A Model-Based Methodology for Automated Verification of ROS 2 Systems

Lukas Dust
lukas.dust@mdu.se
Mälardalen University
Västerås, Sweden

Mikael Ekström
mikael.ekstrom@mdu.se
Mälardalen University
Västerås, Sweden

Rong Gu
rong.gu@mdu.se
Mälardalen University
Västerås, Sweden

Saad Mubeen
saad.mubeen@mdu.se
Mälardalen University
Västerås, Sweden

Cristina Seceleanu
cristina.seceleanu@mdu.se
Mälardalen University
Västerås, Sweden

ABSTRACT

To simplify the formal verification of ROS 2-based applications, in this paper, we propose a novel approach to the automation of their model-based verification using model-driven engineering techniques. We propose a methodology starting with ROS 2 execution traces, generated by *ROS2_tracing* and using models and model transformations in Eclipse to automatically initialize pre-defined formal model templates in UPPAAL, with system parameters. While the methodology targets the simplification of formal verification for robotics developers as users, the implementation is at an early stage and the toolchain is not fully implemented and evaluated. Hence, this paper targets tool developers and researchers to give a first overview of the underlying idea of automating ROS 2 verification.

Hence, we propose a toolchain that supports verification of implemented and conceptual ROS 2 systems, as well as iterative verification of timing and scheduling parameters. We propose using four different model representations, based on the *ROS2_tracing* output and self-designed Eclipse Ecore metamodels to model the system from a structural and verification perspective. The different model representations allow traceability throughout the modeling and verification process. Last, an initial proof of concept is implemented containing the core elements of the proposed toolchain and validated given a small ROS 2 system.

KEYWORDS

ROS 2, Robotic systems, Formal Verification, Model Checking, Model-Based Engineering

ACM Reference Format:
Lukas Dust, Mikael Ekström, Rong Gu, Saad Mubeen, and Cristina Seceleanu. 2024. A Model-Based Methodology for Automated Verification of ROS 2 Systems. In *2024 ACM/IEEE 6th International Workshop on Robotics Software Engineering (RoSE '24), April 15, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3643663.3643970

1 INTRODUCTION

Various forms of verification are needed to ensure the correctness of any robotic system design and implementation, in order to guarantee the required behavior. Formal methods, such as model checking [10], are well-known for providing mathematical and rigorous analysis of complex systems during the design time. Model checking enables an automatic exploration of the system's model state space, based on which an exhaustive verification is conducted to check if the system model satisfies its specifications. Model checking as a model-based verification approach can reveal potential errors not detected by trial-and-error approaches, such as simulation and experiments [13]. Nevertheless, one drawback of formal model-based approaches is the complicated process of modeling, mainly when the system is distributed and complex. In the *model-checking* world, this problem becomes even worse as the learning curve for robotic developers to get acquainted with the mathematical syntax and semantics of the formal modeling language is often very steep. While it might be worth applying model checking in robotic systems, and studies have been done investigating the application of model checking in such systems [12], the high initial effort creates a usage barrier for the industry. Therefore, expensive and error-prone trial-and-error approaches remain the primary method that the robotics developers use to verify the correctness of systems [25].

The Robot Operating System (ROS) [22, 23] has been developed as an open-source middleware that allows fast prototyping and deployment, especially for robotic systems. ROS-based systems with potential safety-critical applications often have strict timing requirements, leading to real-time demands on the middleware. Especially in distributed systems, real-time capabilities are influenced by multiple system components, including communication, task scheduling and execution. Due to the lack of real-time capabilities in ROS, the latter has evolved to ROS 2 [23] by adding real-time capable communication utilizing Data Distribution Service (DDS) [9]. While DDS is a standardized and scientifically regarded approach for real-time communication [9], the ROS 2 task scheduling is missing extended analysis. Early analysis, such as response-time analysis [8, 11], shows the design of the mainline scheduler being vulnerable to blocking, leading to non-deterministic timing behavior. Execution analysis tools like *ROS2_tracing* [6] and *Autoware_perf* [19] have been implemented, allowing to trace system execution and conduct analysis based on such execution traces.

Such analysis is more experimental than exhaustive, and potential system errors might be missed.

In contrast, model checking can provide exhaustive verification that is capable of discovering all potential bugs in the model. However, only a few formal methods are available for ROS 2 systems, that are error-prone and time-consuming due to their manual application and the requirement of background knowledge of those methods. In our previous work [13], we have used the well-known UPPAAL model checker [2] to create reusable templates for verification of timing behavior and buffer overflow. With template-based verification, systems are modeled by instantiating formal model templates. Hence, there is no need to construct the system model from scratch. Therefore, template-based verification facilitates the application of formal verification. Nevertheless, utilizing such model templates requires knowledge of static and runtime system parameters. Furthermore, extended knowledge about the modeling language is needed to represent given requirements as verification properties. Due to the manual nature and interdisciplinary knowledge needed, the process is vulnerable to errors. The needed parameters can be obtained by a manual, hence error-prone, process of source-code analysis and runtime evaluation.

1.1 Problem Definition and Paper Contributions

In our previous work [14], ROS 2 scheduling-related timing issues have been exposed, and formal model templates have been created that can be composed and initialized to verify the identified timing issues of buffer overflow and high latencies [13]. Nevertheless, the formal model templates require system parameters to be determined and added manually, which is time-consuming and error-prone. This paper aims to simplify the formal verification of ROS 2 systems using the already defined formal model templates and a model-based methodology for parameter determination and model initialization. While the end users of the methodology are robotics developers, this paper is the first proposal of the methodology with only partial implementation of a toolchain. Therefore, with this paper, we aim to prove the feasibility of the proposed methods, which makes this paper suitable for researchers and tool developers. We present the following contributions:

(1) A novel methodology for model-based verification of ROS 2 systems proposing a potential toolchain that includes UPPAAL, Eclipse and *ROS2_tracing*.
(2) Inclusion of Ecore metamodels to capture system structure and support verification activities.
(3) Automated model-to-model transformations from generated ROS 2 execution traces to Ecore models and automated transformation of Ecore models to UPPAAL models.
(4) We demonstrate the main workflow of the proposed toolchain in the form of a proof of concept implementation covering partial aspects of the proposed toolchain architecture.

The remainder of the paper is organized as follows. In Section 2, we present the concept of model checking, ROS 2 and model-driven engineering using Eclipse. In Section 3, we present the proposed methodology and toolchain architecture with the potential workflow before answering the first two research questions. Section 4 presents the implemented proof of concept with an answer to the last research question, before in Section 5, we give an overview of related work. The paper concludes with final remarks in Section 6.

2 BACKGROUND

In the following, we give an introduction to the model-checking technique and the tool UPPAAL, before we present the concept of ROS 2 and the tool *ROS2_tracing*. Finally, we give an introduction to the Eclipse Modeling Framework and related tools.

2.1 Model Checking and UPPAAL

Model checking [4] is a formal method that provides mathematical and rigorous analysis by automatically exploring a system's entire state space. Furthermore, exhaustive verification can be conducted in the early design phases of a system to verify that the system satisfies its requirements and specifications. Due to the rigorous analysis and full state-space coverage, model checking can reveal potential errors not detected by trial-and-error approaches. Therefore, a formal model representation of major system components that contains a system's needed parameters and behavior needs to be created. Different model checkers are available to use in real-time and distributed robotic systems [17, 26]. In this work, we use UPPAAL [17], a tool that supports modeling, simulation, and model checking of timed automata [2]. UPPAAL extends timed automata with new features, among which we use the modeling, simulation and model checking of networks of timed automata (UTA) in this paper. UTA are modeled as templates that can be instantiated by assigning values to their parameters. UPPAAL has an inbuilt model checker that can verify time computation tree logic properties such as **invariance** where for all paths, for all states in each path, a specified proposition is satisfied. We refer to the work of Alur et al. [2], and Behrmann et al. [17] for a detailed introduction. For this work, understanding the theory of timed automata and the mechanism of model checking in UPPAAL is optional.

2.2 ROS 2

The Robot Operating System 2 (ROS 2) [23] is an open-source middleware for the fast development and prototyping of robotic systems. Despite the name, ROS 2 is not a standalone operating system (OS) and must be installed on top of an underlying host OS, mostly Linux. ROS 2 has been developed with industrial needs such as fault tolerance and real-time constraints in mind. In order to meet real-time demands, ROS 2 utilizes data distribution service (DDS) [15] as the communication middleware. DDS has been developed by the Object Management Group (OMG) [15] and enables communication between distributed applications. The core components of ROS 2 systems are the so-called *nodes* that communicate over designated channels in the DDS. There are two types of ROS 2 communication [7], that is, *Publisher&Subscriber* and *Service&Client*. With *Publisher-Subscriber* communication, a node can either *subscribe* to and receive all posted messages or *publish* messages in a specific topic. Messages in topics are received by all nodes subscribed to the same topic. In contrast, *Service-Client* communication is directed, where one node uses the service channel to trigger a function in another specified node and executes another function when the response is received. An example of a ROS 2 system consisting of two nodes communicating over four channels is presented in Figure 1. Despite the communication channels, system timers can be utilized to trigger functions inside a node at specific times.

Nodes are executable in the host OS and consist of several functions called *callbacks*, ROS 2's atomic schedulable entities. Callbacks

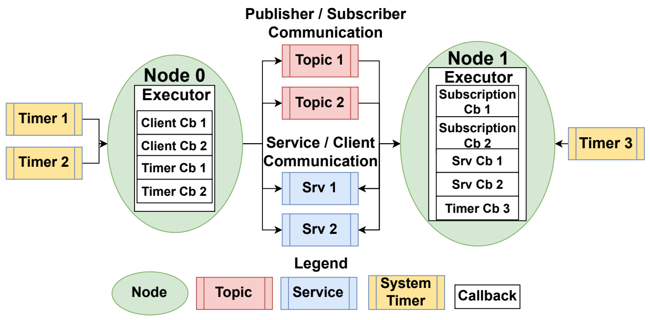


Figure 1: An example ROS 2 system showing the concepts of Node, Timer, Topic, Publisher, Subscriber, Service, and Client.

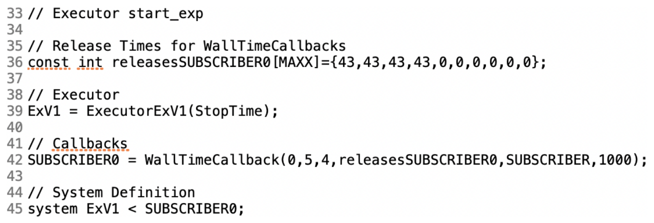
are released on events, such as the arrival of data in the input buffer of a specific channel or an event triggered by system timers. Hence, four types of callbacks exist: timer, subscriber, service, and client. Besides the buffer and the callbacks, a node consists of an executor, which is the internal scheduling component of a node [8, 11]. The executor schedules and executes callbacks while utilizing single or multiple threads in the underlying OS, dependent on the chosen configuration. The latest versions of ROS 2 do not have options to configure priorities of callbacks, which makes their execution vulnerable to to blocking, in the worst case, leading to buffer overflow and instance misses [13, 14].

2.3 ROS2_tracing

System traces can be used to profile the execution of a system. A trace is a profile of a system's execution, where timestamps are recorded and assigned to specified events happening during the system execution. System traces can be used to get information about releases and execution times of callbacks, as well as initialization of system components and message passing in a system. One of the tools to generate traces in ROS 2 is the so-called *ROS2_tracing* toolbox [6]. It is included in the ROS 2 installation. *ROS2_tracing* is a low overhead framework based on the Linux Trace Toolkit next generation (LTTng). In *ROS2_tracing*, traces are based on trace points added to the mainline ROS 2 source code. To analyze generated traces, *ROS2_tracing* comes with a toolbox called tracetools_analysis, containing a python library to transform the LTTng traces into a defined ROS 2 data model, which is a textual representation of the trace, in the form of pandas python objects.

2.4 Pattern-Based Verification of ROS 2 Timing

As a response to the vulnerabilities of the executor, pattern-based verification of the ROS 2 execution behavior using UPPAAL is proposed in our previous work [13]. In the given work, the *callback latency* and *input buffer sizes* of callbacks are selected as properties to be verified. The latency of a callback is defined as the maximum time between the release of a callback instance and the time of completing its execution. The second property, the input buffer size, verifies that the input buffer is large enough to buffer the incoming data for a given system configuration constantly. To formally model a ROS 2 system, we created three types of UTA templates that can be composed to represent ROS 2 Nodes [13]. The UTA templates represent Wall-Time-Callbacks, released at defined times, Periodic Callbacks, released periodically and Executors, representing different available executor versions. To compose a system using the created templates, the developer needs to know static and runtime parameters. Static parameters include the buffer sizes, release periods, and types of callbacks. Runtime parameters include actual release times of callbacks and execution times. An example of a system initialization using the given templates is shown in Figure 2. In the Figure, an array is created holding the release times for a Wall-Time-Callback. In this case, the callback is released four times at 43 ms. Next, an executor is initialized with a defined stop time that describes the interval for which the verification will be conducted. Next, the Wall-Time-Callback template is instantiated, where the following parameters are passed: the ID of the callback, the execution time, the number of releases, the release time array, the type of callback, and the buffer size. Last, the system is defined as a composition of the executor and the callback.


```
// Executor start_exp

// Release Times for WallTimeCallbacks
const int releasesSUBSCRIBER0[MAXX]={43,43,43,43,0,0,0,0,0,0};

// Executor
ExV1 = ExecutorExV1(StopTime);

// Callbacks
SUBSCRIBER0 = WallTimeCallback(0,5,4,releasesSUBSCRIBER0,SUBSCRIBER,1000);

// System Definition
system ExV1 < SUBSCRIBER0;
```

Figure 2: Example UPPAAL System Initialization

We refer to the literature [13] for a detailed description of the UTA templates, system compositions and proposed mapping of ROS 2 system components to the UTA templates. It is important to note that in the given versions of the UTA templates, only individual executors can be verified at a time and communication is abstracted to the respective release time of the receiving callback. After the modeling, the system can be simulated using the UPPAAL simulator, and the wanted properties can be verified in the inbuilt model checker. As the composition of UTA templates simulates the execution of a ROS 2 system, the verification takes all possible execution scenarios given the parameters into consideration [13]. As the models represent the ROS 2 core only, effects of any underlying operating system are neglected and only issues based on the internal ROS 2 executor are exposed. Extensive system knowledge and analysis are required to determine the needed static and runtime parameters to model a ROS 2-based application using the given approach. Furthermore, a user needs to know how to compose the templates and how to write the verification properties. Hence, interdisciplinary knowledge between formal modeling and robotic systems is needed to apply the proposed verification. Furthermore, the process is manual, where developers need to instantiate the given templates by hand based on the obtained system knowledge.

2.5 Eclipse Modeling Framework (EMF)

Despite the formal modeling in UPPAAL, the Eclipse Modeling Framework is chosen throughout this paper as a tool to conduct further model-driven development. Generally, Eclipse is a Java-based, open-source, integrated development environment (IDE). Numerous projects extend the framework and contain specific tools and development environments [27]. An example is the Eclipse Modeling Framework (EMF) [27]. Its focus is on modeling and code generation based on structured data models. EMF allows the definition of metamodels that are instances of the so-called Ecore metamodel. A metamodel is a model's definition, where further models represent model elements and a model is an instantiated metamodel. With EMF, metamodels can be created in a new application plugin. In addition to EMF, tools such as QVT-O [21] and Acceleo [1] allow model navigation and model-to-model (M2M) and model-to-text (M2T) transformations. With an M2M transformation, models based on one metamodel can be mapped to models based on another metamodel. A model-to-text transformation can generate text based on the model elements, e.g., source code.

3 PROPOSED TOOLCHAIN

In this section we propose a methodology that allows automation of verification of legacy and new ROS 2 systems. In order to present the methodology, we propose a toolchain architecture and workflow. In the toolchain, we utilize system execution traces generated with *ROS2_tracing* and model-driven techniques in Eclipse to compose the predefined UTA templates presented in Section 2.4 to an UPPAAL system, allowing verification of callback latency and buffer sizes. Furthermore, the proposed framework aims to decouple the process of formal modeling and the actual system development while allowing traceability throughout the whole application of the toolchain. That allows toolchain users to focus entirely on the system development and reduce the required knowledge to perform formal verification. Note that the proposed toolchain in this section is conceptual and not all proposed parts are actually implemented.

3.1 Architectural Overview

Besides the actual ROS 2 system, the proposed toolchain consists of three tools, namely *ROS2_tracing*, Eclipse and UPPAAL. Each tool has different tasks in the automatic creation of UTA. Figure 3 presents an architectural overview of the toolchain. The layers of the toolchain, the role of the specific tools, and the included artifacts and actions in the toolchain are explained in the following paragraphs, where we use Figure 3 to visualize the architecture.

3.1.1 ***ROS 2 Layer.*** The **ROS 2 Layer** **0** (light blue) represents a ROS 2 system implementation, including static information (dark blue) and dynamic information (white).

3.1.2 ***Tracing Layer.*** The first layer of the toolchain, closest to the ROS 2 system, is the **Tracing Layer** **1** with all actions and artifacts marked in green. *ROS2_tracing* is used to generate a trace from the execution of a system. The trace contains static information such as callback periods, buffer sizes and dynamic information such as execution times. As the tracing format of LTTng traces are not human readable, the Python library Trace_Analysis transforms the traces into the `ROS 2 Data model` **M0**, a textual description of the collected trace information, the so-called ROS 2 Data Model. Generally, the information collected by *ROS2_tracing* and the ROS 2 Data Model can be customized, allowing future extensions and adaptions in case of missing parameters.

3.1.3 ***Modeling Layer.*** The **Modeling Layer** **2** (light yellow) contains the metamodels and modeling actions that are needed to automate the verification process, starting from the `ROS 2 Data model` to the final code generation. With **MM1**, **MM2**, and **MM3**, we propose the usage of three different metamodels as EMF model representations of a ROS 2 application, each focusing on a selected perspective. That allows verification of legacy system implementation and conceptual system design and enables traceability. The three proposed metamodels are presented in the following paragraphs.

MM1 **EMF Data Metamodel** The goal of the `EMF Data Metamodel` is to map the `ROS 2 data model` to an EMF model that can be used in Eclipse. Hence, the EMF Data Metamodel is a complete representation of the tracing output. The EMF Data Metamodel is created to parse the trace results from *ROS2_tracing*, allowing analysis of the tracing results in Eclipse and to trace parameters throughout the toolchain.

MM2 **EMF Verification Metamodel** The EMF Verification Metamodel is an EMF model representation of the UTA templates and their parameters. As part of the toolchain, the metamodel is used to generate the UPPAAL artifacts, namely the systems composition and UTA template initialization. In future adaptions, it is desirable to include requirements such as maximum latency in this metamodel to allow the automatic generation of UPPAAL verification queries.

MM3 **EMF System Metamodel** The purpose of the EMF System Metamodel is to represent a ROS 2 system from an architectural perspective. While it is not essential for verification based on system traces, it gives developers the opportunity to visualize the overview of the system architecture. Therefore, the model contains the main system components and the static system parameters. Developers should be able to make changes in this model and then utilize the model transformations to verify different system parameters without repeatedly running the actual system. Furthermore, it takes away the need to fully understand the verification metamodel. On the other hand, the EMF System Metamodel can be used as a starting point for conceptual systems without an existing source code implementation. With future adaptions, it is intended that this model can be used for static source code generation to give a framework for a ROS 2 systems implementation, allowing model-based development and generation of real systems.

Besides the presented metamodels, with **T1** - **T7**, the toolchain contains seven components that transform different model representations into each other, which are presented in the following.

T1 **Parsing: ROS 2 Data Model to EMF Data Metamodel. Parsing is the translation of a textual model description into a model representation.** A parsing function is implemented in the first step of creating the automated transformations. The goal of the function is to take the ROS 2 Data Model and to create the XML file that can be imported as an EMF Data Metamodel. The parsing is done by reading the data of the traces and printing them in an XML document with the desired formatting.

T2 **M2M: EMF Data Metamodel to EMF Verification Metamodel.** When doing a model-to-model transformation from an EMF Data Model to an EMF Verification Model, it is essential to filter the required information. Furthermore,

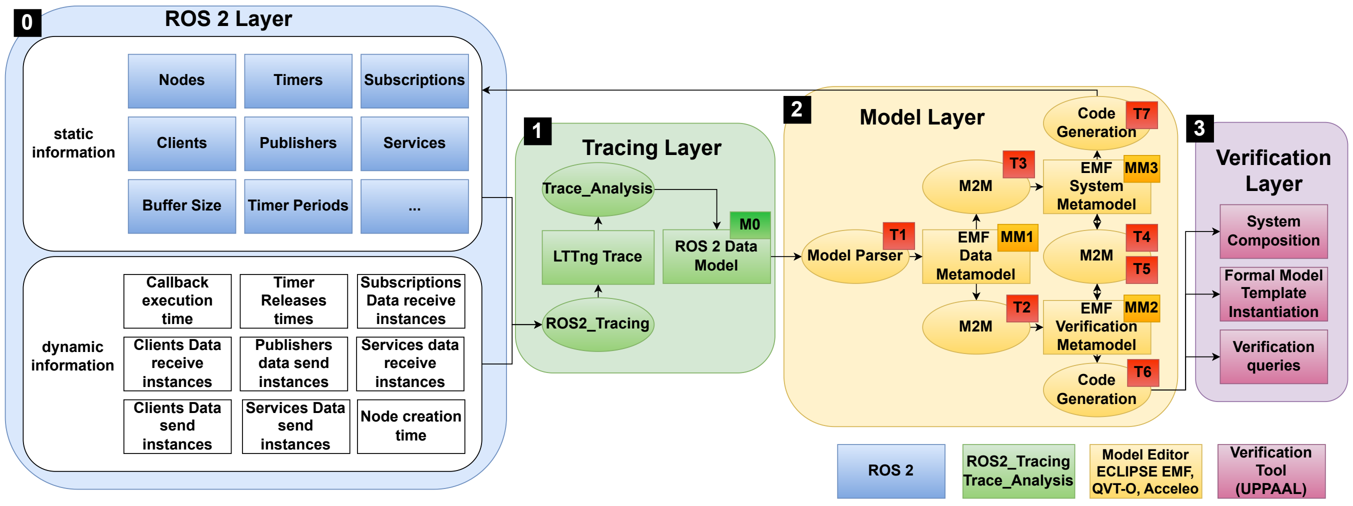


Figure 3: Architectural overview of the proposed toolchain for automation of verification

model components need to be transformed into their corresponding representation in an EMF Verification Model.

T3 **M2M: EMF Data Model to EMF System Model.** Transforming an EMF Data Model to an EMF System Model, the static system parameters and components in the EMF Data Model are taken to create a structural system model according to the EMF System Metamodel.

T4 T5 **M2M: EMF Verification Model to EMF System Model. M2M: EMF System Model to EMF Verification Model.** T4 and T5 are needed to help developers transform between an EMF Verification Model and an EMF System Model. This step is needed for developers that start with a system model instead of an implementation, as it allows to use the toolchain without the need to generate system traces and the EMF Data Model.

T6 **M2T: Verification Model to UPPAAL Verification code.** With the Model-to-Text transformation, an EMF Verification Model can be transformed into UPPAAL Code containing the declaration of a system, instantiating the given templates and composing them to a system. Furthermore, the verification queries are generated in the code.

T7 **M2T: EMF System Metamodel to ROS 2 System Implementation** For the completeness of the toolchain and to show the concept's potential, we include an M2T transformation that can be used to generate a ROS 2 source code implementation based on the static system components from the EMF System Metamodel. That includes architectural components, such as nodes, callbacks and communication channels. This transformation allows a verified conceptual system design to transform into source code automatically. Nevertheless, we have no actual implementation proving that the concept works, and it is a research topic itself, with various works being conducted by fellow researchers.

3.1.4 ***Verification Layer.*** The **Verification Layer** 3 as the last layer of the proposed toolchain contains the generated UPPAAL artifacts as shown in section 2.4. The desired format is C-like code that is the output of the Model-to-Text transformation in the modeling layer and can be opened and run in UPPAAL directly.

3.2 Workflow

Given the proposed architecture of the toolchain, we present the workflow when using the toolchain in this section. We give an overview of the workflow in Figure 4. With the proposed toolchain architecture, it is possible to verify legacy systems (**Start A**), as well as new system designs (**Start B**).

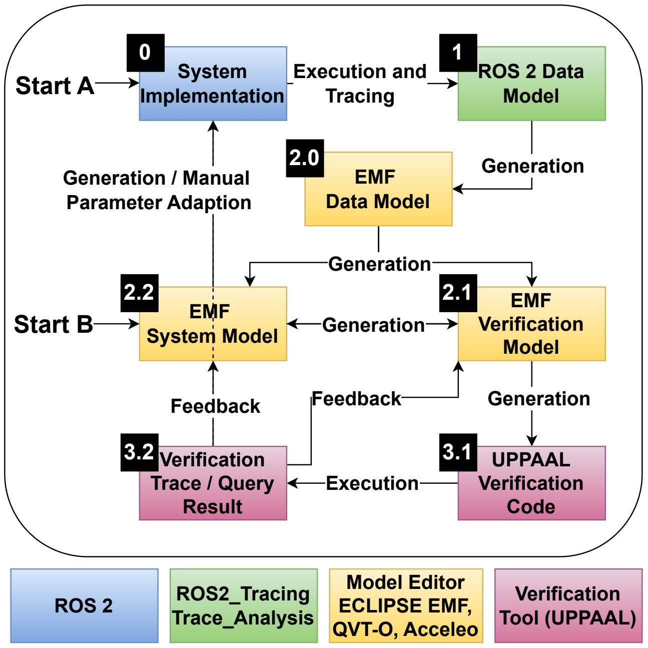


Figure 4: Workflow covered by the toolchain

3.2.1 ***Verification of Legacy Systems.*** Starting at **Start A** with a **ROS 2 system implementation** 0, the first step is to generate an execution trace using *ROS2_tracing*. Next, the traces need to be transformed into the **ROS 2 Data Model** 1, followed by the transformation to the **EMF Data Model** 2.0 using the parsing function. From the EMF Data Model, the **EMF Verification Model** 2.1 is generated using the according M2M transformation. Then, the model can be transformed into the **UPPAAL Verification Code** 3.1 before verification is performed 3.2. If changes are needed, they can be conducted manually in the system implementation 0, and the steps can be followed again. Furthermore, the **EMF System Model** 2.2 can be generated from 2.0 or 2.1 to iteratively verify the changes of parameters at the model layer before these changes are implemented in the real system.

3.2.2 ***Verification of New Systems.*** Starting with a conceptual system design, at **Start B**, the system can be modeled as an **EMF System Model** 2.2 using the proposed metamodel. Based on the EMF System Model, an **EMF Verification Model** 2.1 and the **UPPAAL verification Code** 3.1 can be generated. That process can be followed iteratively until a satisfactory system design has been created and verified. Now, the M2T transformation can be used to generate the a ROS 2 framework for a system implementation 0. Next, the actual implementation can be verified in the following steps: 1 → 2.0 → 2.1 → 3.1 → 3.2.

4 PROOF OF CONCEPT

In this section, we implement a small prototype as a proof of concept of the proposed methodology. For the sake of simplicity and given time constraints, the implementation only focuses on excerpts of the toolchain containing subscription callbacks and periodic timers for the following actions: **ROS 2 System** 0 → **ROS 2 Data Model** 1 → **EMF Data Model** 2.0 → **EMF Verification Model** 2.1 → **UPPAAL Verification Code** 3.1. Despite the fact that customization of the trace information is possible, for this proof of concept we decide to use the mainline trace points with only simple adaptions to the Trace_Analysis library. In the following, an introduction to the implemented metamodels, transformations and mappings is given before the toolchain is used on a simple example.

4.1 Implementation

4.1.1 Metamodel 1 - Eclipse Data Metamodel. In the first step, we create the EMF Data metamodel containing all system components given by the ROS 2 Data Model, such as subscriptions, callbacks, and timers as classes and parameters as attributes. Figure 5 contains an exceerpt from the Meta Model Implementation. The yellow boxes represent the classes in the metamodel, where the arrows between the boxes show the dependencies. Here, all classes that represent system components are child objects of a master class that represents the system. Even though in a ROS 2 implementation, e.g., Publishers are contained in Nodes, in this metamodel, the assignments of Publishers to Nodes are done by identification handlers modeled as attributes. That is done to follow the representation of the ROS 2 Data Model with the EMF Data Model.

4.1.2 Metamodel 2 - EMF Verification Metamodel. The second metamodel is the EMF model representation of the Uppaal templates

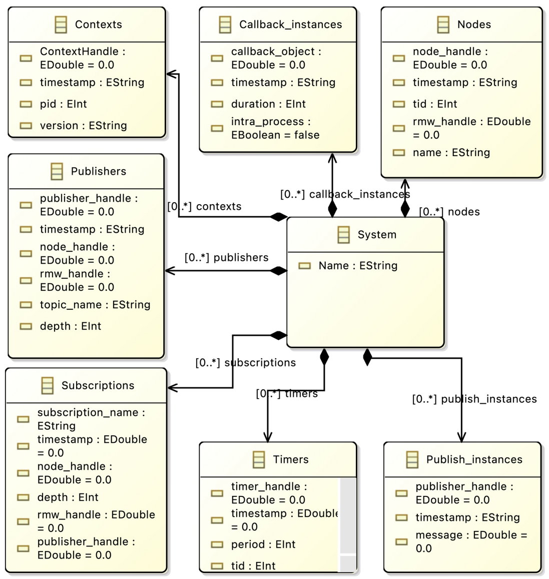

Contexts
ContextHandle : EDouble = 0.0
timestamp : EString
pid : EInt
version : EString
Callback_instances
callback_object : EDouble = 0.0
timestamp : EString
duration : EInt
intra_process : EBoolean = false
Nodes
node_handle : EDouble = 0.0
timestamp : EString
tid : EInt
rmw_handle : EDouble = 0.0
name : EString
Publishers
publisher_handle : EDouble = 0.0
timestamp : EString
node_handle : EDouble = 0.0
rmw_handle : EDouble = 0.0
topic_name : EString
depth : EInt
[0..*] contexts
[0..*] callback_instances
[0..*] nodes
System
Name : EString
[0..*] publishers
[0..*] subscriptions
[0..*] timers
[0..*] publish_instances
Subscriptions
subscription_name : EString
timestamp : EDouble = 0.0
node_handle : EDouble = 0.0
depth : EInt
rmw_handle : EDouble = 0.0
publisher_handle : EDouble = 0.0
Timers
timer_handle : EDouble = 0.0
timestamp : EDouble = 0.0
period : EInt
tid : EInt
Publish_instances
publisher_handle : EDouble = 0.0
timestamp : EString
message : EDouble = 0.0

Figure 5: Data Metamodel in Eclipse

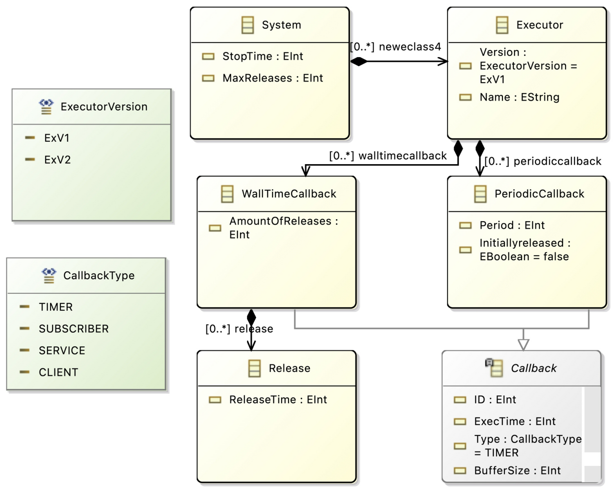


Figure 6: Verification Metamodel in Eclipse

used for formal verification. An overview can be found in Figure 6. In the metamodel, each of the three UPPAAL model templates (Executor, WallTimeCallback and PeriodicCallback) are modeled as individual classes, which are child objects of a system. As described in Section 2.4 each template contains parameters, such as *ids* and buffer sizes. Besides the classes, there are datatypes defined for the attributes of callback type and executor version that can be selected in an instance of the model. While in the toolchain proposal, we mention the modeling of requirements such as maximum latency for callbacks, we leave that part to future work.

4.1.3 Parsing: ROS 2 to EMF Data Model. We implement the parsing as a python function in Trace_Analysis. The function takes the ROS 2 Data Model and creates an XML file that can be imported as a model (using the EMF Data Metamodel) in the Eclipse workspace. The parsing is done by reading the data of the ROS 2 Data Model and printing them in an XML document with the desired formatting of the EMF Data Model.

4.1.4 *M2M: EMF Data Model to EMF Verification Model.*

In this model-to-model transformation, the components of the EMF Data Model are mapped to the components in the verification model using QVT-O. In the first step, the periodic timers are mapped to periodic callbacks with the type attribute set to TIMER. The period and execution time is extracted from the attributes in the EMF Data model and passed to the verification model. Furthermore, for timers, the buffer size is set to one. The second mapping is between the subscription callback and the wall-time-callback of the type SUBSCRIBER. A subscription callback is released on the reception of data. As neither the release time of the callback, nor the reception of the data is initially contained in the system traces, we map the publishing time of the data in the topic the callback is subscribed to as the release time. Furthermore, we pass the execution time and buffer size as further parameters.

4.1.5 *M2T: EMF Verification Model 2 to UPPAAL Code.*

With the Model-to-Text transformation using Acceleo, the EMF Verification Model is translated into the UPPAAL code. Therefore, the classes contained in the EMF Verification Model are mapped to specific code snippets, e.g., representing the UPPAAL template instantiation. Then, the code is dynamically filled with the needed parameters based on the class attributes.

4.2 Evaluation and Discussion

To evaluate and test the prototype, a small ROS 2 System containing two nodes, similar to Figure 1, is created. We test the proof of concept implementation by generating the UPPAAL code using the proposed workflow, where an extract is shown in Figure 2. Instead of needing to analyze the system by hand and calculate potential release times, the automatic generation of traces and the model transformations generate executable UPPAAL code. We execute the generated code with UPPAAL and validate the correctness of the generated results during all steps. Focusing on subscription and timer callbacks only, the generated traces contain the needed information to either directly determine or infer the needed parameters, that are callback periods, callback release times, buffer sizes and callback execution times. Some needed parameters for services and clients and even subscription callback release times are currently missing in *ROS2_tracing*. Nevertheless, the existence of such parameters for subscribers and periodic timers shows the feasibility of the toolchain. Furthermore, it is possible to add such trace points customly. While that requires expert knowledge, we are working on including needed trace points in the mainline releases of ROS 2. Based on our observations, the model-based verification of ROS2 applications can be automated by using systems execution traces and model-driven engineering to fill model-based verification templates automatically.

5 RELATED WORK

A first analysis of the ROS 2 execution behavior has been conducted with response time analysis proposed by Casini et al. [11], Blass et al. [8]. The two papers are essential for the formal verification of ROS 2 timing behavior, where they are used to create the formal model templates and to model timing requirements. Formal verification of ROS 2 communication between nodes, using UPPAAL, has been proposed by Halder et al. [16]. The work takes low-level parameters, such as queue sizes and timeouts, into their TA models to verify queue overflow. While the work focuses on the modeling and verification of ROS 2 using UPPAAL, our toolchain focuses on the automation of verification. Carvalho et al. [10] employ an Alloy extension called Electrum to implement a model-checking technique that automatically creates models from configurations extracted in continuous integration and specifications. Their approach focuses on high architectural-level verification, whereas our toolchain aims to verify low-level behavior such as system execution. Work on formal verification of requirements of robotic systems and ROS 2 message passing in DDS using different model checkers has been conducted by Webster et al. [28] and Liu et al. [20]. While the application of the presented approach still is manual using a different model checker than UPPAAL, our approach focuses on automation of verification. Nevertheless, an extension of our toolchain to formal verification using different model checkers might be desirable. Robo Fuzz Framework proposed by Kim and Kim [18] is a framework that can be used in fuzzing robotic systems for finding bugs in system implementations. Their frameworks focus on the implementation of data types mutation and violation of physical laws and hardware specifications, while our framework focuses on timing and execution verification of ROS 2 applications. Furthermore, fuzz testing is not exhaustive. Towards the development of certified ROS 2 systems, using a "correct-by-construction" approach, ROSCoq has been presented by Anand and Knepper [3]. Their approach is different and not applicable to legacy systems, yet complementary to the verification conducted in our work. Research towards model-based development of DDS-based systems such as ROS 2 has been conducted by Beckmann et al. [5]. It describes how the DDS architecture allows and supports the application of model-based development. This work only focuses on model-based development in contrast to our work on verification. Work containing Ecore models to model QoS Requirements for ROS 2 has been conducted by Parra et al. [24]. While their work focuses on Requirements and QoS attributes specifically, it is complementary to our work, where we focus on architectural components and verification regarding task scheduling. Dal Zilio et al. [12] propose a toolchain for runtime and offline verification of general robotic systems beyond ROS. While they focus on general robotic systems and application code with internal logic, with our toolchain, we focus on verifying timing issues induced by using ROS 2 as middleware. Furthermore, our toolchain uses model-driven engineering in the Eclipse environment to allow iterative verification and model-based development at the same time.

6 CONCLUSION AND FUTURE WORK

To simplify the formal verification of ROS 2-based applications, in this paper, we propose a novel approach to the automation of model-based verification using model-driven engineering techniques. Based on verification of potential timing issues exposed in [14], utilizing formal model templates proposed in [13] we propose a novel model-based methodology that uses ROS 2 system traces to automate the verification process. We propose a potential toolchain using ROS 2 execution traces to utilize models and model transformations to automatically instantiate pre-defined formal model templates. The toolchain focuses on verifying issues induced by using the ROS 2 middleware, which might be overlooked by verification focusing on the application code only. The concept of the toolchain supports verification of implemented as well as conceptual systems. We propose four different model representations, enhancing traceability throughout the process. In addition, parameter refinement and iterative verification of system parameters without repeated source code adaption can be conducted. Due to the complexity of the proposed toolchain, a proof of concept is given with a focus on selected attributes of ROS 2 systems only. In the evaluation we show that *ROS2_tracing* has the potential to cover the needed trace points. Nevertheless, customization is needed to include the needed parameters. Furthermore, an evaluation needs to be conducted to determine to what regard execution times determined by one system execution are sufficient to perform verification. Nevertheless, this paper is showing the feasibility of automatic parameter determination using system traces and model-based development for formal verification. The toolchain leaves space for the automated model-based generation of ROS 2 application code and modeling of requirements, which are not covered by this work and require further analysis and evaluation. While such generation features would lead to a higher degree of automation, they are optional to show the feasibility and novelty of the proposed methodology. Furthermore, despite the toolchain being proposed and demonstrated with specific tools (*ROS2_tracing*, Eclipse EMF, Acceleo, QVT-O and UPPAAL), we are optimistic that the approach can be implemented using different tools like other tracing tools, different model editors and other verification tools. Due to the premature state of the toolchain implementation, this paper is a first proposal and proof of feasibility of the methodology only, which makes this paper suitable to researchers and tool developers. More evaluation and implementation is needed to make this toolchain usable on real robotics systems. To allow more extensive verification, the proposed meta-models and transformations need to be refined and extended. Further formal model templates need to be implemented and added to the toolchain. In the implementation, the modeling of requirements and verification properties is not included and will be conducted in future work. The data model and output of *ROS2_tracing* need to be adapted and refined to contain more system parameters. Furthermore, formal proof of correctness of the specific implementation of the toolchain is desirable. Other future work might be the automation of verification and simulation feedback to the models, including modeling of requirements and the automated source-code generation of the ROS 2 application code. Last but not least, investigation may be performed on the ease of use of the proposed toolchain.

Acknowledgement We acknowledge the support of the Swedish Knowledge Foundation via the synergy project ACICS – Assured Cloud Platforms for Industrial Cyber-Physical Systems, grant nr. 20190038, the Hög project SEINE – Automatic Self-configuration of Industrial Networks, grant nr. 20220230, and the research profile DPAC - Dependable Platform for Autonomous Systems and Control project, grant number: 20150022.

REFERENCES

[1] 2011. Acceleo - Transforming Models into Code.

[2] Rajeev Alur and David L. Dill. 1994. A Theory of Timed Automata. *Theoretical Computer Science* 126 (1994).

[3] Abhishek Anand and Ross Knepper. 2015. ROSCoq: Robots Powered by Constructive Reals. In *Interactive Theorem Proving*, Christian Urban and Xingyuan Zhang (Eds.). Springer International Publishing, Cham, 34–50.

[4] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT press.

[5] Kai Beckmann and Marcus Thoss. 2010. A model-driven software development approach using OMG DDS for wireless sensor networks. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 95–106.

[6] Christophe Bédard, Ingo Lütkebohle, and Michel Dagenais. 2022. ros2_tracing: Multipurpose Low-Overhead Framework for Real-Time Tracing of ROS 2. *IEEE Robotics and Automation Letters* 7, 3 (2022), 6511–6518.

[7] K. Birman and T. Joseph. 1987. Exploiting virtual synchrony in Distributed Systems. *Proceedings of the eleventh ACM Symposium on Operating systems principles* (1987). https://doi.org/10.1145/41457.37515

[8] Tobias Blaß, Daniel Casini, Sergey Bozhko, and Björn B Brandenburg. 2021. A ROS 2 Response-Time Analysis Exploiting Starvation Freedom and Execution-Time Variance. In *IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 41–53.

[9] C. S. V. Gutiéerrez. 2018. Towards a Distributed and Real-Time Framework for Robots: Evaluation of ROS 2.0 Communications for Real-Time Robotic Applications. *Tech. Rep* (2018).

[10] Renato Carvalho, Alcino Cunha, Nuno Macedo, and André Santos. 2020. Verification of system-wide safety properties of ROS applications. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[11] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn Brandenburg. 2019. Response-time analysis of ros 2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems*. 1–23.

[12] Silvano Dal Zilio, Pierre-Emmanuel Hladik, Félix Ingrand, and Anthony Mallet. 2023. A formal toolchain for offline and run-time verification of robotic systems. *Robotics and Autonomous Systems* 159 (Jan. 2023), 104301.

[13] Lukas Dust, Rong Gu, Cristina Seceleanu, Mikael Ekström, and Saad Mubeen. 2023. Pattern-Based Verification of ROS 2 Nodes Using UPPAAL. In *International Conference on Formal Methods for Industrial Critical Systems*. Springer, 57–75.

[14] Lukas Dust, Emil Persson, Mikael Ekström, Saad Mubeen, Cristina Seceleanu, and Rong Gu. 2023. Experimental Evaluation of Callback Behavior in ROS 2 Executors. In *28th International Conf. on Emerging Technologies and Factory Automation*.

[15] Object Management Group. 2022. Object Management Group. https://www.omg.org/ Accessed: 2022-11-7.

[16] Raju Halder, José Proença, Nuno Macedo, and André Santos. 2017. Formal Verification of ROS-Based Robotic Applications Using Timed-Automata. In *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*. 44–50.

[17] M. Hendriks, Wang Yi, P. Petterson, J. Hakansson, K.G. Larsen, A. David, and G. Behrmann. 2006. UPPAAL 4.0. In *Third International Conference on the Quantitative Evaluation of Systems - (QEST'06)*.

[18] Seulbae Kim and Taesoo Kim. 2022. RoboFuzz: fuzzing robotic systems over robot operating system (ROS) for finding correctness bugs. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 447–458.

[19] Zihang Li, Atsushi Hasegawa, and Takuya Azumi. 2022. Autoware_Perf: A tracing and performance analysis framework for ROS 2 applications. *Journal of Systems Architecture* 123 (2022), 102341.

[20] Yanan Liu, Yong Guan, Xiaojuan Li, Rui Wang, and Jie Zhang. 2018. Formal analysis and verification of DDS in ROS2. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. IEEE.

[21] OMG. 2011. MOF 2.0 Query/View/Transformation Spec. V1.1.

[22] OpenRobotics. 2023. ROS : Distributions. http://wiki.ros.org/Distributions

[23] OpenRobotics. 2023. ROS 2: Documentation. https://docs.ros.org/en/humble

[24] Samuel Parra, Sven Schneider, and Nico Hochgeschwender. 2021. Specifying QoS Requirements and Capabilities for Component-Based Robot Software. In *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*.

[25] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. 2010. Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th design automation conference*. 731–736.

[26] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S De Boer. 2004. Modeling and verification of reactive systems using Rebeca. *Fundamenta Informaticae* 63, 4 (2004), 385–410.

[27] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

[28] Matt Webster, Clare Dixon, Michael Fisher, Maha Salem, Joe Saunders, Kheng Lee Koay, Kerstin Dautenhahn, and Joan Saez-Pons. 2016. Toward Reliable Autonomous Robotic Assistants Through Formal Verification: A Case Study. *IEEE Transactions on Human-Machine Systems* 46, 2 (2016), 186–196.