V1.0 | Last updated: 26 october 2020

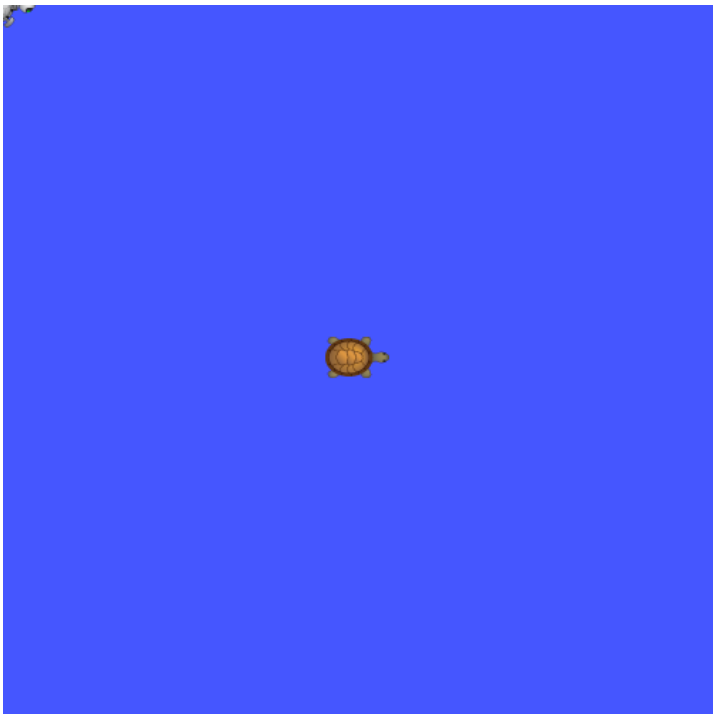
TURTLE-BEHAVIOURS PROJECT

Objective: Execute a range of TurtleSim Behaviours.
Duration: 1.5 weeks. Depends mainly on time spent in prep and setup time.
Learning outcomes: Merging technologies with ROS and the foundations of behaviour planning.

Note: this document was last updated in October 2020. Please signal any issues at thomas.carstens@edu.devinci.fr .

Suggested Output: TurtleSim Demo.
https://www.youtube.com/watch?v=tNqYDqC6wo4&t=92s&ab_channel=ThomasCarstens

STEP 0	**| LEARNING SMACH**	**| 4 days**
STEP 1	**| SETUP**	**| 2 days**
STEP 2	**| PROTOTYPE**	**| 2 days**
STEP 3	**| DEMO**	**| 1 day**



INTRODUCTION

Official WillowGarage Intro: https://www.youtube.com/watch?v=F3slROzVNbc&feature=emb_logo

Splitting the robot motion into states is a technique to be able to create various robot applications by **revisiting** states at will. We make use of a state machine to be our 'decision-maker' (state transitions) between the sensor input (callbacks) and specific motions of the bot (actions). SMACH is a simple Python implementation of state machines, and for the purposes of developing various usecases, you can get up and running easily.

The different programs that I reviewed were ROSPlan, RSM and Python Smach. The solution needed to work with ROS nodes, but also function independently for the sake of modularity. However, it was not necessary to have a performance-oriented state machine such as ROSPlan, rather one that would be useful for quick prototyping. An added plus would be integration into server-client setups for smooth demoing. The Python SMACH library promised all these elements.

V1.0 | Last updated: 26 october 2020

PRELIMINARY

Installation of Ubuntu (dual boot, preferably), of ROS, and finally of Gazebo.
The Ubuntu and ROS steps are outlined here:
https://www.youtube.com/playlist?list=PL1ezH_EwB1WV0NmtxSxzI_leFI_YRmhL7

It is always better to be prepared when approaching a new program. I advise you go through the ROS tutorials, UNIX if necessary, and Gazebo.

The Rosbash suite is an example of how ROS can be used better with preparation. The Rosbash suite contains commands like roslaunch, rosed and roscd. Learning to use these actually speeds up development time, as it gives basic file-management and automation tools.

1. Driving a car in Gazebo

Objective: Have the car drive through the gate, autonomously.
Duration: 1 week. (depends on how fast you do the ROS tutorials)

Steps

Step 1: set up ROS, then Gazebo simulator.
Step 2: set up the environment (the gate)
Step 3: Setting up the car
Step 4: Setting up a ROS script to control the car's wheels.
Step 5: Setting up a camera on the car (and rqtplot)
Step 6: Connecting OpenCV node to recognise obstacles.
Step 7: Making the car move to the gate.

ROS tutorials

Il est toujours bon de réviser ses fondamentaux et je conseille de faire les tutos Débutant, Intermédiaire et On Your Custom Robot pour découvrir toutes les nouvelles fonctionnalités:
http://wiki.ros.org/fr/ROS/Tutorials

***Si vous êtes nouveau sur Linux**: Vous trouverez peut-être utile de commencer par faire un rapide tutoriel sur les outils communs en ligne de commande pour Linux. Un bon est ici.*

Pour toute fonctionalité à implémenter sur Gazebo, voir http://gazebosim.org/tutorials

RESOURCES

ROS Textbook - https://drive.google.com/file/d/147I-EPXTe4QW_H9m5FHB2yH3JsMd5oMQ/view?usp=sharing (request access first)

GITHUB

Always stay up to date with the github. Star the repository and learn to $git pull
https://github.com/ThomasCarstens/txa-dvic-projects-tutos

Now, we can get started!

Contents

STAGE 1: UNDERSTANDING ROS-SMACH CONFIGURATION

LEARNING SMACH

I followed the official set of tutorials for the Smach documentation. These tutorials work in conjunction with TurtleSim, frequently used to test ROS functionality. The final result of this process is a simulation that is better viewed as a video rather than in pictures, please access it here.

You will be following various tutorials for using smach as seen here:

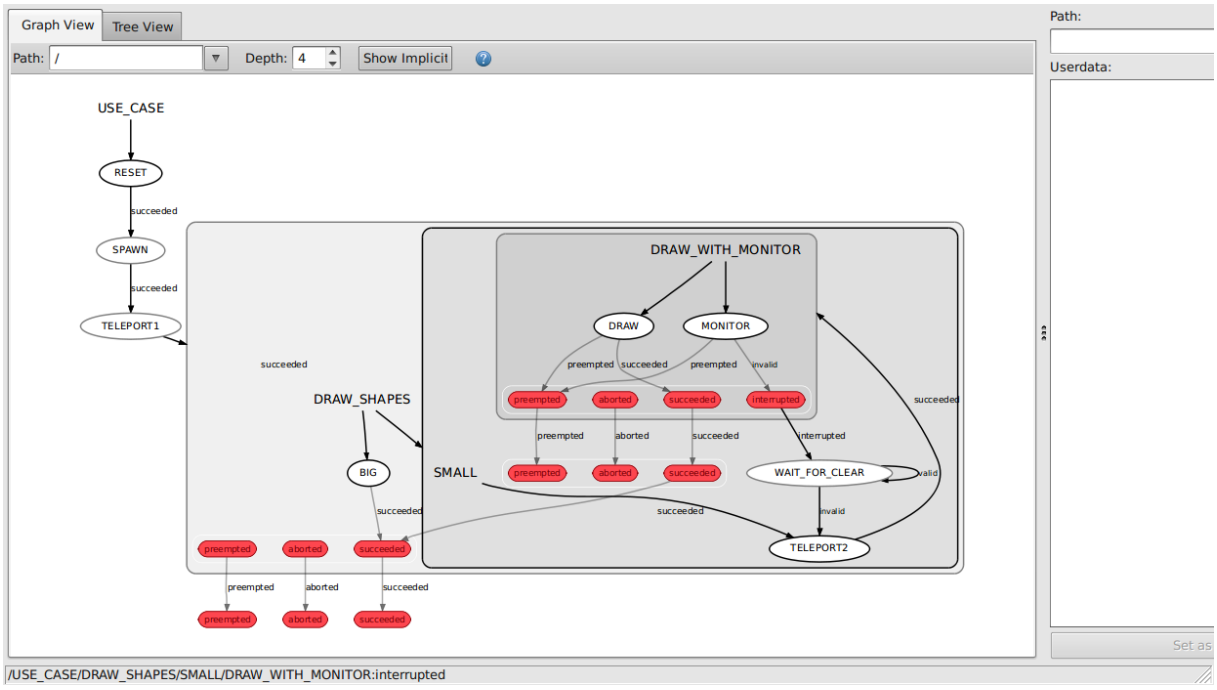
http://wiki.ros.org/smach (follow links to Documentation and Tutorial pages)

RUNNING THE CODE FOR THE FIRST TIME

Error1: Change imports to look something like this:

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

import roslib; #roslib.load_manifest('smach_usecase')

import threading
import rospy
import smach
import smach_ros
from smach_ros import ServiceState, IntrospectionServer, SimpleActionState, MonitorState#, Concurrence
from math import sqrt, pow

import std_srvs.srv
import turtlesim.srv
import turtlesim.msg
import turtle_actionlib.msg

```

By the end of the first part of this tutorial, you will have something like this!

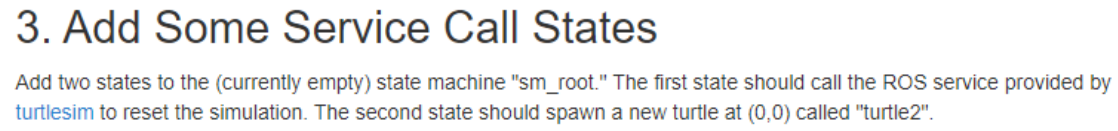


The first part of this tutorial follows the tutorial below closely.

http://wiki.ros.org/smach/UseCase

Please start their tutorial and refer here for any additional guidance.

LISTING ALL THE TURTLESIM SERVICES



Find how to list all the Services used with TurtleSim.

UNDERSTANDING THE TURTLESIM PARAMETER SERVER

TurtleSim has a parameter server to manage information like the TurtleSim background colour. It is worth exploring to understand the full capacities of the simulator.

In order to call existing services and make your own. Rosparam get / ((what is.)), rosparam load my_params.yaml ; rosparam list

http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams

https://roboticsbackend.com/ros-param-yaml-format/ ROS params.

LAUNCH FILE FOR ALL SCRIPTS AT ONCE + COMMON LOG FOR ALL SCRIPTS

This tutorial introduces ROS using rqt_console and rqt_logger_level for debugging and roslaunch for starting many nodes at once.

Roslaunching. But also different nodes for different turtles.

http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch

Creating a common log from all the different nodes launched.

http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch

ALGORITHM: ESTABLISHING A PERIMETER FOR EACH TURTLE

Look at this section once you arrive at step 7:

```
StateMachine.add('DRAW_WITH_MONITOR',
        draw_monitor_cc,
        {'interrupted':'WAIT_FOR_CLEAR'})

with draw_monitor_cc:
    Concurrence.add('DRAW',
            SimpleActionState('turtle_shape2',turtle_actionlib.msg.ShapeAction,
                goal = polygon_small))
    def turtle_far_away(ud, msg):
        """Returns True while turtle pose in msg is at least 1 unit away from (9,5)"""
        if sqrt(pow(msg.x-9.0,2) + pow(msg.y-5.0,2)) > 2.0:
            return True
        return False
    Concurrence.add('MONITOR',
            MonitorState('/turtle1/pose',turtlesim.msg.Pose,
                cond_cb = turtle_far_away))

StateMachine.add('WAIT_FOR_CLEAR',
        MonitorState('/turtle1/pose',turtlesim.msg.Pose,
            cond_cb = lambda ud,msg: not turtle_far_away(ud,msg)),
        {'valid':'WAIT_FOR_CLEAR','invalid':'TELEPORT2'})
```

HOW TO START MACHINE REMOTELY: WRAPPING CODE UP IN AN ACTION SERVER

We're on the last stage! Finally, a central decision planner is used in preparation for scaling up the network. To do this, a ROS Action Server wraps the actions. Using a mission_state ROS node, you can run a full Python Server that in itself runs **an action** in its callback.

9. Wrap SMACH Executive Into An Action

- Remove execute call
- Add wrapping code
- Create a python script that calls the action
- Run

```
    # Attach a SMACH introspection server
    sis = IntrospectionServer('smach_usecase_01', sm0, '/USE_CASE')
    sis.start()

    # Set preempt handler
    smach_ros.set_preempt_handler(sm0)

    # Execute SMACH tree in a separate thread so that we can ctrl-c the script
#   smach_thread = threading.Thread(target = sm0.execute)
#   smach_thread.start()
##################################
# Construct action server wrapper
    asw = ActionServerWrapper(
            'my_action_server_name', MachineAction,
            wrapped_container = sm0,
            succeeded_outcomes = ['did_something','did_something_else'],
            aborted_outcomes = ['aborted'],
            preempted_outcomes = ['preempted'] )

    # Run the server in a background thread
    asw.run_server()
    # Signal handler
```

First make sure you understand how to perform an **action.** Follow this tutorial. You can later adapt it to your specific action.

http://wiki.ros.org/actionlib

https://index.ros.org/doc/ros2/Tutorials/Actions/Writing-an-Action-Server-Python/

Using this server, you can create a custom message within a client that returns the amount of time it takes for the robot before a collision.

CLIENT: *Note that actionlib_tutorials is specific to the tutorial we did just above!*

```
def fibonacci_client():
    # Creates the SimpleActionClient, passing the type of the action
    # (FibonacciAction) to the constructor.
    client = actionlib.SimpleActionClient('my_action_server_name', actionlib_tutorials.msg.MachineAction)

    # Waits until the action server has started up and started
    # listening for goals.
    client.wait_for_server()

    # Creates a goal to send to the action server.
    goal = actionlib_tutorials.msg.MachineGoal(order=20)

    # Sends the goal to the action server.
    client.send_goal(goal)

    # Waits for the server to finish performing the action.
    client.wait_for_result()

    # Prints out the result of executing the action
    #result = fibonacci_client()
    result = client.get_result()
    print("Result: finished", ', '.join([str(n) for n in result.sequence]))

    return client.get_result()  # A MachineResult
```

STAGE 2: NEATENING IT UP FOR FURTHER DEVELOPMENT

Let me know when you get to this stage!!

STAGE 3: SETTING IT UP FOR A DEMO

MANAGING MULTIPLE DEMOES IN PARALLEL

If we can manage multiple state machines in parallel, we can have various configurations for the demoes.

Error: using the threading library to manage multiple threads:

```
if __name__ == '__main__':
    try:
        # Initializes a rospy node so that the SimpleActionClient can
        # publish and subscribe over ROS.
        rospy.init_node('fibonacci_client_py')
        image_sub = rospy.Subscriber("gate_state", String, GateCallback)
        message = rospy.wait_for_message("gate_state", String)
        t1 = threading.Thread(target=fibonacci_client)
        t2 = threading.Thread(target=lidar_client)
        t1.start()
        t2.start()
        #result = fibonacci_client()
        #otherwise = lidar_client()
        #print (result)
        #print (message.data)
        #rospy.spin()
```

Custom actions can deliver specific information to the user once the action has completed!

MAKING AN APPLICATION LAYER FOR YOUR DEMO

Please ask txa for more indications – have not added all the details yet.

Creating an interface :

https://www.youtube.com/watch?v=JBME1ZyHiP8&ab_channel=sentdex

To actually have a button activate the program, you might want to generate a button remotely:

Generating a button (code available as gen_button.py on github)

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

import rospy
from std_msgs.msg import String
import random
def put_on_default():
    pub = rospy.Publisher('go', String)
    rospy.init_node('test_button', log_level=rospy.INFO)
    rospy.loginfo("Default on wait until pressed")
    pub.publish("wait")
    while not rospy.is_shutdown():

        rospy.sleep(0.05)

if __name__ == '__main__':
  try:
    put_on_default()
  except Exception, e:
    print "done"
```