ROS-Maude Bridge: Formal Verification for ROS2 Systems

Renjie Sun

May 12, 2025

1 Introduction

This project integrates **Maude** with **ROS2**, enabling the modeling, simulation, and verification of ROS2 applications using formal methods. By treating ROS2 components (publishers, subscribers, topics) as external objects in Maude, robotic systems can be rigorously analyzed while maintaining compatibility with real-world ROS2 deployments.

2 Background

2.1 Rewrite Logic and Maude

Maude is a rewriting-logic-based executable formal specification language and high-performance analysis tool for distributed systems. Formal analysis methods include: simulation, reachability analysis, LTL model checking, and theorem proving. A Maude module specifies a rewrite theory $(\Sigma, E \cup B, R)$, where:

- Σ is an algebraic signature; i.e., a set of sorts, subsorts, and function symbols.
- $(\Sigma, E \cup B)$ is a membership equational logic theory specifying the system's data types, with E a set of conditional equations and membership axioms, and B a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed modulo the axioms B.
- R is a collection of labeled conditional rewrite rules $[l] : t \longrightarrow t'$ **if** *cond*, specifying the system's local transitions.

2.2 Object Oriented Configuration

A class declaration $classC|att_1 : s_1, ..., att_n : s_n$ declares a class C of objects with attributes $att_1, ..., att_n$ of sorts $s_1, ..., s_n$. An object of class C is represented as a term $< o : C|att_1 : val_1, ..., att_n : val_n >$, where o, of sort `Oid`, is the object's identifier, and where $val_1, ..., val_n$ are the current values of the attributes $att_1, ..., att_n$. A message is a term of sort `Msg`. A system state is modeled as a term of the sort `Configuration`, and has the structure of a multiset made up of objects and messages. The dynamic behavior of a system is axiomatized by specifying its transition patterns as rewrite rules. For example, the rule in 1

```

rl [initTalker] :
    < ME : Talker | publisher : na, words : INT WORDS >
    createdPublisher(ME, ROS2, any, NEW)
⇒
    < ME : Talker | publisher : NEW, words : WORDS >
    publish(NEW,ME,[stamp: (sec: ros#int32(INT), nanosec:
       ros#uint32(0)), frame-id: ros#string("a")]) .
```

Listing 1: exmaple 1

defines a family of transitions in which a message `createPublisher(ME,ROS2,any,NEW)` is read and consumed by the object `ME` of class `Talker`, whose attribute `puiblisher` is changed to NEW, and a new message `publish(NEW,ME,...)` is generated. `ME` might have other Attributes then `publisher` and `words`, but they need not be mentioned in this rule since they do not change in the transitions defined by this rule.

2.3 Custom Special Operators

User-defined and many predefined functions in Maude are specified with equations, but the prelude also includes some special operators whose behavior is internally defined in the C++ code of the interpreter. Most operations on the built-in types `Nat`, `Float`, `Qid`, and `String`, some polymorphic operators like equality `==`, and most descent functions in the META-LEVEL module are examples of special operators. The maude python binding binding allow declaring custom special operators whose behavior against equational reduction and/or rule rewriting is defined in Python. In the Maude side, the operator should be declared first with the special attribute and its id-hook SpecialHubSymbol option. For instance, the gamma function that extends the factorial to real (and complex) numbers can be declared as the following gamma operator within a module.

Listing 2: Special Hook declaration

```
op gamma : Float -> Float [special (
    id-hook SpecialHubSymbol (gamma)
)] .
```

On the Python side, as in 3, we have to define and register the callback that is invoked when a term with gamma on top is reduced or rewritten. This is done by inheriting the maude.Hook class and implementing its run method, and then calling the functions `connectEqHook` and/or `connectRlHook` to register an object of the class as the handler for the special operator. Finally, connecting the `maude.hook` instance to maude will do.

Listing 3: Special Hook Implementation

```
class GammaHook(maude.Hook):
    def run(self, term, data):
        module = term.symbol().getModule()
        argument , = term.arguments()
        value = math.gamma(float(argument))
        return module.parseTerm(str(value))
hook = GammaHook()
maude.connectEqHook('gamma', hook)
```

2.4 ROS2

ROS2 (Robot Operating System 2) is a middleware framework designed for building distributed robotic systems, addressing critical requirements in modern robotics through its adaptable architecture. Its applications span autonomous vehicles, where it enables sensor fusion across decentralized nodes using LiDAR and cameras; industrial automation, facilitating synchronization between robotic arms and IoT devices; healthcare, supporting surgical robots in performing real-time, fault-tolerant operations; and drone technology, allowing swarm coordination with collision avoidance capabilities. The framework's design emphasizes scalability, enabling the addition or removal of nodes without system redesign, while enhancing fault tolerance by isolating component failures such as crashed nodes. Additionally, ROS2 ensures interoperability through a unified interface that integrates heterogeneous hardware, making it a versatile solution for diverse robotic challenges. ROS2 is tailored for distributed robotic systems, leveraging a data-centric architecture built on the Data Distribution Service (DDS) protocol to enable real-time, scalable communication between components. At its core, ROS2 operates through nodes-modular processes that perform specific tasks—which interact via a suite of communication APIs designed for flexibility and reliability. These APIs support multiple interaction models:

- **Topics**: Implement a publish-subscribe pattern for asynchronous, many-to-many communication. Nodes publish messages to named topics (e.g., sensor data, navigation goals) using Publisher APIs, while subscribers receive updates via Subscription callbacks. This model is ideal for high-frequency data streams, such as LiDAR scans or camera feeds in autonomous vehicles.
- **Services**: Enable synchronous request-reply interactions via Service and Client APIs. A service node processes requests (e.g., triggering a robotic arm movement) and returns a response, ensuring deterministic behavior for tasks like industrial automation or surgical robot command execution.

Underlying these APIs is ROS2's DDS-based middleware, which provides granular Quality of Service (QoS) policies to tailor communication reliability, deadline constraints, and durability. For example, sensor data might use a `Best Effort` QoS for low latency, while surgical robot commands prioritize `Reliable` delivery. Nodes auto-discover each other via decentralized discovery, eliminating centralized coordination and enhancing scalability. Additionally, ROS2's Client Library (`rcl`) abstracts DDS complexities, offering consistent interfaces in C++ (`rclcpp`) and Python (`rclpy`), while ensuring interoperability across platforms. By decoupling hardware-specific layers (via Hardware Abstraction Layers or HALs) and isolating node failures (e.g., a crashed sensor node doesn't disrupt the entire system), ROS2's communication architecture directly underpins its scalability, fault tolerance, and adaptability to heterogeneous robotic ecosystems.

3 Publisher-Subscriber Routing Model: Formal foundations

In this report, we focus on a subset of ROS2's features, namely its Publisher-Subscription model of communication in a distributed robotic system. We formalized this model of concurrency as follows.

3.1 System Definition

Given node space $\mathcal{N}$, messages space $\mathcal{M}$, publisher space $\mathcal{P}$, subscriber space $\mathcal{S}$, and topic space $\mathcal{T}$, a ROS2 router's state is a tuple:

$$\langle P, S, I, O \rangle \in State$$

where:

- $P \subseteq \mathcal{P} \times \mathcal{N} \times \mathcal{T}$: Publishers (nodes writing to topics)
- $S \subseteq \mathcal{S} \times \mathcal{N} \times \mathcal{T}$: Subscribers (nodes reading from topics)
- $I : \mathcal{P} \times \mathcal{N} \times \mathcal{T} \rightsquigarrow \mathcal{M}^*$: publish queue for each publisher.
- $O : \mathcal{S} \times \mathcal{N} \times \mathcal{T} \rightsquigarrow \mathcal{M}^*$: subscription queue for each subscription.

3.2 Message Passing Semantics

With (p', n', t') ranges $\mathrm{dom}(I)$ and (s', n', t') ranges $\mathrm{dom}(S)$ in their contexts, define

- **Issue**: For $p = (n, t) \in P$, sending message $m \in \mathcal{M}$ to topic $t \in T$:

$$\frac{\begin{gathered}\langle P, S, I, O\rangle \in \mathit{State}\\ (p, n, t) \in P, (m) \cdot I'_{(p,n,t)} = I_{(p,n,t)}\\ I' = \begin{cases}(p, n, t) \mapsto I'_{(p,n,t)}\\ (p', n', t') \mapsto I_{(p',n',t')}\end{cases} \qquad O' = \begin{cases}(s', n', t) \mapsto O_{(s',n',t')} \oplus m\\ (s', n', t') \mapsto O_{(s',n',t')}\end{cases}\end{gathered}}{\mathbf{issue}(p, n, t, m) : \langle P, S, I, O\rangle \to \langle P, S, I', O'\rangle}$$

  where $\cdot$ is list concatenation that comes with $\mathcal{M}^*$. Intuitively we could find a unique computation, if any, without m being specified given the left-hand state because $I_{(p,n,t)} \in \mathcal{M}^*$ has an unique starting element. Furthermore, parameter t is also redundant if we assume empty initial state $\langle \emptyset, \emptyset, \emptyset, \emptyset\rangle$. It follows from the fact that creating two publishers who share identifier and owner node but differ in topic is impossible, for the definition below.

- **Create Publisher**: For $n \in \mathcal{N}$, create publisher on topic $t \in T$:

$$\frac{\begin{gathered}\langle P, S, I, O\rangle \in \mathit{State}\\ p \in \mathcal{P}, n \in \mathcal{N}, t \in \mathcal{T}, \forall t' \in \mathcal{T} : (p, n, t') \notin P\end{gathered}}{\mathbf{publisher}(n, t) : \langle P, S, I, O\rangle \to \langle P \cup \{(p, n, t)\}, S, I, O\rangle}$$

- **Create Subscription**: For $n \in \mathcal{N}$, subscribe topic $t \in T$:

$$\frac{\begin{gathered}\langle P, S, I, O\rangle \in \mathit{State}\\ s \in \mathcal{S}, n \in \mathcal{N}, t \in \mathcal{T}, \forall t' \in \mathcal{T} : (s, n, t') \notin S\end{gathered}}{\mathbf{subscription}(s, n, t) : \langle P, S, I, O\rangle \to \langle P, S \cup \{(s, n, t)\}, I, O\rangle}$$

  Note that $(p, n, t') \notin P$ and $(s, n, t') \notin S$ in the conditions are to ensure no two publishers or subscriptions are created with same identifier and node but different topic.

- **Publish**: For $n \in \mathcal{N}$, publish $m \in \mathcal{M}$ through publisher $p \in \mathcal{P}$:

$$\frac{\begin{gathered}\langle P, S, I, O\rangle \in \mathit{State}\\ (p, n, t) \in P, m \in \mathcal{M}, I' = \begin{cases}(p, n, t) \mapsto I_{(p',n',t')} \oplus m\\ (p', n', t') \mapsto I_{(p',n',t')}\end{cases}\end{gathered}}{\mathbf{pub}(n, p, t, m) : \langle P, S, I, O\rangle \to \langle P, S, I', O\rangle}$$

- **Recieve**: For $n \in N$, recieve $m \in \mathcal{M}$ from subscription $s \in S$:

$$\frac{\begin{gathered}\langle P, S, I, O\rangle \in \mathit{State}\\ (p, n, t) \in P, m \in \mathcal{M}, (m) \cdot O'_{(s,n,t)} = O_{(s,n,t)}, O' = \begin{cases}(s, n, t) \mapsto O_{(s,n,t)} \oplus m\\ (s', n', t') \mapsto O_{(s',n',t')}\end{cases}\end{gathered}}{\mathbf{recv}(n, s, t, m) : \langle P, S, I, O\rangle \to \langle P, S, I, O'\rangle}$$

3.3 Properties

For each computation step defined in the previous section, we define an state proposition that holds on and only on a state that is derived from its previous state through the specific computation.[1] e.g. $\mathit{state} \models \mathbf{pub}(n, p, m)$ i.i.f. the previous computation is $\mathbf{pub}(n, p, m)$. Moreover, we

[1]Note that such propositions are not necessarily well defined as functions $\mathit{State} \to \mathit{Bool}$ with our current definition of State, and therefore our assumptions here are not legal. However, it could be easily resolved w.l.o.g. by a tiny extension on the state definition to keep track on the previous computation.

extend such propositions with argument place-holder _ to indicate existence of any argument so to hold. e.g. $\mathbf{pub}(n, p, \_)$ holds if $\mathbf{pub}(n, p, m)$ holds for any m. Those said, we can now define the desired properties for a ROS2 system.

- **liveliness**[2]: Every publisher always eventually publishes, and every subscription always eventually receives. That is, for all $p \in \mathcal{P}$ and $n \in \mathcal{N}$.

$$\Box(\mathbf{ispublisher}(p, n, t) \to \diamond\mathbf{pub}(p, n, t, \_))$$

$$\Box(\mathbf{issubscription}(s, n, t) \to \diamond\mathbf{recv}(s, n, t, \_))$$

Note: Above are what we ASSUME, and below are what we can actually PROVE

- **liveness**(Eventual Delivery): Every message published will eventually be received. That is, for any $n, n' \in \mathcal{N}, p \in \mathcal{P}, s \in \mathcal{S}, t \in \mathcal{T}, m \in \mathcal{M}$

$$\mathbf{pub}(n, p, t, m) \wedge \mathbf{issubscription}(s, n', t) \Rightarrow \diamond\mathbf{recv}(n', s, t, m)$$

  As supposed to, it does not exclude the case that a subscription created after the publication still receive the message.

- **FIFO Order**: Message published later is received later. i.e. before a previous published message m is received, every publish implies a corresponding receive after m is recieved. That is, for any $n, n', n'' \in \mathcal{N}, p, p' \in \mathcal{P}, s \in \mathcal{S}, t \in \mathcal{T}, m_1, m_2 \in \mathcal{M}$

$$\mathbf{pub}(n, p, t, m_1) \wedge \mathbf{issubscription}(s, n'', t) \to$$
$$\mathbf{recv}(n'', s, t, m_1)\mathcal{R}(\mathbf{pub}(n', p', t, m_2) \to \diamond(\mathbf{recv}(n'', s, t, m_1) \to \diamond\mathbf{recv}(n'', s, t, m_2))$$

4 Towards an Object Oriented Configuration

4.1 Interaction Interface

We provide the following `Msgs` as an interface to interact with ROS2:

```
sort Ros2Msg. subsort Ros2Msg < Msg .
msg createPublisher : Oid Oid String → Ros2Msg
msg createSubscription : Oid Oid String → Ros2Msg
msg publish : Oid Oid Raw → Ros2Msg .
msg recieve : Oid Oid → Ros2Msg
op ROS2 : → Oid .
```

Listing 4: Ros2Msg

where each message serves as a trigger for the corresponding computation rule defined in 3.2. For example, `createPublisher(ROS2,n,t)` triggers (n, t), and `publish(publisher,n,m)` triggers $\mathbf{pub}(n, p, t, m)$, where t is uniquely decided by publisher and node for how (n, t) is designed. Upon success of or unexpected error during handling a request, ROS2 create a respond to indicate the result, as defined below:

```
msg createdPublisher : Oid Oid String PublisherOid → Ros2Msg .
msg createdSubscription : Oid Oid Oid → Ros2Msg .
msg published : Oid Oid →  Ros2Msg .
msg recieved : Oid Oid Raw → Ros2Msg .
msg ros2Error : Ros2Msg → Ros2Msg .
op ROS2 : → Oid .
```

Listing 5: Pub/Sub Classes

[2]terminology comes from ROS2 Qos setting. Ref to `https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Quality-of-Service-Settings.html`

4.2 A ROS2 simulator for formal verification

In Maude, A ROS2 router is simulated with the following rewrite rules:

- **States**: The state of a ROS2 router is simulated with a virtually centralized portal

```
class ROS2Portal | publisherCount : Int, subscriptionCount : Int,
    subscriptions : Map{String,List{Oid}} .
```

Listing 6: Portal Class

and several publisher and subscription objects

```
class Publisher | topic : String, dataType : MsgType, node : Oid,
    queue : List{RawTRIV}, size : Int .
class Subscription | topic : String, dataType : MsgType, node :
    Oid, queue : List{RawTRIV}, size : Int .
```

Listing 7: Pub/Sub Classes

Attribute size is to model the Qos Setting of ROS2 that enforce a maximum size on the publisher/subscription's message queue, discarding the oldest message enqueued.

- **Publisher Creation**:

```
    rl [createPublisher] :
    < ROS2 : ROS2Portal | publisherCount : COUNT >
    createPublisher(ROS2,FROM,DT,TOPIC,SIZE)
=>
    < ROS2 : ROS2Portal | publisherCount : s(COUNT) >
    < publisher(COUNT,DT) : Publisher | topic : TOPIC, dataType :
        DT, node : FROM, queue : noRaw, size : SIZE >
    createdPublisher(FROM,ROS2,TOPIC,publisher(COUNT,DT))
```

Listing 8: Publisher Creation

We introduce an name counter in the portal state to secure the $(p, n, t') \notin P$ condition in the **Create Publisher** computation rule in 3.2.

- **Subscription Creation**:

```
rl [createSubscription] :
    < ROS2 : ROS2Portal | subscriptionCount : COUNT, subscriptions
       : TOPIC2SUBS >
    createSubscription(ROS2,FROM,DT,TOPIC,SIZE)
=>
    if $hasMapping(TOPIC2SUBS,TOPIC) then
        < ROS2 : ROS2Portal | subscriptionCount : s(COUNT),
           subscriptions : insert(TOPIC,(TOPIC2SUBS[TOPIC]
           subscription(COUNT,DT)),TOPIC2SUBS) >
    else
        < ROS2 : ROS2Portal | subscriptionCount : s(COUNT),
           subscriptions :
           insert(TOPIC,(subscription(COUNT,DT)),TOPIC2SUBS) >
    fi
    < subscription(COUNT,DT) : Subscription | topic : TOPIC,
        dataType : DT, node : FROM, queue : noRaw , size : SIZE >
    createdSubscription(FROM,ROS2,subscription(COUNT,DT))
```

Listing 9: Subscription Creation

Note that subscriptions are registered on the topic upon creation.

- **Message Issuing**: A publisher with nonempty queue will spontaneously trigger the **Issue** computation. To issue a message, the publisher dequeue the message from its message queue and create a `issue` to the portal. Upon receiving the `issue`, the portal create a `send` to all subscriptions that have been registered under the target topic. Finally, the subscriptions involved will enqueue the message into their message queue, to be receive.

```
msg issue : Oid Oid String Raw → Msg .
msg send : List{Oid} Oid Raw → Msg .
msg send : Oid Oid Raw → Msg .
ceq send(SUB SUBS,FROM,RAW)
= send(SUB,FROM,RAW) send(SUBS,FROM,RAW)
if SUBS =/= noOid .
eq send(noOid,FROM,RAW)
= none .
rl [issue] :
    < ME : Publisher | topic : TOPIC, queue : RAW RB >
⇒
    < ME : Publisher | topic : TOPIC, queue : RB >
    issue(ROS2,ME,TOPIC,RAW)
.

rl [issue] :
    < ROS2 : ROS2Portal | subscriptions : (TOPIC |→ SUBS,
        TOPIC2SUBS)>
    issue(ROS2,FROM,TOPIC,RAW)
⇒
    < ROS2 : ROS2Portal | >
    send(SUBS,ROS2,RAW)
.

rl [issue] :
    < ME : Subscription | dataType : DT, queue : B, size : SIZE >
    send(ME,FROM,RAW)
⇒
    if size(B) < SIZE then
        < ME : Subscription | queue : B RAW >
    else
        < ME : Subscription | queue : tail(B) RAW >
    fi
.
```

Listing 10: Message Issuing

Those defined, rewrite rules for `publish` and `receive` are pretty straightforward:

- **Publish/Receive**:

```
rl [publish] :
    < ME : Publisher | dataType : DT, node : FROM, queue : RB,
        size : SIZE >
    publish(ME,FROM,DATA)
⇒
    if size(RB) < SIZE then
        < ME : Publisher | dataType : DT, node : FROM, queue : RB
            DATA >
    else
        < ME : Publisher | dataType : DT, node : FROM, queue :
            tail(RB) DATA >
    fi
    published(FROM,ME)
.

rl [recieve] :
    < ME : Subscription | node : FROM, queue : DATA B >
    recieve(ME,FROM)
=>
    < ME : Subscription | node : FROM, queue : B >
    recieved(FROM,ME,DATA)
.
```

Listing 11: Publish/Receive

4.3 Why This Model Matters

- **Decoupling**: Publishers/subscribers operate independently. Enabling executing different nodes in different maude sessions on even different machines, which match the need of our next step – running Maude code as distributed ROS components.
- **Verification**: Maude rules enable checking safety/liveness (e.g., LTL properties). With this model, one can verify distributed systems as a whole in a Maude configuration.

5 Architecture

We use the Maude Python Bridge and `rclpy` to bring our previous simulated ROS2 router to implementation. First in the interaction interface defined previously, let's add the nessesary `specialhook` meta data to the `Msg` definitions:

```
    msg createPublisher : ROS2Oid Oid MsgType String Int -> Ros2Msg
    [ctor special (
            id-hook SpecialHubSymbol (roshook)
            op-hook publisher (publisher : Nat MsgType ~> PublisherOid)
            op-hook createdPublisher (createdPublisher : Oid ROS2Oid
               String PublisherOid ~> Msg)
            op-hook rosType (rosType : MsgType ~> String)
    )].
    msg createSubscription : ROS2Oid Oid MsgType String Int -> Ros2Msg
    [ctor special (
            id-hook SpecialHubSymbol (roshook)
            op-hook subscription (subscription : Nat MsgType ~>
               SubscriptionOid)
            op-hook createdSubscription (createdSubscription : Oid Oid
               SubscriptionOid ~> Msg)
            op-hook rosType (rosType : MsgType ~> String)
    )].
    msg publish : PublisherOid Oid Raw -> Ros2Msg
    [ctor special (
            id-hook SpecialHubSymbol (roshook)
            op-hook published (published : Oid PublisherOid ~> Msg)
            op-hook typecheck (typecheck : MsgType  Data ~> Bool)
            op-hook cat (_,_ : Raw Raw ~> Raw)
            op-hook mapping (_|->_ : String RawOrSimple ~> Raw)
            term-hook true (true)
    )].
    msg recieve : SubscriptionOid Oid -> Ros2Msg
    [ctor special (
            id-hook SpecialHubSymbol (roshook)
            op-hook recieved (recieved : Oid SubscriptionOid Raw ~> Msg)
            op-hook rosType (rosType : MsgType ~> String)
            op-hook cat (_,_ : Raw Raw ~> Raw)
            op-hook mapping (_|->_ : String RawOrSimple ~> Raw)
   )].
```

Listing 12: interface with special hook

And in the python side, we have

Listing 13: special hook implementation

```python
class RosMaudeNode(Node):
    def run(self, term:maude.Term, data:maude.HookData):
        reply = None
        if symbol == 'createPublisher':
            ...
        elif symbol == 'createSubscription':
            ...
        elif symbol == 'publish':
            ...
        elif symbol == 'recieve':
            ...
        return reply
```

In this report, we will not go deep into the python code – In short, we call the corresponding ROS2 api in python for each symbol, and set `reply` to the corresponding respond.

6 Case Study: Talker-Listener

6.1 Overview

In this case study, we model a very simple distributed system where we have 1 producer and 2 consumers. We expect that each consumer get an identical copy of each message being produced, and stop receiving upon seeing a termination word "<DONE>".

```
pr ROS2 .
pr ROSMAUDE#MSG#STRING * (op type to str) .
pr STRING .
pr LIST{String} * (op nil to noString) .

class Talker | publisher : [Oid], words : List{String} .
class Listener | subscription : [Oid], got : List{String} .

op na : -> [Oid] .
ops t a1 a2 : -> Oid .
op init : -> Configuration .
eq init =
    < t : Talker | publisher : na, words : noString "HI" "Im T" "Hello
       World" "<DONE>" >
    < a1 : Listener | subscription : na, got : noString >
    < a2 : Listener | subscription : na, got : noString >
    createPublisher(ROS2,t,str,"/speech",3)
    createSubscription(ROS2,a1,str, "/speech",5)
    createSubscription(ROS2,a2,str, "/speech",5) .
```

Listing 14: Talker Listener Initial State

In order to interact with ROS2, one have to provide a `MsgType`(message type) which decides the what sort of terms for the publisher to take and the subscription to receive. To get access to a `MsgType` constant that enables sending Maude String through ROS2 *std_msgs/msg/String* interface, We include `ROSMAUDE#MSG#STRING` in protection mode and rename `op type` to `str`. To create a publisher, we send to `ROS` a `createPublisher` msg including the `MsgType`, the topic name(a `String`), and the maximum size of its message queue(an `Nat`).

```
    rl [initTalker] :
        < ME : Talker | publisher : na, words : STR WORDS >
        createdPublisher(ME, ROS2, NEW)
    =>
        < ME : Talker | publisher : NEW, words : WORDS >
        publish(NEW,ME,[STR]) .

    rl [initListener] :
        < ME : Listener | subscription : na >
        createdSubscription(ME, ROS2, NEW)
    =>
        < ME : Listener | subscription : NEW >
        recieve(NEW, ME) .
```

Listing 15: Talker Listener Initialization

Upon a successful publisher/subription creation, `ROS2` will respond with the new object's `Oid`. And that's when we might want to store it somewhere for later use.

```
    rl [talk] :
        < ME : Talker | publisher : O, words : STR WORDS >
        published(ME,O)
    =>
        < ME : Talker | publisher : O, words : WORDS >
        publish(O,ME,[STR]) .

    rl [listen] :
        < ME : Listener | got : WORDS >
        recieved(ME, FROM, [STR])
    =>
        if STR == "<DONE>" then
            < ME : Listener | got : WORDS >
        else
            < ME : Listener | got : WORDS STR >
            recieve(FROM, ME)
        fi .
```

Listing 16: Talker Listener Communication

Once initialized, we code the talker to keep producing words to the `"/speech"` topic and each listener will receive a copy of each of the words through their subscriptions.

6.2 formal verification

To simulate this distributed system with our logical model for ROS2, we can do `rew init <ROS2-Logical>` in maude cli after loading the above configuration[3]. And the result we got is

```
   < ROS2 : ROS2Portal | publisherCount : 1, subscriptionCount : 2, subscriptions
: "/speech" |-> subscription(0, str) subscription(1, str) >
< t : STalker | publisher : publisher(0, str), words : noString >
< a1 : SListener | subscription : subscription(0, str), got : ("Im T" "Hello World")
```

[3]check out "rosmaude/test_ros.maude" in our repo

```
>
< a2 :  SListener | subscription :  subscription(1, str), got :  ("Im T" "Hello World")
>
< publisher(0, str) :  Publisher | topic :  "/speech", dataType :  str, node :  t,
queue :  noRaw, size :  3 >
< subscription(0, str) :  Subscription | topic :  "/speech", dataType :  str, node
:  a1, queue :  noRaw, size :  5 >
< subscription(1, str) :  Subscription | topic :  "/speech", dataType :  str, node
:  a2, queue :  noRaw, size :  5 >
```

Through simulation, we easily observed a potential concern of this system design: since ROS2 is not responsible and not able to check if the targeted listeners are online when the publisher start to talk, we were not able to ensure delivery of all messages(note that the listeners do not receive the starting message "HI"). This is not a problem of our ROS2 nor Maude, but a problem of the Talker-Listener system itself.

7 deployment

After showing how we could simulated a ROS base system design with our `<ROS2-Logical>` model, let put the system online so it can make actually communication with real world ROS node. If the system do not depend on other custom special hooks, the easiest way to do this run the maude code as a real ROS2 system is to use our `run.py` script[4]. For example, to run the Talker-Listener system, one only need to do

```
$ python3 run.py test_ros.py
...
erew [80] results in: < t : Talker | publisher : publisher(0, str),
    words : noString > < a1 : Listener | subscription : subscription(0,
    str), got : ("HI" "Im␣T" "Hello␣World") > < a2 : Listener |
    subscription : subscription(1, str), got : ("HI" "Im␣T" "Hello␣World"
    ) >
```

To check that the communication is performed through ROS2, one can do `ros2 topic echo` in another bash session before running the pythons script, and will see output like in 7

```
$ ros2 topic echo /speech std_msgs/msg/String
data: HI
---
data: Im T
---
data: Hello World
---
data: <DONE>
---
```

[4]rosmaude/run.py in our repo

8 The D Transformation

8.1 Overview

We define the transformation family $D_d : M \mapsto M_{D_d}$, mapping a Maude model M of a distributed system to a distributed RosMaude program M_{D_d} deployed on different machines. Multiple concurrent Maude sessions may run on the same machine. The transformation D takes as input:

- an object-oriented Maude module M defining an actor system (see below);
- an initial state `init` of sort `Configuration`, which is a set of objects $< o_1 : C_1|att_1 > ... < o_n : C_n|att_n >$ with distinct names o_i ;
- a distribution information function $d : o_1, ..., o_n \to \mathcal{N}$ assigning to each (top-level) object o_i in `init` a node identifier.

The transformation D then gives us:

- A RosMaude program M_{D_d} that runs on each distributed Maude session; and
- an initial state $\texttt{init}_{D_d}(n)$ for each node identifier $n \in \mathcal{N}$.

The object-oriented module M should model an "actor" system, which reuires that its rewrite rules must have the form

```
(to o from o' :  mc) < o :  C |...  > => < o :  C |...  > msgs [if ...]   (†)
```

or

```
< o :  C |...  > => < o :  C |...  > msgs [if ...]   (‡)
```

where msgs is a term of sort Configuration in form of a multiset of messages like

$$(\texttt{to}\ \texttt{o}_1\ \texttt{from}\ \texttt{o}:\ \texttt{mc}_1)...(\texttt{to}\ \texttt{o}_k\ \texttt{from}\ \texttt{o}:\ \texttt{mc}_k)$$

To be explicit, our goal here is to define a extended model on the original Maude model that simulates the two kinds of rule where each computational step requiring only the local state, without relying on Maude's commutativity and associativity axioms.

8.2 Model Definition

Given a set of `Node` in the form of `node(n,conf,p,s)` where $n \in \mathcal{N}$ is a node identifier, p a publisher, and s a subscription, we model a distributed system with the following state definition

$$\langle \{\texttt{node}_n\}_{n\in\mathcal{N}}, \texttt{ros}\rangle = \langle \{\texttt{node}(n, \texttt{conf}_n, \texttt{p}_n, \texttt{s}_n)\}_{n\in\mathcal{N}}, \texttt{ros}\rangle$$

where $\texttt{ros} = \langle P, S, I, O\rangle$ is as introduced in 3.1, and computational rules

$$\frac{\begin{array}{c} n \in \mathcal{N}, \texttt{node}_m = \texttt{node}(m, \texttt{conf}_m, \texttt{p}_m, \texttt{s}_m) : \texttt{Node}\ \text{for}\ m \in \mathcal{N}, \texttt{ros} : \mathit{State}, \texttt{conf}'_n : \texttt{Configuration}, \\ \texttt{conf}_n = \texttt{conf}'_n \cdot \texttt{to t from o : mc} \\ \texttt{t}\ \text{not in}\ \texttt{conf}_n \\ \textbf{publish}(n, \texttt{p}_n, \texttt{topic}, \texttt{to t from o : mc}) : \texttt{ros} \to \texttt{ros}' \end{array}}{\textbf{send}(n, \texttt{to t from o : mc}) : \langle \{\texttt{node}_m\}_{m\in\mathcal{N}}, \texttt{ros}\rangle \to \langle \left\{ \begin{cases} \texttt{node}\ \text{if}\ m \neq n \\ \texttt{node}(n, \texttt{conf}'_n, \texttt{p}_n, \texttt{s}_n)\ \text{if}\ m = n \end{cases} \right\}_{m\in\mathcal{N}}, \texttt{ros}'\rangle}$$

and

$$\frac{\begin{array}{c} m \in \mathcal{N}, \mathtt{node}_m = \mathtt{node}(m, \mathtt{conf}_m, \mathtt{p}_m, \mathtt{s}_m) : \mathtt{Node} \text{ for } m \in \mathcal{N}, \mathtt{ros} : \mathit{State}, \mathtt{conf}'_m : \mathtt{Configuration}, \\ \mathbf{recv}(n, \mathtt{p}_n, \mathtt{topic}, \mathtt{to\ t\ from\ o :\ mc}) : \mathtt{ros} \to \mathtt{ros}' \\ \mathtt{t\ in\ conf}_n \\ \mathtt{conf}'_n = \mathtt{conf}_n \cdot \mathtt{to\ t\ from\ o :\ mc} \end{array}}{\mathbf{consume}(n, \mathtt{to\ t\ from\ o :\ mc}) : \langle \{\mathtt{node}_m\}_{m \in \mathbb{N}}, \mathtt{ros} \rangle \to \langle \left\{ \begin{cases} \mathtt{node} \text{ if } m \neq n \\ \mathtt{node}(n, \mathtt{conf}'_n, \mathtt{p}_n, \mathtt{s}_n) \text{ if } m = n \end{cases} \right\}_{m \in \mathcal{N}}, \mathtt{ros}' \rangle}$$

where $\mathtt{conf} \cdot \mathtt{msg}$ stands for "append `msg` to `conf`". Under this definition, the model

$$\langle \{\mathtt{node}(m, \mathtt{conf}_i, \mathtt{p}_i, \mathtt{s}_i)\}_{m \in \mathcal{N}}, \langle \{\mathtt{p}_m\}_{m \in \mathcal{N}}, \{\mathtt{s}_m\}_{m \in \mathcal{N}}, \varepsilon, \varepsilon \rangle \rangle$$

simulates

$$\bigoplus_{m \in \mathcal{N}} \mathtt{conf}_m$$

where $\bigoplus$ is aggregation over concatenation $\oplus$(or `_,_` in Maude). Proof is pending.

8.3 Distributed Initial State

Given the original initial state `init`, we extract the necessary information for initializing the initial state as a tuple of a node identifier and a sub-configuration.

$$(n, \mathtt{conf}_n) = (n, \mathtt{init} \cap (\{\mathtt{obj} : \mathtt{Obj} \mid \mathtt{d}(\mathtt{obj}) = \mathtt{n}\} \cup \{(\mathtt{to\ t\ from\ f} : \mathtt{mc}) : \mathtt{Msg} \mid \mathtt{d}(\mathtt{f}) = \mathtt{n}\}))$$

so that $\mathtt{conf}_n$ is the part of the original configuration that belongs to node `n` as configured with `d`.

```
omod NODE is
    pr ROSMAUDE#MSG#TERM{DMsg} * (op type to msg) .
    ...
    class Node | $ : Configuration, publisher : Maybe{Oid},
        subscription : Maybe{Oid}, ups : Set{Oid}, waiting : Set{Oid} .
    rl [init] :
        O < CONF >
    =>
        < O : Node | $ : CONF, publisher : null, subscription : null,
            ups : empty, waiting : empty >
        createSubscription(ROS2,O,msg,"/d",queueSize(0))
        createPublisher(ROS2,O,msg,"/d",queueSize(0)) .
endom
```

Listing 17: Node Class

Given a tuple `(n,conf)` of a node identifier and a configuration, we initialize a `Node` object `node(n) < conf >` with rule `[init]` as defined in 17, where `msg` is a provided ROSMaude message interface which allows sending Maude terms as ROS strings. This way, we wrap each sub-configuration in an object, and effectively cut off access from the inside to any Maude message outside. This ensures atomicity of each distributed sub-configuration even when multiple of them run on the same machine and the same maude session(e.g. in verification stage), and to make life easier when we try to define a global transformation on a distributed sub-configuration.

Such initialization of nodes can happen under logical setting(with a `<ROS2-Logical>` portal that simulates actual ROS communication for verification) or actual setting(`erew` the nodes with our `rosmaude` special hook connected).

Upon successful initialization of a node, we obtain our desired initial state

$$\mathtt{init}_{D_d(n)} = \mathtt{node}(n, \mathtt{conf}_n, \mathtt{p}_n, \mathtt{s}_n)$$

8.4 Alive-ness and Greeting Mechanism

However, correctness is compromised during the initialization stage of the nodes, to be explicit, when some node is initialized and tries to send a outgoing `msg` to an object in another node for whom ROSMaude has not yet created a subscription on `"/d"` topic – a case whrere the outgoing `msg` is properly published as a ROS message, but may not reach to the node holding the targeted object. To eliminate this concern, we introduce a greeting mechanism to ensure any message delivery can happen only after the targeted node first appears online. More formaly, we describe the mechanism with the following rule

1. node publish a greeting when coming online,
2. node checks its book of other living nodes for the target node before sending the message,
3. node pings the target node if it does not find it in its book,
4. node register the sender node when seeing any messages published, and
5. node replies to a ping with an ack.

9 Conclusion

10 Conclusion

The ROS-Maude Bridge successfully integrates formal verification into ROS2-based robotic systems, enabling rigorous analysis of critical properties such as message delivery guarantees and FIFO order while maintaining compatibility with real-world deployments. By modeling ROS2 publishers, subscribers, and topics as Maude objects, the bridge allows developers to leverage Maude's formal methods—including simulation, reachability analysis, and LTL model checking—without modifying existing ROS2 components. The case study of the Talker-Listener system demonstrated the practical viability of this approach, both in simulation and deployment, showcasing how formal verification can uncover subtle issues like message loss during initialization.

A key strength of the bridge lies in its support for distributed systems through the D-transformation, which partitions Maude configurations across nodes while preserving formal guarantees. The greeting mechanism ensures reliable communication during initialization, addressing challenges in distributed coordination. This scalability is critical for real-world applications, where robotic systems often span multiple hardware components.

While the current implementation focuses on the publish-subscribe model, future work could extend the bridge to support ROS2 services, actions, and Quality of Service (QoS) policies. Additionally, optimizing the integration for real-time performance and expanding the range of verifiable properties (e.g., fault tolerance under node failures) would further enhance its utility. By bridging formal methods and robotic middleware, this work lays a foundation for building safer, more reliable distributed robotic systems through mathematically grounded design and analysis. Code: `https://github.com/rjsun06/rosmaude`