

Step1

According to Detailed guidance we use sm.Cascade, sm.Parallel, sm.Constant and sm.Wire to make a state machine.

Figure1(The original structure):

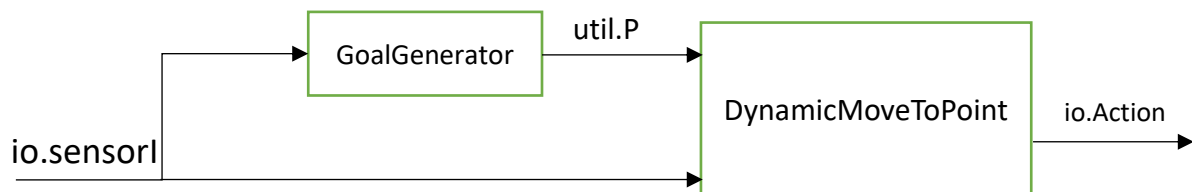
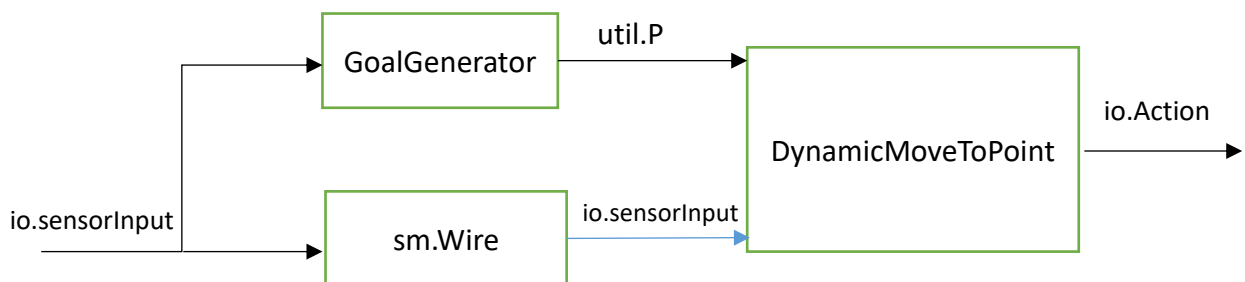


Figure2(Improved structure):



My code for constructing this composite machine into moveBrainSkeleton.py.

```

import os
labPath = os.getcwd()
from sys import path
if not labPath in path:
    path.append(labPath)
print('setting labPath to', labPath)

import math
import lib601.util as util
import lib601.sm as sm
import lib601.gfx as gfx
from soar.io import io

# Remember to change the import in dynamicMoveToPointSkeleton in order
# to use it from inside soar
import dynamicMoveToPointSkeleton
reload(dynamicMoveToPointSkeleton)

import ffSkeleton
reload(ffSkeleton)

from secretMessage import secret

# Set to True for verbose output on every step
verbose = True

# Rotated square points
squarePoints = [util.Point(0.5, 0.5), util.Point(0.0, 1.0),
                 util.Point(-0.5, 0.5), util.Point(0.0, 0.0)]

GoalGenerator = (0.5, 0.5)
DynamicMoveToPoint = dynamicMoveToPointSkeleton.DynamicMoveToPoint()
mySM=sm.Cascade(sm.Parallel(GoalGenerator,sm.Wire()),DynamicMoveToPoint))
    
```

Step2

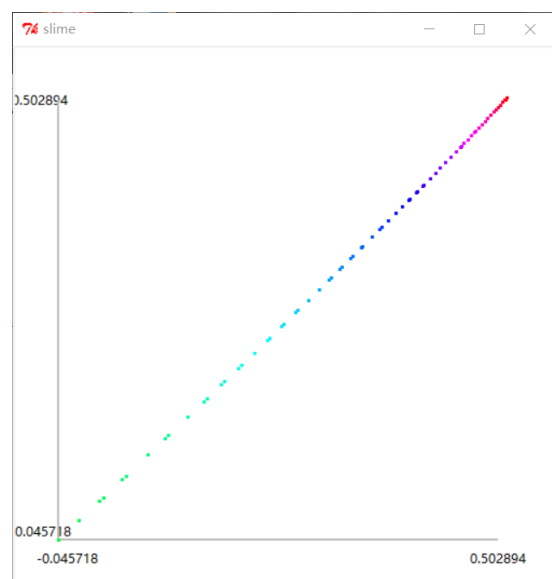
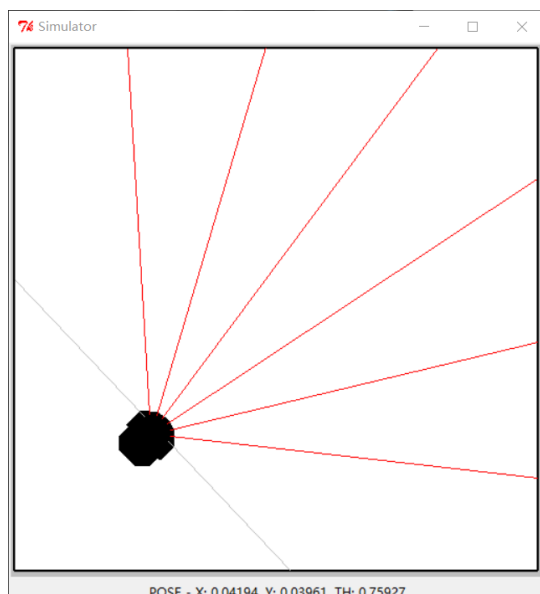
My code to implement the above functions into dynamicMoveToPointSkeleton.py

```
import lib601.sm as sm
import lib601.util as util
import math

# Use this line for running in idle
#import lib601.io as io
# Use this line for testing in soar
from soar.io import io

class DynamicMoveToPoint(sm.SM):
    def getNextValues(self, state, inp):
        (goalPoint,sensors)=inp
        p0=sensors.odometry
        while abs(p0.point().distance(goalPoint))<0.02:
            return (state,io.Action(fvel=0,rvel=0))
        if abs(util.fixAnglePlusMinusPi(p0.theta-math.atan2(goalPoint.y-p0.y, go
            return (state,io.Action(fvel=0,rvel=(util.fixAnglePlusMinusPi(math.a
        else:
            return (state,io.Action(fvel=(p0.point().distance(goalPoint))/1.5,r
        print('DynamicMoveToPoint', 'state=', state, 'inp=', inp)
        assert isinstance(inp,tuple), 'inp should be a tuple'
        assert len(inp) == 2, 'inp should be of length 2'
        assert isinstance(inp[0],util.Point), 'inp[0] should be a Point'
        return (state, io.Action())
```

The simulation results(target goal (0.5,0.5), starting point(0,0)):



Checkoff 1. Explain your strategy for implementing this behavior and your answers to the questions above to a staff member.

According to Detailed guidance, when state machine dynamicMoveToPoint-Skeleton get an input, it should adjust the direction first until the robot is in line with the target point or the angle between robot's orientation and the line between robot and the target point is less than 0.03 rad. Then it should go straight until the distance between robot and the target point is less than 0.02 meters.

Current robot pose	goalPoint	Action
(0.0, 0.0, 0.0)	(1.0,0.5)	Rotate left then move forward
(0.0, 0.0, $\pi/2$)	(1.0,0.5)	Rotate right then move forward
(0.0, 0.0, $\tan^{-1} 0.5$)	(1.0,0.5)	Move forward
(1.0001, 0.4999, 0.0)	(1.0,0.5)	Don't move

Step3

First, we comment out 'from soar.io import io' and uncomment 'import lib601.io as io'. Then we test the code in idle, by running the testMove procedure from testMove.py

The experiment results of testMove:

```
>>>
[SMCheck] the start state of your state machine is 'None'
The actual inputs are (target, sensorInput) pairs; here we
are just showing the odometry part of the sensorInput.

Input: (point:(1.000000, 0.500000), pose:(0, 0, 0))
Output: Act: (fvel = 0, rvel = 0.463648)

Input: (point:(1.000000, 0.500000), pose:(0, 0, 1.570796))
Output: Act: (fvel = 0, rvel = -1.107149)

Input: (point:(1.000000, 0.500000), pose:(0, 0, 0.463648))
Output: Act: (fvel = 0.931695, rvel = 0)

Input: (point:(1.000000, 0.500000), pose:(1.000100, 0.499999, 0))
Output: Act: (fvel = 0, rvel = 0)
>>>
```

Check Yourself 1. Be sure you understand what the answers to the test cases in this file ought to be, and that your code is generating them correctly.

The correct result of code generation is as above.

For these outputs, if the robot and the target point are not on the same line, robot should rotate first. The angle to turn is the angle between the orientation of the robot and the target point. If the robot is in line with the target point, it should move straight. If the robot is near the target point, it should be still.

Check Yourself 2. The robot always starts with odometry reading (0, 0, 0), so it ought to move to somewhere close to the point (1.0, 0.5) and stop.

We modify the parameters of the original statement as follows.

```

import os
labPath = os.getcwd()
from sys import path
if not labPath in path:
    path.append(labPath)
print('setting labPath to', labPath)

import math
import lib601.util as util
import lib601.sm as sm
import lib601.gfx as gfx
from soar.io import io

# Remember to change the import in dynamicMoveToPointSkeleton in order
# to use it from inside soar
import dynamicMoveToPointSkeleton
reload(dynamicMoveToPointSkeleton)

import ffSkeleton
reload(ffSkeleton)

from secretMessage import secret

# Set to True for verbose output on every step
verbose = True

# Rotated square points
squarePoints = [util.Point(0.5, 0.5), util.Point(0.0, 1.0),
                 util.Point(-0.5, 0.5), util.Point(0.0, 0.0)]

GoalGenerator = (1.0, 0.5)
DynamicMoveToPoint = dynamicMoveToPointSkeleton.DynamicMoveToPoint()
mySM=sm.Cascade(sm.Parallel(GoalGenerator,sm.Wire()),DynamicMoveToPoint)
    
```

The simulation results(target goal (1.0,0.5), starting point(0,0))



Step4

The experiment results of testFF:

```
>>>
[SMCheck] the start state of your state machine is '0'
The actual inputs are whole instances of io.SensorInput; here we
are just showing the odometry part of the input.

Input: pose:(0, 0, 0)
Output: point:(0.500000, 0.500000)

Input: pose:(0, 1, 0)
Output: point:(0.500000, 0.500000)

Input: pose:(0.499000, 0.501000, 2)
Output: point:(0.000000, 1.000000)

Input: pose:(2, 3, 4)
Output: point:(0.000000, 1.000000)
>>>
```

Step5

We modify the parameters of the original statement as follows.

The modified part in moveBrainSkeleton.py

```
import os
labPath = os.getcwd()
from sys import path
if not labPath in path:
    path.append(labPath)
print('setting labPath to', labPath)

import math
import lib601.util as util
import lib601.sm as sm
import lib601.gfx as gfx
from soar.io import io

# Remember to change the import in dynamicMoveToPointSkeleton in order
# to use it from inside soar
import dynamicMoveToPointSkeleton
reload(dynamicMoveToPointSkeleton)

import ffSkeleton
reload(ffSkeleton)


from secretMessage import secret

# Set to True for verbose output on every step
verbose = True

# Rotated square points
squarePoints = [util.Point(0.5, 0.5), util.Point(0.0, 1.0),
                util.Point(-0.5, 0.5), util.Point(0.0, 0.0)]

GoalGenerator = ffSkeleton.FollowFigure(squarePoints)
DynamicMoveToPoint=dynamicMoveToPointSkeleton.DynamicMoveToPoint()
mySM=sm.Cascade(sm.Parallel(GoalGenerator,sm.Wire()),DynamicMoveToPoint)
```

Change a target point to a list of target points



The modified part in ffSkeleton.py

```
import lib601.sm as sm
import lib601.util as util
import math

class FollowFigure(sm.SM):

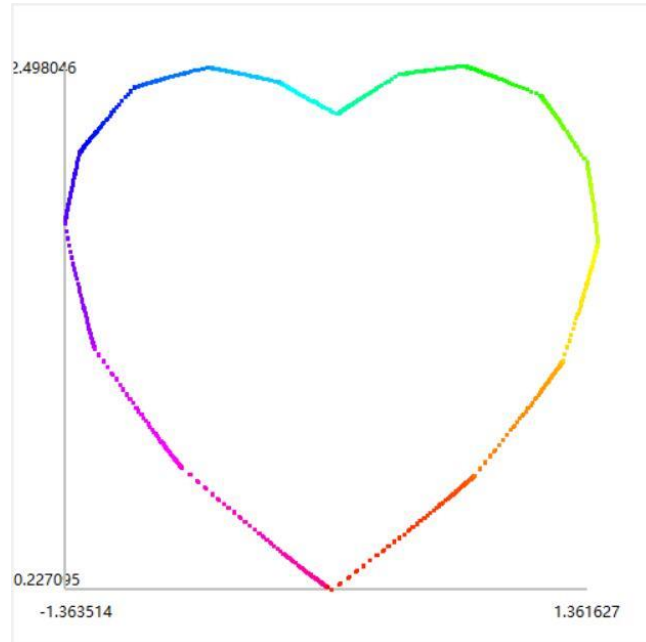
    def __init__(self, points):
        self.startState = 0
        self.points = points

    def getNextValues(self, state, inp):
        lens = len(self.points)
        p0=inp.odometry.point()
        i=state
        while i < lens-1 :
            if p0.isNear(self.points[i],0.03):
                return (state+1,self.points[i+1])
            else:
                return (state,self.points[i])
        return (state,self.points[i])
```

The simulation results:



We use the heart function to generate a bunch of points and let robot dance.



Checkoff 1. Show the slime trail resulting from the simulated robot moving in a square or other interesting figure to a staff member. Explain why it has the shape it does. Take a screenshot of the slime trail and save it for your interview. Mail the code and slime trail you have so far, to your partner.

My code for squarePoints and lovePoints into moveBrainSkeleton.py is as follows.(You can generate different trajectories by defining any list of points and changing the parameters in ffSkeleton.FollowFigure)


```

import os
labPath = os.getcwd()
from sys import path
if not labPath in path:
    path.append(labPath)
print('setting labPath to', labPath)

import math
import lib601.util as util
import lib601.sm as sm
import lib601.gfx as gfx
from soar.io import io

# Remember to change the import in dynamicMoveToPointSkeleton in order
# to use it from inside soar
import dynamicMoveToPointSkeleton
reload(dynamicMoveToPointSkeleton)

import ffSkeleton
reload(ffSkeleton)

from secretMessage import secret

# Set to True for verbose output on every step
verbose = True

# Rotated square points
squarePoints = [util.Point(0.5, 0.5), util.Point(0.0, 1.0),
                 util.Point(-0.5, 0.5), util.Point(0.0, 0.0)]

lovePoints = [util.Point(0.63,0.51), util.Point(1.00,1.00),
               util.Point(1.14,1.51), util.Point(1.08,1.85),
               util.Point(0.87,2.13), util.Point(0.54,2.24),
               util.Point(0.26,2.19), util.Point(0,2.01),
               util.Point(-0.25,2.18), util.Point(-0.56,2.23),
               util.Point(-0.87,2.13),util.Point(-1.09,1.84),
               util.Point(-1.14,1.53),util.Point(-1.00,1.00),
               util.Point(-0.62,0.50),util.Point(0,0)]

checkpoint = [util.Point(1.0,0.5)]

GoalGenerator = ffSkeleton.FollowFigure(lovePoints)
DynamicMoveToPoint=dynamicMoveToPointSkeleton.DynamicMoveToPoint()
mySM=sm.Cascade(sm.Parallel(GoalGenerator,sm.Wire()),DynamicMoveToPoint)
    
```

Step6 Demonstrate your safe figure-follower to a staff member.

Mail your code to your partner

For this purpose, we use the selection structure and define a stop class. The

additions in moveBrainSkeleton.py are as follows.

```
import os
labPath = os.getcwd()
from sys import path
if not labPath in path:
    path.append(labPath)
print 'setting labPath to', labPath

import math
import lib601.util as util
import lib601.sm as sm
import lib601.gfx as gfx
from soar.io import io

# Remember to change the import in dynamicMoveToPointSkeleton in order
# to use it from inside soar
import dynamicMoveToPointSkeleton
reload(dynamicMoveToPointSkeleton)

import ffSkeleton
reload(ffSkeleton)

from secretMessage import secret

# Set to True for verbose output on every step
verbose = True

# Rotated square points
squarePoints = [util.Point(0.5, 0.5), util.Point(0.0, 1.0),
                 util.Point(-0.5, 0.5), util.Point(0.0, 0.0)]

lovePoints = [util.Point(0.63,0.51), util.Point(1.00,1.00),
              util.Point(1.14,1.51), util.Point(1.08,1.85),
              util.Point(0.87,2.13), util.Point(0.54,2.24),
              util.Point(0.26,2.19), util.Point(0,2.01),
              util.Point(-0.25,2.18), util.Point(-0.56,2.23),
              util.Point(-0.87,2.13),util.Point(-1.09,1.84),
              util.Point(-1.14,1.53),util.Point(-1.00,1.00),
              util.Point(-0.62,0.50),util.Point(0,0)]

checkpoint = [util.Point(1.0,0.5)]

GoalGenerator = ffSkeleton.FollowFigure(squarePoints)
DynamicMoveToPoint = dynamicMoveToPointSkeleton.DynamicMoveToPoint()

def condition(inp):
    for i in range(7):
        if inp.sonars[i]<0.3:
            return True
    return False

class stop(sm.SM):
    def getNextValues(self, state, inp):
        return (state,io.Action(fvel=0,rvel=0))

# Put your answer to step 1 here
mySM = sm.Switch(condition,stop(),sm.Cascade(sm.Parallel(GoalGenerator,sm.Wire

#####
###
###      Brain methods
###
#####

def setup():
    robot.gfx = gfx.RobotGraphics(drawSlimeTrail = True)
    robot.behavior = mySM

def brainStart():
    robot.behavior.start(traceTasks = robot.gfx.tasks(),
                        verbose = verbose)

def step():
    robot.behavior.step(io.SensorInput()).execute()
    io.done(robot.behavior.isDone())

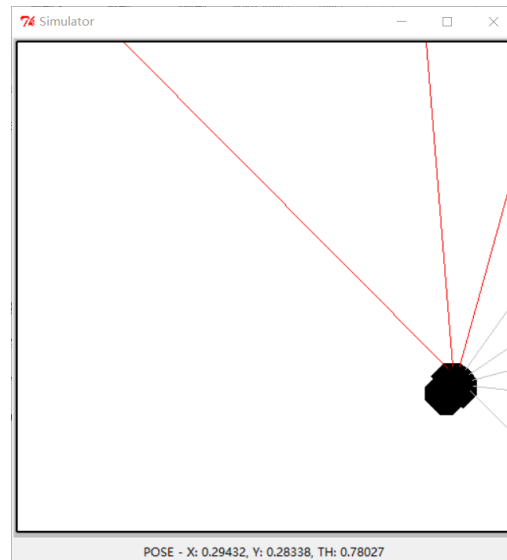
def brainStop():
    pass

def shutdown():
    pass
```

Ln: 24

The robot stopped 0.3 meters away from the obstacle.

The simulation results:



```
pose:(0.294328, 0.283389, 0.780278); Analog: (0.000000, 0.000000, 0.000000, 0.000000) Out: Act: [0, 0, 5.000000] Next State: (None, ((0, None), None))
In: Sonar: [5, 5, 5, 0.612518, 0.385040, 0.303916, 0.290456, 0.437558]; Odo:
pose:(0.294328, 0.283389, 0.780278); Analog: (0.000000, 0.000000, 0.000000, 0.000000) Out: Act: [0, 0, 5.000000] Next State: (None, ((0, None), None))
In: Sonar: [5, 5, 5, 0.612242, 0.385251, 0.303849, 0.290495, 0.437720]; Odo:
pose:(0.294328, 0.283389, 0.780278); Analog: (0.000000, 0.000000, 0.000000, 0.000000) Out: Act: [0, 0, 5.000000] Next State: (None, ((0, None), None))
In: Sonar: [5, 5, 5, 0.613536, 0.385247, 0.303848, 0.290482, 0.437934]; Odo:
pose:(0.294328, 0.283389, 0.780278); Analog: (0.000000, 0.000000, 0.000000, 0.000000) Out: Act: [0, 0, 5.000000] Next State: (None, ((0, None), None))
In: Sonar: [5, 5, 5, 0.611962, 0.385178, 0.303880, 0.290479, 0.437490]; Odo:
pose:(0.294328, 0.283389, 0.780278); Analog: (0.000000, 0.000000, 0.000000, 0.000000) Out: Act: [0, 0, 5.000000] Next State: (None, ((0, None), None))
```

Summary

1. When solving step1, it is difficult to find the serial relationship or parallel relationship between GoalGenerator and DynamicMoveToPoint only by looking at the block diagram in Figure1. However, the introduction of sm.Wire directly shows the serial-parallel relationship between them.

2. When implementing the Avoiding Pedestrians put the bot, we use sm.switch to avoid overlapping 'if' and 'loop' conditions.
3. When we use robot tracking, we adjust its state to a series of integers starting from 0, so as to judge whether the robot has traversed all the target points.