Pattern-Based Verification of ROS 2 Nodes using UPPAAL

Lukas Dust, Rong Gu, Cristina Seceleanu, Mikael Ekström, Saad Mubeen

Mälardalen University, Västerås, Sweden
`(first.last)@mdu.se`

Abstract. This paper proposes a pattern-based modeling and Uppaal-based verification of latencies and buffer overflow in distributed robotic systems that use ROS 2. We apply pattern-based modeling to simplify the construction of formal models for ROS 2 systems. Specifically, we propose Timed Automata templates for modeling callbacks in Uppaal, including all versions of the single-threaded executor in ROS 2. Furthermore, we demonstrate the differences in callback scheduling and potential errors in various versions of ROS 2 through experiments and model checking. Our formal models of ROS 2 systems are validated in experiments, as the behavior of ROS 2 presented in the experiments is also exposed by the execution traces of our formal models. Moreover, model checking can reveal potential errors that are missed in the experiments. The paper demonstrates the application of pattern-based modeling and verification in distributed robotic systems, showcasing its potential in ensuring system correctness and uncovering potential errors.

Keywords: Robot Operating System 2 · Pattern-Based Modeling · Model Checking.

1 Introduction

Robotic systems are often distributed systems consisting of sensors, actuators, and controllers. Communication among these components involves buffers and job scheduling, which demand an elaborate design and extensive testing to ensure the correctness of the system. To provide the foundation and standardize the design of distributed robotic systems, the Robot Operating System (ROS) [13] has been developed as an open-source middleware, which allows fast prototyping for robotics. Since 2015, ROS has been upgraded, and ROS 2 [12] has been released consequently. ROS 2 utilizes Data Distribution Service (DDS) and supports real-time execution and communication. Both generations of ROS have drawn increased interest in academia and industry in recent years [1]. However, the consortium of developers decides to end the support of ROS in 2025 [10]. Hence, robotic systems using ROS need to be updated to ROS 2, with potential behavioral consequences that have not been investigated enough in the community. In order to improve the real-time capability of ROS 2, research on response-time analysis of processing chains [7, 5] and end-to-end timing analysis [16] has been conducted. These proposed analytical methods facilitate the verification of ROS 2 systems, but require manual and intensive computation that varies from system to system. When implementing a ROS 2 system, many options for system configuration are presented to the developer, such as the buffer sizes of DDS communication. Without tool-supported automation, these analytical methods are hard to employ.

Another difficulty of analyzing ROS 2 systems is due to the multiple versions of ROS 2 [11] and the lack of documentation. For instance, ROS 2 comes with an inbuilt module, called executor, which conducts the internal scheduling of functions (a.k.a. *callbacks*) that are triggered on the arrival of data or timers. Every main component of a ROS 2 system (i.e., *node*) consists of at least one such executor, and every communication channel has a configurable size of input and output buffers. In ROS 2 versions up to *Dashing*, timers are scheduled independently from other kinds of callbacks (e.g., publishers and subscribers), whereas in ROS 2 *Eloquent* and *Humble*, timers are scheduled together with other callbacks [5]. These give rise to two different ROS 2 executor semantics.

The different scheduling mechanisms in ROS 2 have been discovered, but not sufficiently regarded in the literature, which hinders ROS 2 developers from predicting the execution order of callbacks. As a result, they may exhibit abnormal phenomena, such as *instance misses* of timers and unexpected *latency* of callbacks. Instance misses occur when a buffer overflow leads to instances being skipped, while the latency of a callback describes the maximum time between the release of a callback instance and the time of completing its execution. Furthermore, although robotic systems, as a type of cyber-physical systems [4], are mostly safety-critical, trial-and-error approaches are dominant in verifying and testing them [14]. Such methods are not systematic, are hard to automate and error-prone.

Formal methods, such as *model checking*, are well-known for providing mathematical and rigorous analysis of complex systems. Model checking enables an automatic exploration of the system's state space, based on which an exhaustive verification is conducted to check if the system satisfies its specification. However, one of the drawbacks of using model checking is the complicated process of formal modeling, especially when the system is distributed and complex. To overcome such a difficulty, we propose a *pattern-based* modeling approach for ROS 2 systems. Additionally, ROS 2 systems are often real-time, which means that they are subjected to timing requirements such as scheduling a periodic callback every 2 ms within a certain level of jitter. Consequently, we employ *timed automata* (TA) [2] as the modeling language and Uppaal [9] as the model checker in this paper, due to their ability of expressing and verifying such requirements, respectively.

We report our experimental results on ROS 2 systems, which reveal problems of scheduling in different versions of ROS 2. In addition, we present our pattern-based modeling of ROS 2 systems, and demonstrate the capability of model checking in the automatic verification of ROS 2 systems. Our TA models are experimentally validated, as the behavior of the ROS 2 systems shown in our experiments are also presented in the verification of the TA models. Moreover, we show that our Uppaal-based verification can even reveal potential errors that are not detected via experiments.

In summary, we answer the following research questions. **RQ1**: Given the two behavioral semantics of single-threaded executors in ROS 2, how to ensure the correctness of a design of ROS 2-based systems with respect to the behavior of timer callbacks, callback latency, and the sizes of input buffers of callbacks? **RQ2**: What are the patterns for modeling ROS 2 systems, which can be reused in verification? **RQ3**: Can model checking find the errors of ROS 2 systems, which are shown in experiments, but also reveal potential errors that are not discovered via experimental evaluation? Our contributions are as follows:

– We demonstrate the difference in callback scheduling and the potential errors in different versions of ROS 2, through experiments and formal verification, which has not been addressed in the literature.
– We design patterns for modeling ROS 2 systems, which simplify the complex construction of formal models that require configuration of parameters.
– Our formal models of ROS 2 systems are validated by experiments. The behavior of ROS 2 presented in the experiments is also exhibited when model-checking our formal models. Moreover, we show that model checking is able to reveal potential errors that are missed in the experiments.

2 Background

To make the remainder of the paper comprehensible, this section introduces the concepts in ROS 2 first, including the internal scheduling mechanism. Next, Timed Automata and Uppaal are briefly overviewed.

2.1 ROS 2

ROS 2 is an open-source middleware that enables fast prototyping and developing distributed robotic systems. ROS 2 is continuously updated, and distributions are released as stable versions that do not change.

ROS 2 system model. ROS 2 systems are composed of the so-called *nodes* representing the core elements in the system that are communicating with each other. An example of such a system is given in Fig. 1, where two nodes, represented by green ellipses, communicate with each other over defined communication channels of ROS 2. There are two types of communication, *publisher-subscriber* and *service-client* communication, represented by the red and blue boxes in Figure 1, respectively.

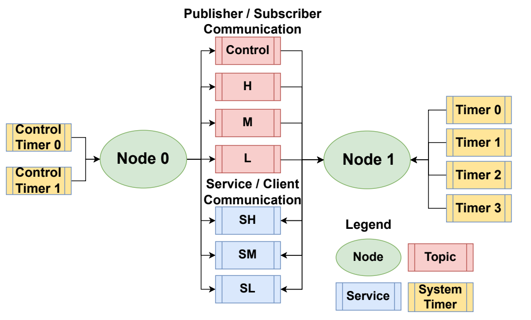


Fig. 1. Schematic example of a ROS 2 system containing two nodes communicating with each other.

The publisher-subscriber communication is unidirectional: a sending node sends data, by utilizing the so-called *publisher*, to all nodes that subscribe to the specific communication channel called *topic*. The event of incoming data in the receiving node triggers a subscription-specific function called *subscription callback*. The second kind of communication, is the service-client communication. This kind of communication is bidirectional. A directed request is sent from the requesting node (Node0) to the receiving node (Node1). Triggered by that request, a *service callback* starts executing in Node1. The response of the *service callback* triggers a *client callback* on arrival in the requesting Node0. Every registered access to a communication channel where data is received (Subscriber, Service, Client) has an individual FIFO input buffer, whose size is configurable. Besides communication, there is another method of triggering callbacks in ROS 2, via the system *timers*, denoted in yellow in Fig. 1. Timers can be configured as periodic or sporadic, whereas in the latter case, timer callbacks are triggered several times sporadically (i.e. when given a set of wall times).

ROS 2 scheduling and execution. The execution of ROS 2 relies on a host operating system that assigns resources to each node that executes the ROS 2 internal scheduler module, called *the executor*. The executor's task is to schedule the execution order of the presented four types of callbacks (subscriber, service, client, timer). The default executor operates on a single thread in the host operating system. The scheduling performed by the executor is non-preemptive and polling-point based, meaning that a callback cannot interrupt an other callback's execution, and only the callbacks that have been released before the time when the executor polls are considered for scheduling. Furthermore, an instance of polling is performed when only one instance of every released callback from the previous polling instance has finished its execution. While ROS 2 has evolved, the executor has changed so that there exist two versions of the executor (ExV1 [7] and ExV2 [5]). The main difference between the two versions is that timers are polled continuously after each execution of a callback in ExV1, while in ExV2, timers are polled after all the callbacks from the previous polling instance have finished their execution. Specifically, in ExV2, timers are polled together with any other types of callbacks.

2.2 Timed Automata and UPPAAL

In this subsection, we introduce the formal definitions of Timed Automata (TA) and the semantics as well as a TA-based model checker UPPAAL. In the interest of space, we refer to the literature [2, 9] for detailed and precise introduction of these concepts. Understanding the theory of Timed Automata and the mechanism of model checking in UPPAAL is not required for this work.

Definition 1. *A Timed Automaton (TA) [2] is a tuple:*

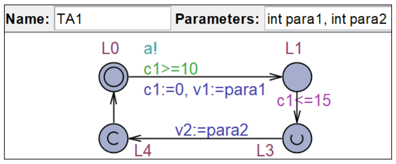
$$\mathcal{A} =< L, l_0, C, \Sigma, E, Inv >, \tag{1}$$

where L is a finite set of locations, l_0 is the initial location, C is a finite set of non-negative real-valued variables called clocks, Σ is a finite set of actions, $E \subseteq L \times \mathcal{B}(C) \times \Sigma \times 2^C \times L$ is a finite set of edges, where $\mathcal{B}(C)$ is the set of guards over C, that is, conjunctive formulas of constraints of the form $c_1 \bowtie n$ or $c_1 - c_2 \bowtie n$, where $c_1, c_2 \in C$, $n \in \mathbb{N}$, $\bowtie \in \{<, \leq, =, \geq, >\}$, 2^C is a set of clocks in C that are reset on the edge, and $Inv : L \to \mathcal{B}(C)$ is a partial function assigning invariants to locations. □

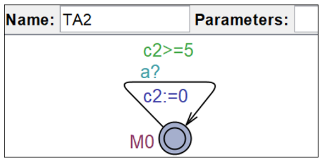
Definition 2. *Let $< L, l_0, C, \Sigma, E, Inv >$ be a TA. Its semantics is defined as a* labelled transition system *$< S, s_0, \to >$, where $S \in L \times \mathbb{R}^C$ is a set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\to \subseteq S \times (\mathbb{R}_{\geq 0} \cup \Sigma) \times S$ such that:*

1. *delay transition: $(l, u) \xrightarrow{d} (l, u \oplus d)$, where $d \in \mathbb{R}_{\geq 0}$, and $u \oplus d$ is a new evaluation of clocks such that $\forall d' \leq d, u \oplus d' \models Inv(l)$, and*
2. *action transition: $(l, u) \xrightarrow{a} (l', u')$, if there exists $e = (l, g, a, r, l') \in E$ such that $u \in g$, $u' = [r \mapsto 0]u$ is a new evaluation of clocks that resets $c \in r$ and keeps $c \in C \setminus r$ unchanged, and $u' \models Inv(l')$.* □

Uppaal [9] is a tool that supports modeling, simulation, and model checking of an extension of TA (*UTA*). In Uppaal, UTA are modeled as *templates* (see Fig. 2) that can be instantiated. UTA extends TA with data variables, synchronization channels, urgent and committed locations, etc. In Uppaal, UTA can be composed in parallel as a *network* of UTA (NUTA) synchronized via *channels*. Fig. 2 depicts an example of a NUTA in Uppaal, where blue circles are *locations*



(a) UTA template *TA1*



(b) UTA template *TA2*

Fig. 2. An example of UTA templates in Uppaal

that are connected by directional *edges*. Double-circled locations are the *initial* locations (e.g. `L0`). Encircled "u" denotes *urgent* locations (e.g. `L3`), and encircled "c" denotes *committed* locations (e.g. `L4`). UTA require that time does not elapse in those two kinds of locations and committed locations are even stricter, that is, the next edge to be traversed must start from one of them. On the edges, there are assignments resetting clocks (e.g., `c1:=0`) and updating data variables (e.g., `v1:=para1`), guards (e.g., `c1>=10`), and synchronization channels (e.g., `a!` and `a?`). At location `L1`, an invariant `c1<=15` regulates that clock `c1` must never exceed 15 time units. In Uppaal, templates of UTA can have parameters (e.g., `para1` in TA1) that are assigned values when the UTA are instantiated.

The Uppaal queries that we verify in this paper are of the form: (i) **invariance**: `A[] p` means that for all paths, for all states in each path, `p` is satisfied, and (ii) **supremum evaluation**: `sup{con}:list` evaluates the supremum of the expressions in the `list` only when the condition (i.e., `con`) is *true*.

3 Modeling and Verification of ROS 2 Nodes in UPPAAL

In order to answer RQ1 and RQ2, this section explains our modeling approach. Uppaal is the tool for modeling and verification in this paper. In order to derive the needed features of ROS 2 and explain the level of abstraction to understand the conducted modeling, this section starts by defining the constructs and representation of all elements that are needed for the verification of *timer blocking*, *callback latencies*, and *callback input buffer sizes* in ROS 2 systems. Furthermore, the verification should be conducted in a way that allows a simple configuring of models. Therefore, a template-based approach of modeling is followed, and the implementation of the templates is explained.

ROS 2 Feature Selection and System Abstraction. As denoted in section 2.1, a ROS 2 system is composed of multiple nodes communicating with each other over different communication channels, which generates various configurations of parameters, such as buffer sizes, distribution of communication channels and Quality-of-Service (QoS) settings for the communication. Nevertheless, verification in this paper involves only a part of a ROS 2 system, which means that irrelevant components and parameters can be removed or represented as abstract elements of our formal model.

We verify three features of ROS 2 systems, that is, the *latency* and *input buffer sizes* of callbacks, respectively, and the behavior of *timer* callbacks. Next, we introduce the modeling of these three features.

– *Callback latency.* The latency of a callback describes the maximum time between the release of a callback instance and the time of completing its execution. In order to verify the latency of a callback, all components and parameters that influence its execution need to be modeled. In ROS 2, those components and parameters include the scheduling algorithm of the executor, as well as the release and execution mechanisms of all callbacks in the same executor. Overall, in order to generalize the modeling to fit all types of callbacks, it is important to find modeling patterns that can be easily configured without changing the model. The patterns include the modeling of the callback execution, the scheduling of callbacks, which is an abstraction of the executor, and the events triggering the releases of callbacks, which are the arrival of messages and system timer events. An example of a ROS 2 node containing two timers, one subscriber, one service, and one client is shown in Fig. 3. The figure reduces components that are irrelevant to our verification of ROS 2, e.g., all sending communication channels. Attached to the corners of rectangles that represent the ROS 2 components, circles show the corresponding UTA templates. In particular, callbacks and executors are modeled as UTA templates, i.e., encircled *E* and *W*. Specifically, every

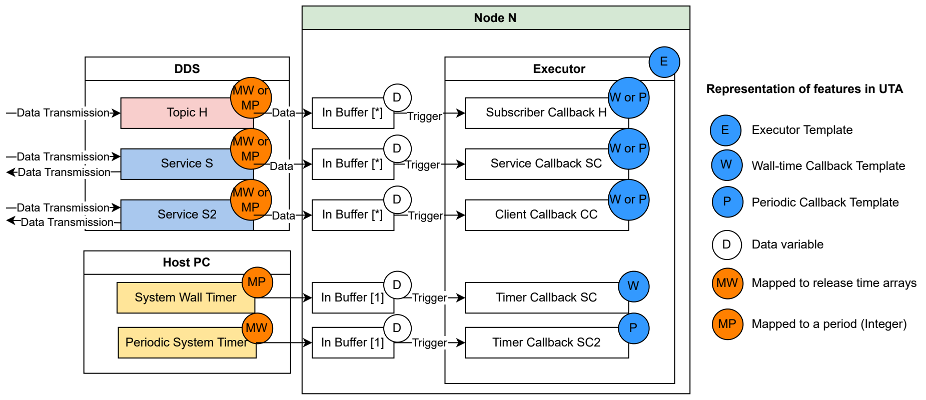


Fig. 3. Selected ROS 2 component features and their abstraction in the UTA model

callback is released on a specific event that is unique for that callback. The event can be either an arrival of data in the input buffer of the callback or an event triggered by an active system timer, called timer event. The arrival of data is dependent on the execution of the sending nodes, which are not included in our verification. Instead, we model the arrival times as integers or multiple parameters of UTA templates, depending on the frequency. Concretely, for representing the periodically arriving data, it is sufficient to pass the length of the period as a parameter of the UTA template of callbacks. Sporadically arriving data can be modeled as an array of integers representing the specified arrival times relative to the global time. Therefore, only two different templates of callbacks are needed. Other specifications that are callback specific and that influence the scheduling and execution are passed as parameters to the UTA template of that particular callback. Such parameters include the type, the ID, and the execution time of a callback. Only the execution time but not the function of a callback is relevant, as the ROS 2 features that we select for verification do not concern the functions of callbacks. Furthermore, the input buffer of a callback also affects its execution and is contained in the UTA template. To verify callback latency, we also need to model the executors of ROS 2. Since there are two versions of the single-threaded executor, we need to design the model to allow simple switching between these two versions. This is achieved by creating a UTA template for each version accordingly. Furthermore, the UTA templates of the callbacks are designed to work with both templates.

– *Callback input buffer size.* The second verification goal of the paper is to verify if the designated input buffer size is sufficient for the data communication and scheduling of callbacks. In the UTA templates of the callbacks, the size of a buffer is passed as an input parameter. Note that timers have a fixed buffer size, which is one, while other callbacks' buffer sizes are configurable. In UTA, we model the actual utilization of a buffer as a counter instead of modeling the buffer itself. From the perspective of verification, actual data in the buffer is irrelevant, but the utilization of the buffer matters. When a callback is released, it is inserted into a buffer, waiting for being scheduled. In our UTA, the corresponding variable of buffer utilization increases, representing the callback being inserted into the buffer. At the beginning of each instance of execution of the callback, the variable is decreased, representing removing the callback instance from the buffer. Therefore, the counter always represents the actual number of elements in the buffer at any point during execution. This allows to perform queries to check if a specific utilization is reached during execution.
– *Timer behavior.* ROS 2 timer callbacks have the highest priority of all callback groups. Nevertheless, the buffer size is statically set to one and cannot be changed by users. This means that, if a timer is released two times before it is executed once, an overflow occurs, where one instance of execution is skipped. In ExV1 (Executor Version 1), timers are considered for scheduling after each callback execution. Therefore, the maximum blocking time of callbacks is the sum of execution time of its previous callback and higher priority timers. In ExV2, timers are only considered for scheduling when all callbacks polled before it have finished their execution, leading the worst-case blocking time to be the sum of the execution time of all callbacks in the system and plus the execution time of timers that have higher priority. Therefore, ExV2 is more vulnerable to blocking, leading to buffer overflow and skipped timer instances. Nevertheless, instance misses caused by buffer overflow can still occur in ExV1. The overflow detection for buffers, which we model in the UTA callback templates, can be used to detect such instance misses.

To summarize, to model and perform verification on latency and buffer sizes for callbacks, only two types of callback templates and an executor model template are needed. The following sections show the modeling approach of the callback and the executor templates.

Modeling of callbacks. As stated in the previous section, our model consists of two UTA templates representing callbacks. The first UTA template models the periodic callbacks. The second UTA template models sporadic callbacks released at specified time points defined in an array. In the following, the general model for both types is derived, and later, the differences in the templates are presented. A simplified version of the UTA templates for the periodic and sporadic callback is presented in Fig. 4, where guards, updates, and invariants are partially shown[1]. Generally, each callback type has four components, the `Waiting`, `Released`, `InReadySet` and execution components, represented by the blue boxes in Fig. 4. The initial location is location `Waiting`. The callback is not ready to be scheduled at that location, as it is waiting for an event to arise, which triggers the callback. At location `Released`, the triggering event has occurred, and the callback is ready to be polled by the executor, meaning that it is to be scheduled. This polling is modeled as an edge from location `Released` to location `InReadySet`, labeled by a broadcast channel `poll` that synchronizes the callback UTA with the executor UTA. Traversing this edge models ROS 2's behavior of including a callback into the so-called ready set in the executor. The executor only considers callbacks included in such a ready set for scheduling. A callback being scheduled is modeled by the edge from `InReadySet` to `StartExecution`, where the execution component of the UTA template starts.

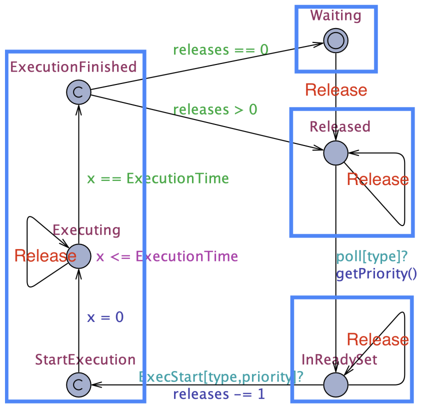


Fig. 4. UTA template overview representing callbacks.

A callback being executed is modeled as the execution component of the UTA template. The component contains three locations for the sake of facilitating the verification. Location `StartExecution` is where the execution starts. At location `Executing` and its outgoing edge to `ExecutionFinished`, we have an invariant and a guard, respectively, meaning that the callback model is forced to stay at location `Executing` for `ExecutionTime` long. This models the execution time of the callback. Finally, the execution finishes when the UTA can go back to location `Waiting` or location `Released`, depending on the availability of releases.

The type of callback is essential for the scheduling of callbacks. Therefore, the callback type is passed as an input parameter to the template. Furthermore, the priority of a callback is important to determine the execution order among all other callbacks of the same type. The priority is determined by the callback registration time, where the earliest registered callback has the highest priority. In our model, priorities are passed to the callback template as input parameters, where 0 is the highest priority, representing the callback that is initialized first. Priorities must be unique within each group with the same type of callbacks.

To check the utilization of the input buffer of callbacks, the UTA contains an integer named *releases* representing the number of elements contained in the buffer that stores callback instances. With a model parameter representing the maximum buffer size, those two elements are sufficient to model the input buffer

[1] The complete model is published: https://sites.google.com/view/pbvros2nodes

of a callback. When a callback instance is released, the callback input buffer is checked before the instance is stored in a buffer and waiting for being scheduled. When the number of instances waiting in the buffer exceeds the maximum buffer size, an overflow happens. Our verification of the buffer size checks whether the value of *releases* ever exceeds the maximum buffer size. Therefore, a designated boolean variable *NoOverflow*, true by default, is set to false when an overflow occurs. Therefore, it can be used as a property in the verification, indicating buffer overflow, and generating counterexamples of cases where an overflow occurs.

In the callback UTA (Fig. 4), every location has a self-loop edge because a new instance of a callback can be released at any point during the life cycle of an existing callback. We label the self-loop edges with red `Release` statements. The `Release` edges are guarded by the next releasing time, meaning that callbacks can only be released when the releasing time comes. While transferring via the `Release` edge, the variable *releases* is increased, representing the filling of the buffer, and the subsequent release time is calculated. Furthermore, a timer, specific to the instance and contained in the array *ReleaseTimers* is reset.

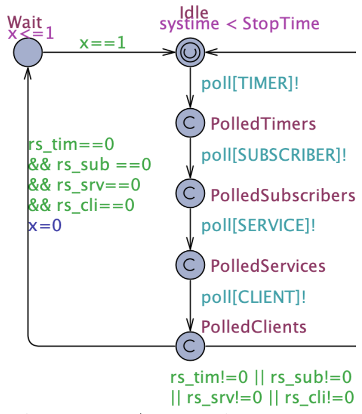
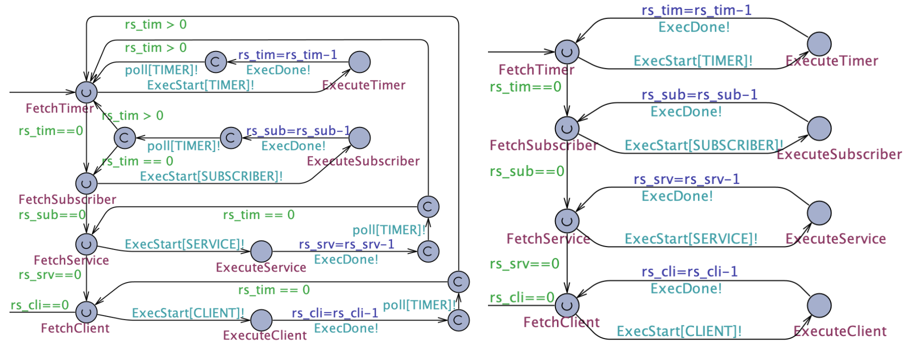


Fig. 5. UTA template extract showing the polling of callbacks in the executor

ROS 2 Executor. The executor model is needed to schedule the callbacks following the ROS 2 executor algorithm. As explained in Section 2.1, two versions (ExV1 and ExV2) of the single-threaded executor exist, which are specific to ROS 2 distributions. The executor versions differ in their approach to polling callbacks. Both executors are provided as a template in this section so that the executor can be chosen in later systems configurations by instantiating the template of the desired version.

In both versions of the executor, the model consists of two main components. The idle component presented in Fig. 5 is valid for both versions of the executor. The execution component differs between ExV1 (Fig. 6(a)) and ExV2 (Fig. 6(b)). In the Idle component in Fig. 5, at initial location `Idle`, we have an invariant constraining the system time and the UTA input parameter *StopTime*. This invariant forces a deadlock after reaching the stop time and is used to reduce the model's state space. When reaching location `Idle` in Fig. 5, both executors use broadcast channels to poll the released callbacks. One broadcast channel is used for each callback type, where the final locations are committed for the polling not be interrupted. Global variables are included in the model to track the number of released callbacks for each type. The variables are increased by each released callback that is using the polling channel (*rs_tim*, *rs_sub*, *rs_srv*, *rs_cli*). If all sets are zero after the polling, the UTA goes to location `Wait`. At location `Wait` and its outgoing edge to `Idle`, we use an invariant and a guard, respectively, meaning that the callback model is forced to stay at location `Wait` for one time unit. That allows the model to progress in time while the executor is idle.

After the polling of all callbacks (location `PolledClients`), if any of the sets is not zero, the UTA continues with the execution component. The execution component is shown in Fig. 6(a) for ExV1 and in Fig. 6(b) for ExV2. In the execution component of both versions of the executor, the execution of callbacks



(a) UTA template extract showing the execution component of ExV1

(b) UTA template extract showing the execution component of ExV2

Fig. 6. Execution component of the executor UTA templates

is performed, ordered by the group and the priorities of callbacks. Timers are executing first, followed by subscribers, services, and clients. This is modeled by the executor component starting at location `FetchTimer`. At this location, if any timer is contained in *rs_tim* the synchronized channel `ExecStart` is used to model the beginning of the execution of a callback. The implemented channel logic only allows one callback (of the corresponding type) at a time to use the channel. The order is determined by the priority of the callback. Triggered by the callback UTA, in ExV1 (Fig. 6(a)), once the execution time has passed, the channel `ExecDone` is decreasing *rs_tim*. The target location is leading to another channel `poll[TIMER]` that performs the polling of timers only, leading back to location `FetchTimer`. In ExV2 (Fig. 6(b)), once the execution time has passed, the channel `ExecDone` leads back to `FetchTimer` immediately, such that no polling of timers is carried out. When *rs_tim* is empty, the next group is executed following the same process. In ExV1, if the variable *rs_tim* is increased under the polling of timers, the UTA goes back to `FetchTimer` in order to execute the obtained timer first.

System Declaration. When modeling a ROS 2 node, the created UTA templates are instantiated in the system declaration. Every system model consists of one executor and one or many callbacks. In our system model, assigning the executor the lowest priority of all system components is essential to generate the right results and allow progression in time while the executor remains idle.

Verification. To simplify the verification of occurring buffer overflow, the model of each callback contains the variable `NoOverflow`, which is true as long as no overflow occurs. Hence the absence of a buffer overflow is confirmed when the following invariance holds, where `cb` stands for any callback in our UTA model, and A[] means for all states in every possible path:

$$\texttt{A[] cb.NoOverflow} \tag{2}$$

A buffer overflow is always equivalent to a missed callback instance. Furthermore, it can be verified if a specified amount of releases in the buffer of any callback is reached at any time during execution. This is achieved by checking the variable `releases`, as shown in the following example query (Query (3)). Nevertheless, given the model implementation approach, the value of the variable `releases` will never exceed the defined buffer size. Hence, only values smaller than the configured parameter `bufferSize` can be verified using that approach, showing if a smaller buffer size would be sufficient.

```
A[] cb.releases <= 5                                   (3)
```

When the query passes, the buffer of the verified size (e.g., 5 in Query (3)) is sufficient, and the buffer size can be adapted.

The latency of a callback is defined as the maximum time from the release of any instance of a callback until the end of execution of that specific instance. The latency of any callback instance can be determined by reading the value of the instance-specific timer contained in `ReleaseTimers` at location `ExecutionFinished`. Using the following supremum evaluation in Uppaal, the longest latency of a callback is determined.

```
sup{cb.ExecutionFinished}:cb.releaseTimers[cb.instance]    (4)
```

4 Evaluation of Pattern-based Verification

To evaluate the proposed model templates, we compare the simulation and verification of the model to system traces gained by system execution. The evaluation is performed on two scenarios of ROS 2 message passing between two nodes, utilizing the ROS 2 system shown in Fig. 1. The first scenario (SC1) focuses on sporadic callbacks only, while the second scenario (SC2) combines sporadic and periodic callbacks. In SC2, using ExV2, a buffer overflow occurs, leading to instance misses. The experimental system and the scenario composition are explained briefly, followed by the comparison of model-based verification with the execution trace of one system execution.

Experimental System. The experimental ROS 2 system (Fig. 1) consists of two nodes that communicate. `Node0` is the control node used to control the experiment by sending sequences of messages and service requests to `Node1` at the defined times, and `Node1` is the system under verification. Communication channels are named after the priority of the callback in `Node1`, which they belong to. The topics are called *H, M, L* (High, Medium, Low) and services *SH, SM, and SL* (Service High, Service Medium, Service Low), respectively. Furthermore, one control topic exists to control the experiment by setting the timers in `Node1`. Each callback in `Node1` is programmed to execute 500 ms, for simplifying evaluation. Moreover, each callback records its execution information in the ROS 2 log when execution is started and finished. By using this information, we can compare the execution trace of our model to the actual execution of callbacks in a real ROS 2 system. The system is set up on a single computer using docker to create a controlled environment and enable simple switching among the different ROS 2 distributions. The selected distributions for the experiments are ROS 2 *Dashing* (the last distribution that contains ExV1), *Eloquent* (the first distribution that contains ExV2), and *Humble* (the latest long time stable distribution that contains ExV2). In this evaluation, each scenario comprises different message sequences. The specific sequences and the resulting execution traces from one actual execution are shown in the following.

Scenario 1 (SC1): Sporadic-Callbacks. The first scenario is taken from the experimental evaluation of the scheduling algorithm of ExV1 conducted in Casini et al. [7]. The scenario utilizes four timers, three subscribers, and three services.

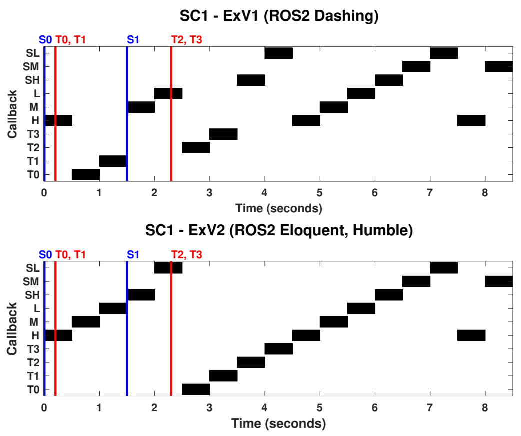


Fig. 7. Execution diagram of Node1 executing SC1 on different ROS 2 distributions. Sequence releases are marked in blue and releases of timers in red.

All callbacks in `Node1` can be modeled as sporadic callback UTA, each executing a defined number of times. This scenario transmits two message sequences (*S0, S1*). The first sequence *S0* is released at time 0, triggering the following callbacks: <*L; M; H; SH; SL; L; M; H; SH; SL*>. The second sequence *S1* triggers $< SM; SM; H >$ and is sent at 1.5 seconds. Timers T0 and T1 are configured to release after 0.2 seconds, while T1 and T2 release after 2.3 seconds. Fig. 7 shows the resulting execution traces when executing the scenario with ExV1 and ExV2. There are significant differences in execution. After timers are released (e.g., T0 and T1), ExV1 (upper trace) schedules them immediately after the execution of the current callback, whereas ExV2 (bottom trace) schedules timers after all the callbacks remaining in the ready set are finished.

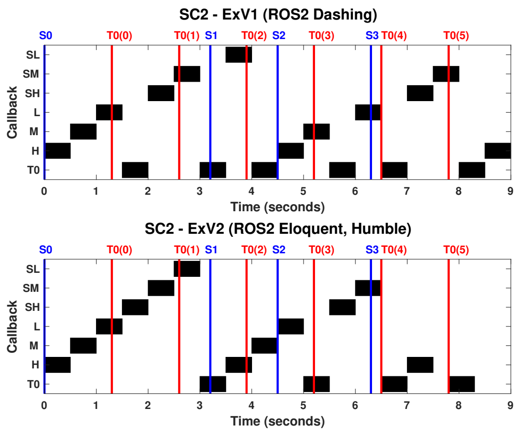


Fig. 8. Execution diagram of Node1 executing SC2 on different ROS 2 distributions. Sequence releases are marked in blue and releases of timers in red.

Scenario 2 (SC2): Periodic Timer and Buffer Overflow. SC2 utilizes only one timer (`T0`) in `Node1`, releasing periodically every 1.3 seconds. Four sequences (*S0, S1, S2, S3*) are contained in SC2. *S0* is released at time 0, triggering the following callbacks in `Node1`: $< H; M; L; SH; SM; SL >$. *S1* at 3.2 seconds triggers $< H; M; L >$. *S2* is released at 4.5 seconds, triggering $< SH; SM >$ followed by *S3* triggering $< H >$ at 6.3 seconds. Fig. 8 shows the resulting execution traces of one real execution. Differences in execution can be seen. For SC2-ExV2, it is visible that even though the timer `T0` is released six times during the experiment, only four instances are executed. This is caused by the timer getting blocked over consecutive releases and the input buffer overflowing. In SC2-ExV1, instead, `T0` is executed for every release instance and is scheduled during the next scheduling action after its release.

UPPAAL Model Configuration. The proposed UTA templates are instantiated to model the created scenarios. For example, to model the execution of `Node1`, the model must contain one executor and all included callbacks. An excerpt of instantiated templates can be found in Tab. 1. The initialization of callbacks `H` and `T0` are shown. Release time arrays and other variables such as *releasesH* and *StopTime* are declared at the beginning of the system configuration. The given example shows the initialization of ExV1. One time step in the created model is chosen to be equivalent to 100 ms, as it is the greatest common divisor of all release and execution times. Sporadic callbacks are instantiated with the following parameters: the ID, the execution time (5, representing 500 ms), the number of releases in the scenario, the release times, the type of callback, and the buffer size. The passed parameters to the periodic timer `T0` are the ID, the execution time (5 for 500 ms), the period (13 for 1.3 seconds), the type, the initial release (0 for not released at initialization), and a constant one as the buffer size.

Table 1. Excerpt from the UPPAAL system configuration for SC2 instantiated system

```
H = WallTimeCallback(0, 5, 3, releasesH, SUBSCRIBER,10);
T0 = PeriodicCallback(0, 5, 13, TIMER, 0, 1);
ExecV1 = ExecutorV1(StopTime);
system ExecV1 < H, M, L, SH, SM, SL, T0;
```

System Simulation in UPPAAL. The symbolic simulator in UPPAAL generates random traces of the model. An excerpt from such a trace can be found in Fig. 9, where timer `T0` is executed. The simulated traces are compared to the traces of the actual execution, out of which one crucial finding emerges. In the execution of ExV1 in SC2, there can be a deviation between the simulation and the trace of the actual execution. In some cases, the order of execution of one instance of callback `SH` and `T0` changes compared to the actual execution. This is because some transitions, such as releases, which occur independently from the scheduling of callbacks in ROS 2, may be running in parallel with the execution of other callbacks and influence the execution order nondeterministically. Therefore, the scheduling outcome differs depending on the action that the model takes first. As the experiments using the actual execution are not exhaustive, only one of all possible paths is obtained is this case. The difference between the simulation and the actual execution result shows the model's usefulness in determining worst-case scenarios. During verification, all possible paths are considered in the model. Nevertheless, it is possible to fix the execution order at the model level, by prioritizing the UTA of `T0` over the other callback UTAs in UPPAAL. This impacts the execution order in the model so that `T0` always executes before `SH`. Hence, prioritizing T0 can make the UTA model only reflect the observed real execution.

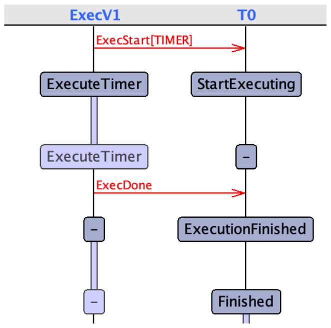
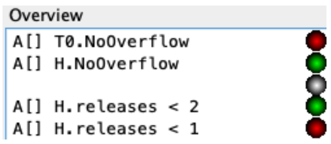


Fig. 9. A part of the UPPAAL simulation trace of ExecV1 and T0

Verification of Buffer Sizes. In order to verify buffer sizes for detecting buffer overflow, the model contains a Boolean variable for each callback, which turns true when a buffer overflow occurs. Now, we define CTL queries in UPPAAL for verification. Examples of the queries and results in UPPAAL can be found in Fig. 10. It can be seen that a buffer overflow is detected in timer `T0` (i.e., the first query is unsatisfied). In general, all verification results correspond to the actual execution. To refine and verify specific buffer sizes that are smaller than the


Overview

```
A[] T0.NoOverflow          🔴
A[] H.NoOverflow           🟢
                           ⚪
A[] H.releases < 2         🟢
A[] H.releases < 1         🔴
```

Fig. 10. Example of queries verifying buffer overflow and utilization

configured buffer size in the model, we design queries containing specific numbers of buffer sizes. For example, the last two queries in Fig. 10 check whether the buffer of callback H always contains less than two, and less than one element, respectively.

Verification of Latencies. In order to find out the latency of a callback, the created array with the release times of each instance of a callback is utilized, and a query that searches for the highest possible value of the time elapsed from release time to the end of the execution for each callback is generated. It is essential to notice that the model disregards instances missed due to a buffer overflow, only showing the latency for executed callbacks. Therefore, it is always essential to check for buffer overflow as well. Furthermore, for periodic callbacks, buffer overflow is also determinable by the latency being higher than one period plus the execution time. During this evaluation, we calculate the latency of each callback using the model and compare the results to the extracted latencies from the actual execution (Fig. 7 and Fig. 8). The results from the model and the actual execution can be found in Tab. 2. The table shows that for all callbacks, the latency is the same as in the actual execution, except in ExV1 for T0 (marked in red). This is related to the possibility of one timer instance executing after the SH callback, and Uppaal calculating the latencies of all possible executions.

Table 2. Comparison of determined latencies

	SCENARIO 1				SCENARIO 2			
	ExV1		ExV2		ExV1		ExV2	
Callback	Uppaal	Real	Uppaal	Real	Uppaal	Real	Uppaal	Real
SL	7.5	7.5	7.5	7.5	4.0	4.0	3.0	3.0
SM	7.0	7.0	7.0	7.0	3.5	3.5	2.5	2.5
SH	6.5	6.5	6.5	6.5	3.0	3.0	2.0	2.0
L	6.0	6.0	6.0	6.0	3.3	3.3	1.8	1.8
M	5.5	5.5	5.5	5.5	2.3	2.3	1.3	1.3
H	6.5	6.5	6.5	6.5	2.7	2.7	1.2	1.2
T0	0.8	0.8	2.8	2.8	1.0	0.9	2.2	2.2
T1	1.3	1.3	3.3	3.3	-	-	-	-
T2	0.7	0.7	1.7	1.7	-	-	-	-
T3	1.2	1.2	2.2	2.2	-	-	-	-

Answer to the Research Questions. Based on the experimental evaluation of our method, we answer the research questions as follows:

RQ1: Given the two behavioral semantics of single-threaded executors in ROS 2, how to ensure the correctness of a design of ROS 2-based systems with respect to the behaviour of timer callbacks, callback latency, and the sizes of input buffers of callbacks?

Answer: In this work, we apply formal modeling and model checking to verify callback latency and input buffer sizes. Utilizing pattern-based verification, we simplify the modeling process by creating templates for callbacks and executors that can be used to model any ROS 2 node.

RQ2: What are the patterns for modeling ROS 2 systems, which can be reused in verification?

Answer: A selection of essential features matching the verification goal is chosen during the creation of the formal models. The communication and execution of other nodes in the processing chain can be abstracted away and represented through release times of callbacks. Generally, any callback can be modeled as a periodic or a sporadic callback with defined releases. Other parameters, such as the type of callback, execution time, and priority, are parameters passed to the template. Callback input buffers can be represented as counters, where buffer size can be configured for callbacks triggered by communication, and set to one for timer callbacks. The data sent in the communication is irrelevant to the verification outcome. Two executor templates can be used to model ExV1 and ExV2, as both can execute the callback templates. The validity of the patterns is shown by comparing the simulation traces of the model and actual system execution.

RQ3: Can model checking find the errors of ROS 2 systems, which are shown in experiments, but also reveal potential errors that are not discovered via experimental evaluation?

Answer: Via a comparison between model-based calculated latencies and experimental results on a real system, we conclude that the model-based verification delivers the same results as the experiments. Furthermore, Uppaal explores all possible execution scenarios. Therefore, model checking can find potential errors in ROS 2 systems that are not discovered in experiments.

5 Related Work

The response time analysis for callbacks in a node using the single-threaded executor and reservation-based scheduling has been proposed by Blass et al. [5], who assume the executor model that includes timers as part of the ready set; next, Tang et al. [15] look into the same problem yet using the executor model with timers being excluded from the ready set. Both works are significant for the research that we present in this paper, and the experiments conducted repeatedly show the differences in the revised executor model.

Formal verification of the communication and computation aspects of ROS 2 has gained significant attention in recent years. Halder et al. [8] propose an approach to model and verify the ROS 2 communication between nodes, using Uppaal. Similar to our approach, the authors consider low-level parameters, such as queue sizes and timeouts, in their TA models, to verify queue overflow, yet they do not consider callback latency verification and do not take into account the two models of ROS 2 single-threaded executors. Moreover, the work does not validate the formal models as we do in our work by using the results of simulation experiments.

Carvalho et al. [6] present a model-checking technique to verify message-passing system-wide safety properties, based on a formalization of ROS launch configurations and loosely specified behaviour of individual nodes, by employing an Alloy extension called Electrum and its Analyzer. The Electrum models are automatically created from configurations extracted in continuous integration and specifications provided by the domain experts. The approach focuses on a high-level architectural-level verification of message passing, whereas our work focuses on a lower-level model checking of the scheduling of single-threaded executors assuming two kinds of timer semantics as found in different distributions of ROS 2. Webster et al. [17] propose a formal verification approach of industrial robotic programs using the SPIN model checker, focusing on behavioral refinement and verification of selected robot requirements. The solution is applied to an existing personal robotic system, it is not ROS-specific, and focuses only on the verification of high-level decision-making rules.

Anand and Knepper [3] present ROSCoq, a Coq framework for developing certified systems in ROS, which involves the use of CoRN's theory of constructive real analysis to reason about computations with real numbers. This work adopts a "correct-by-construction" approach, a different yet complementary approach to ours. However, we verify various timers scheduling mechanisms found in ROS 2 distributions, which is not within the scope of the mentioned work.

6 Conclusions and Future work

In this paper, we introduce and demonstrate the application of pattern-based verification in the context of distributed robotic systems. We create UTA templates for modeling ROS 2 callback execution, allowing pattern-based verification of latencies, buffer overflow, and buffer utilization. Our UTA templates cover two kinds of callbacks and all existing versions of the single-threaded executor in ROS 2. All callbacks in ROS 2 systems using the single-threaded executors can be modeled by those templates. Finally, we show how verification of latencies and input buffer overflow can be performed in two scenarios and compare the verification results with the experimental results of actual system execution. The comparison shows that model checking is able to find potential system errors that might be overlooked by experiments. The verification produces counterexamples, which are infeasible configurations of the system. The models only include the scheduling of ROS 2, assuming 100 percent availability of a system thread in the underlying operating system for the executor module. Detected infeasible configurations remain when fewer resources are available in the underlying OS. Nevertheless, feasible designs in the model may not be feasible in a real system when limited system resources are available to the ROS 2 thread.

The model allows future refinements and improvements, including more system parameters, such as offsets for periodic callbacks, communication latencies, and jitter. In ROS 2, even multi-threaded executors exist for the callbacks, which are not considered in this paper and might become the object of future modeling. In general, the modeling proposed in this paper can be considered a first step towards pattern-based verification in the context of distributed robotic systems using ROS 2.

Acknowledgments

We acknowledge the support of the Swedish Knowledge Foundation via the profile DPAC - Dependable Platform for Autonomous Systems and Control, grant nr: 20150022, the synergy ACICS – Assured Cloud Platforms for Industrial Cyber-Physical Systems, grant nr. 20190038, and HERO - Heterogeneous systems – software-hardware integration, grant nr: 20180039.

Bibliography

[1] Albonico, M., Dordevic, M., Hamer, E., Malavolta, I.: Software engineering research on the robot operating system: A systematic mapping study. J. Syst. Softw. **197**(C) (mar 2023). https://doi.org/10.1016/j.jss.2022.111574, https://doi.org/10.1016/j.jss.2022.111574

[2] Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science **126** (1994)

[3] Anand, A., Knepper, R.: Roscoq: Robots powered by constructive reals. In: Urban, C., Zhang, X. (eds.) Interactive Theorem Proving. pp. 34–50. Springer International Publishing, Cham (2015)

[4] Baheti, R., Gill, H.: Cyber-physical systems. The impact of control technology **12**(1), 161–166 (2011)

[5] Blaß, T., Casini, D., Bozhko, S., Brandenburg, B.B.: A ros 2 response-time analysis exploiting starvation freedom and execution-time variance. In: IEEE Real-Time Systems Symposium. pp. 41–53. IEEE (2021)

[6] Carvalho, R., Cunha, A., Macedo, N., Santos, A.: Verification of system-wide safety properties of ros applications. In: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (2020)

[7] Casini, D., Blaß, T., Lütkebohle, I., Brandenburg, B.: Response-time analysis of ros 2 processing chains under reservation-based scheduling. In: 31st Euromicro Conference on Real-Time Systems. pp. 1–23 (2019)

[8] Halder, R., Proença, J., Macedo, N., Santos, A.: 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). pp. 44–50 (2017)

[9] Hendriks, M., Yi, W., Petterson, P., Hakansson, J., Larsen, K., David, A., Behrmann, G.: Uppaal 4.0. In: Third International Conference on the Quantitative Evaluation of Systems - (QEST'06) (2006)

[10] OpenRobotics: Ros : Distributions (2023), http://wiki.ros.org/Distributions

[11] OpenRobotics: Ros 2: Distributions (2023), https://docs.ros.org/en/humble/Releases

[12] OpenRobotics: Ros 2: Documentation (2023), https://docs.ros.org/en/humble

[13] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y., et al.: Ros: an open-source robot operating system. In: ICRA workshop on open source software. vol. 3, p. 5. Kobe, Japan (2009)

[14] Rajkumar, R., Lee, I., Sha, L., Stankovic, J.: Cyber-physical systems: the next computing revolution. In: Proceedings of the 47th design automation conference. pp. 731–736 (2010)

[15] Tang, Y., Feng, Z., Guan, N., Jiang, X., Lv, M., Deng, Q., Yi, W.: Response time analysis and priority assignment of processing chains on ros2 executors. In: IEEE Real-Time Systems Symposium. pp. 231–243 (2020)

[16] Teper, H., Günzel, M., Ueter, N., von der Brüggen, G., Chen, J.J.: End-to-end timing analysis in ros2. In: 2022 IEEE Real-Time Systems Symposium (RTSS). pp. 53–65 (2022)

[17] Webster, M., Dixon, C., Fisher, M., Salem, M., Saunders, J., Koay, K.L., Dautenhahn, K., Saez-Pons, J.: Toward reliable autonomous robotic assistants through formal verification: A case study. IEEE Transactions on Human-Machine Systems **46**(2), 186–196 (2016)