Using a state machine to perform YMCA

Finite-state machines are powerful mechanisms for controlling the behavior of a system, especially robotic systems. ROS has implemented a state machine structure and behaviors in a Python-based library called SMACH. The SMACH library is independent of ROS and can be used with any Python project. SMACH provides an architecture for implementing hierarchical tasks and mechanisms to define transitions between these tasks. The advantages of using SMACH for a system include the following:

- Rapid prototyping of a state machine for testing and use
- Defining complex behaviors using a clear, straightforward method for design, maintenance, and debugging
- Introspection of the state machine, its transitions, and data flow using SMACH tools

For a complete set of documentation and tutorials on SMACH, examine these websites:

- `http://wiki.ros.org/smach`
- `http://wiki.ros.org/smach/Tutorials`

Some basic rules for implementing state machines on a robot are as follows:

- A robot can be in one—and only one—state at a time.
- A finite number of states must be identified.
- The state that a robot transitions to, will depend on the state that just completed. These behaviors are encapsulated in the states to which they correspond.
- Transitions between states are specified by the structure of the state machine.
- All possible outcomes of a state should be identified and corresponding behaviors should address those outcomes.
- States that only have one transition condition cannot fail and only have one outcome.

To underscore the usefulness of the SMACH package, we devised a simple and fun example for Baxter, which has been implemented by Mikal Cristen, a recent UHCL graduate. Because the UHCL Baxter is such a main attraction on our campus, this project was to endow the robot with an entertainment skill, specifically, dancing to YMCA.

This state machine has five states corresponding to Baxter's arm poses for each letter: Y, M, C, A, and a fifth state for a neutral pose. When one pose of the arms completes, the state will successfully complete and the next state will begin. The code for this state machine is implemented in the `YMCAStateMach.py` code that follows and will be described in subsequent paragraphs.

> The code for `YMCAStateMach.py`and `MoveControl.py`can be found in the`Chapter6`folder of the Packt GitHub website for this book or at the website`https://github.com/FairchildC/ROS-Robotics-By-Example-2nd-Edition`

SMACH compels state machines to be implemented using Python procedures to provide flexibility in their implementation. Notice in the following code, the ROS convention for state machines is that the state names are identified in ALL_CAPS and the transition names are in lowercase:

```
#!/usr/bin/env python

import rospy
```

```
from smach import State,StateMachine

from time import sleep
from MoveControl import Baxter_Arms

class Y(State):
  def __init__(self):
    State.__init__(self, outcomes=['success'])

    self.letter_y = {
'letter': {
'left':  [0.0, -1.0, 0.0, 0.0,  0.0, 0.0, 0.0],
'right': [0.0, -1.0, 0.0, 0.0,  0.0, 0.0, 0.0]
                       } }
      #DoF Key [s0,s1,e0,e1,w0,w1,w2]

  def execute(self, userdata):
    rospy.loginfo('Give me a Y!')
    barms.supervised_move(self.letter_y)
    sleep(2)
    return 'success'

class M(State):
  def __init__(self):
    State.__init__(self, outcomes=['success'])

 self.letter_m = {
'letter': {
'left': [0.0, -1.50, 1.0, -0.052, 3.0, 2.094, 0.0],
'right':[0.0, -1.50, -1.0, -0.052, -3.0, 2.094, 0.0]
                       } }
               #DoF Key [s0,s1,e0,e1,w0,w1,w2]

  def execute(self, userdata):
    rospy.loginfo('Give me a M!')
    barms.supervised_move(self.letter_m)
    sleep(2)
    return 'success'

class C(State):
  def __init__(self):
    State.__init__(self, outcomes=['success'])

    self.letter_c = {
'letter': {
'left': [0.80, 0.0, 0.0, -0.052,  3.0, 1.50, 0.0],
'right':[0.0, -1.50, -1.0, -0.052, -3.0, 1.0, 0.0]
                       } }
            #DoF Key [s0,s1,e0,e1,w0,w1,w2]

  def execute(self, userdata):
    rospy.loginfo('Give me a C!')
    barms.supervised_move(self.letter_c)
    sleep(2)
```

```
        return 'success'

    class A(State):
      def __init__(self):
        State.__init__(self, outcomes=['success'])

        self.letter_a = {
    'letter': {
    'left': [0.50, -1.0, -3.0, 1.0, 0.0, 0.0, 0.0],
    'right':[-0.50, -1.0, 3.0, 1.0,  0.0, 0.0, 0.0]
                           } }
                #DoF Key [s0,s1,e0,e1,w0,w1,w2]

      def execute(self, userdata):
        rospy.loginfo('Give me an A!')
        barms.supervised_move(self.letter_a)
        sleep(2)
        return 'success'

    class Zero(State):
      def __init__(self):
        State.__init__(self, outcomes=['success'])

        self.zero = {
    'letter': {
    'left': [0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
    'right':[0.00, 0.00, 0.00, 0.00,  0.00, 0.00, 0.00]
                           } }
                             #DoF Key [s0,s1,e0,e1,w0,w1,w2]

      def execute(self, userdata):
        rospy.loginfo('Ta-da')
        barms.supervised_move(self.zero)
        sleep(2)
        return 'success'

    if __name__ == '__main__':

      barms = Baxter_Arms()
      rospy.on_shutdown(barms.clean_shutdown)

      sm = StateMachine(outcomes=['success'])
      with sm:
        StateMachine.add('Y', Y(), transitions={'success':'M'})
        StateMachine.add('M', M(), transitions={'success':'C'})
        StateMachine.add('C', C(), transitions={'success':'A'})
        StateMachine.add('A', A(), transitions={'success':'ZERO'})
        StateMachine.add('ZERO', Zero(), transitions={'success':'success'})

      sm.execute()
```

This Python script imports the following packages:

- `rospy`: This ROS package is used for information messages and to control Baxter's arms in the event a system shutdown occurs.
- `smach`: This ROS package imports the `State` and `StateMachine`classesand their methods.
- `time`: This Python package imports the `sleep`function.
- `MoveControl`: This package imports the `Baxter_Arms`classand the methods for interacting with Baxter's arms. Look for this Python script in the `Chapter06`folder of the Packt GitHub website for this book.

Next, classes are defined for each of the states of the state machine: `Y`, `M`, `C`, `A`, and `Zero`. Each of these classes creates a new, initialized instance of the SMACH`State` class, identifying all of the possible outcomes for that state. For these states, `success` is the only outcome. Next, each of Baxter's 14 arm joints are assigned a value (in radians). These values are assigned in the specific order they are expected (ultimately) by the `baxter_interface` package. This order, as indicated in the code comment, is S0, S1, E0, E1, W0, W1, W2.

Each of the states also implements an execute method, where the actual work is done. In each of these state execute methods, the `supervised_move` function for moving Baxter's arms is called passing the argument with all of the joint angles for Baxter's arms. After the function is called, the process sleeps for 2 seconds to allow all of Baxter's arm movements to stop. When the state finishes, the `success` flag is returned as an outcome to the calling function.

The main program of `YMCAStateMach.py` creates an instance of the`Baxter_Arms` class and initializes the attributes for the instance. An instance of a `StateMachine` class is also created and a list of possible outcomes is passed as an argument. The empty `StateMachine` (`sm`) instanceis opened and populated with the different states we defined. Each state is added with the add function. For example, the first `StateMachine.add` function adds the state `Y`, with an instance of the class `Y()` and, on completion with a successful outcome, the `StateMachine`classwill transition to state `M`. Similarly, thestate `M`is added with an instance of the class `M()` and, with a successful outcome, will transition to state `C`. The states `C` and `A` are added in a similar manner, as is the last state `ZERO`. Upon completion, the state `ZERO`will return the successful outcome for `StateMachine`.

Be sure that the `MoveControl.py` and `YMCAStateMach.py` scripts are in your directory and have execute permissions. Real Baxter or Baxter Simulator should be running and enabled (preferably in an untucked pose). Then, run the state machine with the following command:

```
$ python YMCAStateMach.py
```

You should see Baxter's arms transition to a Y pose, then an M pose, then a C pose, then an A pose, and end in a pose similar to the `arms_to_zero_angles.py` pose. The`INFO` messages to the terminal window should be similar to the following:

```
[  INFO ] : State machine starting in initial state 'Y' with userdata:
[]
[INFO] [1502141862.054272]: Give me a Y!
[INFO] [1502141862.054742]: Movement in progress.
[INFO ] : State machine transitioning 'Y':'success'-->'M'
[INFO] [1502141867.569880]: Give me a M!
[INFO] [1502141867.570851]: Movement in progress.
[INFO] [1502141872.782220]: Robot Disabled
[  INFO ] : State machine transitioning 'M':'success'-->'C'
[INFO] [1502141874.784842]: Give me a C!
```

```
[INFO] [1502141874.785353]: Movement in progress.
[INFO] [1502141876.906888]: Robot Disabled
[INFO ] : State machine transitioning 'C':'success'-->'A'
[INFO] [1502141878.909487]: Give me an A!
[INFO] [1502141878.910383]: Movement in progress.
[INFO ] : State machine transitioning 'A':'success'-->'ZERO'
[INFO] [1502141889.262132]: Ta-da
[INFO] [1502141889.262614]: Movement in progress.
[ INFO ] : State machine terminating 'ZERO':'success':'success'
```