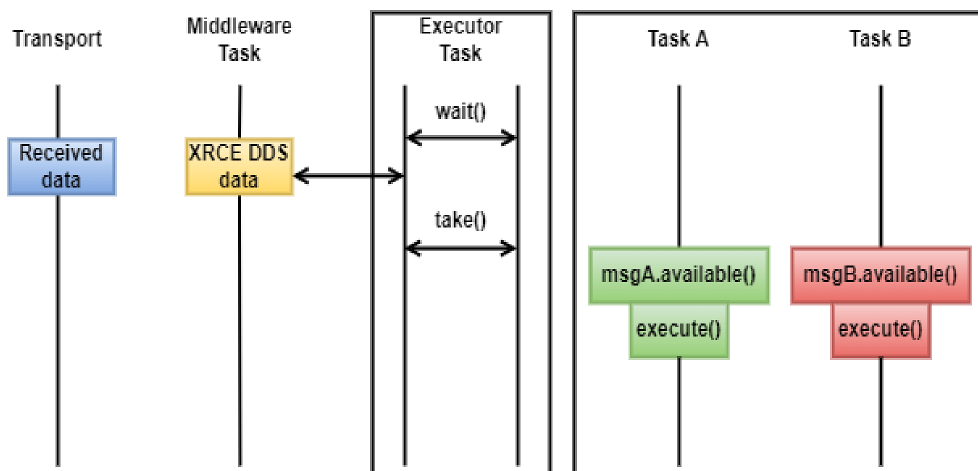

Figure 4.4: Execution Management mapped to AUTOSAR task scheme

Figure 4.4 depicts the execution management of micro-ROS with two callbacks mapped to AUTOSAR Classic OSEK-like OS with Task scheme as explained in the last paragraph. The data is received via the defined transport interrupt service routine. The middleware function running in a high-frequency task unpacks the data into XRCE DDS message. The executor function running in the same or slightly lower frequency task takes the incoming XRCE DDS message and will make it available for the corresponding callback. The callback function is not invoked immediately, but is executed in tasks often at lower frequency than executor. Therefore, from the application perspective, it must be ensured that these functions run into completion within the respective assigned period.

4.4 Implementation on BODAS RC18-12/40

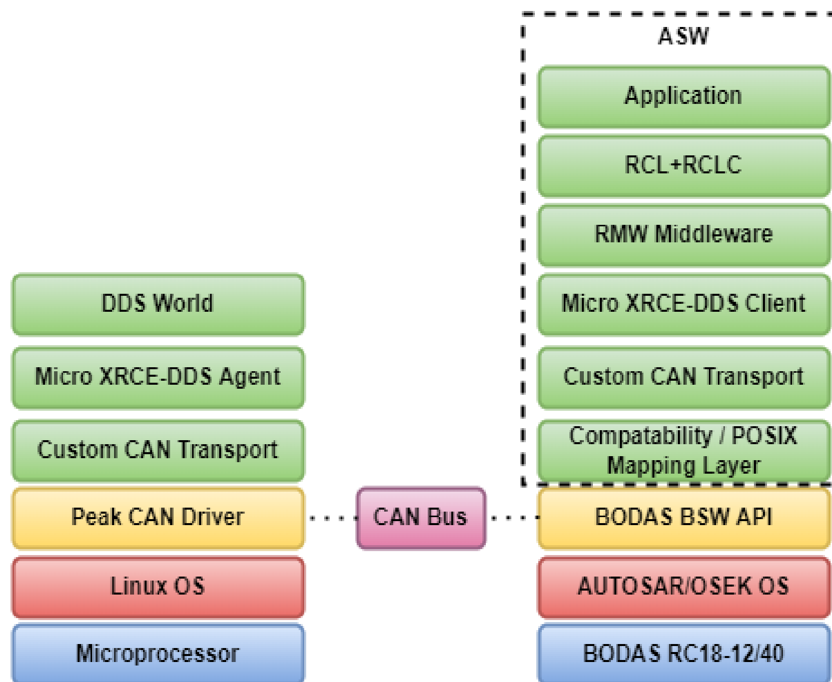

Figure 4.5: Micro-ROS stack on BODAS RC18-12/40

This section explains the porting of micro-ROS Client stack on one of the vehicle control units, which is an AUTOSAR-based platform and popular in off-highway market. Figure 4.5 depicts the micro-ROS Agent stack on a computer running Linux OS and the micro-ROS Client stack on BODAS RC18-12/40 ECU running AUTOSAR/OSEK like OS with the existing BSW. Each

layer on BODAS is added in the ASW region step-by-step from Compatibility/POSIX mapping layer to RCL + RCLC layer. The communication between Client and Agent is setup via standard CAN at 250 kbits/sec. On the Agent side, the hardware support for CAN interface is provided by Peak CAN drivers which is accessed by SocketCAN framework as a network device. On the Client side, two CAN channels are available for user, out of which one is used for communication with the Agent and other one is used for debugging. Before actual implementation on the chosen BODAS platform, the concepts developed were first realized and tested over the Linux platform.

4.4.1 Additional Layers: Support on BODAS

To adapt the layers of the stack to support the tasks-like periodic execution model in AUTOSAR, two more additional layers that must be introduced are *compatibility layer* which gives the support for standard C library functions and *POSIX mapping layer* which maps the necessary standard POSIX functions to base software API function calls on BODAS. Some of the functions in other layers of the stack will be dependent on the functions developed in these two layers.

Compatibility Layer: Support for C Library

BODAS BSW lacks support for standard C library functions which are used by functions in different layers of the Client software stack. Some of these functions are dependent on the architecture and the underlying OS. Therefore these functions are developed and are added in the files: errno.h, nlibc.h, stdbool.h, stdint.h, stdlib.h and string.h to support the dependency. Few of the predominant functions re-implemented are:

- Memory Functions: malloc(), calloc(), realloc(), free() are standard C functions related to dynamic memory management. On BODAS with AUTOSAR/OSEK like OS, dynamic memory management is not allowed. From the memory profiling analysis of micro-ROS Client by eProsima [81], it is shown that the dynamic memory consumption is only during initialization of parameters like nodes, publishers/subscribers etc and not during run-time. Therefore, on BODAS platform, a simple memory management was implemented where a pool of memory was reserved on the stack to initialize the necessary parameters. To support this implementation, malloc(), calloc(), realloc() methods were re-implemented and not free() method because the deallocation of memory happens irrespectively during the shut-down of device.

- String functions: vsnprintf(), snprintf() are standard C functions used to format string and write it to string buffer. These functions are re-implemented using memcpy() method as string concatenation functions which handles the concatenation of two, three and four strings. The following code snippet shows the function for concatenation of two strings.

```
#define CLIB_COMPAT_RET_OK 0
#define CLIB_COMPAT_RET_NOT_OK 1
static int string_concatenate_two(char * buffer, size_t
    buffer_size, const char * str1, const char * str2)
{
  if (buffer_size < (strlen(str1) + strlen(str2)))
  {
    return CLIB_COMPAT_RET_NOT_OK;
  }
  memcpy(buffer, str1, strlen(str1));
  memcpy(buffer+strlen(str1),str2, strlen(str2));
  return CLIB_COMPAT_RET_OK;
}
```

POSIX Functions Mapping Layer: Based on BODAS API

- Clock Function: One of the parameters in the clock function used in RCL layer (rcl_clock_init()) is mapped to BODAS System timers to get the current time of the clock using BODAS BSW API. It was an easy implementation, mainly handling the data conversion as shown in the following code snippet.

```
static int16_t clock_gettime(clockid_t clk_id, struct
    timespec *tp)
{
  uint64 operating_time_u64;
  operating_time_u64 = bds_sys_getTime();//BODAS BSW API
  tp->tv_nsec = (operating_time_u64 % 1000000) * 1000;
  tp->tv_sec = operating_time_u64 / 1000000;
  return 0;
}
```

- Atomic Functions: BODAS platform does not support the standard atomic kind of operations. Therefore, all the atomic operations used in the micro-ROS Client software stack had to be re-implemented as per BODAS architecture with lock and release API from BODAS BSW. Some of the atomic functions re-implemented for BODAS platform are: atomic_get(), atomic_add(), atomic_load(), atomic_exchange() and

atomic_fetch_add(). The following code snippet shows the macro defined for atomic_fetch_add().

```
#define atomic_get_bodas(obj, value) do{ \
  (*value) = (obj)->__val;                \
} while (0)

#define atomic_add_bodas(obj, value) do{ \
  (obj)->__val += (value);                \
} while (0)

#define rcutils_bodas_atomic_fetch_add(object, out, arg)
   ({ \
  BDS_SYS_GET_LOCK_COMMON();              \
   __typeof__ (*object) temp = *object;  \
   atomic_add_bodas(object, arg);         \
   atomic_get_bodas(&temp,&out);          \
   BDS_SYS_RELEASE_LOCK_COMMON();         \
   })
```

4.4.2 Micro XRCE-DDS over Custom Transport

In the automotive industry, the widely used protocol to communicate between the network control units is standard CAN. Its advantages in terms of robustness, cost, wiring requirements, reliability and flexibility when compared to other protocols makes it a de-facto standard in a large network systems. With the view of micro-ROS application to be developed on BODAS RC18-12/40 Series Controllers, CAN protocol is decided as the first-choice for the communication between the Client and Agent.

Custom Transport Configuration

An application developed with micro-ROS stack assumes Micro XRCE-DDS as the middleware of choice. At the present moment, the communication between Micro XRCE-DDS Client and Micro XRCE-DDS Agent is supported natively with TCP, UDP and Serial protocols at the transport layer. In a recent work [7], it is shown that the communication is also possible with standard CAN wrapped under serial protocol. But recently, eProsima came up with the concept of custom transport and developed the necessary template functions in the Client and Agent libraries [82] [83], so that the user can decide his choice of transport layer protocol. In order to also support the transmission virtually over any network, two general modes are also supported from the framing perspective:

1. Stream-oriented mode: Necessary headers, overheads and CRC bytes are added along with the XRCE message and follows a stream logic for transmission

2. Packet-oriented mode: XRCE message is transmitted as a packet and the whole packet can be sent

The required mode can be selected by enabling or disabling the framing parameter during the setting of uxr_set_custom_transport_callbacks() method in both Client and Agent. For our application, we have selected the packet-oriented mode by disabling the framing parameter because the CAN protocol itself has lot of security mechanisms and also we want to keep the minimum number of frame transmissions for any response-reply mechanism between Client and Agent. Along with the framing parameter, the callbacks related to initializing, closing, reading and writing functions are also set in the uxr_set_custom_transport _callbacks() method.

- transport: custom transport available to every other callbacks
- framing parameter: disabled i.e., packet-oriented mode is selected
- my_custom_transport_open: callback for initialization of transport
- my_custom_transport_close: callback for closing of transport
- my_custom_transport_write: callback for writing the data to transport
- my_custom_transport_read: callback for reading the data to transport

CustomEndPoint in Micro XRCE-DDS Agent

Micro XRCE-DDS Agent can receive and reply the messages to multiple clients. As a result, the Agent must be able to differentiate between the clients, hence should know the information about the source and the destination of the message, referred to as endpoint parameters in DDS terms. These parameters are handled in Agent using the *CustomEndPoint* class. This class has three methods that is used along with the callbacks responsible for reading and writing the data to CAN.

- add_member(): method for adding new member to endpoint definition
- set_member_value(): method for setting the value to the added member
- get_member(): method for getting the value of the added member

To communicate with the Agent, each Client has a unique CAN Identifier. As a result, a variable is added using the add_member() method to hold the value of CAN Identifier. In the callback responsible for reading the data, this variable is set using the set_member_value() method based on the identifier field in the received CAN frame. Correspondingly, in the callback responsible for writing the data, this variable value is fetched using the get_member() method and written in the identifier field in the CAN frame to be sent. This ensures that the Agent receives the response from the Client and replies to the same Client respectively.

CAN Data Communication

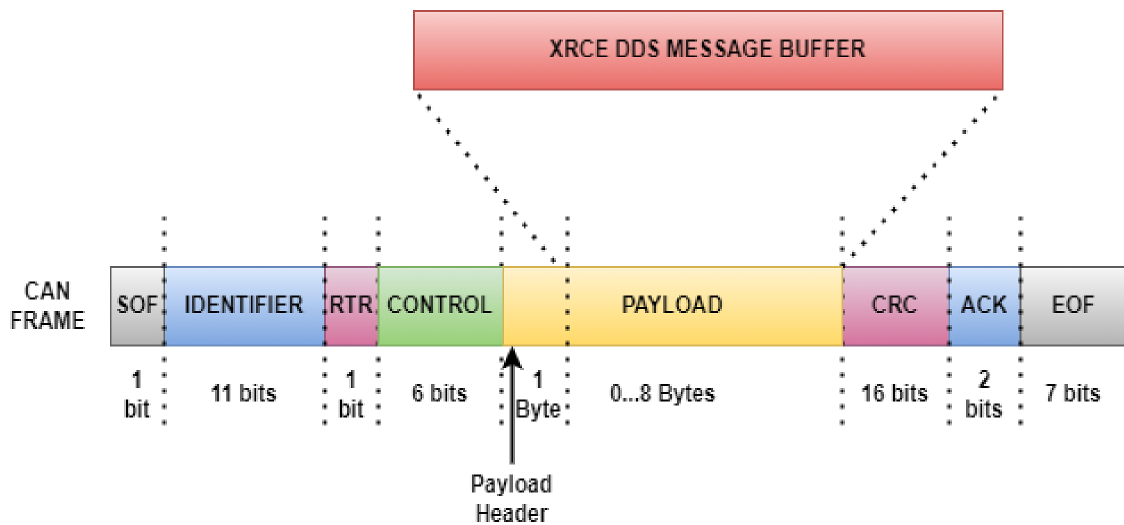

Figure 4.6: Format of standard CAN frame

Value	Description
0x00	Intermediate block of data
0x01	First block of data
0x02	Last block of data
0x03	Single block of data only

Table 4.2: Payload Header value description

Standard CAN Frame with a 11 bit identifier field operated at a frequency of 250 Kbits/sec is used for the communication between Client and Agent

at the transport layer. It supports a payload size of upto 8 bytes. The first byte in the payload of every frame is termed as *Payload Header* and the next 7 bytes are a block of data of XRCE-DDS message buffer as shown in the Figure 4.6. Bit0 of Payload Header represents the first block of data and Bit1 of Payload Header represents the last block of data. Table 4.2 presents the possible 4 values of Payload Header.

CAN Transmission

The data transmission from the middleware layer to the operating system layer via the CAN transport layer is illustrated in the steps below along with the Figure 4.7.

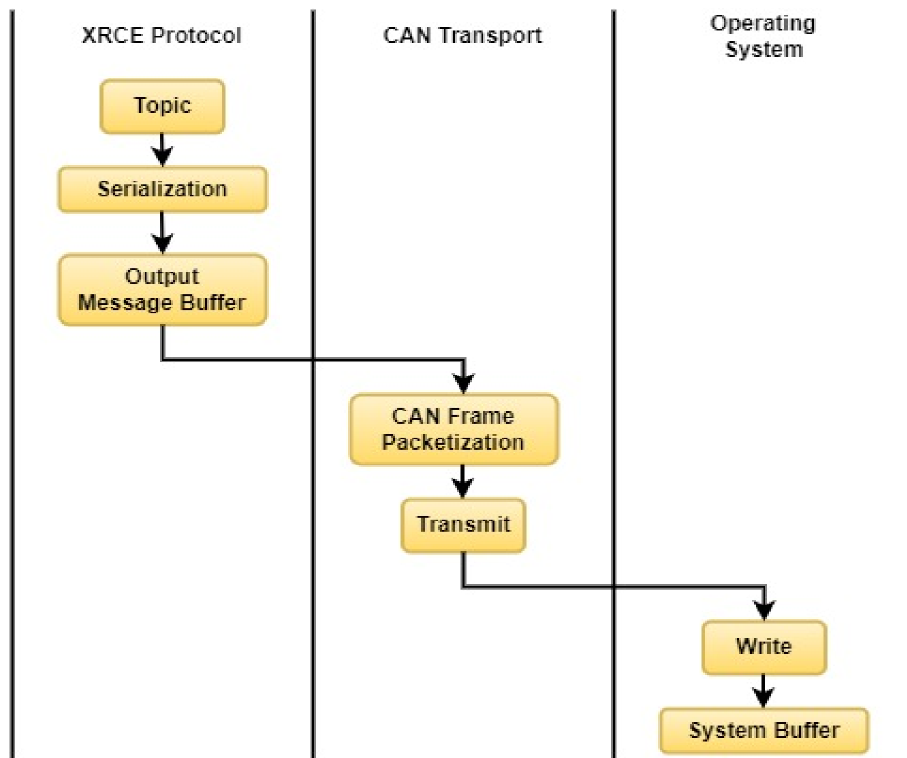

Figure 4.7: CAN data transmission workflow

1. The data to be sent for the given topic is serialized using the MicroCDR library and the XRCE message is added to the output message buffer with all the necessary information.
2. At the transport layer, every CAN frame supports only a payload size of maximum 8 bytes including Payload Header. As a result, if there

is more than 7 bytes of data in the output message buffer, then it can no longer be delivered in a single frame and must be split over many frames. In each frame, 7 bytes of output message buffer (referred to as block) can be transmitted. As presented in the Table 4.2, based on the position of block of data, Payload Header value is appended in the first byte and then the CAN frame is transmitted.

3. Finally, the operating system writes to the system buffer. Thus, the data is transmitted over CAN on the selected channel.

CAN Reception

The data reception from the operating system layer to the middleware layer system via the CAN transport layer is illustrated in the steps below along with the Figure 4.8.

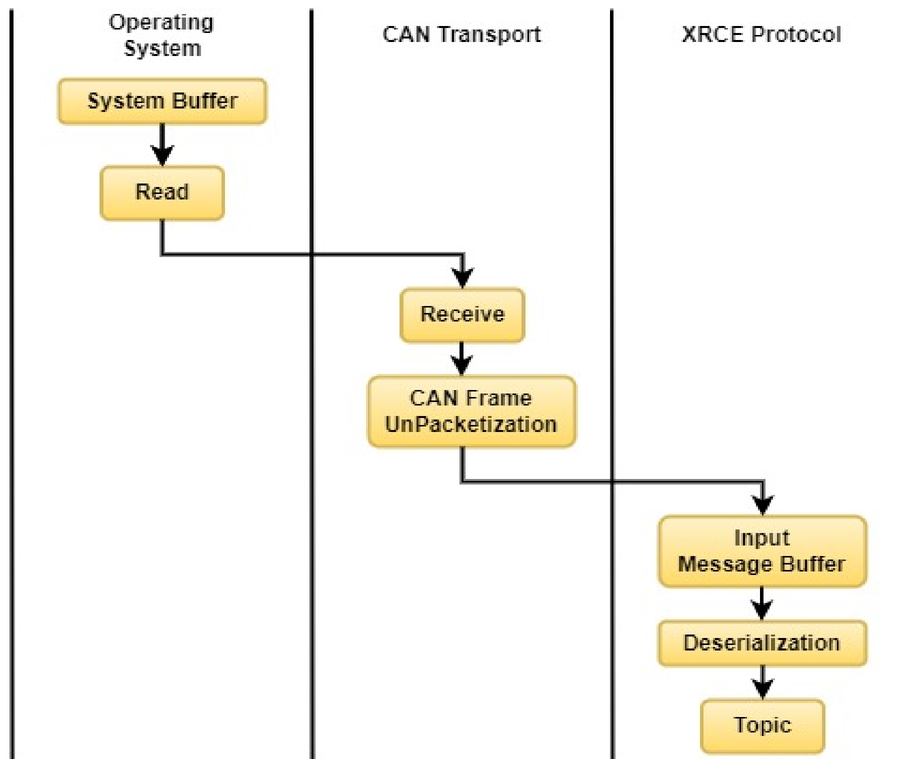

Figure 4.8: CAN data reception workflow

1. The operating system reads the data from the system buffer. The data is received over CAN on the selected channel.

2. At the transport layer, the payload is extracted based on the Payload Header value as explained in the CAN transmission workflow and is further sent to input message buffer.

3. The XRCE message is deserialized using the MicroCDR library and finally the data is extracted for the given topic and is sent to the above layers for further processing.

4.4.3 Other Layers: RCLC, RCL, RMW

As explained in the Section 4.2, for each entity creation in the Client, API calls in RCLC, RCL, RMW layers has to be adapted to handle the request-response mechanism through the state-machine model approach. Global variables are introduced to ensure that request from the Client is sent in one cycle and response from the Agent is read in next cycle to avoid blocking function call. The following code snippet shows the APIs updated in each layer for node entity creation in the Client. Similarly, APIs are updated in each layer for other entities creation as mentioned in Table 4.1.

```
// ---------- RCLC Layer ----------
rcl_ret_t rclc_node_init_default(..)
{
  // initial NULL checks
  // ...
  if(!rclc_node_init_default_initialization)//global variable
  {
    // parameters initialization
    // ...
    rclc_node_init_default_initialization = true;
  }
  rc = rclc_node_init_with_options(..);
  // ...
}
rcl_ret_t rclc_node_init_with_options(..)
{
  // initial NULL checks
  // ...
  if(!rclc_node_init_with_options_initialization)//global
   variable
  {
    // parameters initialization
    // ...
    rclc_node_init_with_options_initialization = true;
  }
  rc = rcl_node_init(..);
```

```
  // ...
}

// ---------- RCL Layer ----------
rcl_ret_t rcl_node_init(..)
{
  // initial NULL checks
  // ...
  if(!rcl_node_init_initialization)//global variable
  {
    // parameters initialization
    // ...
    rcl_node_init_initialization = true;
  }
  node->impl->rmw_node_handle = rmw_create_node(..);
  // ...
}

// ---------- RMW Layer ----------
rmw_node_t * rmw_create_node(..)
{
  rmw_node = create_node(..);
}
rmw_node_t * create_node(..)
{
  // Initial NULL Checks
  // ...
  if(!create_node_initialization)//global variable
  {
    // Parameters initialization
    // ...
    participant_req = uxr_buffer_create_participant_bin(..);
   // Node Entity Request Created
    create_node_initialization = true;
  }
  if (!run_xrce_session(..) // Request Sent
  {
    // Node Entity Response from Agent
  }
  // ...
}
```

Therefore, with the analysis, concepts and implementation details explained in the sections of this chapter, an application developed with the micro-ROS Client framework can be ported on BODAS platform or any other embedded platform running AUTOSAR/OSEK like OS with a simple configuration change related to chosen hardware.

Chapter 5

Performance Modeling and Analysis

This chapter mainly addresses the second and third research question [Q2 & Q3] mentioned in Section 3.4 in three steps. In the first step, we list all the properties that influence the real-time characteristics of micro-ROS application and develop a generic model for micro-ROS independent of the modeling language. In the second step, we decide for one of the standard modeling languages popular in automotive, i.e., AMALTHEA and model the generic model using AMALTHEA. In the third step, we simulate the timing behavior of the AMALTHEA model with a performance analysis tool, in our case, chronSUITE. By this generic approach, one could in the future apply the generic model to different modeling languages and performance analysis tools. The above three steps are realized in the sections of this chapter. In Section 5.1, a generic performance model of an application with micro-ROS framework on AUTOSAR-based platform is explained. In Section 5.2, an introduction to System Modeling with AMALTHEA is explained. In Section 5.3, the modeling of generic performance model with such a modeling language is discussed. The last Section 5.4 explains the performance analysis of models designed in earlier sections of this chapter using simulation tool like chronSUITE.

5.1 Generic Performance Model

Micro-ROS aims at providing a generic robotics framework for the entire microcontroller family as a whole. As explained in the Chapter 1, it is gaining popular in the development of automation applications with advanced features in the automotive sector, especially on the BODAS platforms in the

off-highway market. System designed with these applications must meet the specified deadlines i.e., the response must be obtained within the specified timing constraint. Therefore, enabling advanced automation functions in an application leads to increase in the complexity of development. This demands the model-based performance analysis with report on real-time properties like response time, jitter, deadlines, end-to-end latencies etc in the early stages of development.

As far as micro-ROS is concerned, even with it's popularity and demand, there does not exist a common modeling framework i.e., to describe the *timing characteristics* of micro-ROS application. At system level, one of the most important timing properties is end-to-end latency between perception and actuation. Such latency describes the overall processing between a sensor and an actuator and thus the reaction time of the system for this processing path. Therefore, to obtain such a latency from the modeling perspective, the timing properties related to an application developed with micro-ROS framework on an AUTOSAR-based platform like BODAS has to be investigated. Below is the list of properties that influence the timing characteristics of micro-ROS application on BODAS platform.

- **Number of Processing Cores**
  The performance of the system is directly dependent on the number of processing units on the chosen platform. In order to utilize the hardware platform completely and all the tasks are within specified timing constraint, the allocation of software modules (referred to as tasks) to the available cores is important [84]. The intra-core and inter-core task communication has a significant impact on the system level timing properties.

- **Memory Size**
  Each core in the multi-core system typically has a small amount of local memory (cache) and share a larger amount of RAM with other cores. The duration of communication between the tasks is dependent on whether it must happen through fast local memory or slower shared memory between the cores [84].

- **OS Scheduling Policy**
  The scheduling policy supported by the OS on the chosen embedded platform has a direct impact on the real-time performance of the system. The BODAS platform provides OSEK OS which supports fixed periodic preemptive scheduling scheme.

- **Task Period**
  Depending on how frequent the functions are executed, they are assigned to periodic tasks running at different periods like 1 ms, 10 ms, 100 ms etc.

- **Subscriptions**
  Subscription to a topic is handled by the subscriber within the given node. A callback function is called on every ROS message received on the topic. Hence, the execution time of such a callback function depends on the message size. Thus, the overall response time of such a callback function depends on its execution time and the duration for which it is preempted by other high priority tasks.

- **Publishers**
  Publishing to a topic is handled by the publisher within the given node. A callback function is called to publish the ROS message on the topic at a defined interval, which is handled using timers.

- **Middleware**
  As explained in the Section 4.1, the middleware puts the received data via the defined transport (in our case, standard CAN) into DDS queue. Thus, the timing related to middleware depends on the chosen transport, bit-rate configured, packet format, payload length etc. The executor takes the message from the queue and process the corresponding callback further. As a result, the execution time of the executor depends on the number of callbacks and message size of data in each received topic.

Next, the listed properties has to be grouped, so that it would help us to know the minimum number of models that would be necessary describing the timing behavior, like response times, latencies and jitter of an application with micro-ROS framework on an AUTOSAR-based platform like BODAS Controller. Therefore, Figure 5.1 depicts an overview of necessary individual models that we developed with the main objective to abstract the execution timing behavior of the entire middleware into a defined execution time.

- **Application Model**: The properties that the user is allowed to define.

- **Generic micro-ROS Model on BODAS**: The properties that is specific to micro-ROS stack.

- **Hardware Model**: The properties that is specific to hardware platform

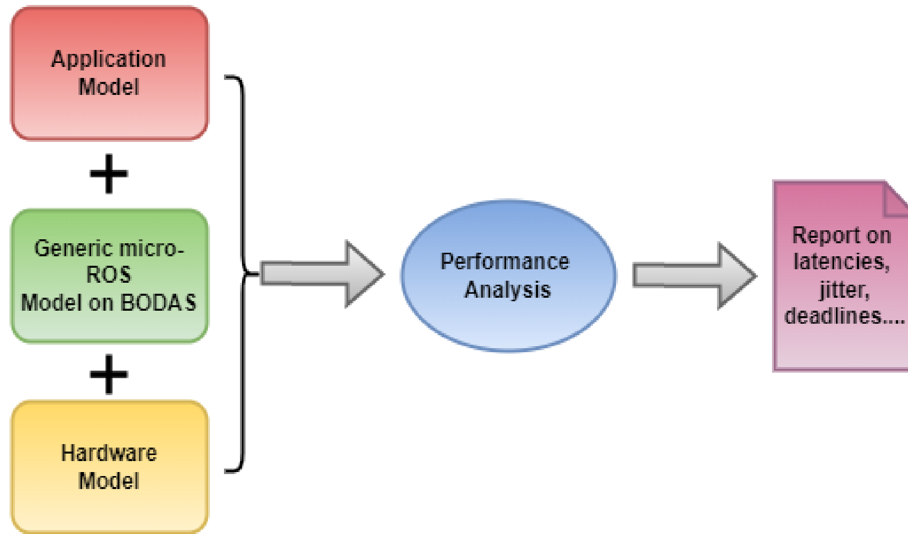

Figure 5.1: Abstract overview of models

From the properties listed above, Number of Processing Cores and Memory Size depends on the chosen platform and hence it could be associated with the *hardware model*. For our application with micro-ROS framework on BODAS platform, OS Scheduling Policy and Task Period is defined and hence it could be associated with the *generic micro-ROS model on BODAS*. From the application developer perspective, the number of Subscriptions and Publishers based on the application designed must be known in advance and hence it is associated with the *application model*. The Execution Time (Callback) parameter for the Subscriptions, Publishers, Middleware properties has to be measured by profiling methods on the chosen platform. Hence, it is dependent on the hardware model. As explained in the Section 4.3, based on the execution time, the Frequency parameter could be mapped to corresponding period in the Task Period. The Message Size parameter depends on the data in the topic. Finally, from the modeling perspective, the Middleware property that also includes executor function, is mandatory in every micro-ROS application and hence it is associated with the generic micro-ROS model on BODAS. Figure 5.2 depicts all the three models with their associated properties and relations between them.

Configuration Parameters in the Middleware

In the RMW_microxrcedds, most of the configuration related to memory management is carried out. An application developer should be aware of such a configuration before trying to model or design an application with

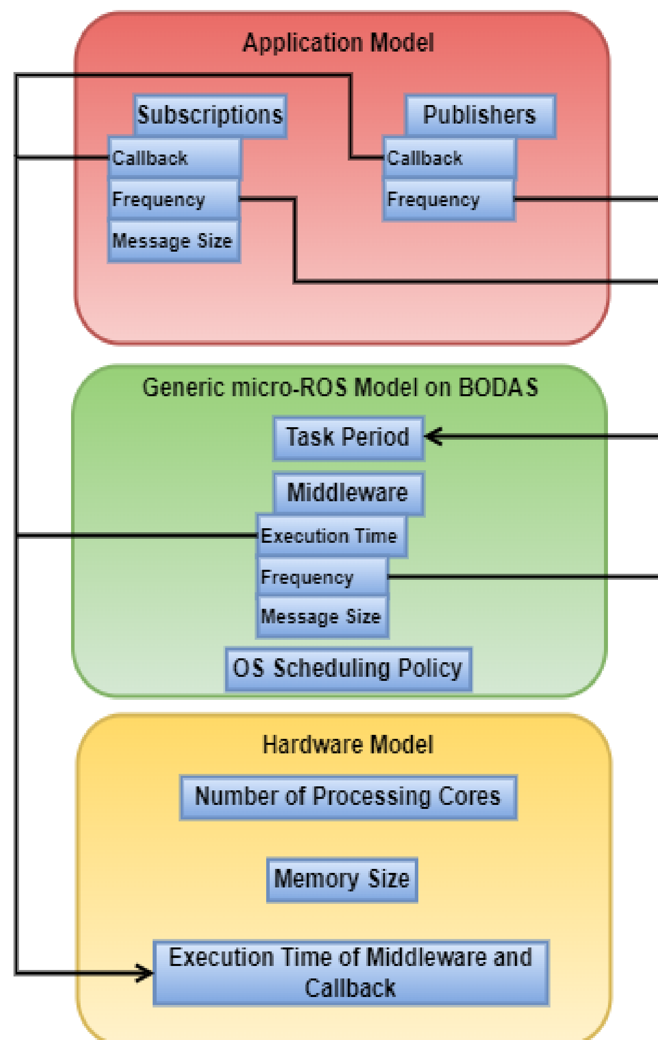

Figure 5.2: Generic Performance Model

micro-ROS framework. Some of the properties mentioned above are also directly dependent on these parameters. Some of the parameters are [85]:

- RMW_UXRCE_MAX_SESSIONS: Number of sessions supported
- RMW_UXRCE_MAX_NODES: Number of nodes supported
- RMW_UXRCE_MAX_TOPICS: Number of topics supported
- RMW_UXRCE_MAX_PUBLISHERS: Number of publishers supported
- RMW_UXRCE_MAX_SUBSCRIPTIONS: Number of subscribers supported

- RMW_UXRCE_MTU: MTU is middleware transport dependent and creates an internal buffer of memory block to store the messages [86]
- RMW_UXRCE_STREAM_HISTORY: Number of MTUs, including input and output buffers

5.2 System Modeling with AMALTHEA

In the automotive industry, during the development of software on automotive control units, modeling the system benefits from reduced hardware costs, faster time to market, higher quality systems and rapid adoption [87]. To support advanced features and functionalities in any application, modern cars also need to consider the system with multi-core architectures, which means the hardware is more complex and heterogeneous. The increase in the complexity of algorithms in these applications on such platforms is forcing the designers to consider the model-based performance analysis at the early stages of development. As explained in the Section 3.2.1, few of the popular standards available in the automotive industry to develop models are ASAM MDX, AUTOSAR, and AMALTHEA. Among these, *AMALTHEA* stands out as a qualified supplement, especially when it comes to the modeling on platform with multi- and many-core systems.

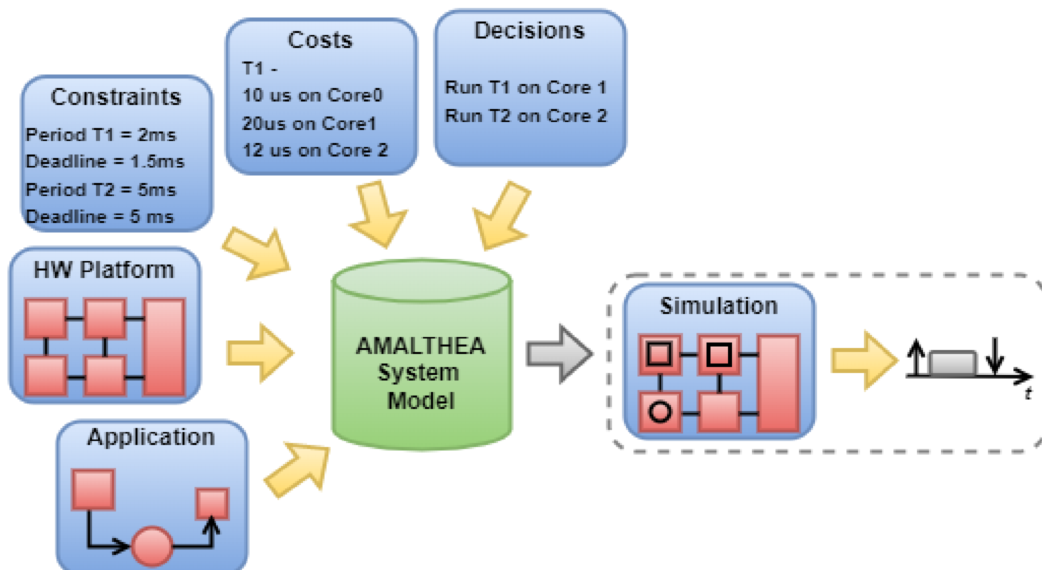

Figure 5.3: Overview of AMALTHEA system model [88]

APP4MC [88] is an open-source Eclipse platform that provides AUTOSAR compliant common data models namely AMALTHEA, whose system model combines various partial models which includes hardware model, software model, timing-constraint model etc. It includes processing tools for basic visualization and validation of these models. It supports interoperability, extensibilty and plugins for code generation from the models [89]. Thus, the system model contains all the information needed to run a performance simulation, which results in the system's timing behavior, which may then be analyzed to help with system optimization. Figure 5.3 depicts an overview of models which combine to form AMALTHEA System Model, which could be given as input to simulation tools for performance analysis of the system.

5.2.1 AMALTHEA Data Models

AMALTHEA, an open tool platform for automotive embedded-system engineering, mainly focuses on design, implementation and optimization of software for multi-core systems based on model-driven methodology [90]. It describes the distributed system on different level of details with the help of sub-models. Figure 5.4 depicts an overview of AMALTHEA sub-models and how they are referenced to each other.

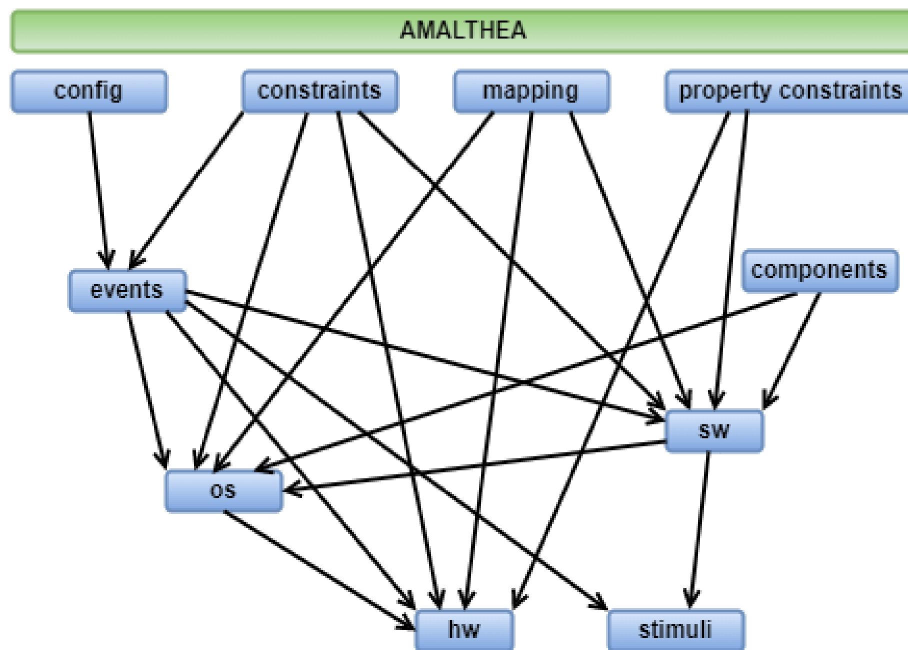

Figure 5.4: AMALTHEA data models
[91]

The AMALTHEA model is mainly divided into ten data models, where each model covers a specific aspect of the system which is considered under development. An overview of each data model is given below [65][91].

- **Components Model**
  It is the model which is comprised of elements, referred to as components that describe the system at the top-level like a black box with ports, composites. These elements, which can be instantiated multiple times are connected using component interfaces. They also help in exchanging information during early phases of development.

- **Software Model**
  It is the model which is comprised of elements that describe the functional behaviour of software in the system, that mainly includes processes, process functions, ISRs, variables etc. The elements of this model are referred to as either task, runnable or label. Process is generally referred to as task, process function as runnable. A task can consist of one or more runnables. Runnables to be executed at a certain period are mapped to the task running at that period. Tasks and runnables have ticks and counters as sub-elements to specify their execution time. The number of ticks is the number of cycles of processing unit. Counters help to describe the activation of a task or runnable element to happen only every n^{th} time. Label is simple data element like variable located in a defined area of a given memory. It has data type and size as sub-elements to represent the type and size of data element received or sent by a task/runnable instance.

- **Hardware Model**
  It is the model which is comprised of hierarchical elements, that describes the entire hardware in the system, that mainly includes ECUs, microcontrollers, processing units (number of cores), memory information, connections, additional peripherals etc. At the top-level, it contains definitions, domains, features and structures as elements. Definition element contains ProcessingUnitDefinition, MemoryDefinition as sub-elements.

- **Operating System Model**
  It is the model which is comprised of elements that give an abstract information about the underlying operating system with the schedulers, scheduling algorithm, buffers, semaphores, resources etc supported on it. It contains task scheduler as one of the elements. Task scheduler contains scheduling algorithm as sub-element.

- **Mapping Model**
  It is the model which is comprised of elements, which mainly contains the information about mapping of elements among hardware model, software model and operating system model. It allows to define the:

  1. mapping of tasks to schedulers
  2. scheduling parameters like priority, based on mapping
  3. mapping of data elements such as variables to memory modules

- **Constraints Model**
  It is the model which is comprised of elements that describe the constraints mainly related to the timing of software, execution order of functions, the affinity of functions to each other etc.

- **Property Constraints Model**
  It is the model that is used to constrain the design space by providing information about the specific hardware properties required by certain functions, which are necessary for allocation or target mapping from a software perspective.

- **Stimuli Model**
  It is the model which is comprised of elements that describe stimulus and clock objects. Some of the available stimulus are periodic stimulus, event stimulus, inter-process stimulus etc.

- **Event Model**
  It is the model which is comprised of elements, referred to as events that describe mainly the event chains. Event chains are useful in defining the specific task-chain sequence that help in tracing configuration, timing constraints of entities like processes, labels, runnables, end-to-end latencies etc.

- **Configuration Model**
  It is the model which is comprised of elements, that contain definitions and configuration information mainly related to simulation, hardware tracing, building system model etc.

5.3 Generic Performance Model using AMALTHEA

This section explains the modeling of generic performance model that includes hardware model, generic micro-ROS model on BODAS and application model with their associated properties and dependencies designed in

Section 5.1 using available AMALTHEA Data models. Figure 5.5 depicts the realization of generic performance model using data models in AMALTHEA.

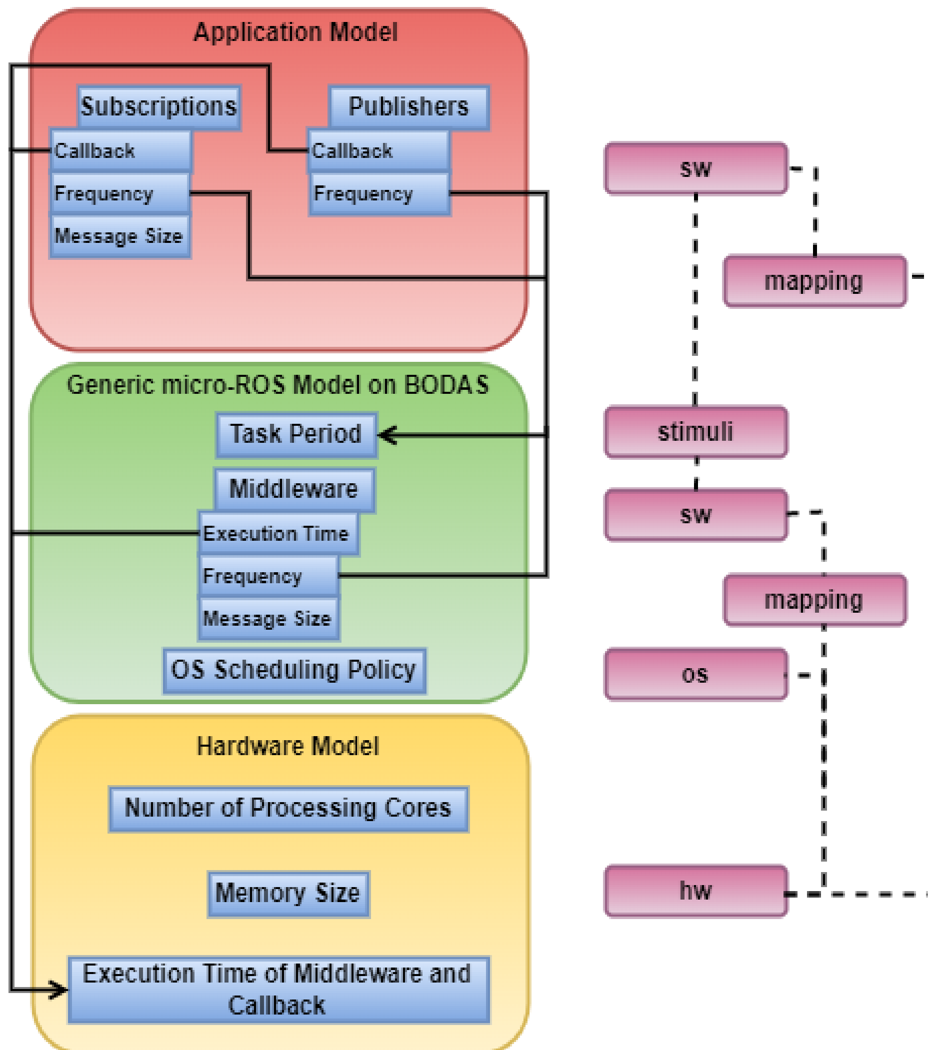

Figure 5.5: Generic Performance Model using AMALTHEA data models

5.3.1 Hardware Model

Hardware model, which is part of the generic performance model could be realized using hardware data model in AMALTHEA with Number of Processing Cores and Memory Size properties described using definition element.

Other hardware details like clock frequency, microcontroller etc could be described using respective elements. Figure 5.6 depicts the visualization of a simple hardware data model with single core on a microcontroller described in AMALTHEA.

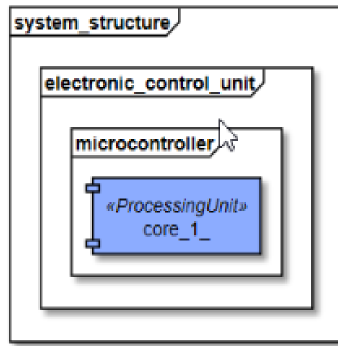

Figure 5.6: Hardware data model

5.3.2 Generic micro-ROS Model on BODAS

Generic micro-ROS Model on BODAS, which is part of the generic performance model can be modeled in a generic way so that this model could be used for all kinds of micro-ROS-based applications on BODAS platform. Thus, each and every property belonging to this model is realized using specific data models in AMALTHEA.

The Task Period property could be described using stimuli data model with 1 ms, 10 ms, 100 ms etc as PeriodicStimulus sub-elements. Next, the OS Scheduling Policy property could be described using operating system data model with the fixed periodic preemptive scheduling algorithm sub-element. Finally, the Middleware property that also includes executor function could be described using software data model with task, runnable and label as the elements. The Execution Time parameter could be described using ticks and if the execution time is varying every n^{th} interval, it could be described using counters. The Frequency parameter could be mapped to respective PeriodicStimulus sub-element of stimuli data model(1 ms, 10 ms or 100 ms) depending on how frequent the task or runnable should be executed. The Message Size parameter could be described using label element with size as sub-element depending on the topic. Figure 5.7 depicts the visualization of a simple software data model with a single task consisting of three runnables and two labels described in AMALTHEA.

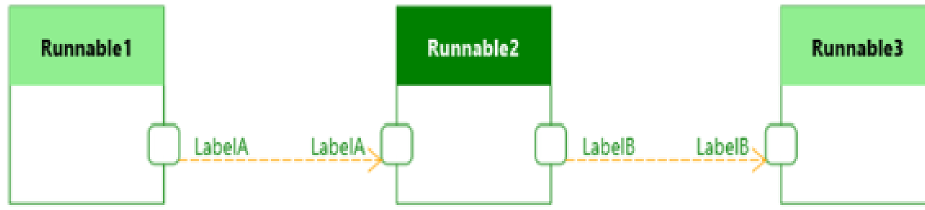

Figure 5.7: Software data model

5.3.3 Application Model

Subscriptions and Publishers properties belonging to application model, which is part of the generic performance model could be described using software data model in AMALTHEA. The Execution Time, Frequency and Message Size parameters could be described using respective elements belonging to data models as explained in the last paragraph.

Therefore, from the modeling perspective, with the analysis, concepts and realization details explained above, a system model depicting an application developed with the micro-ROS framework on a embedded platform like BODAS Controller running AUTOSAR/OSEK like OS could be developed using data models in AMALTHEA.

5.4 Performance Analysis using chronSUITE

In Section 3.2.2, few of the popular standard tools available for timing and performance analysis in the automotive industry are mentioned. Among these, the proprietary tool *chronSUITE* from Inchron stands out as a well-researched supplement, mainly because it supports the analysis of complex task dependencies and heterogeneous, distributed architectures and also an importer called *am2inc* [92] developed in its recent version, which reads the AMALTHEA system model and directly convert them into chronSUITE timing model which could be further used for performance analysis.

Inchron [93], an active member of the AUTOSAR consortium provides a wide-variety of tools for the developers, testers, integrators and system architects to excel in real-time properties throughout the entire real-time system development life cycle. Its start-of-the-art methods enable the analysis, design, development, testing and optimization of complex embedded applications, thus predicting its quality, costs and time to market [94]. Among

the products supported from Inchron, chronSUITE stands out as a leading toolkit for building efficient and reliable embedded real-time systems. Some of its key features are [95][96]:

- Modeling of complex systems with EventSequences and DataFlows
- Simulation of many and multi-core systems on a chip (SoC) and memory bus congestion effects
- Interactive trace visualization and automated timing and performance requirements verification
- Importer for APP4MC AMALTHEA system models

Thus, the system model developed as explained in the previous section using AMALTHEA data models could be given as input to chronSUITE using am2inc importer. The importer, available under the Eclipse Public License 2.0 [97], helps in direct conversion of AMALTHEA system model to chronSUITE timing model. Further, the timing model could be simulated and a detailed report on real-time properties like response time, jitter, preemption time and slack time of the tasks, end-to-end latencies between specified task-chains, deadlines, CPU load, RTOS failure etc could be obtained.

Chapter 6

Evaluation

In this chapter, we evaluate the effectiveness of the approach as discussed in Chapter 4 and Chapter 5 regarding modeling and timing analysis by a micro-ROS test application on a BODAS RC18-12/40 Series Controller as mentioned in Section 1.2. This chapter also addresses the fourth research question [Q4] mentioned in Section 3.4. In Section 6.1, a micro-ROS test application is explained. In Section 6.2, the modeling of same application using the approach from Chapter 5 is explained. In Section 6.3, the test setup and results obtained through the implementation on the BODAS platform and through performance analysis of the developed model using the simulation tool are compared and discussed. In addition, the test setup and timing results obtained with external hardware are also discussed.

6.1 Micro-ROS Test Application

As mentioned in the Section 1.2, an application with the micro-ROS framework has been implemented on a BODAS RC18-12/40 Series Controller as shown in the Figure 6.1 with the initialization phase, operation phase and implementation details as explained in Chapter 4.

As a use-case, in a micro-ROS test application considered, an executor, which is initialized with the node, is configured to handle two callbacks. One of the callbacks is referred to as *Control Function* (timer callback) which reads the input(i) test signal, multiplies the obtained value with the factor(f) and the result is seen at the output(o) as shown in the Figure 6.2. This callback is associated with a timer which is created and added to the executor and it is responsible to call the callback periodically (in our case, 1 kHz period). This timer callback is referred to as *Task B*. The factor(f) is to be updated

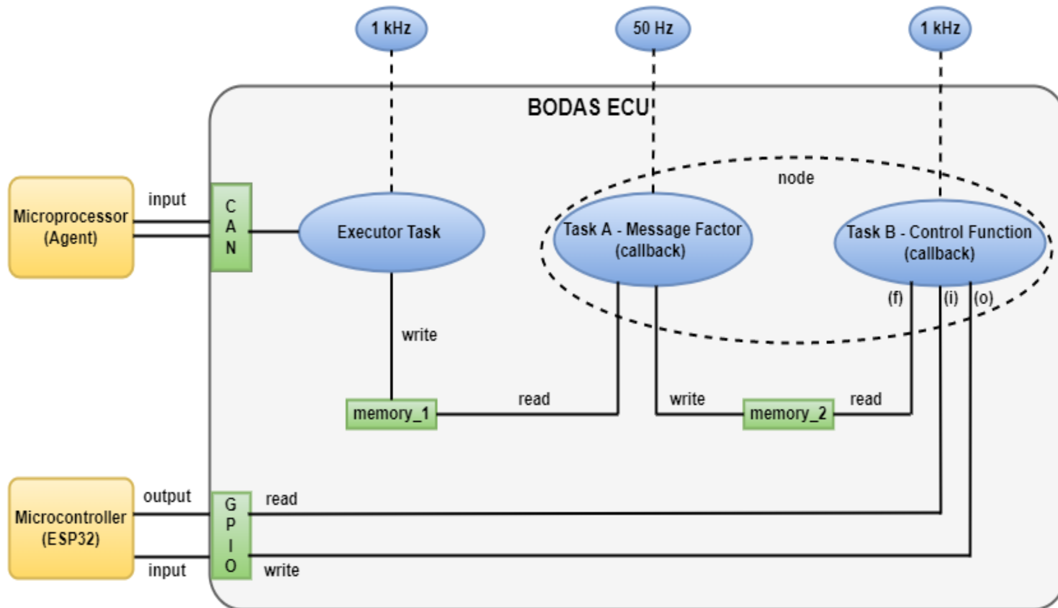

Figure 6.1: Micro-ROS test application on BODAS RC18-12/40

at a certain interval from the Agent (in our case, 50 Hz). The other callback, referred to as *Message Factor*, which handles the updated factor, is mapped to a task running at 50 Hz and is referred to as *Task A*. Finally, as explained in Section 4.3, the executor, which serves the middleware and notifies the callbacks, should be running at 1 kHz and is referred to as *Executor Task*.

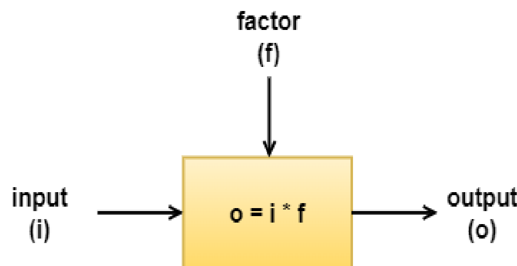

Figure 6.2: Control Function

The micro-ROS Agent, which is responsible for the micro-ROS Client on BODAS platform talking to the DDS world, is running on a microprocessor (in our case, a Lenovo T15 Laptop with Intel i7 CPU running Ubuntu 20.04 Linux in a VM under Windows 10). The communication between Client and Agent is through standard CAN interface, configured based on custom transport profile as explained in Section 4.4.2. The Agent sends the data containing information on the factor at a regular interval (50 Hz) and the Executor

Task receives this data and writes to a memory location (memory_1). Then the Message Factor Task A reads from this memory location and updates to a memory location (memory_2) from which the Control Function TASK B reads it and it modifies the signal read through GPIO input pin as per the factor received. Finally, this modified signal is written through GPIO output pin. From the hardware perspective, the input test signal is considered from an externally connected ESP32 microcontroller [98] to BODAS ECU and the signal is read through a GPIO input pin by the Control Function callback and this signal is modified as per the factor and is written as output signal through a GPIO output pin, which is read by ESP32 microcontroller. Thus, ESP32 also helps in signal validation according to the factor.

6.2 Modeling of micro-ROS Test Application

The system model depicting a micro-ROS application on a BODAS RC18-12/40 Series Controller as explained in previous section has been modeled using AMALTHEA data models as shown in the Figure 6.3 with the modeling concepts and realization details as explained in the Chapter 5.

The hardware data model is built for single-core with clock frequency of 1 MHz and is not built with the hardware specifications of BODAS as explained in Section 2.4.1. In the operating system data model, fixed periodic preemptive scheduling scheme is selected as the sub-element for the task scheduler element. In the stimuli data model, 1 ms and 20 ms periods are defined using respective periodicstimulus elements. In the software data model, three tasks are defined using respective task elements. The execution time of each task is given as a measure of number of ticks. Since the clock frequency is considered as 1 MHz, the value of 1 tick would be equal to 1μs. The message received externally at a certain interval regarding factor is defined as a runnable with its interval defined using counter sub-element. In the mapping data model, task priorities, mapping of three tasks to scheduler and scheduler to core are defined. Executor Task has the highest priority, Control Function Task has medium priority and Message Factor Task has the lowest priority.

The system model is successfully validated for INCHRON specifications in APP4MC. This validation ensures the direct conversion of system model in AMALTHEA into timing model in chronSUITE using am2inc importer.

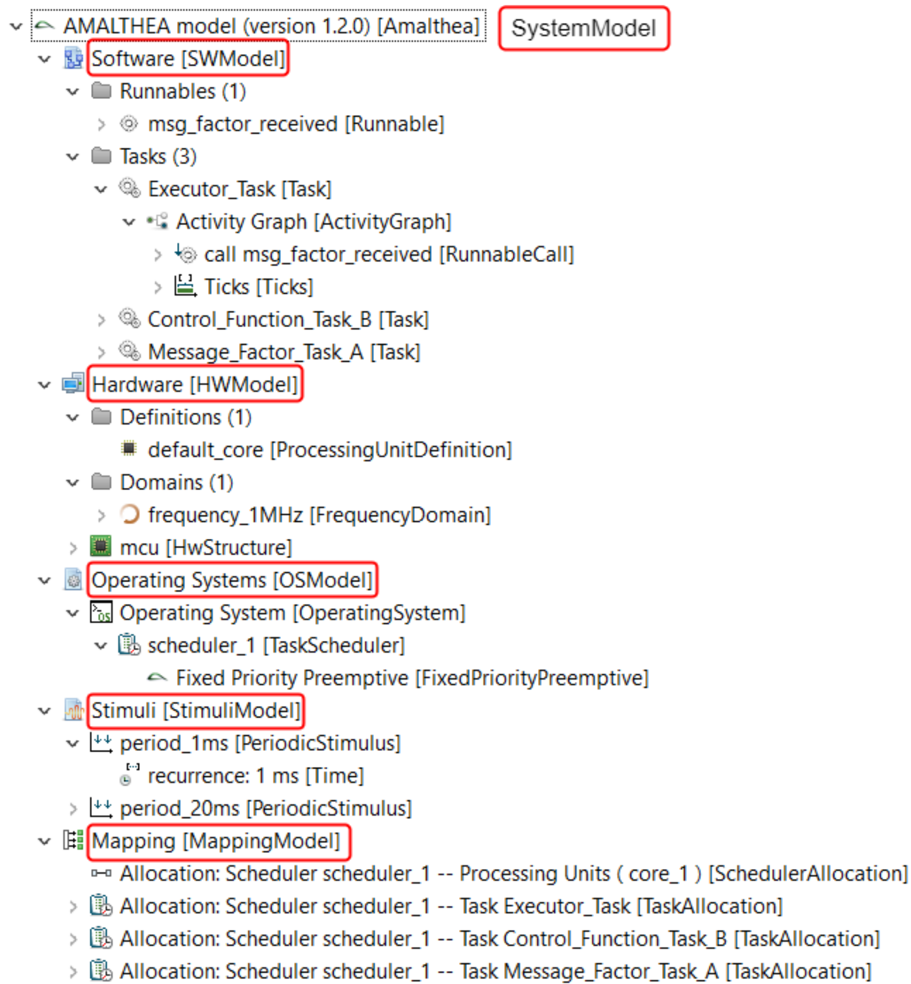

Figure 6.3: Micro-ROS test application model in AMALTHEA

6.3 Test Experiments

6.3.1 Initialization Phase

Test Setup

During the initialization phase, on the Client side, once the entity creation request is sent, it also sends a CAN message via the debug interface CAN channel with the identifier 0x180000AAA. On the Agent side, it receives the request and generates an appropriate response. In the next iteration, once the successful response from the Agent is acknowledged, the Client sends a CAN message via the debug interface CAN channel with the identifier 0x18000BBB.

Result

Figure 6.4 depicts the result of micro-ROS initialization phase implemented using the state-machine model approach. It shows the request-response mechanism for entities creation handled sequentially on the micro-ROS Client (left) and micro-ROS Agent (right).

Figure 6.4: Micro-ROS Initialization Phase

6.3.2 Operation Phase

As explained in the Section 6.1, the executor and the callbacks are mapped to respective tasks running at certain interval. The result obtained on the BODAS hardware regarding response time of the Control Function via debug interface CAN channel is compared to the simulation result obtained using chronSUITE for two approaches. Then the result regarding latency and jitter of the Control Function is calculated with another experimental setup, i.e., using externally connected device (in our case, ESP32) to BODAS. Finally, along with this setup, the result obtained on the BODAS hardware regarding end-to-end (E2E) latency of a specified task chain are compared to the simulation result obtained using chronSUITE. All the experiments have a run-time for around 5 minutes. The data collected via debug interface CAN channel and ESP32 are given as input to the tool respectively, which is developed using python and it helps to generate necessary statistics information regarding timing properties of the respective task.

Approach 1: Simple Model

Test Setup
In the *first approach*, the minimum, average and maximum execution time of each task in isolation are measured using the BODAS API, which offers a built-in function for task run-time measurements and sent via debug interface CAN channel every second. For the Executor Task, single execution time is considered irrespective of its variation in execution during the reception of message from the Agent. In the next step, similar to execution time, the response time of the Control Function, which is executed as timer callback, is measured using the BODAS API and sent via debug interface CAN channel every second. From the modeling perspective, a *simple model* of the application is developed with a single execution time for the Executor Task and the average execution time of other tasks, which are given as input respectively to the tasks in AMALTHEA. Finally, the result obtained on the BODAS hardware regarding response time is compared with the simulation result obtained using chronSUITE with the timing model generated from the simple model of application as input.

Result

	Executor Task (1 kHz)	**Message Factor Task (50 Hz)**	**Control Function Task (1 kHz)**	**Executor + Control Function (1 kHz)**
min net time [µs]	25	25	25	25
avg net time [µs]	53.08	32.2	45	**68.83**
max net time [µs]	92	41	73	100
input to model (execution time)	12	2	4	**16 (output of model)**

Table 6.1: Results based on Simple Model

Table 6.1 gives the information on minimum, average and maximum execution time of each task measured using BODAS API in isolation and the last column gives the response time of the Control Function (implemented as a timer callback by an executor). As a baseline, initially, the average execution time of an empty 1 kHz task and 50 Hz task was measured and it was found to be around 41 µs and 31 µs respectively (includes context switching time

etc). Hence, for the model, the execution time of the tasks (excluding baseline execution time) in the form of ticks are given as input as per last row in the Table 6.1. Figure 6.5 depicts the response time of the Control Function obtained using chronSUITE tool. Adding the baseline execution time, the overall value would be 57 µs.

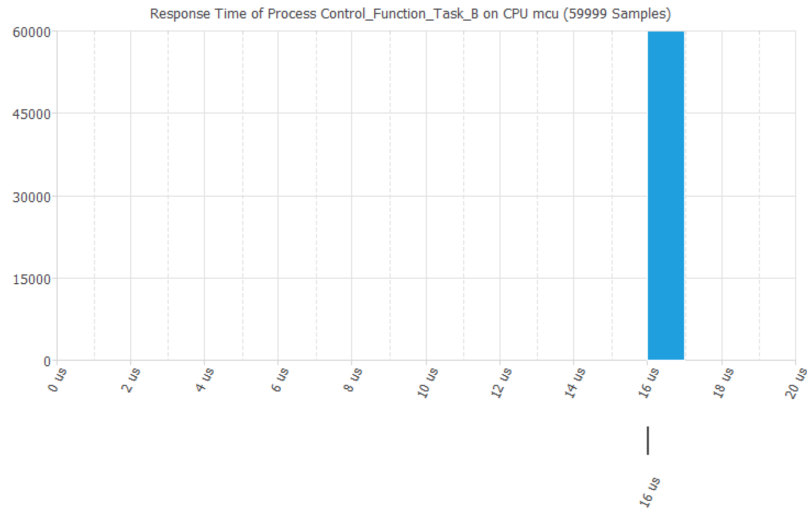

Figure 6.5: Response Time of Control Function using Simple Model

From the results, it is observable that there is an increase of around 16.5% in the response time value measured on the BODAS hardware. This is because the Control Function callback is not running independently on the BODAS hardware i.e., it is executed as a timer callback. Whereas from the modeling perspective, it is realized as an independent task without calling a timer. Therefore, the difference is because of the additional overhead due to timer for the Control Function Task in the executor, which is not considered in the model.

Approach 2: Advanced Model

Test Setup

The *second approach* is similar to the first approach except that, for the Executor Task, two execution times are considered, i.e., one during message received from the Agent and one during message not received from the Agent. Accordingly, an *advanced model* of the application is developed with two execution times for the Executor Task. In this approach, since the model is developed in detail regarding the micro-ROS stack (Executor Task), it is

expected that it predicts the response time of the Control Function better than the first approach.

Result

	Executor Task without message received	**Executor Task with message received**	**Executor + Control Function**
min net time [µs]	25	25	25
avg net time [µs]	53	66.3	**68.83**
max net time [µs]	53	73	100
input to model (execution time)	12	13	**16.65 (output of model)**

Table 6.2: Results based on Advanced Model

Table 6.2 gives the detailed information on the execution time of Executor Task if a message is not received from the Agent and if a message is received from the Agent. On an average, its execution time is increased by 13 µs every 20 ms when it receives the message from Agent. From the modeling perspective, this additional execution time for the Executor Task is modeled using counter sub-element as explained in the Section 6.2. Other inputs to the model are same as the earlier approach. Figure 6.6 depicts the response time of the Control Function obtained using chronSUITE tool. Adding the baseline execution time, the overall value would be 57.65 µs.

From the results, it is observable that on an average there is an increase in the response time of the Control Function by 0.65 µs. This is because every 20 ms there is an increase in the execution time of Executor Task due to the reception of message from Agent. In our case, since the message size is very small, there is not significant increase in the execution time of Executor Task when it receives the message. But in general, the model predicts the better response time of the tasks/functions if it is given the detailed information on the execution time of Executor Task.

Determining Latency and Jitter of the Control Function

Test Setup
Figure 6.7 depicts the setup of BODAS ECU and ESP32 and Table 6.3 gives

Name	Min	Avg	Max
GenericSystem			
RtosConfig			
IsrScheduler			
scheduler_1			
Executor_Task			
Metrics			
Load		1.27%	
Net Execution Time	12.000 us	12.650 us	25.000 us
Gross Execution Time	12.000 us	12.650 us	25.000 us
Response Time	12.000 us	12.650 us	25.000 us
Preemption Time	0 us	0 us	0 us
Initial Pending Time	0 us	0 us	0 us
Start to Start Time	1.000 ms	1.000 ms	1.000 ms
Slack Time	975.000 us	987.349 us	988.000 us
Functions			
Control_Function_Task_B			
Metrics			
Load		0.40%	
Net Execution Time	4.000 us	4.000 us	4.000 us
Gross Execution Time	4.000 us	4.000 us	4.000 us
Response Time	16.000 us	16.650 us	29.000 us
Preemption Time	12.000 us	12.650 us	25.000 us
Initial Pending Time	12.000 us	12.650 us	25.000 us
Start to Start Time	987.000 us	999.999 us	1.013 ms
Slack Time	971.000 us	983.349 us	984.000 us

Figure 6.6: Response Time of Control Function using Advanced Model

the GPIO details of the same.

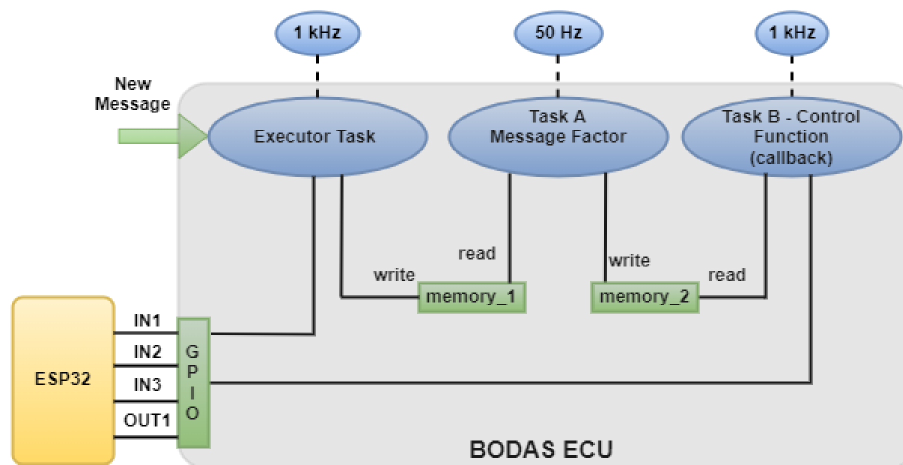

Figure 6.7: BODAS-ESP32 setup

	BODAS	**ESP32**
Input PINs	A43	17 (IN1), 16 (IN2), 4 (IN3)
Output PINs	K86, K85, K84	22(OUT1)

Table 6.3: Pin connections between BODAS and ESP32

The latency of the Control Function is calculated as time interval for the Control Function Task to write the value read from OUT1 to IN3 (with respect to ESP32). t_1 and t_2 are the defined start time and end time on ESP32.
t_1 = write(OUT1)
t_2 = read(IN3)
Time interval i.e., latency measurement on ESP32 = t_2 - t_1
The jitter of the Control Function is calculated as the expected value of the squared deviation from the average duration value.

Result

	Control Function without micro-ROS stack (1 kHz)	**Control Function with micro-ROS stack (1 kHz)**
min latency [µs]	1443	1505
avg latency [µs]	7192	7418
max latency [µs]	39919	38114
jitter [µs]	3787	4100
CPU load	25.62%	28.3%

Table 6.4: Measurement on ESP32

Table 6.4 gives the information on the latency as well as the jitter of the Control Function with and without micro-ROS stack. The last row also gives the information on the CPU load. Figure 6.8 depicts the histogram of latency of the Control Function without micro-ROS stack running in a 1 kHz task. Figure 6.9 depicts the histogram of latency of the Control Function with micro-ROS stack running in a 1 kHz task.

From the results, it is observable that the numbers related to latency and jitter calculated on ESP32 are quite high and distributed than the goals defined in the Section 1.2 because there was unexpected high delays observed in the I/O subsystem of BODAS hardware. This was verified by running

Control Function without micro-ROS stack in different tasks on BODAS hardware like 1 kHz, 50 Hz etc. It was also verified with simple I/O tests by connecting the ESP32 outputs directly with its inputs that there was no delay on ESP32 side. Regarding CPU load, it is observable that there is an increase of around 3% due to the micro-ROS stack.

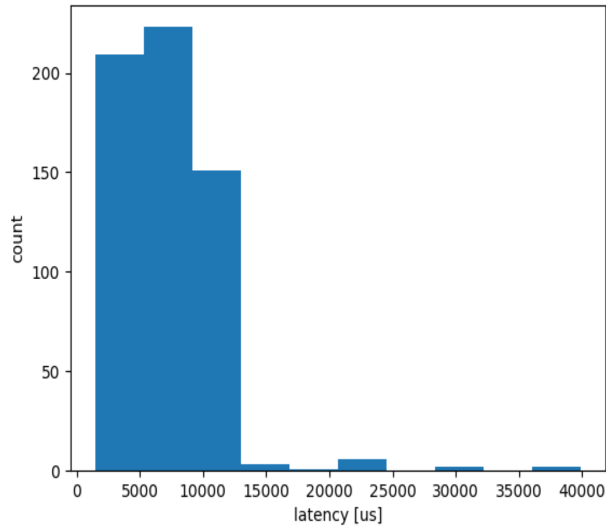

Figure 6.8: Distributed latency of Control Function without micro-ROS stack on ESP32

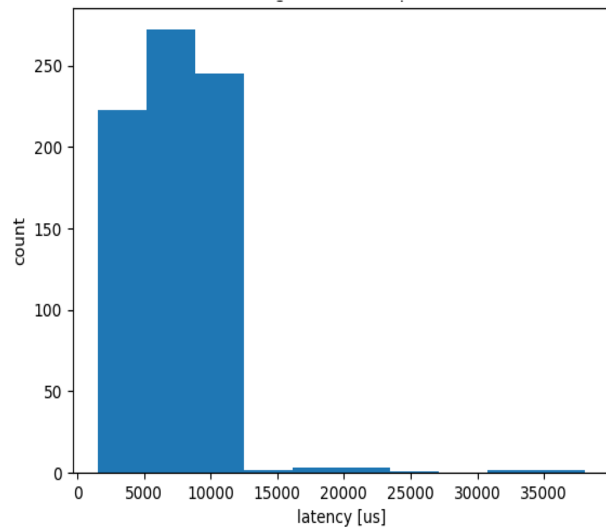

Figure 6.9: Distributed latency of Control Function with micro-ROS stack on ESP32

Determining E2E Latency of a Task Chain

Test Setup

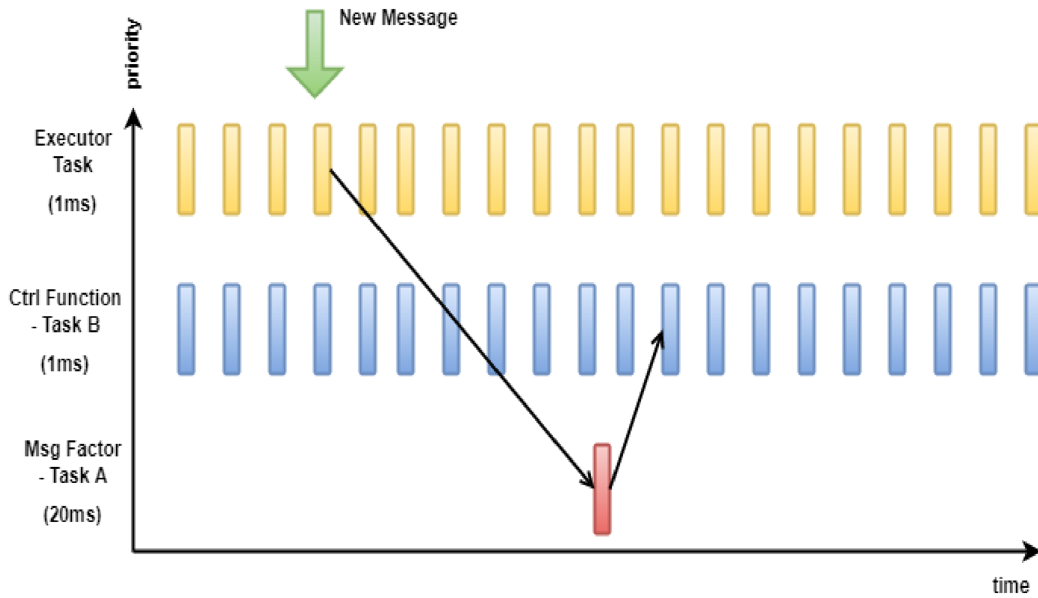

Figure 6.10: Specified task chain

The end-to-end latency of a task chain is calculated as time interval from the incoming new message (factor) received by the Executor Task to update of signal by the Control Function Task at the output as per the factor reception. Figure 6.10 depicts the specified task chain as per the micro-ROS test application on BODAS. To observe the minimum, average and maximum E2E latency of a task chain, the message from Agent is sent in a interval of (20 ms + offset). This offset varies from 0 to 20 ms.

Measurement on BODAS

As soon as the new message is received by an Executor Task, it sends the timestamp (start time) via the debug interface CAN channel. Then, once the Control Function updates the input signal as per the new factor, it sends the timestamp (end time) via the debug interface CAN channel. The difference between these two timestamps gives the E2E latency of a task chain.

Measurement using Simulation Tool (Model)

Figure 6.11 depicts the timing model in chronSUITE which is the result of the system model imported using am2inc importer. In this thesis work, from the modeling perspective to predict the end-to-end latency of a task

chain, event sequence is not defined using event data model in AMALTHEA, because currently the importer am2inc doesn't support the automatic conversion of AMALTHEA event chains. Hence, the event sequence as shown in the Figure 6.10 is defined in the timing model in chronSUITE. Also, the distributed message reception from the Agent could not be realized in the timing model in chronSUITE and hence only the minimum and maximum E2E latency of a task chain are calculated.

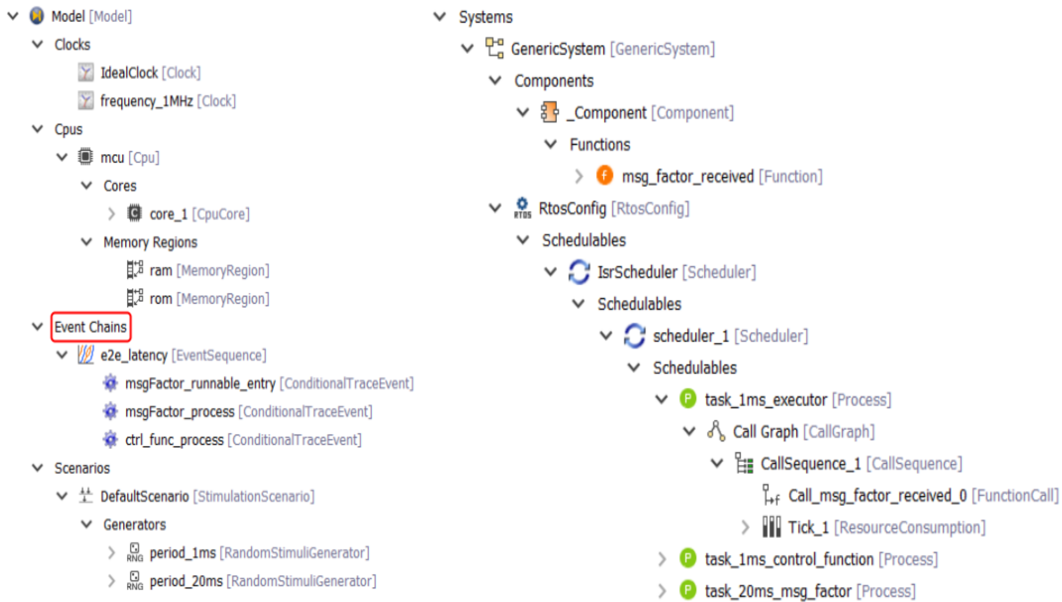

Figure 6.11: Timing Model in chronSUITE

Measurement on ESP32

t_3 and t_4 are the defined start time and end time on ESP32 i.e.,

t_3 = read(IN1) , Executor Task updates pin as soon as new message is received

t_4 = read(IN2), Control Function Task updates pin after the update of signal as per the factor reception

Time interval i.e., E2E latency on ESP32 = t_4 - t_3

Result

Figure 6.12 depicts the histogram of E2E latency of a task chain calculated from the measured values on BODAS hardware.

Figure 6.13 depicts the timing diagram of an event chain with the minimum E2E latency i.e., when all the three tasks are ready for execution and the

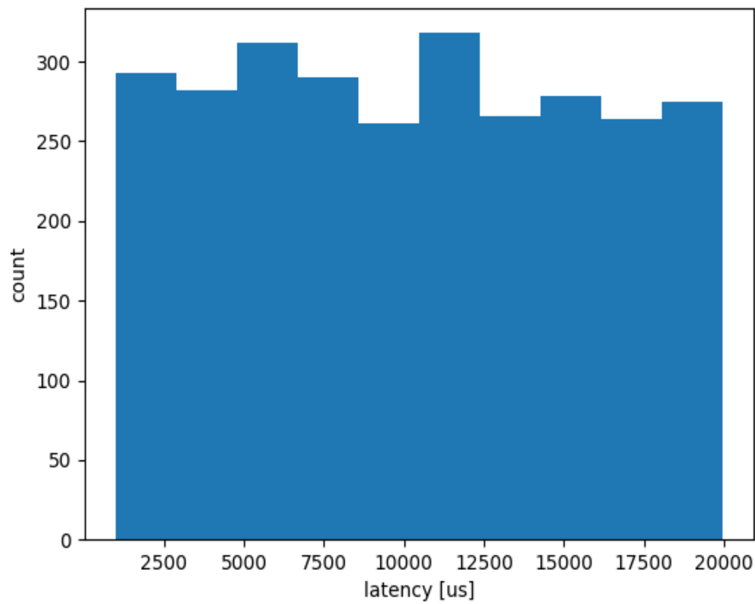

Figure 6.12: Distributed E2E latency of a task chain

message is received from Agent. Figure 6.14 depicts the timing diagram of an event chain with the maximum E2E latency i.e., when the Message Factor Task A started its execution and then immediately the message is received from Agent.

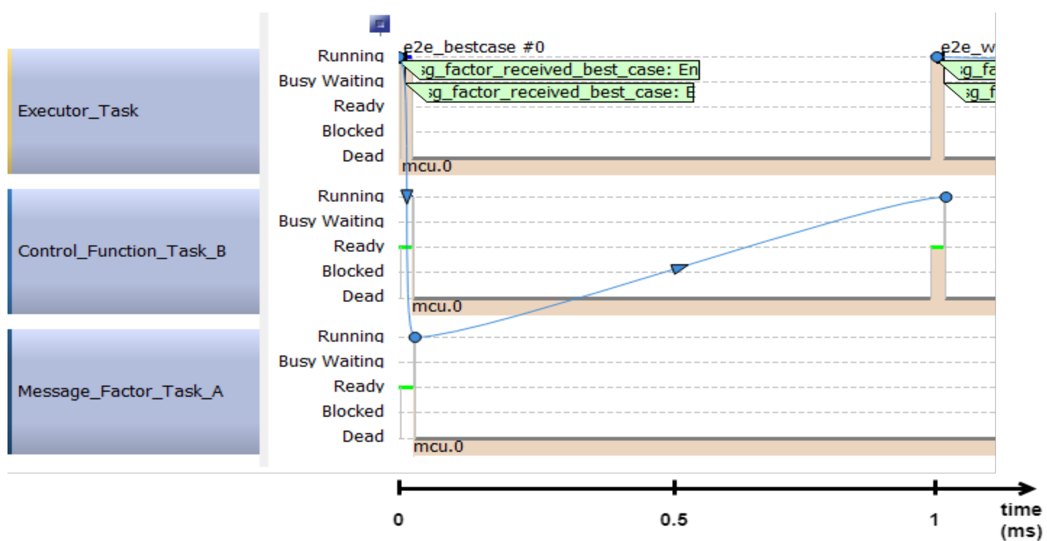

Figure 6.13: Minimum E2E latency of a task chain

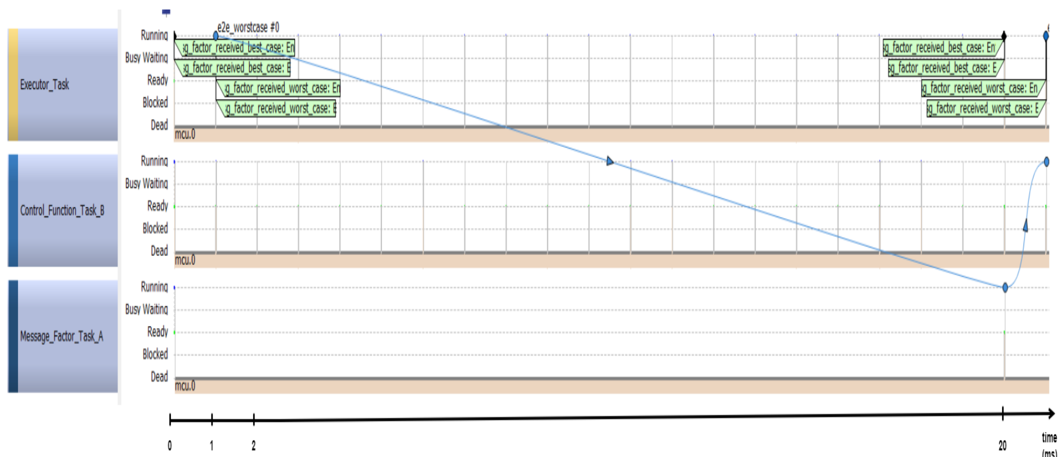

Figure 6.14: Maximum E2E latency of a task chain

Table 6.5 gives the information on the minimum, average and maximum E2E latency of a task chain calculated from the measured values on BODAS, ESP32 and using simulation tool chronSUITE. It is observed that the results obtained on BODAS and the simulation tool (Model) are almost same. As explained before, there is an unexpected delay observed at the I/O subsystem of BODAS, hence the values are quite high on ESP32.

	BODAS	**Model**	**ESP32**
min latency [µs]	970	1029	9790
avg latency [µs]	10320	-	15080
max latency [µs]	19978	20070	40192

Table 6.5: Measurement of E2E latency of a task chain

Chapter 7

Conclusion

This chapter summarises the contributions and conclusions drawn from this thesis work. It also suggests some topics for future work.

7.1 Summary

The successful porting of micro-ROS stack on one of the most popular AUTOSAR-based ECUs in off-highway applications proves that the robotics framework could also be used in the development of advanced automation functions along with automotive proven standards. In this thesis work, the analysis of micro-ROS framework and AUTOSAR Classic framework helped to formulate the four research objectives, which mainly addressed the modeling and timing requirements to be followed for any micro-ROS application developed on an AUTOSAR-based embedded device.

The *first question* [Q1] addressed mainly from the timing perspective, which primarily focused on the concept of mapping event-based execution model in micro-ROS to fixed periodic scheduling scheme in AUTOSAR. Along with the execution management, it also addressed the issue of handling the memory allocation for tasks with single stack, in contrast to global memory of the OSEK operating system. The second to fourth question addressed mainly from the modeling perspective. In the *second question* [Q2], the timing properties that influence the characteristics of micro-ROS stack was investigated and with the main objective to abstract the execution timing behavior of the entire middleware, these properties were grouped to individual necessary models. In the *third question* [Q3], how these individual models on a specific microcontroller could be analysed with one of the most popular modeling languages in automotive, i.e., AMALTHEA was described. Finally, the *fourth*

question [Q4] answered how these developed models allowed to predict the latencies using an appropriate simulation tool like chronSUITE, which is also quite popular in automotive for performance and timing analysis. From the results, it was observed that with detailed modeling of executor i.e., mainly with respect to its execution time, the model could predict better timing statistics related to latencies, response time of the tasks/functions involved in the application.

From the implementation perspective, on the BODAS ECU, additional layers were introduced to support the missing standard C library functions and necessary POSIX functions. The support for standard CAN interface between micro-ROS Client and micro-ROS Agent based on the configuration of custom transport profile was implemented. Along with that, to support tasks-like execution model instead of threads-like execution model, the API function calls in the layers of the micro-ROS Client stack were identified and adapted using a state-machine model approach. The complete implementation was first realized on Linux and later with the hardware interface on the BODAS ECU. Finally, a test application that involved a simple Control Function with the micro-ROS framework was also demonstrated on a BODAS RC18-12/40 Series Controller, which is quite popular in off-highway applications.

7.2 Future Work

The possible improvements and extensions of the thesis work could be in the following paths:

- The hardware data model is built for single-core only. It did not have much impact on the results as the micro-ROS test application on BODAS was very simple. But when designing more advanced application, hardware data model must be built with the actual hardware specifications of BODAS RC40 Series Controllers, so that the simulation results are more accurate.
- The message size of the topic received from the Agent was not considered to be part of the modeling as it was very small. But, we observed that it had an impact on the execution time of the executor. Therefore, this has to be investigated further and studied in detail.
- The event chains are developed in the timing model in chronSUITE tool. To have more accurate E2E latency results, event chains have to

be developed in the event data models in AMALTHEA and have to be converted automatically using am2inc importer.

- From the modeling perspective, the impact of micro-ROS timer associated to the executor has to be investigated, as we observed it also had an effect on the execution time of the executor.

Bibliography

[1] Jason M. O'Kane. *A Gentle Introduction to ROS*. Independently published, October 2013. Available at http://www.cse.sc.edu/~jokane/agitr/.

[2] M. Quigley, K. Conley, and B. Gerkey et al. ROS: an open-source Robot Operating System. *ICRA Workshop on Open-Source Software*, 3(3.2):5, 2009.

[3] Robocademy. What is ROS? https://robocademy.com/2020/07/01/what-is-ros/. Accessed: Dec 2021.

[4] Brian Gerkey. Why ROS 2. http://design.ros2.org/articles/why_ros2.html. Accessed: Dec 2022.

[5] AUTOSAR Offical Website. AUTOSAR. https://www.autosar.org/. Accessed: Dec 2021.

[6] Qiang Wang, Baiyu Xin, Chao Li, and Hong Chen. The realization of reusability in vehicular software development under AUTOSAR. *IEEE Conference Anthology*, China, 2013.

[7] Kaiwalya Kalyan Belsare. Off-Road Vehicle Control Unit with Robot Operating System. *Institute of Architecture of Application Systems, University of Stuttgart*, 2021.

[8] Peter Struss. Model-based Systems in the Automotive Industry. *Technische Universität München*, January 2004.

[9] Open Robotics. The ROS Ecosystem. https://www.ros.org/blog/ecosystem/. Accessed: Jan 2022.

[10] Amanda Dattalo. ROS Introduction. http://wiki.ros.org/ROS/Introduction. Accessed: Jan 2022.

[11] acodez. Microsoft Robotics Developer Studio. https://acodez.in/microsoft-robotics-developer-studio/. Accessed: Jan 2022.

[12] Open RObot Control Software. The Orocos Project. https://orocos.org/. Accessed: Jan 2022.

[13] YARP-Open Source. Yet Another Robot Platform. https://www.yarp.it/latest/. Accessed: Jan 2022.

[14] Object Management Group (OMG). Data Distribution Service for Real-time Systems. https://www.omg.org/spec/DDS/1.2/PDF. Accessed: Jan 2022.

[15] Jose Luis, Poza Luján, and Jose Enrique Simo et al. QoS-Based Middleware Architecture for Distributed Control Systems. *Universitat Politècnica de València*, May, 2014.

[16] Object Management Group (OMG). Data Distribution Service (DDS). https://www.omg.org/spec/DDS/1.4, March 2015. Accessed: Jan 2022.

[17] Dirk Thomas. ROS 2 middleware interface. https://design.ros2.org/articles/ros_middleware_interface.html. Accessed: Jan 2022.

[18] Brian Gerkey. Changes between ROS 1 and ROS 2. http://design.ros2.org/articles/changes.html. Accessed: Jan 2022.

[19] Micro-ROS. Micro-ROS. https://micro-ros.github.io/. Accessed: Jan 2022.

[20] OFERA. OFERA Micro-ROS. http://www.ofera.eu/index.php/micro-ros. Accessed: Jan 2022.

[21] OFERA. OFERA Project. http://www.ofera.eu/index.php. Accessed: Jan 2022.

[22] Micro-ROS. Micro-ROS: Features and Architecture. https://micro.ros.org/docs/overview/features/. Accessed: Jan 2022.

[23] Object Management Group (OMG). DDS For Extremely Resource Constrained Environments. https://www.omg.org/spec/DDS-XRCE/About-DDS-XRCE/. Accessed: Jan 2022.

[24] Ivan Paunovic, Jacob Perron, and William Woodall. rcl. https://github.com/ros2/rcl/. Accessed: Jan 2022.

[25] Open Source Robotics Foundation, Robert Bosch GmbH, and eProsima. The rclc repository. https://github.com/ros2/rclc/. Accessed: Jan 2022.

[26] eProsima. eProsima Micro XRCE-DDS. https://micro-xrce-dds.docs.eprosima.com/en/latest/. Accessed: Jan 2022.

[27] eProsima. eProsima Micro XRCE-DDS Client. https://micro-xrce-dds.docs.eprosima.com/en/latest/client.html. Accessed: Jan 2022.

[28] eProsima. eProsima Micro XRCE-DDS Agent. https://micro-xrce-dds.docs.eprosima.com/en/latest/agent.html. Accessed: Jan 2022.

[29] OpenRobotics. RViz. https://github.com/ros2/rviz. Accessed: Jan 2022.

[30] Micro-ROS. First micro-ROS Application on an RTOS. https://micro.ros.org/docs/tutorials/core/first_application_rtos/. Accessed: Jan 2022.

[31] The Apache Software Foundation. APACHE LICENSE, VERSION 2.0. https://www.apache.org/licenses/LICENSE-2.0. Accessed: Jan 2022.

[32] Free Software Foundation. GNU General Public License. https://www.gnu.org/licenses/gpl-3.0.en.html. Accessed: Jan 2022.

[33] Micro-ROS. License Overview. https://micro.ros.org/docs/overview/license/. Accessed: Jan 2022.

[34] ROS Discourse. ”ROS 2 Embedded Working Group. https://discourse.ros.org/c/embedded/9. Accessed: Jan 2022.

[35] Micro-ROS. Benchmarking. https://micro.ros.org/docs/concepts/benchmarking/benchmarking/. Accessed: Jan 2022.

[36] Micro-ROS. Tutorials Overview. https://micro.ros.org/docs/tutorials/core/overview/. Accessed: Jan 2022.

[37] eProsima. Micro XRCE-DDS. https://github.com/eProsima/Micro-XRCE-DDS. Accessed: Jan 2022.

[38] Steve Heath. *Embedded Systems Design (2ed.)*. 2013.

[39] Tammy Noergaard. Embedded Systems Architecture. *Elsevier's Science Technology*, pages 402–419, 2005.

[40] Xiaocong Fan. Real-Time Embedded Systems. *Elsevier's Science Technology*, (13):339–343, 2015.

[41] Linux. sched(7) — Linux manual page. https://man7.org/linux/man-pages/man7/sched.7.html. Accessed: Jan 2022.

[42] Continental Automotive GmbH. OSEK VDX Portal. https://www.osek-vdx.org/. Accessed: Jan 2022.

[43] G.C. Buttazzo. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. (12):419–423, 2011. Accessed: Jan 2022.

[44] ETAS Group. RTA-OSEK User Guide. (5.0.2), 2001. Accessed: Jan 2022.

[45] ISO. ISO 25119-1:2018. https://www.iso.org/standard/69025.html. Accessed: Mar 2022.

[46] Robert Bosch Press. Rexroth unveils BODAS RC 40 controllers specifically designed for agricultural machinery. https://www.boschrexroth.com/en/xc/company/press/index2-36801. Accessed: Mar 2022.

[47] Bosch Rexroth AG. BODAS Controller RC18-12 series 40 RC27-18 series 40. (RE 95208), August, 2021.

[48] Robert Bosch Press. Rexroth presents BODAS RC 40 control units. https://www.bosch-presse.de/pressportal/de/en/rexroth-presents-bodas-rc-40-control-units-208005.html. Accessed: Mar 2022.

[49] Bosch Rexroth AG. BODAS-service Version 4.3.1. (RE 95087), July, 2021.

[50] Sandeep Parvatikar. A Low Resource Implementation of AUTOSAR. https://link.springer.com/content/pdf/10.1007/s40111-013-0025-z.pdf. Accessed: Mar 2022.

[51] Jan Staschulat, Ralph Lange, and Dakshina Narahari Dasari. Budget-based real-time Executor for Micro-ROS. May 2021.

[52] Daniel Casini, Tobias Blass, Ingo Lütkebohle, and Björn B. Brandenburg. Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling. *Proceedings of 31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, July 2019.

[53] Yukihiro Saito, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. Priority and synchronization support for ROS. *IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 77–82, 2016.

[54] H. Wei, Z. Shao, Z. Huang, R. Chen, Y. Guan, and J. Tan et al. RT-ROS: A real-time ROS architecture on multi-core processors. *Future Generation ComputerSystems*, 2015.

[55] J. Sticha. Validating the Real-Time Capabilities of the ROS Communication Middleware. July, 2014.

[56] Micro-ROS. Micro-ROS: Supported RTOSes. `https://micro.ros.org/docs/overview/rtos/`. Accessed: Jan 2022.

[57] eProsima. micro-ROS example for Mbed RTOS. `https://github.com/micro-ROS/micro_ros_mbed`. Accessed: Jan 2022.

[58] eProsima. micro-ROS for Arduino. `https://github.com/micro-ROS/micro_ros_arduino`. Accessed: Jan 2022.

[59] Arduino. Arduino Official Website. `https://www.arduino.cc/`. Accessed: Jan 2022.

[60] Robert H¨ottger, Lukas Krawczyk, and Burkhard Igel. Model-Based Automotive Partitioning and Mapping for Embedded Multicore Systems. January, 2015.

[61] Justyna Zander-Nowicka. Model-based Testing of Real-Time Embedded Systems in the Automotive Domain. *Technischen Universität Berlin*, Berlin, 2009.

[62] ASAM e.V. ASAM AE MDX. Metadata Exchange Format for Software Module Sharing. June 2015.

[63] Robert Bosch GmbH. AMALTHEA 4 public. `http://www.amalthea-project.org/`. Accessed: Jan 2022.

[64] Robert Höttger, Harald Mackamul, Andreas Sailer, Jan-Philipp Steghöfer, and Jörg Tessmer. APP4MC: Application platform project for multi- and many-core systems. *Bosch, University of Gothenburg*, September, 2017.

[65] Andreas Sailer, Stefan Schmidhuber, Maximilian Hempe, Michael Deubzer, and Jürgen Mottok. Distributed Multi-Core Development in the Automotive Domain – A Practical Comparison of ASAM MDX vs. AUTOSAR vs. AMALTHEA. *Ostbayerische Technische Hochschule (OTH) Regensburg*, Germany, April 2016.

[66] Padma Iyenghar, Lars Huning, and Elke Pulvermueller. Early Synthesis of Timing Models in AUTOSAR-based Automotive Embedded Software Systems. *In Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development*, pages 26–38, 2020.

[67] University of Brittany LISyC Team. The Cheddar project : a free real time scheduling analyzer. `http://mercury.pr.erau.edu/~siewerts/cec450/design/Cheddar-tools/Cheddar-2.1-win32-bin/docs/index.html`. Accessed: Jan 2022.

[68] Universidad de Cantabria. MAST - Modeling and Analysis Suite for Real-Time Applications. `https://mast.unican.es/mast.html`. Accessed: Jan 2022.

[69] Vector. TA Tool Suite. `https://www.vector.com/de/de/products/products-a-z/software/ta-tool-suite/`. Accessed: Jan 2022.

[70] Technische Universität Braunschweig. SymTA/S: Project overview. `https://www.ida.ing.tu-bs.de/forschung/projekte/symtas/project-overview`. Accessed: Jan 2022.

[71] Inchron. chronSUITE. `https://www.inchron.com/chronsuite/`. Accessed: Jan 2022.

[72] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *IEE Proceedings*, 152(2):148–166, March, 2005.

[73] Steffen Kollmann, Victor Pollex, Kilian Kempf, and Frank Slomka. Comparative Application of Real-Time Verification Methods to an Automotive Architecture. *Ulm University*, Germany, 2010.

[74] Alex Lotz, Ralph Lange, Christian Heinzemann, and Jan Staschulat et al. Combining Robotics Component-Based Model-Driven Development with a Model-Based Performance Analysis. *IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots San Francisco, USA*, Dec 13-16, 2016.

[75] D. Stampfer, A. Lotz, M. Lutz, and C. Schlegel. The SmartMDSD Toolchain: An Integrated MDSD Workflow and Integrated Development Environment (IDE) for Robotics Software. *Journal of Software Engineering for Robotics (JOSER), Special Issue on Domain-Specific Languages and Models in Robotics*, 7(1):3–19, 2016.

[76] Tobias Blass, Arne Hamann, Ralph Lange, Dirk Ziegenbein, and Björn B. Brandenburg. Automatic Latency Management for ROS 2: Benefits, Challenges, and Open Problems. *IEEE Conference, Nashville, TN, USA*, May, 2021.

[77] Open Robotics. Understanding ROS 2 nodes. `https://docs.ros.org/en/foxy/Tutorials/Understanding-ROS2-Nodes.html`. Accessed: Jan 2022.

[78] Micro-ROS. Execution Management. `https://micro.ros.org/docs/concepts/client_library/execution_management/#introduction`. Accessed: Jan 2022.

[79] Micro-ROS. Execution Management - Analysis of rclcpp standard Executor. `https://micro.ros.org/docs/concepts/client_library/execution_management/#analysis-of-rclcpp-standard-executor`. Accessed: Jan 2022.

[80] Micro-ROS. Execution Management - rclc Executor. `https://micro.ros.org/docs/concepts/client_library/execution_management/#rclc-executor`. Accessed: Jan 2022.

[81] eProsima. micro-ROS client Memory Profiling. `https://www.eprosima.com/index.php/resources-all/performance/micro-ros-client-memory-profiling`. Accessed: Jan 2022.

[82] eProsima. Micro XRCE-DDS Agent. `https://github.com/eProsima/Micro-XRCE-DDS-Agent`. Accessed: Jan 2022.

[83] eProsima. Micro XRCE-DDS Client. `https://github.com/eProsima/Micro-XRCE-DDS-Client`. Accessed: Jan 2022.

[84] Juarj Felijan. Task Allocation Optimization for Muticore Embedded Systems. *Mälardalen University Sweden*, 2015.

[85] eProsima. RMW Micro XRCE-DDS Implementation. https://github.com/micro-ROS/rmw_microxrcedds. Accessed: Jan 2022.

[86] Micro-ROS. Micro XRCE-DDS memory profiling. https://micro.ros.org/docs/concepts/middleware/memo_prof/. Accessed: Jan 2022.

[87] Albert Bollard, Elixabete Larrea, Alex Singla, and Rohit Sood. Introducing the next-generation operating model. *McKinsey Global Institute*, March, 2017.

[88] Inc Eclipse Foundation. Eclipse APP4MC. http://www.eclipse.org/app4mc/. Accessed: Feb 2022.

[89] Robert Bosch GmbH. org.eclipse.app4mc.git. https://git.eclipse.org/c/app4mc/org.eclipse.app4mc.git. Accessed: Feb 2022.

[90] ITEA4. AMALTHEA. https://itea4.org/project/amalthea.html. Accessed: Feb 2022.

[91] Robert Bosch GmbH. AMALTHEA - Data Models. https://www.eclipse.org/app4mc/help/latest/index.html#section3. Accessed: Feb 2022.

[92] Inchron. am2inc. https://github.com/inchron/am2inc. Accessed: Feb 2022.

[93] Inchron. Inchron-Excellence in Real Time. https://www.inchron.com/. Accessed: Feb 2022.

[94] Inchron. Inchron-Automotive. https://www.inchron.com/automotive/. Accessed: Feb 2022.

[95] Inchron. chronSUITE Feature OverView. https://www.inchron.com/chronsuite-feature-overview/. Accessed: Feb 2022.

[96] Inchron. What's New in chronSUITE 3. https://www.inchron.com/whats-new-chronsuite-3/. Accessed: Feb 2022.

[97] Inc Eclipse Foundation. Eclipse Public License - v 2.0. https://www.eclipse.org/legal/epl-2.0/. Accessed: Feb 2022.

[98] ESPRESIF. ESP32. https://www.espressif.com/en/products/socs/esp32. Accessed: Mar 2022.

Erklärung:

Ich erkläre hiermit, dass die in dieser Arbeit vorgelegte Arbeit vollständig von mir stammt und nach meinem besten Wissen und Gewissen kein Material enthält, das zuvor von einer anderen Person veröffentlicht oder verfasst wurde, mit Ausnahme der Fälle, in denen im Text ein entsprechender Hinweis gegeben wurde. Darüber hinaus bestätige ich, dass diese Arbeit kein Werk enthält, das für die Verleihung eines anderen akademischen Grades oder Diploms in meinem Namen an einer Universität oder einer anderen Einrichtung angenommen wurde. Die elektronische Kopie stimmt mit allen eingereichten Kopien überein.

Declaration:

I hereby declare that the work presented in this thesis is entirely my own and to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other institution. The electronic copy is consistent with all submitted copies.

Stuttgart, 12.04.2022 Suraj Rao Bappanadu