Actor 機構の解説

momoi.org

1. まえがき

ROS の応用分野は広範囲におよびつつあり、そのアプリケーションの構造もますます複雑になっている。さらに、深層学習技術の成果とも連携する必要が生じている。これにより、ロボットの専門家だけでなく、AI や、ビジネスアプリケーション等のロボット以外の専門家がロボットアプリケーションを作成する機会も高まっている。本サンプルアプリケーションは、このような他分野の専門家や、これから ROS を勉強しようとしている学生が、短時間で試行に用いることができるアプリケーションサンプルとして企画されたものである。

従来、ROS のアプリケーション開発プラットフォームとして Turtlebot 仕様が作成され、使用されてきた。その延長として、Python で簡便に記述できる、深層学習を実行できる GPU を備えた Lime によるモバイルマニピュレータのアプリケーションサンプルをここに掲げるものである。

深層学習技術の出現により、画像処理や強化学習などの分野の高度なアルゴリズムが開発されるようになっており、ここでは、それらを簡単に取り込んで種々の試行が行えるように、汎用的なプログラム構造をとることを試みた。さらに、ここで作成したプロトタイプを、実システムにスムースに移行するための機構も求められる。これらの点も考慮して、汎用性を持たせるための簡単なソフトウェアアーキテクチャを構築した上で、アプリケーションサンプルを実装することとする。

2. 実装の対象となるアプリケーション

本コードは、極めてシンプルな構造をとりながらも、統合性と拡張性を考慮した上で、初心者にも扱える構造を持った統合的なサンプルコードになるように配慮して設計されている。そこで最も重要な点は、高度なモバイルマニピュレータを実現するための構造をモデル化することである。

実装の対象となる、典型的なアプリケーションのうち、典型的なものとして、モバイルマニピュレータによるオブジェクトの探索、ピッキング、プレースからなる、一連の動作を考えてみる。これは、SLAM による地図を用いて、オブジェクトを探索し、コーラ缶等の種別と位置を認識し、マニュピュレータによる操作が可能な位置まで接近し、ピッキング動作を行い、移動して所定位置まで運搬し、プレースする作業を、要素として含んでいる。ここであげた、オブジェクトの探索と認識、接近、ピッキング、プレース等は、具体的には、以下のようなものである。

・ オブジェクトの探索：移動しながら、カメラによって作業対象物を発見し、その位置と種類を判別する

・ オブジェクトへの接近（オブジェクトのカテゴリごとに用意）：カメラによって対象物

の位置を確認しながら、ピッキング等の作業ができる位置まで移動する。オブジェクトのカテゴリごとに、接近すべき方角、距離が異なるので、それぞれに別個の戦略を用意する。オブジェクトのカテゴリは、たとえば把持操作の戦略によって分類を行う。大きさ、形状、把持位置等によってカテゴリは異なる。

・　ピッキング（オブジェクトのカテゴリごとに用意）：オブジェクトの形状に対応した把持戦略を実行するため、そのカテゴリごとに用意される。たとえば、コーラ缶のような円筒形であれば中央部の周囲の姿勢と形状を計測してそれに対応した把持動作を行い、ビール瓶のような形状であれば、下部の周囲の姿勢と形状を計測してそれに対応した把持動作を行う。

モバイルマニピュレータアプリケーションは、以上のような動作を機能単位として、それらを組み合わせることによって構成される。

3.　プログラム構造の概念モデル

ROS の MoveIt や Nav 等を用いて、２．であげたような機能を実装するために必要なアーキテクチャについて考えてみる。そこでは、ROS が提供する標準機能だけでは不足であり、特に深層学習を用いた環境認知に関する種々のアプリケーションを組み合わせて実装する必要がある。

従来、ROS での機能構成単位はノードであった。YOLO のような AI 機能もノードとして実装され、MoveIt や Nav のノードと ROS の通信機能によって結合することによって全体が構成される。すでに、NVIDIA の Isaac ROS に見られるように、種々の機能を提供する ROS ノードが提供されるようになっている。しかしながら、このような多数のノードからなる複雑なアプリケーション構造を統合することは容易ではない。

ここでの基本的な目標は、ノードとして実装された多数の機能単位を統合して動作させることができる、“中央ノード”の実装方法を検討することである。これは、多数のノードと通信を行って、リアルタイムで次の全体動作をスケジューリングする機能を持つ必要がある。Behavior　Tree がその候補となるが、Behavior Tree はポーリングに基づく制御構造に基づいているので、リアルタイムで動作するより小さな粒度の機能単位を作成し、それを Behavior Tree から呼び出して利用するようにすることが、有効であると考えられる。

ここでは、そのようなリアルタイム機能単位を’actor’とよぶことにする。各 Actor は、みかけは単純な関数であるが、データフローモデル的な構造原理で構成される。Actor は、Python の callable として構成されるが、同期的な方法でも非同期的な方法でも呼び出すことができ、また、各 actor は、複数回の呼び出しに渡る永続的なコンテクストを持たない。これらによって、リアルタイムプログラミングの複雑さを軽減している。

4.　Actor

ここでは、図１に示すような構造で、ROS アプリケーションを構築することを試みる。

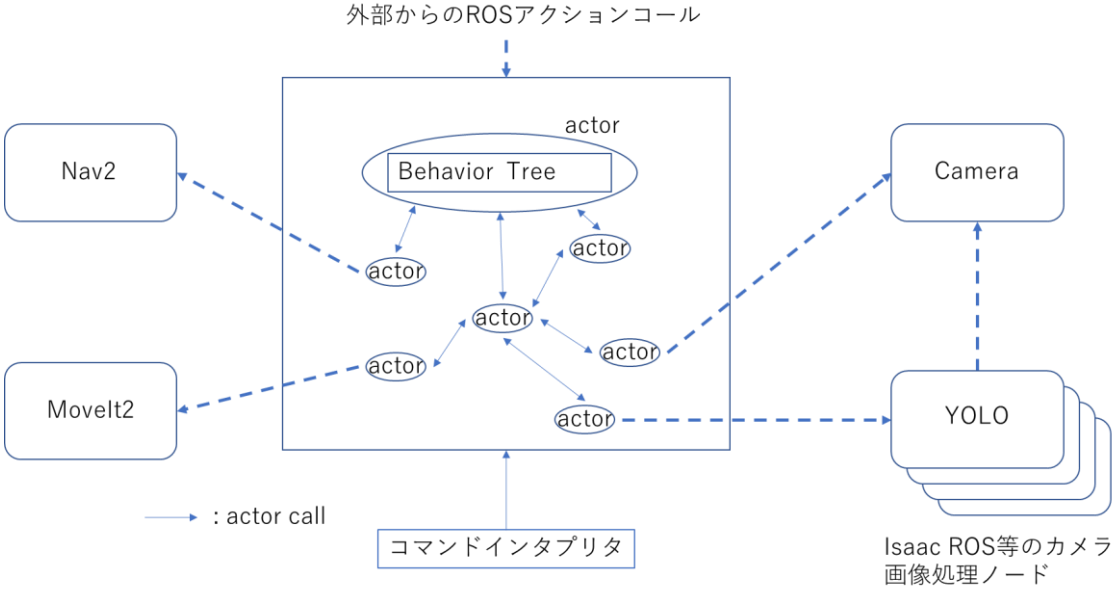

図１　actor による ROS アプリケーション構築

(1) エンジニアリング的側面

Actor は、単なる Python の callable であり、ロボットの単位となる動作を実装することを目的としている。各 Actor には名前が付けられており、名前を指定してその機能を利用することができる。

開発の容易さを向上する観点から、複数の方法で起動することができるようになっている。

- Actor 間での呼び出し
  同期的な呼び出しと、非同期的な呼び出しの双方が可能である
- 外部プログラムからの呼び出し
- コマンドとして、端末から起動
- ROS 通信の action として、他のノードからの呼び出し

が可能になっている。個々の actor を作成した後、コマンドインタプリタから呼び出してデバッグや改良を行えるように配慮されている。また、actor は、action として ROS 通信で呼び出すことが可能なので、より上位の、たとえば、深層強化学習プログラム等から動作単位として呼び出すことが可能である。

さらに、behavior tree 自体も、actor として利用できるようになっているとともに、各 behavior として actor を利用することができる。すなわち、behavior tree と actor はリカーシブに相互接続されている。

(2) Actor の利用方法

Actor の呼び出しは、同期的、あるいは非同期的の、どちらでも呼び出し時に選択できる。

Actor の集合が、一つのまとまった機能を実装しているものを、”SubSystem”とよぶ。SubSystem は Python のクラスとして定義され、コンストラクタで永続的なデータを宣言し、actor をメソッドとして定義する。Actor の定義は、たとえば、

```
Class SampleSystem(SubSystem):

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

のように、actor デコレータを付加するだけで定義できる。 上記の例では、”sample_actor”と命名された actor が定義されている。

他の actor の内部から actor を呼び出すためには、run_actor メソッドを用いる。

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

とすることにより、”hello”が表示される。また、run_actor_mode メソッドを用いることにより、他の形式で呼び出すことができる。たとえば、

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

とすると、非同期的な呼び出しが行われ、sample_actor 実行終了時に、callback が呼び出される。また、

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

とすると、iterator が返され、同一条件で複数回の呼び出しを行うことができる。具体的には、

```
@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
```

とすると、カメラからの画像を受信する、pic_receiver actor に繰り返しアクセスし、画像中

にコーラ缶が発見されまでループ実行を行う actor が実装される。このとき、run_actor_mode の引数として、'timed_iterator'を指定することにより、タイマによるループ回数の上限を 10 秒と設定し、10 秒以上繰り返した場合はループを終了できるようにしている。

Actor では、このように、リアルタイムシステムも、ループ構造によってコーディングすることが想定されている。待ち合わせは iterator 内で行われるので、陽に Semaphore などの同期命令を使用することは推奨されない。また、今後の actor 機構の実装は、thread ではなく、coroutine ベースに移行する予定であるので、同期命令を使用すると動作しなくなることが予想される。

このような記述法により、ロボットアプリケーションを、従来の汎用プログラムと同様の手続き呼び出しに近い形式で作成できるようになり、経験の浅いプログラマに対して大きな助けとなる。たとえば、同期的に実装された actor を非同期的に呼び出す場合、actor の実装機構が自動的にスレッドを割り当てて actor を実行し、終了時に callback 呼び出しを行うので、プログラマが陽に考慮する必要がなくなる。これに対して、従来のロボットアプリケーションの実装では、コールバックによる非同期的な処理が多用され、制御とデータ構造の双方が錯綜する原因となっていた。

actor の機能の中核は、このように、実装側と呼び出し側の間で同期と非同期の関係を分離することにある。したがって、呼び出す側の指定とは独立に、actor 自体の実装も、同期型と非同期型を自由に選択できる。Actor の定義において、

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

とすると、非同期的な実装が可能になる。pic_receiver は、呼び出し時には処理の起動のみを行い、終了時に callback を呼び出すように動作する。

このように、本来、pic_receiver は非同期的に実装されていたが、それを利用する側は、iterator を利用することにより、非同期的な制御構造に触れることなくプログラミングすることが可能になっている。

actor 機構では、さらに、ROS 通信も actor 呼び出しの形式にラッピングされ、区別なく利用できるようになっている。たとえば、

```
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)
```

によって、navigate action clien, motor publisher, pic subscriber が定義でき、通常の actor と同様に呼び出すことができるようになる。

”SubSystem”クラスは、actor を定義するとともに、永続的な変数の定義や、ROS 通信の定義など、外部との連携に関する“エッジ”機能を果たす。これらの宣言は、大半が、SubSystem のコンストラクタ中で行われる。Actor の定義が多数になった場合は、別途”SubNet”クラス中で定義し、SubSystem に埋め込んで使用することができる。すなわち、SubNet クラスは、C 言語の#include と同様の目的で使用されるものであり、そこでは永続的なコンテクストを生成するコンストラクタは作成しないことが推奨される。

(4) Actor の目標

actor 機構の最終的な目標は、ロボットの基本動作群を実装し、データベースの形式にして、LLM 等の上位の深層学習から利用しやすいようにまとめることである。Nav2 や MoveIt2 がすでにそうであるように、Behavior Tree/Task Constructor を用いてこのような基本機能の実装を行うと有効であると考えられ、actor はそれをリアルタイム性の面から支え、ラッピングされた単一インタフェースを提供し、全体を機能データベースとして実装することを可能にする機構である。

このような目的から、現在の実装は、Python のみ対応しているが、将来、C++にも対応し、言語の差異も actor 機構によって吸収できるようにする。

また、コマンドラインから actor を呼び出すのは、プロトタイピングの際、簡単に試行を行うためである。これによって、新規の単位機能をインクリメンタルに実装することが可能になる。特に、最初 Python を用いてプロトタイピングを行い、機能や仕様の確認を行った後に C++で正式な実装を行うような使い方をすることにより、開発効率が高まることが期待される。

5. アプリケーション例

Lime のシミュレーション例を示す。ロボットは、迷路中でカメラを使ってコーラ缶を探索し、発見した場合は接近してピッキングを行い、所定の場所まで運搬する。全体は、以下の behavior tree で実現される。

```
<root>
    <BehaviorTree ID="bt_all">
    <Sequence name="main">
        <bt_search name="global_search"/>
        <bt_catch name="pick_action"/>
        <bt_carry name="carry_action"/>
    </Sequence>
    </BehaviorTree>
</root>
```

bt_search は、カメラを動作させながら、迷路内を移動してコーラ缶を探索する。探索経路は、vector_map ライブラリを用いてベクトル化された SLAM 地図を用いてプランニングする。

bt_catch は、コーラ缶を発見した場合、段階的に近づきながらカメラを用いてコーラ缶姿勢の計測を補正し、ピッキング作業が行える位置まで接近し、アーム角度を設定し、ピッキング作業を行う。

bt_carry は、固定位置にあるパレットまでコーラ缶を運搬し、カメラの深度情報を用いたたビジュアルフィードバックによってパレット上にコーラ缶をプレースできる位置まで移動し、プレース作業を行う。

これらの behavior の実装には、カメラ制御、移動制御、アーム制御に関わる 20 個を超える数の actor が利用されている。