Overview of ROS mobile arm application sample using Behavior Tree and ros_actor

1. Application Overview

The sample application is stored in:

https://github.com/momoiorg-repository/actor_demo

It operates a Turtlebot equipped with OpenManipulator (Mobile Manipulator), and performs various tasks in a Gazebo world. The task includes:

- search for Coke Cans,
- picking Coke Cans,
- transportation and placement of Coke Cans.

The Mobile Manipulator is based on Waffle_pi with RealSense added.

2. Implementation

This sample application is saved in the pytwb_ws workspace in the repository. This can be referenced from within docker as ~/pytwb_ws. Below, file names are written starting from this directory. This workspace is structured according to the specifications of pytwb, a tool for building applications based on py_trees.

The main directories involved are:

- src/cm1/cm1/behavior: Directory containing behavior implementation by Python.
- src/cm1/cm1/trees: Directory containing Behavior Tree description files written in XML.
- src/cm1/cm1/lib: Library of Python classes and functions used in Behavior implementation.
- src/cm1/cm1/lib/actor: Directory containing the ros_actor implementation files used in Behavior implementation.

3. Structural explanation based on Behavior Tree

The application is implemented by calling lower-level Trees, with the 'bt_pick_place' Behavior Tree at the top.

```
<root>
    <BehaviorTree ID="bt_pick_place">
    <Sequence name="main">
        <bt_search name="global_search"/>
```

```
        <bt_catch name="pick_action"/>
        <bt_carry name="carry_action"/>
        <ArmHome name="arm_home" />
        <Mini_Walk name="mini_walk" target="[-20]"/>
        <SetBlackboard name="set_target" key="target_pose" value="[(1.0, 0.0, 0.0)]" />
        <GoToPose name="move_to_place_loc"/>
    </Sequence>
    </BehaviorTree>
</root>
```

The Behavior Trees that make up bt_pick_place are described below.

(1) bt_search

It searches for Coke Cans with the camera as the robot moves around the Gazebo world. It implements a mechanism that automatically divides the world and allows for efficient exploration without overlooking searching targets.

```
<root>
    <BehaviorTree ID="bt_search">
    <Sequence name="main">
        <SetLocations name="set_locations"/>
        <Retry    name="find_loop" num_failures="[10]">
            <Sequence name="seek_target">
                <Parallel name="seek_and_go" policy="SuccessOnOne">
                    <LookForCoke name="look_coke"/>
                    <Sequence name="search_location">
                        <GetLocation name="get_loc"/>
                        <GoToPose name="go_to_loc"/>
                    </Sequence>
                </Parallel>
                <SetWatchLocations name="set_location" debug="True" />
                <GoToPose name="get_near"/>
            </Sequence>
        </Retry>
    </Sequence>
    </BehaviorTree>
```

</root>

The main behaviors used here are as follows.

- SetLocations

uses the vector_map library to create a list of destinations for exploration and save it to the py_trees blackboard.

- LookForCoke

detects a Coke Can from a camera image. The sample version uses OpenCV to detect red areas, but it is also possible to use NL based object detection mechanism such as YOLO.

- GoToPose

It is an interface for Nav2 and executes movement. Here, Retry and Parallel Behaviors are used to execute LookForCoke and GoToPose in parallel, and if it is not found even after reaching to the destination, the robot changes the destination and searches again.

- GetWatchLocations

If the robot see a Coke Can, calculate the coordinates of where the robot saw it and save its location on the blackboard.

(2) bt_catch

It makes the robot approach the found Coke Can, correct the position using the camera, and pick it.

```
<root>
    <BehaviorTree ID="bt_catch">
    <Sequence name="main">
        <Watch name="watch_coke"/>
        <ScheduleDestination name="schedule_final_target1"/>
        <GoToPose name="reach"/>
        <Watch name="watch_coke"/>
        <GetFoundPoint name="get_found_point"/>
        <GoToPose name="reach"/>
        <Sleep name="sleep" actor="sleep" time="[5]" />
        <Face name="face"/>
        <Open name="open"/>
        <Approach name="approach" target="[0.19]"/>
        <Adjust name="adjust"/>
        <Pick name="pick"/>
```

```
            <Fit name="fit1"/>
            <Shift name="shift" target="[0.145]"/>
            <Fit name="fit2"/>
            <Close name="close"/>
        </Sequence>
    </BehaviorTree>
</root>
```

The main behaviors used here are as follows:

- Watch

looks at the Coke Can repeatedly to get accurate coordinates.

- ScheduleDestination

uses the vectorized map provided by vector_map to determine the robot's working position for picking work.

- Face

rotates the robot so that it is facing the Coke Can precisely.

- Open

opens the gripper.

- Approach

moves the robot to a designated distance to an object by using visual feedback.

- Adjust

corrects the arm angle using camera image information.

- Pick

makes the arm perform a picking motion.

- Fit

corrects the arm angle using camera image information. Since the arm is already down and interferes with the camera field of view, the Coke Can position is detected using a different method than Adjust.

- Shift

detects the distance to the Coke Can and moves to an appropriate position for picking.

- Close

closes the gripper to complete picking.

(3) bt_carry

It makes the robot transport and place the picked Coke Cans on a pallet.

```
<root>
    <BehaviorTree ID="bt_carry">
    <Sequence name="main">
        <ArmHome name="arm_home" />
        <SetBlackboard name="set_target" key="target_pose" value="[(1.0, 3.0, 1.57)]" />
        <GoToPose name="move_to_place_loc"/>
        <Approach name="approach" target="[0.13]" />
        <Place name="place_action"/>
    </Sequence>
    </BehaviorTree>
</root>
```

4. Structure description based on ros_actor

All of the above Behavior Trees are implemented using actors. The actor of ros_actor is a normal Python callable and must be defined as a SubSystem or SubNet method. A defined actor can be called by the run_actor method.

For example, the 'home' actor in src/cm1/cm1/lib/actor/manipulator.py is

```
class ManipulatorNetwork(SubNet):
    @actor
    def home(self):
        return self.run_actor('move_joint', 0.0, 0.0, 0.0, 0.0)
```

The actor has the function of returning the arm to its home position. The '@actor' notation allows it to be called from another actor by calling self.run_actor('home'). You can also run it by typing 'home' from the command prompt.

Furthermore, if you call it as

```
self.run_actor_mode('home', 'async', callback)
```

from another actor, it will be an asynchronous call, the call itself will end immediately, and the callback function will be called when the arm operation is finished.

Conversely, it is also possible to make a function that operates asynchronously operate synchronously using run_actor, in which case, it is defined as

@actor(<name>, 'async')

```
def <func>(self, callback, …):
```

The actor allows for both synchronous and asynchronous calls.
Furthermore, ROS publishers, subscribers, and actions can also be defined as actors and called in the same way.

The actors that implement the main functions are listed below. Only major ones are listed, because it is difficult to introduce all the actors implemented in the sample code.

(1) RealSense image processing

The subscribers that receive image information and depth information from RealSense are defined in src/cm1/cm1/lib/actor/system.py as 'pic' actor and 'depth' actor, respectively.

```
class Tb3CameraSystem(SubSystem):
    def __init__(self, name, parent):
        super().__init__(name, parent)
        ---
        self.register_subscriber('pic',Image,"/intel_realsense_r200_depth/image_raw",10)
        self.register_subscriber('depth',Image,"/intel_realsense_r200_depth/depth/image_
        raw",10)
```

The actor that uses ‘pic’ is defined in src/cm1/cm1/lib/actor/cognitive.py.

- pic_receiver actor

```
    @actor('pic_receiver', 'multi')
    def pic_receiver(self, callback):
        def stub(data):
            cv_image = None
            cv_bridge = self.get_value('cv_bridge')
            try:
                cv_image = cv_bridge.imgmsg_to_cv2(data, "bgr8")
            except CvBridgeError as e:
                print(e)
            return callback(cv_image)
        pic_tran = self.run_actor_mode('pic', 'multi', stub)
        return ('close', lambda tran: pic_tran.close(pic_tran)),
```

This is an actor that converts image data received using pic actor into OpenCV format. This is a 'multi' type which allows for asynchronous operation where the callback may be called multiple times.

・ pic_find

This actor searches for Coke Cans from the received image data.

```
@actor
def pic_find(self):
    ret = None
    with self.run_actor_mode('pic_receiver', 'timed_iterator', 10) as pic_iter:
        for cv_image in pic_iter:
            ret = find_coke(cv_image)
            if ret: break
    return ret
```

A ‘multi’ type actor can be called in the form of an iterator. The description

self.run_actor_mode('pic_receiver', 'timed_iterator', 10)

is the calling part and returns an iterator. This is the ‘timed_iterator’ type actor invocation, and it operates as an iterator that automatically ends after 10 seconds.

・ find_object

This is an actor who attempts to find the exact coordinates of a Coke Can by repeating the process multiple times to calculate the coordinates from the location of the Coke Can in the image found using pic_find.

・ measure_center

This is an actor that calculates the center coordinate of a Coke Can by acquiring depth information using a ‘depth’ actor.

・ object_loc

This is an actor that uses find_object to detect the position of the Coke Can, and uses TF to transform the coordinates to calculate the arm angle to face the Coke Can.

5. Behavior implementation

The Behavior used in the sample code mentioned in 3. is implemented using actors. For example, the Open Behavior that opens the gripper is

src/cm1/cm1/behavior/manipulator.py

```
@behavior
class Open(ActorBT):
    desc = 'gripper open'

    def __init__(self, name, node):
        super().__init__(name, 'open')
```

This implementation is completed by simply specifying the open actor to be used for Open Behavior implementation. Inside the base class ActorBt, the structure is such that open actor is called asynchronously and the value of the callback is checked using the Behavior's update method.

Other Behaviors have almost the same structure, and implementation is completed just by calling the actor. The correspondence between actor and Behavior looks like this:

- LookForCoke Behavior: Implemented by object_glance actor and map_trans actor that receives TF.
- GoToPose Behavior: Navigate actor
- Watch Behavior: get_found actor
- Face Behavior: face actor
- Open Behavior: open actor
- Approach Behavior: approach actor
- Adjust Behavior: ad actor
- Pick Behavior: open, pick actor
- Fit Behavior: fit actor
- Shift Behavior: shift actor
- Close Behavior: close actor

In this way, actors are used directly to implement Behaviors in an almost one-to-one correspondence.