OPEN ACCESS
International Journal of Software Engineering
and Knowledge Engineering
(2026) 1–23
© The Author(s)
DOI: 10.1142/S0218194026500439

World Scientific
www.worldscientific.com

Modeling and Verification of ROS 2 Callback Concurrency Using UPPAAL Model Checker

Yong-Jun Shin
Electronics and Telecommunications Research Institute
Daejeon, Republic of Korea
yjshin@etri.re.kr

Received 14 November 2025
Revised 27 April 2026
Accepted 18 May 2026
Published 30 June 2026

Robot Operating System 2 (ROS 2) has become the de facto standard middleware for developing robotic applications. The execution of ROS 2 applications heavily relies on callback functions, which execute concurrently within a ROS 2 node. The concurrency of these callbacks significantly impacts application behavior, yet its thorough analysis remains challenging due to the complex interaction of multiple system components. To address the need for an exhaustive analysis of ROS 2 callback concurrency behavior, this paper presents a novel and practical approach for modeling and verification using the UPPAAL model checker. We introduce a set of callback concurrency models, incorporating key elements such as callbacks, callback groups, threads, and executors, alongside model checking queries. Furthermore, we present a tool that automates the generation of UPPAAL model code from high-level specifications, offering a streamlined solution for practitioners. To demonstrate the validity and effectiveness of our approach, we model three representative cases of callback concurrency configurations in ROS 2 and verify four notable concurrency behaviors. Our results accurately capture callback concurrency behavior compared to the execution logs of real ROS 2 applications, making the proposed models and queries reusable and extendible for analyzing ROS 2. The verification reports on recent ROS 2 versions also reveal issues such as callback starvation and unfairly overloaded threads, providing valuable insights for developers and practitioners working with ROS 2 in the industry.

Keywords: Modeling; verification; model checking; ROS2; callback concurrency; UPPAAL.

1. Introduction

Robotic Operating System 2 (ROS 2) has emerged as the de facto standard library and middleware due to its modular architecture, support for distributed

communication, and rich ecosystem of reusable software packages and tools for developing robotic applications in both academia and industry [1]. As robotics systems become more complex, ROS 2 enables the management of multiple nodes within a system, with each node being responsible for specific tasks. These nodes achieve the tasks via callback functions which are triggered by events such as timers, incoming messages, or service requests. The concurrent execution of callbacks by multiple threads is fundamental to the operation of the node and the whole system.

However, the callback concurrency behavior in ROS 2 is difficult to analyze thoroughly [2]. The behavior of callback execution can differ significantly depending on the configuration. Even when the same set of callbacks is used, the behavior can change drastically based on different configurations, as seen in the example execution logs in Listings [5–7]. Misunderstanding or misconfiguring callback concurrency can lead to unexpected outcomes. Some of these may cause crucial issues in the system's operation, such as race conditions [3]. These issues can disrupt the system's behavior, potentially leading to unreliable or unpredictable performance. From the perspective of model-based software engineering, while formal modeling and verification can be a powerful solution to exhaustively analyze callback concurrency behavior in ROS 2, formal modeling of callback concurrency has received little attention since the recent emergence of ROS 2.

In this paper, we propose a novel and practical approach for modeling and verifying the callback concurrency behaviors of ROS 2 nodes using the UPPAAL model checker. Our approach addresses the challenges associated with analyzing callback concurrency in ROS 2, enabling exhaustive verification of all possible execution paths. Specifically, we propose models for ROS 2 callback concurrency, a suite of queries for formal verification, and a verification report based on recent ROS 2 implementations. We also provide a tool for automatic UPPAAL model code generation for ROS 2 practitioners who are novices in formal modeling and verification.

In summary, our contributions are as follows:

- novel ROS 2 callback concurrency **models** and **queries** for UPPAAL model checker,
- a **tool** that automatically converts user-friendly specifications of ROS 2 nodes under verification into UPPAAL model specification codes,
- and the **verification report** of three representative cases and four notable behaviors of callback concurrency in the recent versions of ROS 2.

This paper is structured as follows. Section 2 provides background information on ROS 2 callback concurrency and the UPPAAL. Section 3 reviews the novelty of our approach in comparison with related works. We then develop models and queries in Sec. 4, and the tool in Sec. 5. Section 6 reports the case study results. Section 7 highlights threats and potential value of our approach. Section 8 concludes.

2. Background

2.1. *Callback concurrency in ROS 2*

In ROS 2 [1], a robotic application is composed of a set of nodes running as independent processes. A node contains callbacks executed by threads of the process, where each callback is a function associated with a specific event within a node. Callback functions that handle specific events such as processing messages, managing timers, or responding to service requests. Therefore, ROS 2 enables concurrency at two levels: node concurrency and callback concurrency, both of which must be carefully managed to optimize application functionality. This paper specifically focuses on callback concurrency.

The callback concurrency in ROS 2 is influenced by three primary components: the callback, the callback group, and the executor, which collectively determine how callbacks are scheduled and executed. The *callback* is a function that represents a unit of work, with types including *timer*, *subscription*, *service*, and *client*, listed in order of execution priority. The *callback group* organizes callbacks and dictates their execution rules. It can be either *reentrant*, allowing the simultaneous execution of its callbacks, or *mutually exclusive*, ensuring that only one callback is executed at a time. The *executor*, which can be either single-threaded or multi-threaded, is responsible for managing the scheduling of callbacks. Single-threaded executors process callbacks sequentially, while multi-threaded executors allow concurrent execution of callbacks based on the number of threads set by the user, within the limits of the operating system's capacity. The combination of these components determines the callback concurrency behavior in ROS 2, making its analysis inherently challenging due to the numerous possible configurations and the complex interplay between executors, callback groups, and callbacks. This interaction results in dynamically determined and often non-deterministic scheduling behavior, which is difficult to systematically analyze and verify in practice.

In this paper, we thoroughly analyze ROS 2's callback concurrency using the UPPAAL model checker by modeling the behaviors of these core components.

2.2. *UPPAAL model checker*

To systematically model and verify the complex and non-deterministic callback concurrency behavior in ROS 2, we employ the UPPAAL model checker. UPPAAL [4] is a powerful tool for the modeling, simulation, and verification of real-time systems. It employs automata as a mathematical framework to represent the system's behavior through the non-deterministic transitions between finite states. A key feature of UPPAAL is its ability to visualize the simulation of system models, allowing users to execute the system model step-by-step and observe the system's behavior. Another key feature is its formal verification using model checking which exhaustively explores all potential system states to ensure that specified properties are satisfied. If a property is violated, UPPAAL can also search for and visualize a counterexample path.

UPPAAL supports five types of symbolic queries to express properties of the system model under verification:

- `A[] p` (invariantly): Property `p` holds in all states of all reachable paths.
- `A<> p` (eventually): Property `p` will hold in a state of all reachable paths.
- `E[] p` (potentially always): There exists a path where `p` holds in all states.
- `E<> p` (possibly): There exists a path where `p` holds in a state.
- `p` $\rightarrow$ `q` (leads to): In all paths, whenever `p` holds, `q` will eventually hold.

In this paper, we introduce (1) UPPAAL models that represent the callbacks, callback groups, and executors of ROS 2, and (2) a set of UPPAAL model checking queries designed to verify the callback concurrency behavior in ROS 2.

3. Related Work

Since the first release of ROS 2 (i.e. ROS 2 Ardent) in 2017, numerous studies have been conducted to analyze its behavior. Table 1 summarizes these works, focusing on their coverage of ROS 2 callback concurrency components introduced in Sec. 2. In

Table 1. Comparison of related works analyzing the concurrency of ROS 2.

Ref.	Callback concurrency components analyzed			Formal modeling and verification	Analysis goal
	Callback	Callback group	Executor		
[3, 5]	O	X	O	X	Response time analysis of the naive multi-threaded executor of ROS2
[2, 6]	O	O	O	X	
[7–9]	O	X	O	X	Response time analysis of advanced multi-threaded executors
[10–12]	O	O	O	X	
[13–16]	O	X	X	O	Verification of buffer overflow of the ROS2 node communication
[17]	X	X	X	O	Verification of security and liveness of ROS2 data distribution service
[18]	X	X	X	O	Verification of security of ROS2 node communication mechanisms
[19, 20]	X	X	X	O	Verification of the safety of ROS2 applications
Ours	**O**	**O**	**O**	**O**	Verification of the callback concurrency behavior in ROS2 nodes

addition, the table summarizes whether formal modeling and verification methods were employed for exhaustive behavior analysis and their primary analysis goals. The multi-threaded executor of ROS 2 has been a significant focus of several works. [2, 3, 5, 6] These studies primarily analyzed the response time of callback functions assigned to threads by the executors. Similarly, some works conducted response time analysis of advanced multi-threaded executors to enhance their performance.[7–12] While these works addressed callback concurrency as their primary focus, their analyses were limited to simulations and testing within constrained experimental scenarios, lacking exhaustive verification. A notable series of works by Lucas Dust [14–16] and a work of Raju Halder [13] provides formal models to verify the absence of buffer overflows in ROS 2 node communication using the UPPAAL model checker. These studies align with our work in terms of the modeling method but focus on node concurrency rather than callback concurrency. Other studies have explored formal modeling of security [17, 18] and safety [19, 20] properties in ROS 2 components or applications. However, these works also primarily focused on node concurrency and did not comprehensively address callbacks, callback groups, or executors. Distinct from the related works, our study explicitly models callback functions, callback groups, and executors using the UPPAAL model checker to exhaustively verify the callback concurrency behavior in ROS 2.

4. ROS 2 Callback Concurrency Models and Model Checking Queries

This section presents the ROS 2 callback concurrency model templates (which are the reusable and parameterized automata for UPPAAL), which model the four key elements of ROS 2 callback concurrency: **callbacks**, **callback groups**, **threads**, and **executors**, and the verification properties that specify its behavior. The model templates and related codes are provided in our repository.[a] The following subsections introduce the core concepts of each model, 1) callback, 2) callback group, 3) thread, and 4) executor in ROS 2. These templates are based on thorough analysis of the source code of recent versions of ROS 2 (ROS 2 Jazzy, Iron, and Humble) and official documents,[b] as well as related works that describe the behavior of callbacks, callback groups, and executors.[2, 3, 5, 6] Additionally, another subsection introduces seven UPPAAL model checking query templates to specify callback concurrency behaviors that are verifiable on our model.

4.1. *Callback model*

For model checking of ROS 2 callback concurrency, we model the callback, as illustrated in Fig 1. In the model, the circles represent states, while double-lined circles indicate the initial state. Arrows denote transition edges between states, with green

[a] https://github.com/yongjunshin/ROS2_Callback_Concurrency_In_UPPAAL.
[b] https://github.com/ros2/ros2 and https://docs.ros.org.

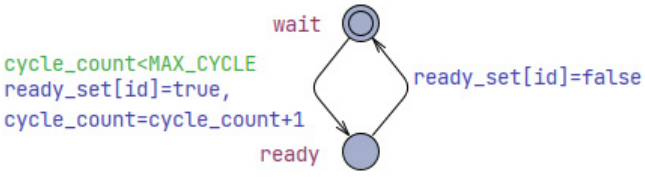

Fig. 1. Callback model template.

text on the arrows specifying the guard conditions for the transitions, and blue text indicating the update expressions representing the effects of the transitions.

A callback is the smallest unit of work used to accomplish a node's task. It becomes ready when the specific condition of its type is met: a timer callback after its interval elapses, a subscription callback upon receiving new data, a service callback when a request arrives, and a client callback upon receiving a response. Therefore, the callback model has two states: `wait` (i.e. non-ready) and `ready` for execution. Each callback has a unique identifier, `id`, and there is a global variable, `ready_set`, whose type is a list of booleans, used for globally tracking the states of all callbacks under verification. `ready_set[id]` denotes the state of the callback with identifier `id`, where `true` indicating ready and `false` indicating wait. From the perspective of a callback, transitions between the two states are uncertain, depending on the satisfaction of the ready condition introduced above. Therefore, guard conditions for the transition are not explicitly modeled, and the UPPAAL model checker resolves this uncertainty exhaustively. However, to ensure efficient model checking and avoid the state-explosion problem, we restrict infinite transitions by limiting the maximum number of callback state transitions, denoted as `cycle_count`. This restriction is implemented as a guard condition and an update expression on an edge.

4.2. *Callback group model*

A callback belongs to a callback group, which determines whether concurrent execution of a set of callbacks is allowed. We model the callback group, as shown in Fig 2. Each group has a unique identifier, `id`, and a type variable, `group_type`, which

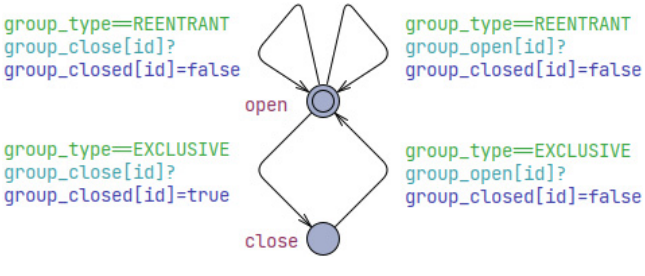

Fig. 2. Callback group model template.

is set to either `REENTRANT` or mutually `EXCLUSIVE`. The two states, `open` and `close`, indicate whether the execution of an additional callback is allowed or restricted, and `open` state is the initial state. When an executor assigns a callback to a thread, the callback group receives a synchronization signal, `group_close[id]`. Conversely, when the execution of a callback is completed, the group receives a synchronization signal, `group_open[id]`. The reception of synchronization signals is represented by sky-blue expressions ending with a question mark (`?`) on the edges. A reentrant group allows concurrent execution of callbacks, remaining in the `open` state even when it receives the `group_close[id]` signal. In contrast, a mutually exclusive group does not allow the concurrent execution, transitioning to the `close` state upon receiving the `group_close[id]` signal. When it receives the `group_open[id]` signal, it returns to the `open` state. Similar to the callback model, the state of the group is stored in a global variable, `group_closed[id]`, which is `true` when the group is closed and `false` when it is open.

4.3. *Thread model*

In ROS 2, a thread is responsible for executing callbacks, and its model template is illustrated in Fig 3. Similar to the callback and callback group models, each thread has a unique identifier, `id`, and its state is recorded in a global variable, `thread_running[id]` for tracking whether the thread is idle or executing a callback as a boolean value. The model explicitly defines two states: `wait` (idle state) and `running` (executing a callback). Additionally, the model includes implicit `committed` states, represented by circles labeled with '`C`' that enforce atomic transitions between states without delay.

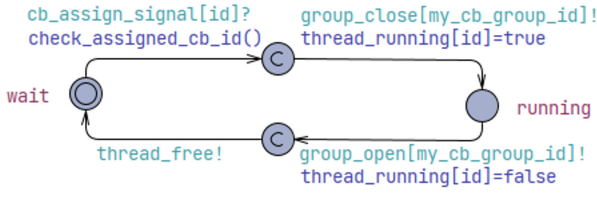

Fig. 3. Thread model template.

The state transitions of a thread proceed as follows: (1) A thread remains in the `wait` state until it receives a callback assignment signal, `cb_assign_signal[id]?`, from the executor. (2) Upon receiving the signal, the thread checks the identifiers of the assigned callback and its group, denoted as `my_cb_group_id`. Before executing the callback, the thread signals the closure of the callback group by sending `group_close[my_cb_group_id]!`. The synchronization signal ending with an exclamation mark (`!`) denotes the sending of a signal to other models. It synchronizes the thread model and the callback group model via edges with `group_close[id]?`.

The thread transitions to the `running` state, where it executes the assigned callback. (3) Upon completing the callback execution, the thread reopens the callback group by sending `group_open[my_cb_group_id]!`. (4) Finally, it notifies the executor that it has returned to the idle state by sending the `thread_free!` signal. The thread model repeats this process continuously.

4.4. *Executor model*

The executor is responsible for scheduling the execution of callbacks across available threads. Its scheduling process is modeled in Fig 4. It continuously monitors callbacks that are ready to execute in a step referred to as polling. During polling, it identifies ready callbacks and copies their identifiers into a ready set, initiating the scheduling process (modeled as the `polling` and `schedule_start` states). It then selects or waits for an available thread (modeled as the `pick_thread` and `wait_thread` states). Next, it chooses the highest-priority callback from the ready set (modeled as the `pick_cb` state). Callback priorities are determined by type (timer,

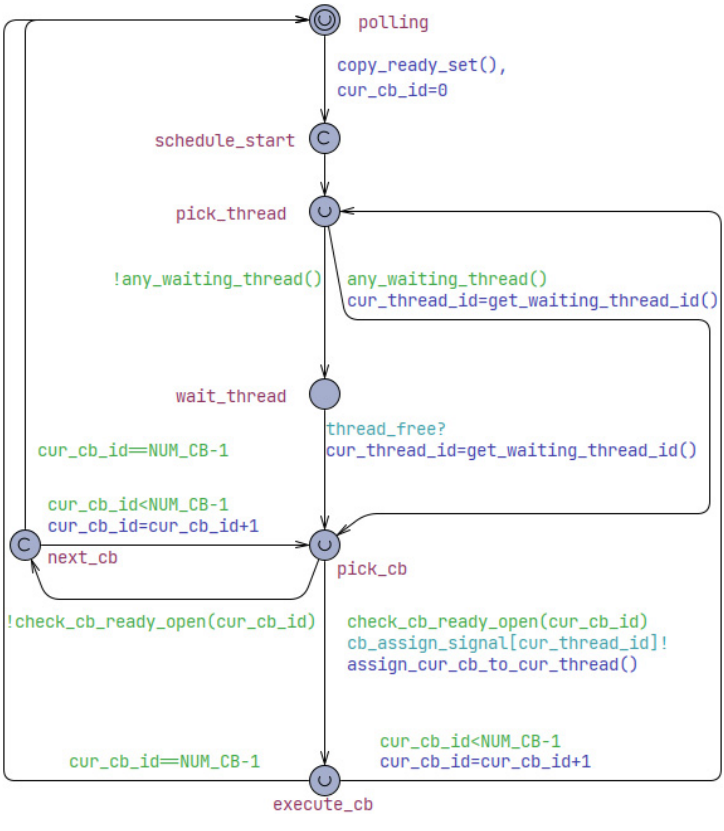

Fig. 4. Executor model template.

subscription, service, or client, in that order) or can be manually configured by the developer. The executor verifies that the selected callback is still ready and that its execution is allowed by its callback group (modeled as the guard condition `check_cb_ready_open(cur_cb_id)`). The executor checks the readiness of subscription, service, and client callbacks from the ready set copied during the polling phase, but it checks the timer callback directly from the ROS 2 middleware [6]. If the condition is not met, the executor removes the callback from the ready set and proceeds to the next callback (modeled as the `next_cb` state), making the callback selection mechanism neither strictly priority-ordered nor first-in-first-out. Once a callback is successfully selected, the executor assigns it to a thread (modeled as the `execute_cb` state). This process repeats until the ready set is empty, at which point the executor returns to the polling step to identify newly ready callbacks.

In this model, states marked with '`U`' represent *urgent* states in UPPAAL, where time cannot elapse. While the ROS 2 executor incurs a small scheduling delay (e.g. on the order of nanoseconds), this delay is assumed negligible. As a result, most states in the executor model are defined as either *urgent* or *committed*, ensuring immediate or atomic transitions. The exception is the `wait_thread` state, which models the situation where all threads are running, requiring the executor to pause until a thread becomes available.

We have introduced four model templates that developers can instantiate for model checking. For example, if the ROS 2 node under verification includes a four-threaded executor, four callbacks, and two callback groups, the corresponding model templates can be instantiated. The model templates along with related code and a set of modeling examples are available in our repository (footnote a). During model checking, the instantiated models dynamically interact with each other. The executor model orchestrates the others by assigning callbacks to threads. Across the numerous state transition paths of these interacting models, the UPPAAL model checker exhaustively verifies the properties defined as model checking queries.

4.5. *UPPAAL Model Checking Queries*

This subsection introduces model checking query templates for verifying ROS 2 callback concurrency behavior in UPPAAL. Table 2 presents seven verification purposes, query templates, and their descriptions. Each query template is written in UPPAAL query syntax, introduced in Sec. 2.2, and includes parameters denoted by the '`$`' symbol. Users can create specific queries by replacing these parameters with concrete model components (e.g. model instance names, variable names, etc.). The first query template is designed to verify the absence of deadlocks in the node, ensuring that the model under verification functions correctly without syntax errors. The remaining templates focus on key specifications of ROS 2 callback concurrency, such as mutual exclusion, concurrent execution, and callback starvation. Users can utilize or extend these query templates based on their descriptions and verification needs.

Table 2. Query templates of ROS 2 callback concurrency model checking.

Verification purpose	Query template	Description
Absence of node deadlock	`A[] not deadlock`	There is no deadlock in the node.
Callback function executability	`E<> cb_running($callback_id)`	There is at least one path where a callback function (`$callback_id`) is executed.
Mutually exclusive execution of callback functions	`A[]not(cb_running($callback_id_1) and cb_running ($callback_id_2))`	Two arbitrary callback functions (`$callback_id_1` and `$callback_id_2`) are never executed simultaneously.
Blocking multi-thread access to the callback group	`A[] cb_running($callback_id) imply group_closed [cb_group_table[$callback_id]]`	When a callback function (`$callback_id`) is executing, its mutually exclusive callback group is closed.
Concurrent execution of callback functions	`E<> (cb_running($callback_id_1) andcb_running ($callback_id_2))`	Two arbitrary callback functions (`$callback_id_1` and `$callback_id_2`) can be executed simultaneously.
Allowing multi-thread access to the callback group	`A[]notgroup_closed($callback_group_id)`	A reentrant callback group (`$callback_group_id`) is always open.
Absence of callback function starvation	`($executor_name.schedule_start and$executor_name. ready_set_copy[$callback_id] and !cb_running ($callback_id)) → cb_running($callback_id)`	If a callback function (`$callback_id`) is scheduled for execution, the executor (`$executor_name`) will eventually execute the callback function.

5. Automated UPPAAL Model Generation Tool

This section introduces an automated UPPAAL model code generation tool supporting our approach. Model checking engineers can specify the ROS 2 nodes under verification and their properties using model and query templates proposed in Sec. 4, requiring an understanding of both the system and UPPAAL. Specifically, engineers must correctly define global variable declarations, system definitions, and queries in UPPAAL with the provided templates to represent their system. While this instantiation task is straightforward, errors may still occur, particularly for novices unfamiliar with UPPAAL.

To address this, we develop an automated model code generation tool (footnote a), as illustrated in Fig 5. With this tool, engineers only need to specify the ROS 2 nodes in a user-friendly format. The tool then automatically generates all mandatory files, including declarations, system definitions, and queries for the UPPAAL model checker. Subsequently, the model checker processes these files and returns verification results.

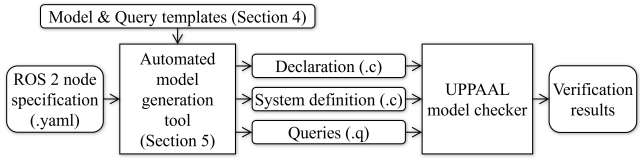

Fig. 5. ROS 2 callback concurrency model checking process with automated model code generation tool.

Listing 1. An example of the ROS 2 node specification for automated code generation.

```yaml
node:
  name: "Example_node"
  callbacks:
    - name: "TMR_callback"
      type: "TMR"
      callback_group: "R_Group"
    - name: "SUB_callback"
      type: "SUB"
      callback_group: "R_Group"
    - name: "SRV_callback"
      type: "SRV"
      callback_group: "ME_Group"
    - name: "CLT_callback"
      type: "CLT"
      callback_group: "ME_Group"
  callback_groups:
    - name: "R_Group"
      type: "REENTRANT"
    - name: "ME_Group"
      type: "EXCLUSIVE"
  executor:
    name: "Executor"
    threads:
      - name: "thread_0"
      - name: "thread_1"
```

The input format for the ROS 2 node specification, the only required input for our tool, is detailed with an example in Listing 1. Users specify the node, including its name, callbacks, callback groups, and executor, using YAML syntax. The node is identified by the `name` field under the `node` key. The `callbacks` section lists the nodes callbacks, each defined by a unique `name`, a `type` (either `TMR`, `SUB`, `SRV`, or `CLT`), and a `callback_group`, which specifies the name of a callback group listed below. The `callback_groups` section defines the groups, including their `name` and `type` (either `REENTRANT` or `EXCLUSIVE`). Finally, the `executor` section specifies the executor managing the node, including its `name` and a list of `threads`, each identified by the `name`.

It simply specifies ROS 2 nodes, requiring only system knowledge and no familiarity with UPPAAL. The tool parses the node specification in YAML, converts it into UPPAAL models (e.g. declarations and system definitions) using the templates, and generates all possible queries by combining query templates with available parameters from the generated models. The tool can be executed with the following command: `python3 uppaal_model_code_generation.py –config_file nodes.`

`yaml`. This significantly lowers the barrier to adopting formal verification in practice, allowing ROS 2 developers to analyze callback concurrency without requiring deep expertise in UPPAAL. In this paper, we do not elaborate on the internal logic of the tool, as it focuses on automatically translating YAML-based specifications into UPPAAL models, but its source code is available in our repository (footnote a).

6. ROS 2 Callback Concurrency Analysis

6.1. *Case study setup*

We conduct case studies to evaluate the correctness of our approach. We model three representative cases of ROS 2 node implementations, each combining multiple callbacks and callback groups with different types, managed by single-threaded or multi-threaded executors. Specifically, the modeled cases are: (1) a mutually exclusive callback group with a single-threaded executor, (2) a mutually exclusive callback group with a multi-threaded executor, and (3) a reentrant callback group with a multi-threaded executor. We then verify the ROS 2 callback concurrency behavior properties defined in UPPAAL queries. Additionally, we compare the traces of model checking and the actual ROS 2 applications to evaluate their consistency.

Listings 2–4 show the code for the three cases. All cases share the same two timer callback functions, `publish_msg_A` and `publish_msg_B` publishing every second, as well as two subscription callback functions, `subscribe_msg_C` and

Listing 2. Case 1 (mutually exclusive callback group and single-thread executor) Python code.

```
class SingleThreadNode(Node):
    def __init__(self):
        self.group = MutuallyExclusiveCallbackGroup()
        self.timer_A = self.create_timer(1, self.publish_msg_A,
          callback_group=self.group)
        self.timer_B = self.create_timer(2, self.publish_msg_B,
          callback_group=self.group)
        self.msg_C_subscriber = self.create_subscription(String, msg_C,
          self.subscribe_msg_C, callback_group=self.group)
        self.msg_D_subscriber = self.create_subscription(String, msg_D,
          self.subscribe_msg_D, callback_group=self.group)
    def publish_msg_A(self):
        logger.info("[TMR1] pub A")
        ... # takes about 0 seconds
    def publish_msg_B(self):
        logger.info("[TMR2] pub B")
        ... # takes about 0 seconds
    def subscribe_msg_C(self, msg):
        logger.info(" [SUB1] sub C")
        ... # takes about 1 seconds
    def subscribe_msg_D(self, msg):
        logger.info(" [SUB2] sub D")
        ... # takes about 2 seconds
def main():
    node = SingleThreadNode()
    executor = SingleThreadedExecutor()
    executor.add_node(node)
    try:
        executor.spin()
```

Listing 3. Case 2 (mutually exclusive callback group and multi-thread executor) Python code.

```
1  class MutuallyExclusiveMultiThreadNode(Node):
2      ... # same with Case 1
3  def main():
4      node = MutuallyExclusiveMultiThreadNode()
5      executor = MultiThreadedExecutor(num_threads=4)
6      ... # same with Case 1
```

Listing 4. Case 3 (reentrant callback group and multi-thread executor) Python code.

```
1  class ReentrantMultiThreadNode(Node):
2      def __init__(self):
3          self.group = ReentrantCallbackGroup()
4      ... # same with Case 1
5  def main():
6      node = ReentrantMultiThreadNode()
7      ... # same with Case 2
```

`subscribe_msg_D`, which subscribe topics, `msg_C` and `msg_D`, as shown in Listing 2 (lines 419). Topics `msg_C` and `msg_D` are published by an external node every second and every two seconds, respectively. The `executor.spin()` call (line 25) initiates the executor to schedule these callbacks. However, the cases differ in combinations of the callback group and executor, which result in varying callback concurrency behaviors. In Listing 2, Case 1 uses a single-threaded executor (line 22), with all callbacks placed in a mutually exclusive callback group (line 3). In Listing 3, Case 2 employs a multi-threaded executor with four threads (line 5) while retaining the same mutually exclusive callback group as in Case 1 (line 2). Finally, in Listing 4, Case 3 uses the same multi-threaded executor as Case 2 (line 7) but replaces the mutually exclusive callback group with a reentrant callback group to manage the callbacks (lines 3).

We specified the three cases in YAML to automatically generate UPPAAL models and 29 model checking queries listed in Table 3 using our tool. For each verification purpose, multiple queries were generated for all possible combinations of query parameters. We performed model checking to verify that the models for each case satisfy these queries (i.e. verification properties).

6.2. *Analysis Results*

Table 4 summarizes the verification results (pass or fail) of the queries for each case, along with their verification time and memory usage. From the results, we can observe that the callback concurrency behaviors specified by the queries vary across the cases. These behaviors are as follows:

- In all paths, our model exhibits no deadlock (Q1 for all cases), and all callback functions are reachable (Q2–5 for all cases).
- Callbacks in mutually exclusive groups are not executed concurrently (Q6–11 and Q16–21 for Cases 1 and 2) in all paths, while paths with concurrent execution of callbacks exist in reentrant groups (Q6–11 and Q16–21 for Case 3).

Table 3. Model checking queries under evaluation instantiated from the query templates.

Id	Model checking query
	— absence of node deadlock —
Q1	`A[] not deadlock`
	— Callback function executability —
Q2	`E<> cb_running(TMR1_id)`
Q3	`E<> cb_running(TMR2_id)`
Q4	`E<> cb_running(SUB1_id)`
Q5	`E<> cb_running(SUB2_id)`
	— Mutually exclusive execution of callback functions —
Q6	`A[] not (cb_running(TMR1_id) and cb_running(TMR2_id))`
Q7	`A[] not (cb_running(TMR1_id) and cb_running(SUB1_id))`
Q8	`A[] not (cb_running(TMR1_id) and cb_running(SUB2_id))`
Q9	`A[] not (cb_running(TMR2_id) and cb_running(SUB1_id))`
Q10	`A[] not (cb_running(TMR2_id) and cb_running(SUB2_id))`
Q11	`A[] not (cb_running(SUB1_id) and cb_running(SUB2_id))`
	— Blocking multi-thread access to the callback group (mutual exclusion) —
Q12	`A[] cb_running(TMR1_id) imply group_closed[cb_group_table[TMR1_id]]`
Q13	`A[] cb_running(TMR2_id) imply group_closed[cb_group_table[TMR2_id]]`
Q14	`A[] cb_running(SUB1_id) imply group_closed[cb_group_table[SUB1_id]]`
Q15	`A[] cb_running(SUB2_id) imply group_closed[cb_group_table[SUB2_id]]`
	— Concurrent execution of callback functions —
Q16	`E<> (cb_running(TMR1_id) and cb_running(TMR2_id))`
Q17	`E<> (cb_running(TMR1_id) and cb_running(SUB1_id))`
Q18	`E<> (cb_running(TMR1_id) and cb_running(SUB2_id))`
Q19	`E<> (cb_running(TMR2_id) and cb_running(SUB1_id))`
Q20	`E<> (cb_running(TMR2_id) and cb_running(SUB2_id))`
Q21	`E<> (cb_running(SUB1_id) and cb_running(SUB2_id))`
	— Allowing multi-thread access to the callback group (reentrant) —
Q22	`A[] not group_closed[cb_group_table[TMR1_id]]`
Q23	`A[] not group_closed[cb_group_table[TMR2_id]]`
Q24	`A[] not group_closed[cb_group_table[SUB1_id]]`
Q25	`A[] not group_closed[cb_group_table[SUB2_id]]`
	— Absence of callback function starvation —
Q26	`executor.schedule_start and executor.ready_set_copy[TMR1_id] and !cb_running(TMR1_id) → cb_running(TMR1_id)`
Q27	`executor.schedule_start and executor.ready_set_copy[TMR2_id] and !cb_running(TMR2_id) → cb_running(TMR2_id)`
Q28	`executor.schedule_start and executor.ready_set_copy[SUB1_id] and !cb_running(SUB1_id) → cb_running(SUB1_id)`
Q29	`executor.schedule_start and executor.ready_set_copy[SUB2_id] and !cb_running(SUB2_id) → cb_running(SUB2_id)`

- In all paths, while a tenant callback is executing, the mutually-exclusive groups prevent other tenant callbacks from executing (Q12–15 and Q22–25 for Cases 1 and 2), whereas reentrant groups allow concurrent execution of other tenant callbacks (Q12–15 and Q22–25 for Case 3).
- Even if the timer callback was in the ready state during the polling phase, it may not be scheduled (Q26 and Q27 for all cases). As explained in Sec. 4.4, this is because the ROS 2 executor checks the readiness of the timer callback directly from the ROS 2 middleware, not from the ready set.
- Except for the timer callback, callback starvation (where a callback is ready during polling but finally not scheduled to a thread) does not occur in the single-threaded executor (Q28 and Q29 for Case 1). On the other hand, in the multi-threaded

Table 4. ROS 2 callback concurrency model checking results. ○ = pass, ○ = fail, with verification time and memory usage.

Id	C1. Single-threaded executor	C2. Multi-threaded executor and mutually exclusive group	C3. Multi-threaded executor and reentrant group
Q1	○ (0.043 s / 64,032 KB)	○ (0.525 s / 92,280 KB)	○ (90.116 s / 4,463,412 KB)
Q2	○ (0.002 s / 62,328 KB)	○ (0.006 s / 62,380 KB)	○ (0.005 s / 62,380 KB)
Q3	○ (0.006 s / 62,328 KB)	○ (0.009 s / 62,376 KB)	○ (0.005 s / 62,376 KB)
Q4	○ (0.006 s / 58,572 KB)	○ (0.013 s / 58,628 KB)	○ (0.008 s / 58,624 KB)
Q5	○ (0.019 s / 58,712 KB)	○ (0.016 s / 58,760 KB)	○ (0.020 s / 58,756 KB)
Q6	○ (0.037 s / 64,028 KB)	○ (0.407 s / 92,284 KB)	○ (0.019 s / 58,628 KB)
Q7	○ (0.030 s / 64,032 KB)	○ (0.397 s / 92,292 KB)	○ (0.027 s / 58,896 KB)
Q8	○ (0.045 s / 64,028 KB)	○ (0.409 s / 92,284 KB)	○ (0.026 s / 62,920 KB)
Q9	○ (0.036 s / 64,024 KB)	○ (0.402 s / 92,288 KB)	○ (0.027 s / 58,896 KB)
Q10	○ (0.040 s / 64,028 KB)	○ (0.393 s / 92,288 KB)	○ (0.037 s / 59,164 KB)
Q11	○ (0.033 s / 64,024 KB)	○ (0.404 s / 92,292 KB)	○ (0.034 s / 60,196 KB)
Q12	○ (0.039 s / 64,032 KB)	○ (0.398 s / 92,288 KB)	○ (0.002 s / 62,380 KB)
Q13	○ (0.042 s / 64,036 KB)	○ (0.401 s / 92,284 KB)	○ (0.010 s / 62,384 KB)
Q14	○ (0.036 s / 64,036 KB)	○ (0.408 s / 92,284 KB)	○ (0.015 s / 58,628 KB)
Q15	○ (0.037 s / 64,036 KB)	○ (0.408 s / 92,284 KB)	○ (0.024 s / 58,764 KB)
Q16	○ (0.044 s / 64,028 KB)	○ (0.408 s / 92,280 KB)	○ (0.010 s / 58,628 KB)
Q17	○ (0.039 s / 64,028 KB)	○ (0.404 s / 92,280 KB)	○ (0.028 s / 58,892 KB)
Q18	○ (0.034 s / 64,028 KB)	○ (0.394 s / 92,280 KB)	○ (0.033 s / 62,916 KB)
Q19	○ (0.045 s / 64,024 KB)	○ (0.398 s / 92,280 KB)	○ (0.025 s / 58,892 KB)
Q20	○ (0.044 s / 64,028 KB)	○ (0.397 s / 92,284 KB)	○ (0.031 s / 59,164 KB)
Q21	○ (0.039 s / 64,032 KB)	○ (0.408 s / 92,284 KB)	○ (0.029 s / 60,192 KB)
Q22	○ (0.004 s / 62,336 KB)	○ (0.002 s / 62,376 KB)	○ (69.093 s / 4,463,416 KB)
Q23	○ (0.004 s / 62,332 KB)	○ (0.005 s / 62,380 KB)	○ (68.651 s / 4,463,420 KB)
Q24	○ (0.002 s / 62,328 KB)	○ (0.005 s / 62,376 KB)	○ (68.615 s / 4,463,416 KB)
Q25	○ (0.005 s / 62,336 KB)	○ (0.007 s / 62,380 KB)	○ (68.676 s / 4,463,416 KB)
Q26	○ (0.003 s / 62,920 KB)	○ (0.005 s / 62,964 KB)	○ (0.005 s / 62,960 KB)
Q27	○ (0.002 s / 62,920 KB)	○ (0.001 s / 62,960 KB)	○ (0.005 s / 62,960 KB)
Q28	○ (0.058 s / 64,208 KB) (*B1*)	○ (0.002 s / 62,964 KB) (*B2, B3*)	○ (89.666 s / 4,785,476 KB) (*B4*)
Q29	○ (0.048s / 65,368 KB)	○ (0.007 s / 62,964 KB)	○ (95.512 s / 4,896,668 KB)

executor, callback starvation can occur for callbacks in mutually exclusive groups due to group occupation (Q28 and Q29 for Case 2), but it does not occur for callbacks in reentrant groups (Q28 and Q29 for Case 3).

These results confirm that the behavior of our model is consistent with that of actual ROS 2, based on our knowledge [2, 3, 5–12]. This indicates that our UPPAAL model accurately represents the functionalities of ROS 2 components, and the verification results are reliable. In addition, the verification results reveal four notable callback concurrency behaviors in the model checking traces (B1–4 in Table 4): (1) round-robin scheduling in the single-threaded executor, (2) callback starvation in mutually exclusive callback groups, (3) thread overload in mutually exclusive callback groups, and (4) starvation-free execution in reentrant callback groups. We further explore these behaviors shown in specific model checking traces with logs of the actual ROS 2 applications.

Listing 5. An example execution log of Case 1 (Single-thread executor and mutually exclusive callback group).

```
 1  [INFO][0][SNode]:[TMR1] pub A       # (T1,0sec)
 2  [INFO][0][SNode]:[TMR2] pub B       # (T1,0sec)
 3  [INFO][0][SNode]: [SUB1] sub C      # (T1,1sec)
 4  [INFO][1][SNode]: [SUB2] sub D      # (T1,2sec)
 5  [INFO][3][SNode]:[TMR1] pub A       # (T1,0sec)
 6  [INFO][3][SNode]:[TMR2] pub B       # (T1,0sec)
 7  [INFO][3][SNode]: [SUB1] sub C      # (T1,1sec)
 8  [INFO][4][SNode]: [SUB2] sub D      # (T1,2sec)
 9  [INFO][6][SNode]:[TMR1] pub A       # (T1,0sec)
10  [INFO][6][SNode]:[TMR2] pub B       # (T1,0sec)
11  [INFO][6][SNode]: [SUB1] sub C      # (T1,1sec)
12  [INFO][7][SNode]: [SUB2] sub D      # (T1,2sec)
```

6.2.1. *Behavior 1: Round-robin scheduling of the single-threaded executor*

The single-threaded executor in ROS 2 schedules callbacks in a round-robin manner, prioritizing callback types in the order of timer, subscription, service, and client. An example execution log is shown in Listing 5. Each log entry recorded by a callback function follows the format: `[INFO][$timestamp_(s)][$node_name]:$message_of_logger # ($scheduled_thread, $execution_time)`. The log reveals a repeated sequence of messages: `pub A`, `pub B`, `sub C`, and `sub D`. This behavior is also reflected in our model checking trace illustrated in Fig. 6. The executor sequentially schedules the highest-priority callback to the single thread and waits for the callback to complete before processing the next one. This scheduling approach ensures that all callbacks are eventually executed (i.e. no callback is starved), as verified by Q28 in Case 1. However, if a callback occupies the thread for a long time, other callbacks may experience significant delays. As a result, the timing constraints of callbacks (e.g. the period of a timer callback) cannot be guaranteed.

6.2.2. *Behavior 2: Callback starvation in the mutually exclusive callback group*

Mutually exclusive callback groups are used with a multi-threaded executor to divide callbacks into multiple independent sets. In this scenario, developers should be aware that callbacks in a mutually exclusive callback group can experience starvation, unlikely to the case of a single-threaded executor. The log in Listing 6 shows that the callback function `subscribe_msg_D` (SUB2) is never executed. In our case study setup in Sec. 6.1, an external node publishes `msg_C` every second and `msg_D` every two seconds. Processing callback `SUB1` takes approximately 1 s, while processing callback `SUB2` takes about 2 s. As a result, whenever callback `SUB2` becomes ready, callback `SUB1` is also ready at the same time. Since the execution of callback `SUB1` is scheduled slightly earlier than callback `SUB2` even if there are multiple threads, `SUB1` occupies the callback group, causing `SUB2` to be skipped over for scheduling. This behavior is also verified through our model checking, as shown in Fig. 7. The trace illustrates that callbacks `TMR1`, `TMR2`, and `SUB1` are scheduled sequentially, and the executor

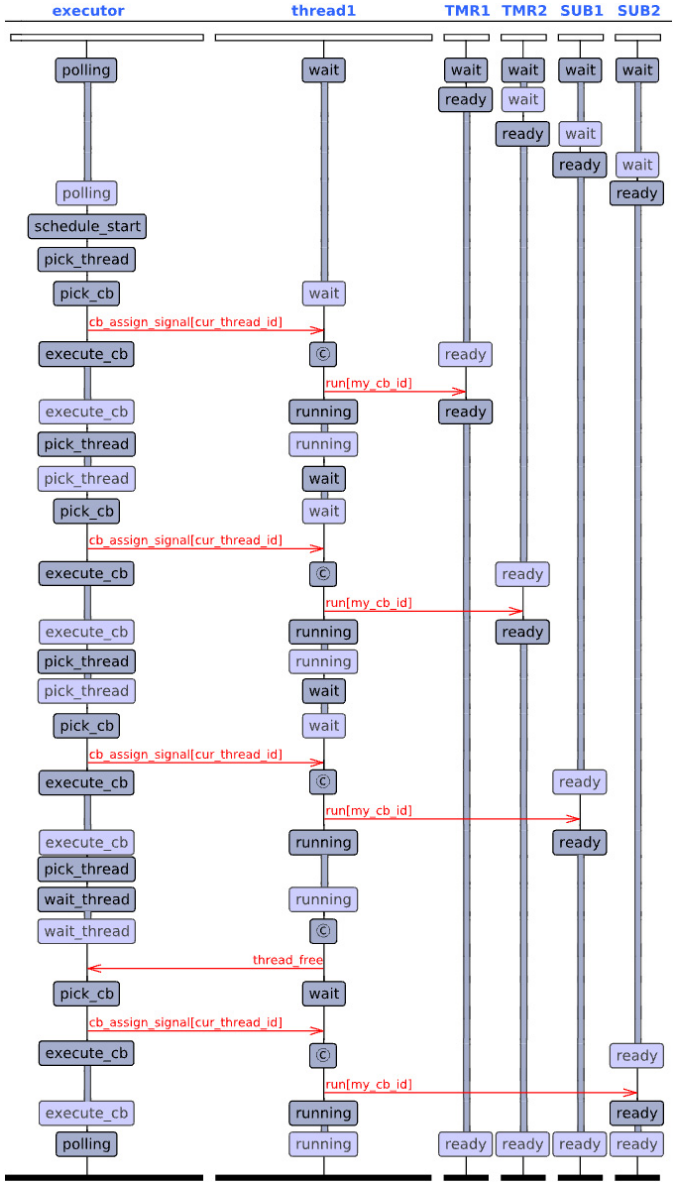

Fig. 6. An example model checking trace of Case 1.

returns to the next *polling* phase, while `SUB2` is not scheduled due to the closed callback group (the trace of the callback group is omitted here for brevity). This occurs even when another thread is available. The potential for callback starvation is exhaustively verified by Q28 in Case 2.

Listing 6. Execution logs of Case 2 (Multi-thread executor and Mutually exclusive callback group).

```
1  [INFO][0][MMNode]:[TMR1] pub A      # (T1,0sec)
2  [INFO][0][MMNode]:[TMR2] pub B      # (T1,0sec)
3  [INFO][0][MMNode]: [SUB1] sub C     # (T1,1sec)
4  [INFO][1][MMNode]:[TMR1] pub A      # (T1,0sec)
5  [INFO][1][MMNode]:[TMR2] pub B      # (T1,0sec)
6  [INFO][1][MMNode]: [SUB1] sub C     # (T1,1sec)
7  [INFO][2][MMNode]:[TMR1] pub A      # (T1,0sec)
8  [INFO][2][MMNode]:[TMR2] pub B      # (T1,0sec)
9  [INFO][2][MMNode]: [SUB1] sub C     # (T1,1sec)
```

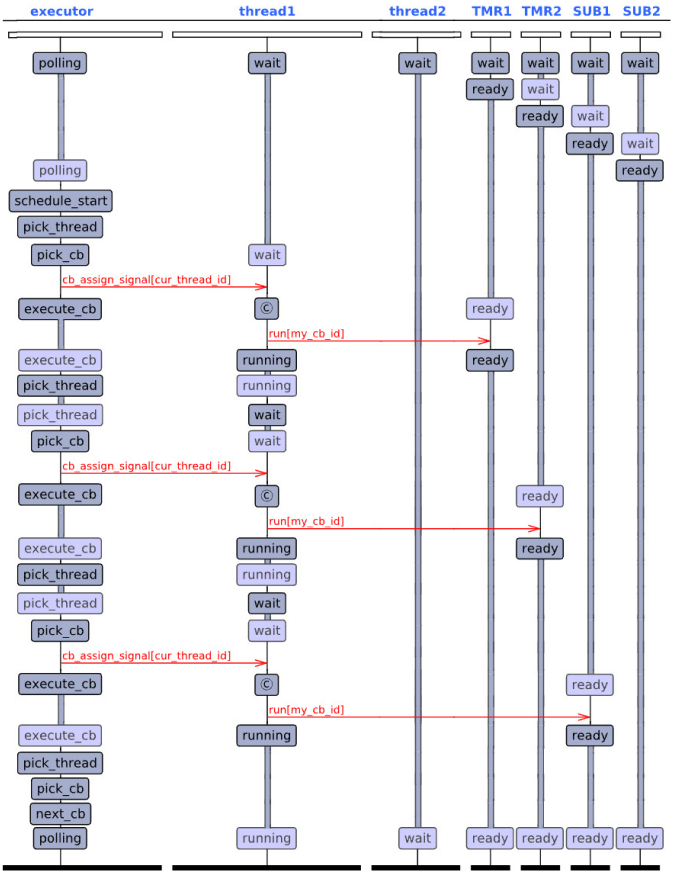

Fig. 7. An example model checking trace of Case 2.

6.2.3. *Behavior 3: Busy thread with the mutually exclusive callback group*

When using a multi-threaded executor, the callback execution load is not guaranteed to be equally distributed across threads, and one thread may become busier than others. When the execution time of a callback is very short, such as timer callbacks publishing small data (e.g. `TMR1` and `TMR2` in our cases), after the callback is assigned

to a thread and completes in a short time, the next ready callback is likely to be assigned to the same thread again. The selection of a thread for scheduling is determined by the operating system and is not directly managed by ROS 2. This can also happen with a reentrant callback group but is more likely to be observed with a mutually exclusive callback group. Both the execution log in Listing 6 and the model checking trace in Fig 7 verifies this experience. They demonstrate that the callbacks (except for a starved callback) are repeatedly assigned to the same thread.

Listing 7. An example execution log of Case 3 (Multi-thread executor and Reentrant callback group).

```
 1  [INFO][0][MRNode]:[TMR1] pub A     # (T1,0sec)
 2  [INFO][0][MRNode]:[TMR2] pub B     # (T2,0sec)
 3  [INFO][0][MRNode]: [SUB1] sub C    # (T3,1sec)
 4  [INFO][0][MRNode]: [SUB2] sub D    # (T4,2sec)
 5  [INFO][1][MRNode]:[TMR1] pub A     # (T1,0sec)
 6  [INFO][1][MRNode]:[TMR2] pub B     # (T2,0sec)
 7  [INFO][1][MRNode]: [SUB1] sub C    # (T3,1sec)
 8  [INFO][2][MRNode]:[TMR1] pub A     # (T1,0sec)
 9  [INFO][2][MRNode]:[TMR2] pub B     # (T2,0sec)
10  [INFO][2][MRNode]: [SUB1] sub C    # (T3,1sec)
11  [INFO][2][MRNode]: [SUB2] sub D    # (T4,2sec)
12  [INFO][3][MRNode]:[TMR1] pub A     # (T1,0sec)
13  [INFO][3][MRNode]:[TMR2] pub B     # (T2,0sec)
14  [INFO][3][MRNode]: [SUB1] sub C    # (T3,1sec)
```

6.2.4. *Behavior 4: Callback starvation-free in the reentrant callback group*

When the number of threads is greater than or equal to the number of callbacks, callbacks in a reentrant group are not starved. Both Listing 7 illustrates this case. The assignment of all callbacks is completed very quickly to threads `T1-4`, so the execution of callbacks starts almost simultaneously (in lines 1-4). Then, the next polling and scheduling begins instantly with the available threads, allowing `T1-3` to execute callbacks while `T4` continues running the callback `SUB2`, which takes two seconds (in lines 5-7). The model checking also covers this possible path, as shown in Fig. 8. The executor assigns a callback even if the previously assigned callback has not yet been completed, as they belong to a reentrant group. This snippet of the trace flows within a short period of real application time, after which the executor proceeds to the next polling phase. Including this trace, the UPPAAL model checker exhaustively verifies that callback starvation does not occur in any path, as verified by Q28 in Case 3.

6.3. *Analysis cost of using model checking*

Model checking is effective for exhaustively verifying all possible execution paths of the system under verification, but it is expensive in terms of computation. The exploration space of our ROS 2 callback concurrency model corresponds to the number of callbacks and threads. In addition, models with a reentrant group have a larger space than those with a mutually exclusive group, as the former allows many

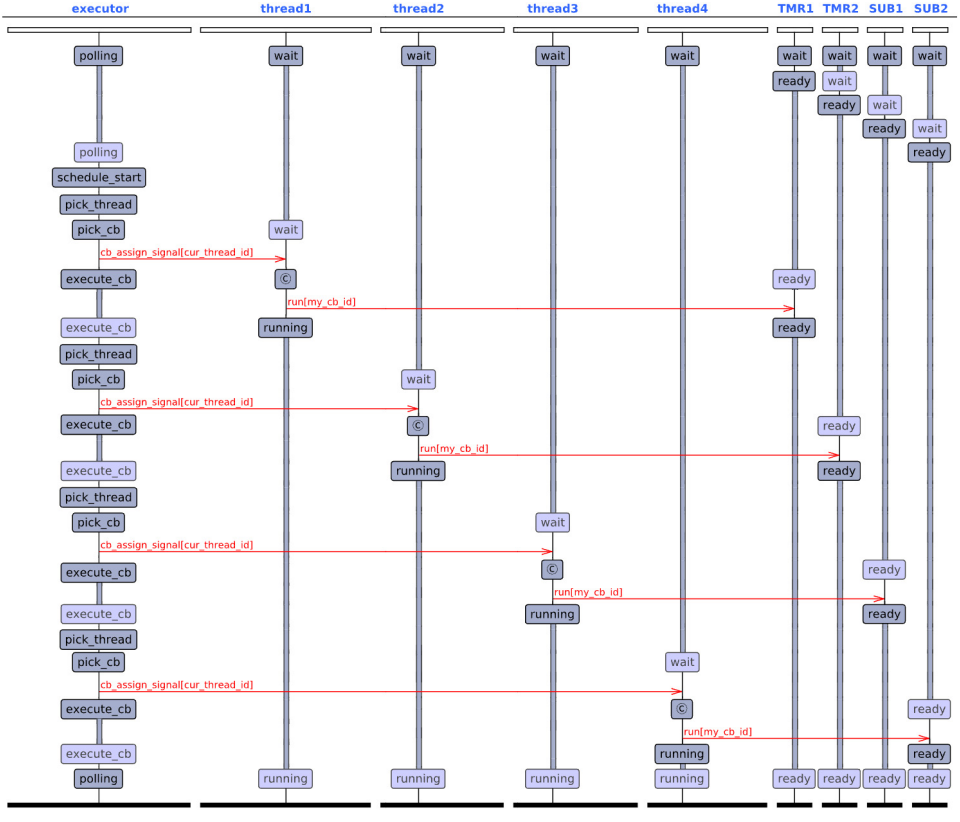

Fig. 8. An example model checking trace of Case 3.

Table 5. Model checking time (seconds) pairs (x/y) by number of callbacks and threads, using either a mutually exclusive (x) or reentrant (y) callback group.

# of threads \ # of call-backs	1	2	3	4	5
1	0.002 / 0.003	0.007 / 0.007	0.016 / 0.024	0.053 / 0.052	0.368 / 0.702
2	0.001 / 0.006	0.014 / 0.011	0.055 / 0.075	0.559 /0.719	6.095/7.524
3	0.005 /0.007	0.009 / 0.043	0.066 / 0.61	0.56 /9.421	6.137 / 113.739
4	0.004 /0.013	0.009 / 0.19	0.06 / 5.403	0.566 /101.186	6.293/ 1531.545
5	0.002 /0.014	0.015 / 1.004	0.067 / 44.338	0.591/ 1051.959	6.341/N/A

callbacks to be executed simultaneously. Therefore, we report the model checking time based on the pairs of the number of callbacks and threads with either a mutually exclusive or a reentrant group in Table 5. We used the '`A[] not deadlock`' query,

which took the longest verification time among our query templates, to show the worst-case scenario.

As expected, we observed that as the number of callbacks and threads increases, the verification time increases exponentially. Moreover, when the callbacks belong to a reentrant group, we observed significantly higher time consumption. For instance, when a node has 5 callbacks in a reentrant group and a four-threaded executor, the model checking takes about 25 min (1531 s). When five threads are modeled, the model checking cannot be completed within a practical time (e.g. a few hours) on a desktop with 64 GB of RAM.

Therefore, to verify ROS 2 callback concurrency using the UPPAAL model checker, the modeling — covering callbacks, callback groups, and threads — should considers the associated computational cost. In ROS 2, designing a node to handle a single task through a small number of core callback functions is a golden rule for modularity and maintainability. Hence, our ROS 2 callback concurrency model is particularly useful for verifying nodes with a limited number of core callbacks, as model checking exhaustively explores all possible paths to provide proof of the system's properties.

7. Discussion

While we have validated that our model accurately represents ROS 2 components affecting callback concurrency, its validity may be internally threatened by potential misunderstandings of ROS 2 behavior. To mitigate this risk, we thoroughly analyzed the ROS 2 source code and official documentation, cross-checking with related works [2, 3, 5, 6]. Additionally, external threats arise from ROS 2's continuous evolution, with frequent updates and new callback scheduling mechanisms [7–12]. However, since our model captures core functionalities, we believe it provides a solid and extensible foundation for future adaptations of our approach.

Given the validity of our approach, several user groups can benefit from our contributions: ***Developers*** can formally verify callback concurrency behaviors in real-world ROS 2 applications using our models, enabling them to identify and prevent concurrency issues, such as callback starvation and thread overload, at the design stage before deployment, supported by an intuitive UPPAAL model generation tool. ***Researchers*** can extend our model for deeper analysis of the new releases of ROS 2 or advanced ROS 2 components. ***ROS 2 beginners*** can explore callback concurrency by simulating our models on UPPAAL model checker, with our verification report carefully structured to highlight key ROS 2 behaviors.

8. Conclusion

Despite the significant impact of ROS 2 callback concurrency on robotic applications, a few studies comprehensively model and verify the interplay of callbacks, callback groups, and multi-threaded executors. In this paper, we proposed a novel and practical approach for modeling and verifying ROS 2 callback concurrency using

the UPPAAL model checker. We developed reusable model templates for callbacks, callback groups, threads, and executors, along with a suite of model checking queries. Additionally, we introduced an automated tool that generates UPPAAL model code from simple ROS 2 node specifications, streamlining verification for practitioners. We validated our approach through three representative ROS 2 callback concurrency cases. The verification reports confirmed the correctness of our models and revealed concurrency issues, such as round-robin scheduling, callback starvation, and busy threads. These findings offer valuable insights for developers and practitioners in ROS 2, and our models and queries facilitate further exploration of ROS 2 callback concurrency in the industry.

In future work, we plan to apply our modeling and verification approach to real-world robotic applications of our industry partners, such as automated wheelchairs and patrol robots, to verify safety concerns in their services. Additionally, we aim to extend our models and queries to use *statistical* model checking [21, 22] for evaluating performance aspects like timing and security, enhancing the scalability and comprehensiveness of our approach [23].

Acknowledgments

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2024-00406245). This work was supported by Electronics and Telecommunications Research Institute(ETRI) grant funded by the Korean government [26CS1100, Development of Proprietary Physical AI-based Small-scale Computers and Integrated Soft Suits].

References

1. S. Macenski, T. Foote, B. Gerkey, C. Lalancette and W. Woodall, Robot operating system 2: Design, architecture, and uses in the wild, *Sci. Robot.* **7**(66) (2022) eabm6074.
2. L. J. Dust, E. Persson, M. Ekström, S. Mubeen, C. Seceleanu and R. Gu, Experimental evaluation of callback behavior in ROS 2 executors, in *Proc. IEEE Int. Conf. Emerg. Technol. Factory Autom.* 2023, pp. 1–8.
3. T. Blaß, D. Casini, S. Bozhko and B. B. Brandenburg, A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance, in *Proc. IEEE Real-Time Syst. Symp.* 2021, pp. 41–53.
4. K. G. Larsen, P. Pettersson and W. Yi, Uppaal in a nutshell, *Int. J. Softw. Tools Technol. Transf.* **1** (1997) 134–152.
5. H. Teper, M. Günzel, N. Ueter, G. von der Brüggen and J.-J. Chen, End-to-end timing analysis in ROS2, in *Proc. IEEE Real-Time Syst. Symp.* 2022, pp. 53–65.
6. D. Casini, T. Blaß, I. Lütkebohle and B. Brandenburg, Response-time analysis of ROS 2 processing chains under reservation-based scheduling, in *Proc. Euromicro Conf. Real-Time Syst.* 2019, pp. 1–23.

7. H. Choi, Y. Xiang and H. Kim, Picas: New design of priority-driven chain-aware scheduling for ROS2, in *Proc. IEEE Real-Time Embed. Technol. Appl. Symp.*, 2021, pp. 251–263.
8. S. Liu, X. Jiang, N. Guan, Z. Wang, M. Yu and W. Yi, Rtex: An efficient and timing-predictable multithreaded executor for ROS 2. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, *IEEE* 43(9) 2024 2578–2591
9. H. Teper, T. Betz, M. Günzel, D. Ebner, G. Von Der Brüggen, J. Betz and J.-J. Chen, End-to-end timing analysis and optimization of multi-executor ROS 2 systems, in *Proc. IEEE Real-Time Embed. Technol. Appl. Symp.* 2024, pp. 212–224.
10. Y. Yang and T. Azumi, Exploring real-time executor on ROS 2, in *Proc. IEEE Int. Conf. Embed. Softw. Syst.* 2020, pp. 1–8.
11. H. Sobhani, H. Choi and H. Kim, Timing analysis and priority-driven enhancements of ROS 2 multi-threaded executors, in *Proc. IEEE Real-Time Embed. Technol. Appl. Symp.*, 2023, pp. 106–118.
12. A. Al Arafat, K. Wilson, K. Yang and Z. Guo, Dynamic priority scheduling of multi-threaded ROS 2 executor with shared resources, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **43**(11) (2024) 3732–3743.
13. R. Halder, J. Proença, N. Macedo and A. Santos, Formal verification of ROS-based robotic applications using timed-automata, in *Proc. IEEE/ACM Int. FME Workshop Formal Methods Softw. Eng.*, 2017, pp. 44–50.
14. L. Dust, R. Gu, C. Seceleanu, M. Ekström and S. Mubeen, Pattern-based verification of ROS 2 nodes using Uppaal, in *Proc. Int. Conf. Formal Methods Ind. Critical Syst.*, 2023, pp. 57–75.
15. L. Dust, M. Ekström, R. Gu, S. Mubeen and C. Seceleanu, A model-based methodology for automated verification of ROS 2 systems, in *Proc. ACM/IEEE Int. Workshop Robot. Softw. Eng.*, 2024, pp. 35–42.
16. L. Dust, R. Gu, C. Seceleanu, M. Ekström and S. Mubeen, Uppaal-based modeling and verification of ROS 2 multi-threaded execution and operating system reservations, in *Proc. Int. Conf. Formal Methods Ind. Critical Syst.*, 2024, pp. 40–59.
17. Y. Liu, Y. Guan, X. Li, R. Wang and J. Zhang, Formal analysis and verification of DDS in ROS2, in *Proc. ACM/IEEE Int. Conf. Formal Methods Models Syst. Des.* (2018), pp. 1–5.
18. S. Yang, J. Guo and X. Rui, Formal analysis and detection for ROS2 communication security vulnerability, *Electronics* **13**(9) (2024) 1762.
19. S. Kortik and T. K. Shastha, Formal verification of ROS based systems using a linear logic theorem prover, in *Proc. IEEE Int. Conf. Robot. Autom.*, 2021, pp. 9368–9374.
20. M. Adam, E. E. Hartmark, T. Andersen, D. A. Anisi and A. Cavalcanti, Safety assurance of autonomous agricultural robots: from offline model-checking to runtime verification, in *Proc. IEEE Int. Conf. Autom. Sci. Eng.*, 2024, pp. 2511–2516.
21. Y.-J. Shin, E. Cho and D.-H. Bae, Pasta: An efficient proactive adaptation approach based on statistical model checking for self-adaptive systems, in *Proc. Int. Conf. Fundam. Approaches Softw. Eng.*, eds., E. Guerra and M. Stoelinga, Lecture Notes in Computer Science, Vol. 12649, 2021, pp. 292–312.
22. A. David, K. G. Larsen, A. Legay, M. Mikučionis and D. B. Poulsen, Uppaal SMC tutorial, *Int. J. Softw. Tools Technol. Transf.* **17** (2015) 397–415.
23. Y.-J. Shin, Runtime verification of cause-effect latency in black-box systems, *Inf. Softw. Technol.* **196** (2026) 108172.