Deadlock- and Starvation-free Formally Verified Client Library for Robots

Yuuki Takano
TIER IV, inc.
Tokyo, Japan
https://orcid.org/0000-0002-6888-9024

Abstract—**Robot middleware is crucial for managing diverse robotic functions, though it often faces issues like deadlock and starvation. Addressing these, this paper introduces two libraries, safe_drive and async_lib, aiming to enhance the efficiency and safety of ROS 2 middleware. These libraries address deadlock and starvation through well-typed APIs and formal verification methods, offering enhanced security.**

The safe_drive library, a single-threaded client library for ROS 2, features a task executor verified to be free from deadlock and starvation using PlusCal and TLA+, establishing credibility. On the other hand, async_lib offers well-typed intra-process communication APIs for multi-threading environments, with verified multi-threaded executors that support safe_drive in complex applications, thereby facilitating efficient robot applications.

Furthermore, the paper proposes using session types and typestate programming for safe communication protocol implementation, capable of detecting protocol violations at both compile and run time, especially useful in complex scenarios like state transitions in action servers where violations could lead to deadlock.

Index Terms—**robotics, formal verification, session types, Rust, TLA+, ROS 2, deadlock, starvation**

I. Introduction

The Robot Operating System 2 (ROS 2) [1], [2] is widely used middleware for robots. Several robots have been implemented using ROS 2, such as [3]–[5] and others. As robots increasingly permeate our society, their safety becomes more and more important. For example, bugs in self-driving cars can cause irrecoverable traffic accidents. However, conventional middleware has several problems. For instance, ROS 2's libraries and APIs can cause deadlock and starvation, and C++, which is used as the main language for ROS 2, can lead to memory access violation errors such as buffer overruns.

To verify properties of software, formal methods such as [6]–[9] and others are used. Formal methods have also been adopted for robots [10], but conventional work focuses on the behavior of robots and does not target middleware. However, middleware is also significant for implementing safe robots. This paper proposes formally verified middleware libraries for robots.

In this paper, safe_drive and async_lib are proposed as new libraries. The task schedulers of these libraries are verified to be deadlock- and starvation-free using TLA+ [7], a model checker and one of the formal methods for specifying and verifying systems. By specifying and verifying the schedulers, several bugs in the libraries were discovered and fixed.

For verification, task executors of safe_drive and async_lib are specified. In addition, the delta list [11], which is a core data structure for timers, is specified to verify that the implementation is indeed correct. To represent time, the head of the delta list is decremented to avoid using a time counter, because such a time counter would cause the number of states to explode. This means that the model checking would never finish.

In addition to formal methods, async_lib adopts session types [12] to provide deadlock-free APIs. Session types are types of a programming language for representing communication protocols, and protocol violations can be detected at compile and run-time. Several programming languages are used to implement session types [13], [14]. This paper uses the Rust language for performance and safety, and adopts a modified version of session types for Rust [15]. Proposed libraries are protected from memory violation errors by Rust.

ROS 2 defines three types of communications, which are called topic, service, and action. async_lib provides session-typed APIs for service and action. Implementing service and action tends to be complex and can cause deadlock, but proposed session-typed APIs can avoid such deadlock at both compile and run-time. For action, this paper proposes more abstracted APIs to make implementation easier. In this paper, these APIs are called well-typed APIs.

Fig. 1 represents an overview of the proposed libraries. rcl and Data Distribution Service (DDS) are responsible for machine-to-machine and inter-process communications in ROS 2. rcl is the client library of ROS 2, and it wraps DDS. safe_drive, proposed in this paper, uses rcl to provide ROS 2 compatible APIs. async_lib, also proposed in this paper, is designed for multi-threading and provides well-typed intra-process communication APIs. safe_drive can use async_lib for multi-threading, and in this case, the verified multi-threaded executor of async_lib is used. This means that using only safe_drive is suitable for simple robot applications, while using both safe_drive and async_lib is appropriate for complex robot applications.

Sec. II summarizes related work. It describes middleware of robots, formal method, and session types. Sec. III shows a preliminary study of ROS 2's task executor. This section

This research is based on results obtained from a project, JPNP21027, subsidized by the New Energy and Industrial Technology Development Organization (NEDO).

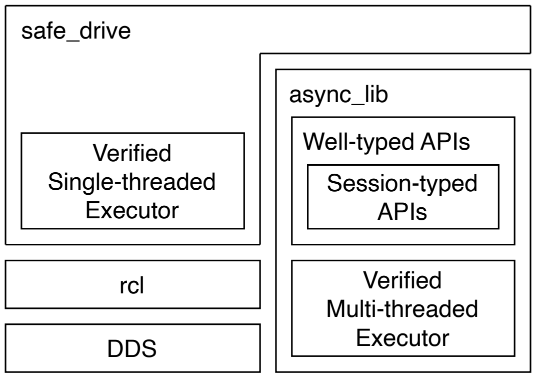
safe_drive
Verified Single-threaded Executor
rcl
DDS
async_lib
Well-typed APIs
Session-typed APIs
Verified Multi-threaded Executor

Fig. 1. Overview

shows ROS 2's executor causes starvation. Sec. IV describes formally verified executors of proposed libraries. This section specifies the proposed executors to verify these are deadlock- and starvation-free. Sec. VI shows implementation of proposed libraries. Sec. VII shows evaluations. This section compares our libraries and others. Sec. VIII describes conclusion.

II. Related Work

Middleware for Robots. The Robot Operating System 2 (ROS 2) [1], [2] is middleware for robots; it is not an operating system like Linux or Windows. ROS 2 uses a communication framework called Data Distribution Service (DDS) for machine-to-machine (M2M) and inter-process communications (IPC). By using DDS, ROS 2 provides three types of communications: topic, service, and action. Topic and service are publish/subscribe and request/response communications, respectively. Action consists of topic and service, and it will be explained in Sec. V-B.

rclcpp [16] and rclpy [17] are official client libraries of ROS 2 for C++ and Python, respectively. R2R [18], ros2_rust [19], and rclrust [20] are client libraries of ROS 2 for Rust. These libraries have a task executor and the design of rclcpp's executor has been explained by D. Casini *et al.* [21]. In Sec. III, it will be shown that the executor of rclcpp causes starvation, and its APIs can lead to deadlock.

In this paper, a ROS 2 client library called safe_drive [22], and an asynchronous library for async/await programming called async_lib are proposed. async_lib can be used as a multi-threaded executor of safe_drive. These libraries are implemented in Rust for safety and security. M. Noseda *et al.* [23] have reported that Rust is more secure than MISRA C/C++, which is a guideline of C/C++ for safety and security.

Formal method. By using formal methods, a system design can be described unambiguously and verified using mathematical logic. Model checking is a formal method that mathematically specifies systems.

TLA+ [7] is a model checker for specifying and verifying systems. It is based on linear temporal logic and provides the PlusCal language, which can be transpiled to TLA+'s expressions. In this paper, PlusCal and TLA+ are used for specification and verification. Tab. I shows notation of TLA+ and Rust.

TABLE I
Notation

TLA+	Rust	Description
`<<c, d>>`	`[a, b]`	sequence/array of `c` and `d`
`{c, d}`		set of `c` and `d`
`c = d`	`c == d`	`c` is equal to `d`
`c /= d`	`c != d`	`c` is not equal to `d`
`c := d`	`c = d`	assign value `c` to variable `d`
`[k |-> v]`	`{k: v}`	construct a structure
`f == E`	`fn f() { E }`	equation `E` is defined as `f`
$P \leadsto Q$		if P is true, Q is eventually true
$\Diamond P$		P is eventually true

```
P   :=   Recv<T, P>        receive T followed by P
     |   Send<T, P>        send T followed by P
     |   Offer<P, P>       offer the former or latter
     |   Choose<P, P>      choose the former or latter
     |   Rec<P>            label to jump
     |   Var<N>            jump to N-th P labeled by Rec
     |   Eps               terminate connection

N   :=   Z                 zero
     |   S<N>              N + 1
```

Fig. 2. Session Types for Rust (Backus-Naur form)

According to M. Luckcuck *et al.* [10], several formal methods have been applied to robots. However, conventional work described in [10] has focused on behavior of robots, and middleware has not been targeted. This paper focuses on middleware for robots and specifies and verifies that it is deadlock- and starvation-free.

Session types. Session types [12] are types to specify communication protocols. By using session types, violation of a protocol can be detected at compile or run time, and it guarantees that the protocol and implementation are deadlock-free.

Session types for Rust [15] is a library of session types for Rust. Session types for Rust can be used for only intra-process communications. X. Liu *et al.* [24] proposed session types for IPC and applied their methods to session types for Rust.

Session types are generally discussed in terms of linearity and the π-calculus [25]. However, this paper uses the notation of session types for Rust as shown in Fig. 2. `P` and `N` represent a protocol and a number of Peano axiom, respectively.

Fig. 3 is a simple protocol to demonstrate session types for Rust. This protocol can be encoded by session types for Rust as follows.

Listing 1. Simple Protocl in Session Types (Rust)
```rust
type Server = Rec<Recv<u64,
                  Send<bool,
                  Offer<Eps, Var<Z>>>>>;
```

Listing 1 presents the protocol from the server's viewpoint. The server receives a `u64` value and then sends a `bool` value. After that, the server offers either `Eps`, to close the connection, or `Var<Z>`, to jump to `Rec`.

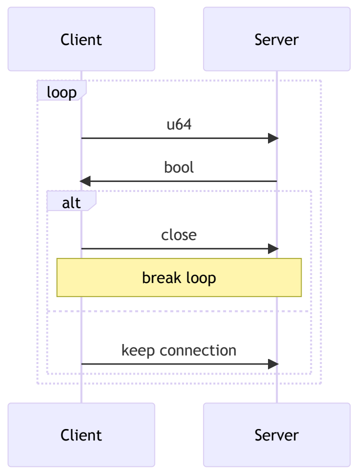

Fig. 3. Simple Protocol

TABLE II
DUALITY

P	P::Dual
`Recv<T, P>`	`Send<T, P::Dual>`
`Choose<P1, P2>`	`Offer<P1::Dual, P2::Dual>`
`Rec<P>`	`Rec<P::Dual>`
`Var<N>`	`Var<N>`
`Eps`	`Eps`

As shown in Fig. 3, a protocol exhibits mirror symmetry, known as duality. Tab. II represents the duality of session types, where `P::Dual` denotes the dual type of `P`. This means that if one endpoint receives a value of type `T`, the other endpoint sends a value of the same type `T`. The same applies to `Offer` and `Choose`. Therefore, the protocol from the client's viewpoint in Fig. 3 can be typed as `Server::Dual`.

III. Preliminary Study

In this section, 2 applications of ROS 2 will be introduced to show that starvation and deadlock can easily occur. This paper uses ROS 2 Humble on Linux 5.15 (Ubuntu 22.04) for experiments.

The following code snippet is an example of starvation. This creates long and short timers (line 5-10), 1 subscriber (line 13-15), and 1 publisher (line 18). The long and short timers are invoked every 1 second and 200 milliseconds, respectively. Note that the long timer sleeps for 1.1 seconds during the callback function (line 24), and the short timer publishes a message during an invocation (line 31).

Listing 2. Starvation of ROS 2 Multi-threaded Executor (C++)

```
1  SampleNode::SampleNode(const rclcpp::NodeOptions &
       options)
2  : Node("sample_node", options)
3  {
4      // timers
5      long_timer_ = this->create_wall_timer(
6          1s, std::bind(
7              &SampleNode::long_timer, this));
8      short_timer_ = this->create_wall_timer(
9          200ms, std::bind(
10             &SampleNode::short_timer, this));
11
12     // subscriber
13     msg_sub_ = this->create_subscription<std_msgs::
       msg::String>(
14         "message", 1,
15         std::bind(&SampleNode::subscribe, this, _1))
       ;
16
17     // publisher
18     msg_pub_ = this->create_publisher<std_msgs::msg
       ::String>("message", 10);
19 }
20
21 void SampleNode::long_timer()
22 {
23     std::this_thread::sleep_for(
24         std::chrono::milliseconds(1100));
25     return;
26 }
27
28 void SampleNode::short_timer()
29 {
30     std_msgs::msg::String msg;
31     msg.data = "Sent from short timer";
32     msg_pub_->publish(msg);
33     std::this_thread::sleep_for(
34         std::chrono::milliseconds(100));
35 }
36
37 void SampleNode::subscribe(const std_msgs::msg::
       String::SharedPtr msg)
38 {
39     std::this_thread::sleep_for(
40         std::chrono::milliseconds(100));
41 }
```

rclcpp provides multi-threaded and single-threaded executors, and users can choose one of the executors depending on workload. If the multi-threaded executor with a callback group is used to execute Listing 2, starvation will occur. More precisely, only the long timer will be invoked eventually because of a problem of the scheduling algorithm of ROS 2.

As described in [21], rclcpp's executor adopts the round-robin scheduling algorithm, and it does not seem that the algorithm and implementation cause such starvation. In other words, it is difficult to predict such starvation from any documentation or source code.

Another example involves a service, and it causes deadlock. The following code snippets, Listing 3 and 4, are for a client and a service, respectively.

In Listing 3, the client first waits for a server (line 1-3). If a server is available, it sends a request (line 5), and waits for a response (line 6).

Listing 3. Deadlock of ROS 2 Client (C++)

```
1 while (!client->wait_for_service(std::chrono::
      seconds(1))) {
2     if (!rclcpp::ok()){ /* error */ return 1; }
3 }
4
5 auto result_future = client->async_send_request(
      request);
6 if (rclcpp::spin_until_future_complete(node,
      result_future) !=
7     rclcpp::FutureReturnCode::SUCCESS)
8 { /* error */ return 1; }
```

`spin_until_future_complete` is a function that waits for a response and blocks the process. Because of the blocking function, a deadlock is caused.

Listing 4 is a callback function invoked when a request arrives. Normally, `response` will be updated to send a response, but it calls `exit` to emulate a crash (line 6).

Listing 4. ROS 2 Server (C++)

```
1 void handle_service(
2   const std::shared_ptr<omitted> request_header,
3   const std::shared_ptr<omitted> request,
4   const std::shared_ptr<omitted> response)
5 {
6     exit(0); // emulate crash
7 }
```

As a result of the crash, no response is sent to the client, and the client waits for a response indefinitely. To avoid such deadlock, programmers must be cautious with lost messages and set a timeout when receiving messages. However, a protocol used by a large project tends to be complex, and it becomes increasingly difficult to ensure that it is deadlock-free.

In this paper, a formally verified executor and well-typed APIs will be proposed to ensure deadlock- and starvation-free. The formally verified executor is a task runner, and its scheduler is verified to be deadlock- and starvation-free using TLA+. The well-typed APIs adopt session types, which guarantee deadlock-freedom at both compile and run-time, for designing and implementing complex protocols.

IV. Formally Verified Executor

safe_drive and async_lib proposed in this paper have a task executor. These executors are formally verified to be deadlock- and starvation-free. In this section, the specification and verification for the executors will be described.

A. Deadlock- and Starvation-free Executor

In this section, the formal specification of our executor will be introduced, and it will be verified that both deadlock-freedom and starvation-freedom are satisfied by TLA+. The full specification can be found on the repository **??**.

To specify a system, entities represented by constants and variables must be defined. The following code snippet is an example of constant definitions for specifying the executor.

Listing 5. Example of Constants (Config, TLA+)

```
CONSTANTS
    Timers = {"timer1", "timer2"}
    Subscribers = {"subscriber1", "subscriber2"}
    Servers = {"server1", "server2"}
    Clients = {"client1", "client2"}
    DeltaRange = 10
```

`Subscribers`, `Servers`, and `Clients` are sets to represent processes, which are equivalent to callback functions in ROS 2. `Timers` is a set of timer processes in ROS 2. Additionally, `Tasks` is defined as $\texttt{Tasks} = \texttt{Subscribers} \cup \texttt{Servers} \cup \texttt{Clients}$.

Timers are represented by the delta list [11]. Fig. 4 is a delta list of a set of timers containing Timer1, Timer2, Timer3,

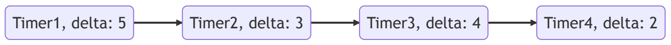

Fig. 4. Delta List

and Timer4. Because Timer1, Timer2, Timer3, and Timer4 wait for 5, 8, 12, and 14 clocks, respectively, the deltas of Timer2, Timer3, and Timer4 in the delta list are $3 = 8 - 5$, $4 = 12 - 8$, and $2 = 14 - 12$, respectively. In other words, a delta represents the difference between the current node's and the previous node's clocks. `DeltaRange` in Listing 5 is the maximum delta of the delta list. The scheduler of the delta list will pop the head node after 5 clocks, and Timer1 will be invoked immediately.

The following code snippet shows variables for the executor. `SetToSeq` converts a set to a sequence and `random_num` generates a random number.

Listing 6. Variables (PlusCal, TLA+)

```
variables
    \* list for timer
    \* example: <<[delta |-> 3, name |-> "timer1"],
    \*             [delta |-> 2, name |-> "timer2"]>>
    delta_list = SetToSeq({
        [delta |-> random_num(0, DeltaRange),
         name |-> x]: x \in Timers});

    \* a set of processes which is executable
    wait_set = {};

    \* a set of processes running now
    running = {};

    \* a set of processes waiting for a event
    waiting = Tasks;
```

`delta_list` is a delta list for handling timers. `wait_set` and `waiting` are subsets of `Tasks`, and `running` is a subset of $\texttt{Tasks} \cup \texttt{Timers}$.

These variables are used to represent state transitions. For example, if `timer1` is running, `subscriber1` is executable, and `client1` is waiting for an event, the states can be represented as $\texttt{timer1} \in \texttt{running}$, $\texttt{subscriber1} \in \texttt{wait\_set}$, and $\texttt{client1} \in \texttt{waiting}$. The scheduling algorithm can be specified by updating these variables.

The following equation is a predicate of the temporal logic to verify that the algorithm is starvation-free.

$$\begin{aligned}\text{starvation\_free} = \forall x \in (\text{Timers} \cup \text{Tasks})(&\\ (x \in \{y.\text{name} \mid y \in \text{delta\_list\_set}\}) \vee&\\ (x \in \text{wait\_set}) \leadsto x \in \text{running})& \quad (1)\end{aligned}$$

delta_list_set is a set derived from `delta_list`. Eq. (1) means that timers and executable processes will be eventually invoked. By using this, the starvation-freedom of our executor has been verified. TLA+ automatically detects deadlock, and deadlock-freedom has also been verified at the same time.

Contrary to the straightforward representation of state transitions, representing a clock requires a trick. The following code snippet is a macro to increment the clock.

Listing 7. Increment Clock (PlusCal, TLA+)

```
macro increment_clock()
begin
    if delta_list /= <<>> /\ delta_list[1].delta > 0
    then
        delta_list[1].delta :=
            delta_list[1].delta - 1;
    end if;
end macro;
```

To represent the incrementing of the clock, it decrements the delta at the head of the delta list. For example, if the delta at the head is 5 and it is decremented to 4, the timer at the head will be invoked after 4 clocks. This is equivalent to one clock passing. If a variable for the clock is introduced instead of decrementing the delta, the number of states will explode, making verification impossible to complete.

B. Verification of Delta List

The insertion algorithm of the delta list is at the core of the scheduler. In this section, the insertion algorithm will be specified and verified by TLA+. The full specification can be found on the repository **??**.

To verify the algorithm, the Rust implementation has been manually transpiled into PlusCal of TLA+. The following code snippet shows a Rust enum type for the delta list.

Listing 8. Delta List (Rust)

```
pub enum DeltaList<T> {
    Nil,
    Cons(Box<UnsafeCell<(Duration, T,
                         DeltaList<T>)>>),
}
```

`DeltaList` is a type name with variants: `Nil`, which has no data, and `Cons`, which has data of `Duration`, `T`, and `DeltaList<T>`. For verification, `DeltaList` in Rust must be encoded to TLA+'s data. The following code snippet shows variables, expressions to encode `DeltaList`, and a procedure for insertion in TLA+.

Listing 9. Delta List (PlusCal, TLA+)

```
1  variables
2      delta_list = nil,
3
4      result_insert_delta = nil,
5
6      result_seq_to_delta = <<>>,
7      result_delta_list_to_seq = <<>>;
8
9  define
10     nil == << "DelaList::Nil", <<>> >>
11     cons(delta, data, next) ==
12         << "DelaList::Cons",
13            <<delta, data, next>> >>
14
15 procedure insert(delta, data) begin
16     BeginInsert:
17         call insert_delta(delta_list, delta, data);
18     EndInsert:
19         delta_list := result_insert_delta;
20         return;
21 end procedure;
```

Note that Sections IV-A and IV-B use different source code. `delta_list` in Listing 6 is a sequence, while `delta_list` in Listing 9 is an encoded value of `DeltaList`.

TABLE III
TYPES OF MORPHISMS

$toDeltaList :$	$\{Duration\} \rightarrow DeltaList$
$toSeqS :$	$\{Duration\} \rightarrow {<<}Duration{>>}$
$toSeqD :$	$DeltaList \rightarrow {<<}Duration{>>}$
$sort, \Delta :$	${<<}Duration{>>} \rightarrow {<<}Duration{>>}$

`result_insert_delta` (line 4) is a variable for storing the result of `insert_delta` (line 17), which implements the insertion algorithm. `nil` and `cons` (line 10-13) are expressions to encode `DeltaList` in Rust. `DeltaList` can be encoded using a nested sequence as described here.

The correctness of the insertion algorithm can be confirmed by ensuring that the following commutative diagram is satisfied.

$$\begin{tikzcd}
\{Duration\} \arrow[rr, "toDeltaList"] \arrow[d, "toSeqS"] & & DeltaList \arrow[dd, "toSeqD"] \\
{<<}Duration{>>} \arrow[d, "sort"] & \circlearrowleft & \\
sorted \arrow[rr, "\Delta"] & & result
\end{tikzcd} \tag{2}$$

$\{Duration\}$ and ${<<}Duration{>>}$ are a set and a sequence of durations, respectively. Tab. III shows the types of the morphisms, where $A \rightarrow B$ represents the morphism which takes A and returns B. $toDeltaList$ takes a set of durations and maps it to a delta list by using `insert` in Listing 9. $toSeqS$ and $toSeqD$ are maps to a sequence of durations. $sort$ is a sort function. Δ calculates differences between adjacent elements. If $output$ and $input$ are the argument and result of Δ, then $output[i] = input[i] - input[i-1]$ for $i > 1$, and $output[1] = input[1]$ where the origin is 1.

`result_seq_to_delta` in Listing 9 can be derived by Eq. 2 as follows.

$$\text{result\_seq\_to\_delta} = \Delta \circ sort \circ toSeqS(\{Duration\})$$

This indicates that what the delta list is.

`result_delta_list_to_seq` in Listing 9 can also be derived as follows.

$$\text{result\_delta\_list\_to\_seq} = toSeqD \circ toDeltaList(\{Duration\})$$

This indicates how the insertion algorithm is implemented. Based on the facts, the correctness of the insertion algorithm has been verified by the following predicate.

$$delta\_equality = \Diamond(\text{result\_seq\_to\_delta} = \text{result\_delta\_list\_to\_seq})$$

V. Well-typed APIs

async_lib proposed in this paper provides APIs for intra-process communications. The APIs adopt session types to achieve deadlock-freedom. In this paper, the design of the APIs called well-typed APIs will be described.

A. Service by Session Type

ROS 2 provides only request and response style service APIs. However, this paper proposes more complex APIs for specifying actions in the following section. For the implementation, session types for Rust [15] are used and modified for async/await.

Fig. 5 shows an example service. In this figure, there are a client, a proxy, and a server. *ClientToProxy* and *Client* are endpoints of the client. *ProxyCli* and *ProxySrv* are endpoints of the proxy. *ServerFromProxy* and *Server* are endpoints of the server. This protocol sends an endpoint (Rx) from the client to the server via the proxy. The client then sends `u64` data to the server using the endpoints, one of which (Rx) is transmitted via the proxy.

Fig. 5 can be specified by session types as follows. `Recv<A, Next>` indicates that the protocol receives `A` and then proceeds with `Next`. `Send<A, Next>` indicates that the protocol sends `A` and then proceeds with `Next`. `Eps` indicates closing the session.

Listing 10. Protocol of Example Service (Rust)

```rust
1  // Client - Proxy
2  type ProxyCli = Recv<EndpointRx, Eps>;
3  type ClientToProxy = <ProxyCli as HasDual>::Dual;
4
5  // Proxy - Server
6  type ProxySrv = Send<EndpointRx, Eps>;
7  type ServerFromProxy = <ProxySrv as HasDual>::Dual;
8
9  // Client and Server
10 type Server = Recv<u64, Eps>;
11 type Client = <Server as HasDual>::Dual;
12
13 // endpoint (Rx)
14 type EndpointRx = Chan<(), Server>;
```

`EndpointRx` (line 14) is an endpoint to be sent from the client to the server, and its protocol is specified by `Server` (line 10). Receiving and sending the endpoint by the proxy are specified by `ProxyCli` (line 2) and `ProxySrv` (line 6.)

The server can be implemented as follows. This function takes an endpoint specified by `ServerFromProxy`.

Listing 11. Server Implementation (Rust)

```rust
1 async fn srv(c: Chan<(), ServerFromProxy>) {
2     // Receive an endpoint.
3     let (pr, server) = c.recv().await;
4     pr.close();
5
6     // Receive a value from the endpoint.
7     let (c, _result) = server.recv().await;
8     c.close();
9 }
```

First, it receives an endpoint (line 3), then closes the session with the proxy (line 4). After that, it receives data through the received endpoint (line 7), and closes the session related to the endpoint (line 8).

If the implementation violates the specification as described in Listing 10, it either cannot be compiled or will panic at run-time. This means that if there is no infinite loop, the implementation is deadlock-free. The client and proxy can also be implemented like the server.

B. Action by Typestate Programming

As shown in 3, inappropriate usage of ROS 2's APIs leads deadlock. To prevent the deadlock, async_lib uses typestate programming along with session types. By using session types, deadlock can be discovered at compile- and run-time. This section describes the design of typestate programming along with session types.

Fig. 6 represents the generic protocol of an action. The client has 2 endpoints, *Client* and *Sender (Client)*, and the server has 2 endpoints, *Server* and *Sender (Server)*. *Client* and *Server* are endpoints of session types, and *Sender (Client)* and *Sender (Server)* are endpoints of a normal channel, not session types. By doing this, feedback values can be transmitted using an unreliable channel, which is suitable for real-time communication. Attributes related to reliability can be configured by `attribute` in Fig. 6.

Fig. 6 can be specified as follows. `Rec<P>` defines a label for a loop, and it is used by `Var<Z>`, which is a jump instruction. `Offer<P, Q>` offers either `P` or `Q`, and `Choose<P, Q>` chooses between `P` and `Q` offered by `Offer`. `bounded::Receiver` is an endpoint of a normal channel, and `bounded::attribute` is an attribute of this.

Listing 12. Protocol Specification of Action (Rust)

```rust
type ProtoServer<G, F, R> =
    Rec<ProtoServerInn<G, F, R>>;

type ProtoServerInn<G, F, R> =
    Offer<Eps /* Close. */,
          ProtoServerGoal<G, F, R>>;

type ProtoServerGoal<G, F, R> =
    Recv<G /* Receive a goal. */,
         ProtoServerGoalResult<F, R>>;

type ProtoServerGoalResult<F, R> =
    Choose<Var<Z> /* Reject. */,
           ProtoServerSendGoalResult<F, R>
           /* Accept. */>;

type ProtoServerSendGoalResult<F, R> =
    S::Send<
        // Send an ID and a response of the goal.
        (u64, GoalResponse),
        ProtoServerFeedback<F, R>,
    >;

type ProtoServerFeedback<F, R> = Recv<
    bounded::Attribute, /* Receive an attribute. */
    Send<bounded::Receiver<F> /* Send Rx. */,
         ProtoServerResult<R>>>;

type ProtoServerResult<R> = S::Send<
    ResultStatus<R>, /* Send a result. */
    S::Var<S::Z>,    /* Goto ProtoServerInn. */
>;
```

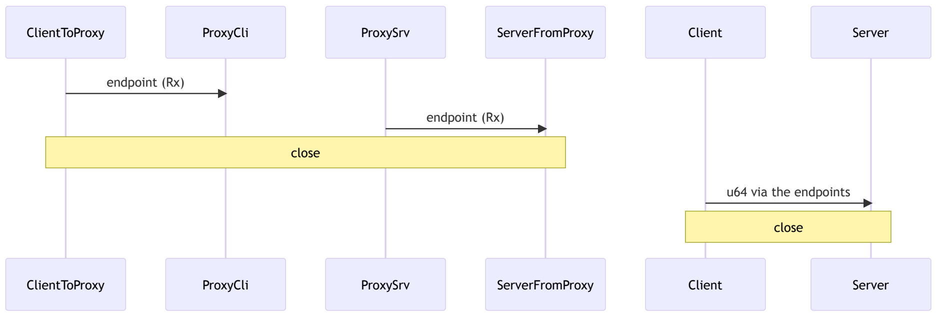

Fig. 5. Example Service

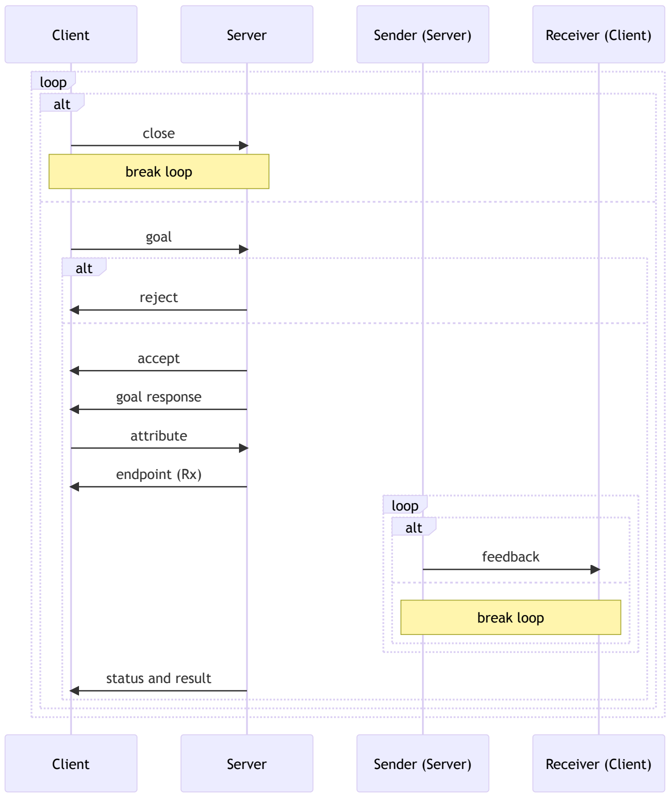

Fig. 6. Protocol of Action

An action can be implemented using the session types in Listing 12, but we provide more abstracted APIs for actions to make implementation easier. To achieve this, this paper defines four states and represents the states using Rust types. This method is called typestate programming.

Fig. 7 represents the states and transitions. The nodes and edges can be implemented as structures and methods in Rust, respectively. The *accept* transition in Fig. 7 performs *accept* in Fig. 6. Other transitions are interpreted straightforwardly.

The following code snippet shows the structures and methods for the action server. `G`, `F`, and `R` indicate types of goal, feedback, and result, respectively. Each structure represents a state in Fig. 7.

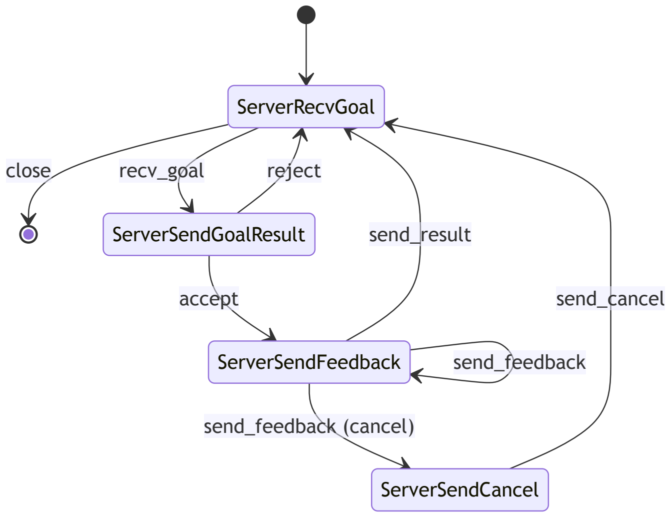

Fig. 7. State Transition of Action Server

Listing 13. Action by Typestate Programming (Rust)

```rust
pub struct ServerRecvGoal<G, F, R>;

impl<G, F, R> ServerRecvGoal<G, F, R> {
    pub async fn recv_goal(self) ->
        Option<(ServerSendGoalResult<G, F, R>, G)>
    { /* omitted */}
}

pub struct ServerSendGoalResult<G, F, R>;

impl<G, F, R> ServerSendGoalResult<G, F, R> {
    pub async fn reject(self) ->
        ServerRecvGoal<G, F, R> { /* omitted */}

    pub async fn accept(self,
                        response: GoalResponse) ->
        ServerSendFeedback<G, F, R> { /* omitted */}
}

pub struct ServerSendFeedback<G, F, R>;

impl<G, F, R> ServerSendFeedback<G, F, R> {
    pub async fn send_feedback(self, feedback: F)
        -> ServerFeedbackOrCancel<G, F, R>
    { /* omitted */}

    pub async fn send_result(self, result: R)
        -> ServerRecvGoal<G, F, R> { /* omitted */}

    pub async fn send_abort(self)
31          -> ServerRecvGoal<G, F, R> { /* omitted */}
32 }
33
34 pub struct ServerSendCancel<G, F, R>;
35
36 impl<G, F, R> ServerSendCancel<G, F, R> {
37     pub async fn send_cancel(self)
38         -> ServerRecvGoal<G, F, R> { /* omitted */ }
39 }
40
41 pub enum ServerFeedbackOrCancel<G, F, R> {
42     Feedback(ServerSendFeedback<G, F, R>),
43     Cancel(ServerSendCancel<G, F, R>),
44 }
```

`ServerRecvGoal::recv_goal` (line 4-6) returns `None` if `close` is chosen. `ServerSendGoalResult` has `reject` (line 12-13) and `accept` (line 15-17) methods to reject or accept the received goal value. `ServerSendFeedback` has `send_feedback` (line 23-25), `send_result` (line 27-28), and `send_abort` (line 30-31) methods to send feedback or result values. If a connection is canceled, `ServerSendFeedback` is used and it sends a cancel result by `send_cancel` (line 37-38).

Note that every method takes `self`. This indicates that violated transitions are prohibited by the compiler due to the move semantics. Client types and methods can also be implemented similarly.

C. Example Usage

The following code snippet is client code of service that violates the protocol.

Listing 14. Protocol Violation (Rust)

```
type Client = Send<u64, Recv<bool, Eps>>;

let client = create_client::<Client>("service".into
    ()).await.unwrap();
let ch = client.send(100).await;
let ch = ch.send(200).await; // Compilation error.
```

The `Client` protocol stipulates that the client must send a `u64` value and then receive a `bool` value. However, in this code, two integer values are sent. This behavior violates the protocol and will be rejected by the Rust compiler. The Rust compiler will report the following error message when compiling Listing 14.

Listing 15. Error Message

```
error[E0599]: no method named `send` found for
    struct `Chan<(), async_lib::session_types::Recv<
    bool, async_lib::session_types::Eps>>` in the
    current scope
  --> userland/src/lib.rs:135:17
    |
135 |     let ch = ch.send(200).await; //
    Compilation error.
    |                 ^^^^ method not found in `Chan
    <(), Recv<bool, Eps>>`
    |
    = note: the method was found for
            - `Chan<E, async_lib::session_types::
    Send<A, P>>`
```

An example action server, which increments a number, can be implemented as follows.

Listing 16. Example Action Server (Rust)

```
1  'outer: loop {
2      // Receive a goal value.
3      let Some((send_goal_result, goal)) =
4          server_recv_goal.recv_goal().await
5          else { /* closed. */ return };
6
7      // Send a goal result.
8      let mut server_send_feedback = send_goal_result
9          .accept(GoalResponse::AcceptAndExecute)
10         .await;
11
12     // Send feedback values.
13     let mut result = 0;
14     for i in 0..=goal {
15         result += i;
16
17         match server_send_feedback.send_feedback(
       result).await {
18             // Feedback.
19             ServerFeedbackOrCancel::Feedback(f) =>
       server_send_feedback = f,
20
21             // Canceled.
22             ServerFeedbackOrCancel::Cancel(c) => {
23                 server_recv_goal = c.send_cancel().
       await;
24                 continue 'outer;
25             }
26         }
27     }
28
29     // Send a result value.
30     server_recv_goal = server_send_feedback.
       send_result(result).await;
31 }
```

First, it receives a goal value (lines 3-5) from a client, and then sends a goal result to the client (lines 8-9). After receiving the goal value, `goal`, it increments a number, `result`, and sends feedback to the client `goal` times (lines 13-27). After that, it sends a result to the client (line 30). If the connection is closed, the action will be canceled (lines 22-24).

Listing 16 is an asynchronous task for a connection. Each connection can be created as follows.

Listing 17. Connection Reception of The Action Server (Rust)

```
1  async fn server_task() {
2      let server = create_server::<u64, u64, u64>
3          ("action_server".into()).unwrap();
4
5      // Accept a connection.
6      while let Ok(server_recv_goal) = server.accept()
       .await {
7          spawn(task, // the example action server
8                SchedulerType::RoundRobin).await;
9      }
10 }
```

An action server is first created by `create_server` (lines 2-3). After that, `server.accept` is called to accept connections (line 6), and the task described in Listing 16 is spawned (lines 7-8). If there are any violations regarding the protocol and state transition, these code snippets cannot be compiled.

VI. Implementation

Both safe_drive [22], which is a formally verified ROS 2 client library, and async_lib that provides ROS 2 style APIs

TABLE IV
COMPARISON

	spec	DL-free	SV-free	well-typed APIs
* safe_drive [22]	✓	✓	✓	
* async_lib	✓	✓	✓	✓
rclpy [17]				
rclcpp [16]				
R2R [18]				
ros2_rust [19]				
rclrust [20]				

*: proposed libraries
spec: formal specification
DL-free: formally verified that scheduler is deadlock-free
SV-free: formally verified that scheduler is starvation-free
well-typed APIs: well-typed APIs for deadlock-freedom

for intra-process communications, have been implemented in Rust.

safe_drive consists of approximately 7.7K LOC, excluding automatically generated files. It depends on safe_drive_msg [26], which is about 2.8K LOC, and idl_parser [27], which is around 4.4K LOC, to generate Rust types from interface definition language (IDL) files. IDL files are used by ROS 2 to define user-defined types.

async_lib uses session types for Rust to provide well-typed APIs described in Sec. V. It consists of approximately 5.1K LOC. async_lib can work on no_std environments. It means that this library can be used in environments without any operating system. To support async/await programming, session types for Rust have been modified.

VII. EVALUATION

In this section, safe_drive and async_lib will be compared with others. Evaluation metrics are as follows: (1) Provision of any formal specification. (2) Formal verification of scheduler's deadlock-freedom. (3) Formal verification of scheduler's starvation-freedom. (4) Provision of APIs or language mechanisms for deadlock-freedom as described in Sec. V.

Tab. IV shows a comparison between the proposed libraries and others. safe_drive is a wrapper for ROS 2's core library, called rcl [28], written in C, and it implements a single-threaded scheduler for the executor. The single-threaded scheduler has been formally specified and verified that it is deadlock- and starvation-free as described in Sec. IV. Because rcl's APIs are connectionless, while session types are connection-oriented, session types cannot be applied to rcl's APIs. As a result, safe_drive does not implement well-typed APIs.

There are two choices to implement multi-threaded applications in Rust; using threads or async/await. async_lib is a library for async/await, and it implements well-typed APIs for intra-process communications.

async_lib has a scheduler and it is also formally specified and verified as described in Sec. IV. Note that the starvation-freedom of async_lib depends on algorithms of mutual exclusion. This means that if the mutual exclusion algorithms used by async_lib is not starvation free, the library is not starvation-free. To achieve starvation-free, async_lib internally uses MCS lock [29], which has been formally verified to be starvation-free [30].

TABLE V
CAPABILITIES

	single	multi	ROS 2 APIs (inter)	WT APIs (intra)
1: safe_drive	✓		✓	
2: ascyn_lib		✓		✓
1 + 2		✓	✓	✓

single: single-threaded executor
multi: multi-threaded executor
ROS 2 APIs (inter): ROS 2 APIs for M2M and inter-process communications
WT APIs (intra): well-typed APIs for intra-process communications

rclpy and rclcpp are official client libraries in Python and C++, respectively. R2R, ros2_rust, and rclrust are also libraries written in Rust. These libraries have not been formally specified or verified at all.

Tab. V summarizes capabilities of safe_drive and async_lib. safe_drive has a single-threaded executor, and async_lib has a multi-threaded executor. safe_drive can be used with async_lib for multi-threading. Well-typed APIs are thus only available for multi-threading and intra-process communications. Providing well-typed APIs for the single-threaded executor of safe_drive is difficult because it is based on callback style concurrent programming, which is less abstracted to apply session types. Well-typed APIs for IPC of ROS 2 are not supported because of rcl described above. safe_drive supports ROS 2 APIs for M2M and inter-process communications, and it can also be used with async_lib.

VIII. CONCLUSION

ROS 2 is middleware for robot applications, but it suffers from deadlock and starvation. As a preliminary study, Sec. III showed that rclcpp causes starvation and misuse of rclcpp's APIs leads to deadlock. In this paper, safe_drive and async_lib have been proposed to avoid such deadlock and starvation. In addition, these libraries are more secure because these are written in Rust as mentioned by M. Noseda *et al.* [23].

safe_drive is a client library of ROS 2 and it has a single-threaded task executor. To verify that it is deadlock- and starvation-free, the scheduling algorithm and the delta-list, which is a core data structure, are specified and verified using PlusCal and TLA+.

To represent time in the specification, the head of delta-list has been decremented instead of introducing a global time counter. A global time counter can also represent time, but it explodes the number of states, because time n and $n+1$ are treated as different states by a model checker like TLA+.

To verify the delta-list itself, a commutative diagram has been introduced. The diagram shows that $\mathit{toSeqD} \circ \mathit{toDeltaList} = \mathit{sort} \circ \mathit{toSeqS}$. $\mathit{toSeqD} \circ \mathit{toDeltaList}$ and $\mathit{sort} \circ \mathit{toSeqS}$ represent how the delta-list is implemented and what the delta-list is, respectively. The correctness can be verified by checking that the equation is satisfied.

Additionally, this paper proposed well-typed APIs for service and action communications to avoid deadlock caused by misuse of APIs. Session types are type systems used to specify communication protocols, and protocol violations which could cause deadlock can be detected at compile-time or run-time. This paper incorporates session types into service communication.

The protocol for action is complex because it uses four endpoints, as described in Fig. 6, and the protocol cannot be specified by session types alone. For usability and safety, this paper proposes more abstracted APIs for action, which describe states using types in Rust, a method known as typestate programming. As shown in Fig. 7, there are five nodes and eight edges in the state transition of a action server. Programmers must implement this state transition flawlessly; otherwise, the implementation could lead to a deadlock. Typestate programming can detect such protocol violations at compile-time.

It should be noted that the verification of the scheduler and the use of well-typed APIs cannot guarantee that any user code will avoid deadlock. For instance, if there is an infinite loop in a client, the communication will result in a deadlock. Therefore, while these measures significantly reduce the risk of deadlocks, it remains essential for developers to review and test their code to ensure correct behavior.

The evaluation demonstrated that safe_drive and async_lib are the only libraries that have been formally verified, with async_lib being the sole provider of well-typed APIs. As robots become more pervasive in our society, their safety grows increasingly important. Therefore, the use of verified and safe APIs should be seen as paramount.

This paper focuses only on ensuring the deadlock- and starvation-free nature of the executors, but other mechanisms, such as the Quality of Service (QoS) of communication channels, should also be verified. Furthermore, such verification is important not only for middleware but also for the operating system. In the future, I plan to design, implement, and verify both middleware and operating systems.

References

[1] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot Operating System 2: Design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[2] “ROS 2: Robot Operating System.” Accessed April 11th, 2023. https://www.ros.org/.

[3] “Autoware - the world’s leading open-source software project for autonomous driving.” Accessed April 19th, 2023. https://github.com/autowarefoundation/autoware.

[4] I. D. Miller, F. C. Ojeda, A. Cowley, S. S. Shivakumar, E. S. Lee, L. Jarin-Lipschitz, A. Bhat, N. Rodrigues, A. Zhou, A. Cohen, A. Kulkarni, J. Laney, C. J. Taylor, and V. Kumar, “Mine Tunnel Exploration using Multiple Quadrupedal Robots,” *CoRR*, vol. abs/1909.09662, 2019.

[5] R. Halder, J. Proença, N. Macedo, and A. Santos, “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.

[6] G. Holzmann, “The model checker SPIN,” *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[7] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, June 2002.

[8] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, *Testing Real-Time Systems Using UPPAAL*, pp. 77–117. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[9] M. Kwiatkowska, G. Norman, and D. Parker, “PRISM 4.0: Verification of Probabilistic Real-time Systems,” in *Proc. 23rd International Conference on Computer Aided Verification (CAV’11)* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *LNCS*, pp. 585–591, Springer, 2011.

[10] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, “Formal Specification and Verification of Autonomous Robotic Systems: A Survey,” *ACM Comput. Surv.*, vol. 52, sep 2019.

[11] D. Comer, *Operating System Design - The Xinu Approach*. CRC Press, 2nd, ed., 2015.

[12] O. Dardha, E. Giachino, and D. Sangiorgi, “Session Types Revisited,” in *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, PPDP ’12, (New York, NY, USA), pp. 139–150, Association for Computing Machinery, 2012.

[13] K. Imai, N. Yoshida, and S. Yuen, “Session-ocaml: A session-based library with polarities and lenses,” *Science of Computer Programming*, vol. 172, pp. 135–159, 2019.

[14] R. Pucella and J. A. Tov, “Haskell Session Types with (Almost) No Class,” *SIGPLAN Not.*, vol. 44, pp. 25–36, sep 2008.

[15] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen, “Session Types for Rust,” in *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, (New York, NY, USA), pp. 13–22, Association for Computing Machinery, 2015.

[16] “rclcpp: ROS Client Library for C++.” Accessed April 11th, 2023. https://github.com/ros2/rclcpp.

[17] “rclpy: ROS Client Library for the Python language.” Accessed April 11th, 2023. https://github.com/ros2/rclpy.

[18] “R2R: Easy to use, runtime-agnostic, async rust bindings for ROS2.” Accessed April 11th, 2023. https://github.com/sequenceplanner/r2r.

[19] “ROS 2 for Rust.” Accessed April 11th, 2023. https://github.com/ros2-rust/ros2_rust.

[20] “rclrust: Yet another ROS2 client library written in Rust.” Accessed April 11th, 2023. https://github.com/rclrust/rclrust.

[21] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, “Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling,” in *Euromicro Conference on Real-Time Systems*, 2019.

[22] “safe_drive: Formally Specified Rust Bindings for ROS2.” Accessed April 11th, 2023. https://github.com/tier4/safe_drive.

[23] M. Noseda, F. Frei, A. Rüst, and S. Künzli, “Rust for secure IoT applications : why C is getting rusty,” WEKA, jun 2022. Embedded World Conference, Nuremberg, Germany, 21-23 June 2022.

[24] X. Liu, Y. Takano, and A. Miyaji, “Design and implementation of session types-based tcp and unix domain socket,” in *2022 7th International Conference on Information and Network Technologies (ICINT)*, pp. 72–79, 2022.

[25] N. Kobayashi, B. C. Pierce, and D. N. Turner, “Linearity and the Pi-Calculus,” *ACM Trans. Program. Lang. Syst.*, vol. 21, pp. 914–947, sep 1999.

[26] “Message Transpiler for safe_drive.” Accessed April 11th, 2023. https://github.com/tier4/safe_drive_msg.

[27] “T4 IDL Parser.” Accessed April 11th, 2023. https://github.com/tier4/idl_parser.

[28] “rcl: ROS Client Library.” Accessed April 11th, 2023. https://github.com/ros2/rcl.

[29] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors,” *ACM Trans. Comput. Syst.*, vol. 9, p. 21–65, feb 1991.

[30] J. Kim, V. Sjöberg, R. Gu, and Z. Shao, “Safety and Liveness of MCS Lock—Layer by Layer,” in *Programming Languages and Systems* (B.-Y. E. Chang, ed.), (Cham), pp. 273–297, Springer International Publishing, 2017.