Challenge Problem 3: ROS1 to ROS2 Conversion

1. Approach Overview

We have employed an architecture-centric approach for the migration of a ROS1 system to ROS2. In our approach, we integrate the architecture description of a ROS system with its implementation. The architecture description specifies the nodes (called components in the architecture specification) that are present in the system and the mechanisms (called connectors in the architecture specification) by which they interact with each other (topics, services, etc.).

The architecture description is used to abstract away from specific ROS versions. In particular, the nodes are implemented without the use of ROS-specific communication channels (topics, services, etc.). Instead, the nodes are programmed to interact using abstract interfaces (called ports). The architecture is responsible for facilitating the reification of the port interfaces by the establishment of appropriate communication channels between the nodes. The architecture description specifies which ports of which nodes communicate using which connectors. We use metaprogramming to generate the code for instantiating communication channels between nodes based on the specification of connectors in the architecture description.

We use Wyvern, a programming language that we have designed, for specifying the architecture description. Additionally, we provide abstractions of ROS library functionality – both communication-related (e.g., topics, services, etc.) and non-communication-related (e.g., angles, time, etc.) – in Wyvern. A ROS1 system implemented in Wyvern using its architectural abstractions as well as ROS library abstractions can be easily converted to ROS2 by only making a small number of changes in the architecture description of the system. These changes involve changing the types of the connectors from those provided by ROS1 to those provided by ROS2.

To migrate a ROS1 system implemented in C++ to ROS2 using our approach, it must first be converted to Wyvern to use our abstractions. This process involves three steps:

1. Extracting the architecture specification of the system
2. Transforming the C++ code to replace usages of ROS functionality with Wyvern's abstractions
3. Writing wrappers in Wyvern that integrate the architecture specification with the transformed C++ code

Each of these steps is described in detail below.

1.1 Architecture Discovery

As the first step in converting a ROS1 system to Wyvern, its architecture must be extracted. We have provided a tool named rosdiscover for this purpose. Rosdiscover performs a static analysis of the ROS1 system's source code and launch files to determine the nodes in the system and the mechanisms by which they communicate with each other. It produces a YAML file that contains this information. We have provided a utility that converts this output YAML file to Wyvern architecture specifications.

1.2 C++ Code Transformation

As mentioned above, the use of ROS library functionality – both communication-related and non-communication-related – needs to be abstracted out from the implementation of the nodes. To this end, the C++ code that implements the nodes in the ROS1 system has to be transformed to replace calls into the ROS library with uses of the abstractions provided by Wyvern. In principle, this step can be automated by scanning the C++ codebase for patterns of ROS library usage and replacing them with corresponding uses of Wyvern's abstractions. We have shown how this may be done for the code for the TurtleBot3 Teleop system (described in section 2 below). We have provided a script that transforms the TurtleBot3 Teleop system's codebase to use Wyvern's abstractions. We use the Comby tool (https://comby.dev) to carry out the transformation. This script, however, is tailored to the TurtleBot3 Teleop example and is not amenable to general use. This limitation is due to the fact that Comby can only analyze code structurally and does not provide any semantic information (e.g., which namespace is this identifier declared in?). Because of this, the ROS library usage patterns encoded in the script use simplifying assumptions about ROS library calls which limits the script's general use.

1.3 Wyvern Wrapper Implementation

Wyvern's architecture specification language is not designed to work with C++ code directly. This is because one of Wyvern's design goals is architectural control i.e., the enforcement of the architecture in the implementation. The use of C++ as an implementation language is not in congruity with this goal as C++ has features that can be used to bypass architectural restrictions. As a result, in order to integrate the transformed C++ code with the architecture specification, wrappers must be written in Wyvern for the nodes implemented by the C++ code. These wrappers act as the glue between the architecture specification and the C++ node implementations. They facilitate interoperability between the Wyvern code generated for the connectors specified in the architecture description and the C++ node implementations.

Implementation of the ROS Abstraction Layer

As previously mentioned, Wyvern provides abstractions for ROS library functionality (both communication-related and non-communication-related) to aid the migration of ROS systems to different ROS versions. These abstractions must be implemented for both ROS1 and ROS2. Even though this can take a large amount of effort, it needs to be done only once for every ROS version. It can then be used to migrate any ROS application between the supported ROS versions.

Conversion to ROS2

Once the ROS1 system written in C++ has been converted to Wyvern, it can be easily migrated to ROS2 by making a small number of modifications in the architecture description of the system. These modifications consist of changing the types of the connectors from the ones implementing ROS1 communication abstractions to those implementing corresponding ROS2 abstractions.

2. TurtleBot3 Teleop Example

We will demonstrate our approach for converting a ROS1 system to ROS2 on a simulation of the TurtleBot3 robot (http://emanual.robotis.com/docs/en/platform/turtlebot3/overview) performing

teleoperation. The system comprises three nodes: 1) pdw_turtlebot3_fake_node, 2) pdw_robot_state _publisher, and 3) pdw_turtlebot3_teleop_keyboard. The pdw_turtlebot3_fake_node node is the node that simulates a TurtleBot3 robot. The pdw_robot_state_publisher node is used to publish the state of the robot to the /tf topic. The pdw_turtlebot3_teleop_keyboard node is used to perform teleoperation using a keyboard.

The ROS computation graph for the TurtleBot3 teleop system as seen in rqt_graph is shown in Fig. 1. The pdw_turtlebot3_teleop_keyboard node accepts keyboard input commands from the user and translates them to velocity commands for the robot which are published on the /cmd_vel topic. The pdw_turtlebot3_fake_node node publishes the joint angles of the robot on the /joint_states topic. The pdw_robot_state_publisher node uses a kinematic tree model of the robot to compute the 3D poses of the robot links from the joint angles and publishes them on the /tf topic.

Fig. 1. ROS Computation Graph for the TurtleBot3 Teleop System

3. Instructions for Running the TurtleBot3 Teleop Example

To run the TurtleBot3 teleop example, you will need an Ubuntu machine or VM with Git and Docker installed. We have tested these instructions on Ubuntu 18.04. To install Git, run `sudo apt install git`. To install Docker, follow the instructions here: https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-18-04. The migration of the TurtleBot3 teleop system from ROS1 to ROS2 using our approach can then be seen in action by performing the following steps:

i. Clone the ros-transmogrifier repository which contains the source code for the demo.

```
$ git clone https://github.com/selvasamuel/ros-transmogrifier.git
```

ii. Change into the ros-transmogrifier/demo directory.

```
$ cd ros-transmogrifier/demo
```

iii. Now, you can start the demo by running the demo.sh script.

```
$ bash demo.sh
```

This script creates a Docker image named brass-pdw/teleop-demo and starts a container from it. This Docker image contains the source code for the TurtleBot3 teleop system transformed to work with Wyvern. During the creation of the Docker image, rosdiscover and the Comby script are run on the original source code of the system. Rosdiscover extracts the architecture description of the system (section 1.1) and the Comby script transforms the C++ source code to use Wyvern's abstractions (section 1.2). When you run this image in a Docker container, you can see the architecture description files TurtleBot3Teleop.wyc and TurtleBot3Teleop.wyd in the /root/brass_pdw/teleop-demo directory. As mentioned in

section 2, the TurtleBot3 teleop system consists of three nodes: 1) pdw_turtlebot3_fake_node, 2) pdw_robot_state_publisher, and 3) pdw_turtlebot3_teleop_keyboard. The pdw_turtlebot3 _fake_node and pdw_robot_state_publisher nodes have been originally implemented in C++. The transformed source code for these nodes can be seen in the /root/brass_pdw/teleop-demo/cpp directory in the Docker container. The Wyvern wrappers for these nodes (section 1.3) can be found in the files pdw_turtlebot3_fake_node.wyv and pdw_robot_state_publisher .wyv in the /root/brass_pdw/telelop-demo directory. The pdw_turtlebot3_teleop_keyboard node has been originally implemented in Python. We rewrote this node to convert it to Wyvern. The Wyvern code for the rewritten pdw_turtlebot3_teleop_keyboard node is in the file pdw_turtlebot3_teleop_keyboard.wyv in the /root/brass_pdw/teleop-demo directory.

iv. Run the Wyvern-converted ROS1 version of the TurtleBot3 teleop system in the Docker container.

```
# source /opt/ros/melodic/setup.bash
# bash scripts/run/run_ros1_teleop.sh
```

After the system is started, the instructions for performing teleoperation on the simulated robot will be displayed. You can make the simulated TurtleBot3 robot move by pressing the keys shown in the instructions.

v. You can now visualize the robot in RViz. To do this, first open another shell in the running Docker container by executing the following command in another terminal:

```
$ sudo docker exec -it $(sudo docker ps \
  --filter ancestor=brass-pdw/teleop-demo -q) /bin/bash
```

Please ensure that you are running only one container from the brass-pdw/teleop-demo image. Otherwise, the above commands might not work properly.

Now, you can start RViz by running the following commands in the new shell:

```
# source /opt/ros/melodic/setup.bash
# bash scripts/run/visualization/ros1_visualization.sh
```

vi. To migrate the TurtleBot3 teleop system to ROS2, first of all terminate the running ROS1 version of the system. Close the RViz window and press Ctrl+C in the terminal in which you started the system in step iii.

vii. Now, the architecture description file must be modified to change the type of connectors from those implementing ROS1 communication mechanisms to those implementing ROS2 communication mechanisms. To do this, open the file TurtleBot3Teleop.wyc in Vim.

```
# vi TurtleBot3Teleop.wyc
```

Make the changes shown in the following table in the TurtleBot3Teleop.wyc file.

Line Number	From	To
18	`connector ROS1Topic`	`connector ROS2Topic`

21	`connector ROS1Tf`	`connector ROS2Tf`
31	`ROS1Topic cmd_vel, joint_states, odom`	`ROS2Topic cmd_vel, joint_states, odom`
32	`ROS1Tf tf, tf_static`	`ROS2Tf tf, tf_static`

Save and close the file.

viii. Compile the ROS2 version of the system.

```
# bash scripts/compile/wyvern/compile_for_ros2.sh
```

ix. Run the ROS2 version of the system.

```
# source /opt/ros/dashing/setup.bash
# bash scripts/run/run_ros2_teleop.sh
```

x. Visualize the robot in RViz by running the following commands in the other shell running in the Docker container:

```
# source /opt/ros/dashing/setup.bash
# bash scripts/run/visualization/ros2_visualization.sh
```