Proposal of next-generation ROS application architecture for ease of development

momoi.org
https://momoi.org/

1. Introduction

The field of ROS application is expanding, and the structure of its applications is becoming increasingly complex. Furthermore, there is a need to collaborate with the results of deep learning technology such as LLM. As a result, opportunities are increasing for not only robot experts but also experts in fields other than robotics, such as AI and business applications, to create robot applications. The main purpose of the proposal is for experts in other fields and students to build ROS programs in a short period of time.

With the advent of deep learning technology, advanced algorithms in fields such as image processing and reinforcement learning have been developed, and there has been a huge demand for making it easy to incorporate them into robotics area. Furthermore, as the commercial use of ROS2 expands, there is a need for a mechanism that can transfer prototype programs to practical systems quickly. Taking these points into consideration, we propose a program architecture that is easy to develop and provides versatility at the same time.

2. Application targeted for implementation

We will specifically consider application architecture by modeling the structure for an advanced mobile manipulator in this chapter.

Traditionally, the Turtlebot specification was created and is being used as a ROS application development standard. As an extension of this, we will take as an example the construction of a mobile manipulator application sample using Turtlebot + OpenManipulator.

As a typical example of an application to be implemented, let us consider a series of operations consisting of searching, picking, and placing objects by a mobile manipulator. The program uses a SLAM map to search for an object such as cola cans, recognizes the type and location of the object, etc., and makes the robot approach a position where it can be operated with a manipulator, perform a picking operation, and then move and transport it to a predetermined location.

The operations mentioned here are as follows:

- Object search: While robot is moving, the camera discovers the work object and determines its location and type.
- Approaching the object (operation detail is prepared for each object category): While checking the position of the object using the camera, move to a position where robots can perform tasks such as picking. Since the appropriate direction and distance to work with each object category are different, separate strategies are to be prepared for each category. Object categories are classified based on, for example, grasping operation strategies.
- Picking (operation detail is prepared for each category of object): In order to execute a grasping strategy that corresponds to the shape of the object, it is prepared for each category. For example, if the object is cylindrical like a cola can, the posture and shape around the center part will be measured and the corresponding gripping motion will be performed. If the object is shaped like a wine bottle, the posture and shape around the bottom will be measured, and performs a corresponding gripping motion.

A mobile manipulator application is constructed by combining the above operations as functional units. The actual sample code is available from the following repository:

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

3. Conceptual model of program structure

In this chapter, let us think about the architecture required to implement the functionality listed above. For this purpose, the standard functions provided by ROS alone are insufficient, and it is necessary to implement a combination of various applications/algorithms related to environmental recognition using deep learning.

Traditionally, the functional unit in ROS is a node. ROS nodes are already available, such as NVIDIA's Isaac ROS, which provide a variety of cognitive functions. The whole including MoveIt and Nav is combined into a single system with the communication function of ROS. However, it is not so easy to integrate a complex application structure consisting of such a large number of various nodes.

The basic policy here is to implement a "central node" that can integrate and operate a large number of functional units implemented as nodes. It needs to have the ability to communicate with a large number of nodes and schedule the next global operation in real time. Behavior Tree is a candidate, but since Behavior Tree is based on a control structure of polling, the idea is to create smaller-grained functional units that operate in real time and call them from

Behavior Tree for use.

Here, we will refer to such real-time functional units as 'actors'. Each actor is configured as a Python callable, but can be called in either synchronous or asynchronous ways, and each actor does not have a persistent context locally across multiple invocations. These reduce the complexity of real-time programming.

The mechanism for implementing actor is published on GitHub as the ros_actor library.

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

4. Actor

Here, we will try to build a ROS application with the structure shown in Figure 1.

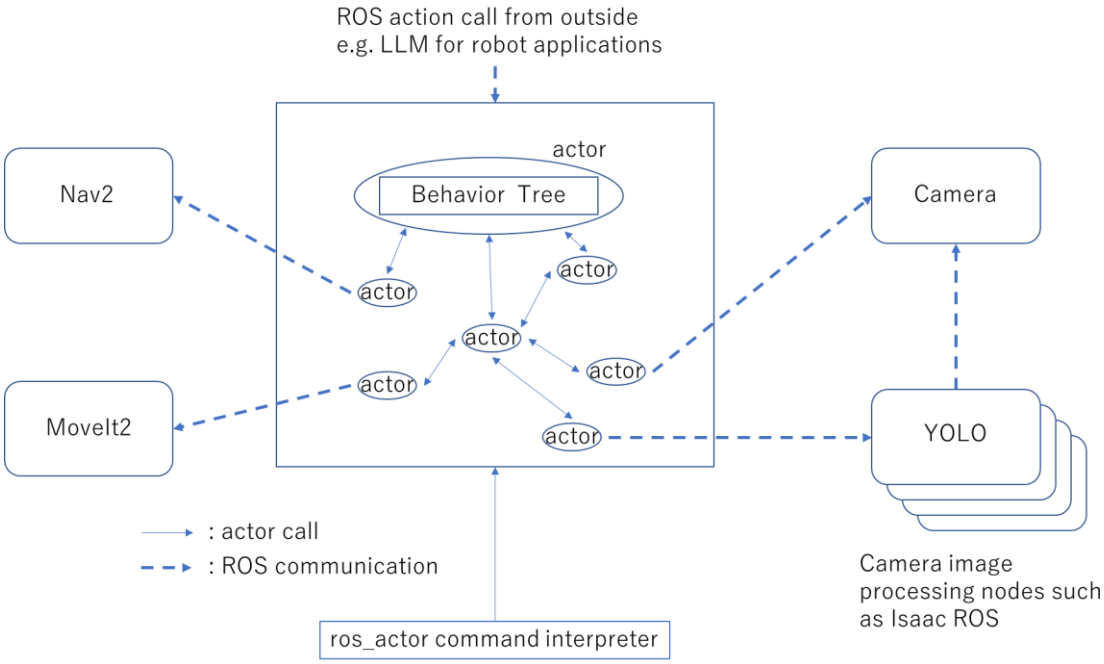

Figure 1 ROS application construction using actor.

(1) Engineering aspects

An actor is simply a Python callable whose purpose is to implement a unit of robot behavior. Each actor has a name, and you can use its function by specifying the name.

To improve ease of development, it can be activated in multiple ways. They are:

・ Calls between actors,

Both synchronous and asynchronous calls are possible.

・ Call from external program,
・ Launch from terminal as command,
・ Call from other nodes as ROS communication action.

After creating individual actors, it is designed so that they can be called directly from the command interpreter for debugging and improvement. Furthermore, since an actor can be called as an action through ROS communication, it can be used as an action unit from higher layer, such as a deep reinforcement learning program.

Additionally, the Behavior Tree itself can be used as an actor, and each Behavior can use actors. That is, the Behavior Tree and actors are recursively interconnected.

(2) How to use Actor

A collection of actors that implements a single unified function is called a "SubSystem". SubSystem is defined as a Python class, declares persistent data in the constructor, and defines actors as methods. The definition of actor is, for example,

```
Class Sample System(SubSystem):

@actor
def sample_actor(self, mes):
        print(mes)
```

It can be defined simply by adding an actor decorator, like this. In the above example, an actor named "sample_actor" is defined.

You can choose whether to call the actor synchronously or asynchronously. To call an actor from inside another actor synchronously, use the run_actor method.

```
self.run_actor(“sample_actor”, “hello”)
```

By doing this, "hello" is displayed. It can also be called in other formats by using the run_actor_mode method. For example, the statement

self.run_actor_mode(“sample_actor”, “async”, callback)

makes an asynchronous call to ‘sample_actor’, and the callback will be called when it finishes executing.

Also,

iterator = self.run_actor_mode(“sample_actor”, “iterator”)

will return an iterator which provides return values of callbacks of sample_actor multiple times with the same conditions.

```
@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
```

pic_find executes a loop with repeatedly accesses the pic_receiver actor until a cola can is found in the image. At this time, by specifying 'timed_iterator' as the argument of run_actor_mode, the upper limit of the duration of loop execution is set to 10 seconds, and the loop will be terminated if it is repeated for more than 10 seconds.

In this way, actor assumes that real-time systems are also coded using loop structures. Since the wait is done within the iterator, it is not recommended to explicitly use synchronization instructions such as Semaphore. This makes it easy to implement complex real-time programs. Additionally, future implementations of the actor mechanism are planned to be based on coroutines rather than threads, so using synchronous instructions may cause them to not work.

This approach makes it possible to create robot applications in a format similar to conventional general-purpose programs, which is of great help to inexperienced programmers. For example, when calling an actor having synchronous structure asynchronously, the implementation mechanism for the actor (ros_actor lib.) automatically

allocates a thread to run the actor and calls the callback upon completion, eliminating the need for the programmer to explicitly consider this.　In contrast, asynchronous processing using callbacks is often used in conventional real-time programs, which causes confusion in both control and data structures.

The core of actor's functionality is thus separating synchronous and asynchronous relationships between the implementer and the caller. Therefore, independent of the caller's specifications, the implementation of the actor itself can be freely selected between synchronous and asynchronous types. In the definition of actor, the notation

```
@actor('pic_receiver', 'async')
def pic_receiver(self, callback):
```

allows for asynchronous implementation. pic_receiver only starts the process when it is called, and calls the callback when it finishes.

In this way, pic_receiver is originally implemented asynchronously, but it can still be invoked in a synchronous way.

```
self.run_actor(‘pic_receiver’)
```

Users of the pic_receiver actor can now program without touching asynchronous control structures.

In the actor mechanism, ROS communication is also wrapped in the form of an actor call, so that it can be used without distinction. for example, the notation

```
self.register_action('navigate', NavigateToPose, "/navigate_to_pose")
self.register_publisher('motor', Twist, 'cmd_vel', 10)
self.register_subscriber('pic',Image,"/intel_realsense_r200_depth/image_raw",10)
```

allows you to define navigate action client, motor publisher, and pic subscriber and call them like normal actors.

The "SubSystem" class not only defines actors, but also performs "edge" functions related to external collaboration, such as defining persistent variables and ROS communication. Most of these declarations are made in the SubSystem constructor. If there are too many actor

definitions, they can be defined separately in the "SubNet" class and used by embedding them in the SubSystem constructor. In other words, the SubNet class is used for the same purpose as #include in C language, and it is recommended not to use an explicit constructor that creates a persistent context in SubNet.

(3) ros_actor's goal

The ultimate goal of the actor mechanism is to implement a group of basic robot movements and compile them into a database format so that they can be easily used by higher-level deep learning systems such as LLM. It is thought to be effective to implement such basic functions using Behavior Tree/Task Constructor, as is already the case with Nav2 and MoveIt2, and actors support this from a real-time perspective. It is a mechanism that provides a single interface and allows the whole to be implemented as a functional database. For this purpose, the current implementation supports only Python, but in the future, it will also support C++, and the actor mechanism will be able to absorb language differences. Also, calling actors from the command line is for easy testing during prototyping. This allows new unit functions to be implemented incrementally. In particular, it is expected that development efficiency will increase by first prototyping using Python, checking the functionality and specifications, and then formal implementation in C++.