UNIVERSITY OF BURGUNDY

ROBOTICS OPERATION SYSTEM PROJECT

SMART Project

Author:
Anas MHANA

Supervisor:
Dr. Ralph SEULIN
Dr. Cansen JIANG
Dr. Raphael DUVERNE

Submition Date is

in the

January 6, 2016

UNIVERSITY OF BURGUNDY

Abstract

Institute of Technology
MsCV

SMART Project

by Anas MHANA

SMART project has been developed to achieve the main purposes of ROBOTIC module. the main purpose are understanding the core of the ROS and achieving deep comprehensive cognition of robot navigation and localization, computer vision and patrolling with different vision tasks. The project has been achieved using hydro distribution of ROS on a turtle-bot robot and a workstation. The turtle-bot would perform different tasks like object tracking, object recognition, tag detection, fire detection and vision decision making.
this report will discuss the details of the practical implementation of the SMART project with some theoretical aspects. After the introduction, building the map and autonomous navigation will be discussed and then Vision tasks which has been implemented in SMART project will be discussed later. Then, State machine implementation of the project will be explained. Finally,the conclusion and some problems which has been faced will be previewed.

Contents

Abstract **ii**

1 Introduction **2**

2 Navigation and Localization **4**
2.1 Motion Control : 4
2.1.1 Motion control goals : 4
2.2 Map Building : 5
2.3 Autonomous Navigation and Localization : 6

3 Computer Vision Tasks **8**
3.1 Object Detection and tracking : 8
3.2 Object following and fire escaping: 9
3.3 Tag Detection: 10

4 State Machine **11**

5 Problems Faced and Conclusion **14**
5.1 Problems Faced : 14
5.2 Conclusion : 14

A Source Code **15**

Bibliography **26**

List of Figures

2.1 Real Map 6
2.2 simulated Map 7

3.1 Fire Escaping 9

4.1 MAIN Project STATE MACHINE 12
4.2 MAIN Project STATE MACHINE with fire escaping 13

List of Abbreviations

ROS	**R**obot **O**perating **S**ystem
SMACH	**S**tate **Mach**achine
SLAM	**S**imultaneous **L**ocalization **a**nd **M**apping

Chapter 1

Introduction

This report will discuss the main details of SMART project which has satisfied the basic aims of the ROBOTIC module in the Master of Computer Vision program. Many tutorials provided by the supervisors has been followed and many references has been read to achieve this project. The main highly recommended reference is "ROS by example" book (Goebel, 2013) which helped a lot in understanding and learning the basics of ROS and achieving different kind of tasks like navigation and vision tasks that increased the knowledge of Robotic principles and expanded the deep awareness of the nature of how the robot act and work. The mentioned tutorials and the mentioned reference became the main block to start working on SMART project.

The main schema of this project could be summarized by specific prototype that the robot have to follow to navigate in different points in known map and perform different intelligent vision tasks at each point.

These tasks has been tested individually before they have been integrated to the main navigation task which is developed using SMACH. These tasks are object tracking, object detection, object recognition, making decisions based on the detected object, tag detection and fire escaping. These tasks has been integrated with the main navigation project and they has been organized in different classes to be be reusable easily at different states of the robot.

SMART project can be reused at different closed environments based on the needs and requirements of these environments. SMART project has been implemented in proposed map which has been designed based on different suggestions.

The proposed scenario in this environment. The robot navigates between four points.It performs object detection and object following at the first point.Then, the robot perform object detection at the second point. Then it navigate to the third point or to the first point based on what it detect. Then it will perform object detection at the third point and make it decision based on the object that it detect again. Finally, the robot will perform tag detection and then it will perform object following task. if the robot detect fire while it perform these tasks.

Building the map and autonomous navigation will be discussed in the

chapter 2. Vision tasks which has been implemented in SMART will be discussed in the chapter 3. State machine implementation in SMART project will be discussed in the chapter 4. Problem faced and conclusion will be discussed in the chapter 5.

Chapter 2

Navigation and Localization

In this chapter, main tasks, which has been implemented on the turtlebot to improve the project maturely, building the map and autonomous navigation will be discussed. these tasks have been the main basic block to build the entire project.

2.1 Motion Control :

Controlling the robot can be performed on different number of levels; direct control of the motor, path planning and SLAM. **Odometry** is the internal motion data of the robot like distance traveled in meters, angle rotated by radians and speed. **base controller** is a single node in ROS that combines PID controller and the driver which is controlling the robot's motor at lowest level of motion control. The base controller node publishes Odometry data on **/odom** topic and listen for motion commands on the **/cmd_vel** topic. That is why we focus on the desired linear and angular velocities in real-world units at this level of control while we are programming and it doesn't matter any more what hardware we are using for our base controller.

At next level, **/move_base** ROS package tries to move the robot to specified goal while avoiding obstacles after we determine the orientation and the position of this goal with respect to reference frame.

At higher level, using gmapping package the robot can create a map of its environment using laser scanner or depth camera. when the map is created, amcl package is used to localize the robot based on the Odometry and scan data.

2.1.1 Motion control goals :

The first step in this project has been to implement small motion tasks(goals) with different levels of control. Twist message type are used with ROS for publishing motion commands to be used by the **base controller** which translate it to motor signals. So, moving the turtle-bot forward, rotating it and moving it to the same position with same orientation have been the first goal. The thing is that the robot doesn't move back to the same position exactly because of not using the Odometry data in the script.

The Odometry message consists of header,child frame id and (TwistWithCovariance and PoseWithCovariance) sub-messages. the position and orientation are recorded in the PoseWithCovariance sub-message meanwhile linear and angular speeds are recorded in the TwistWithCovariance. Both are supported by a covariance matrix which measures uncertainty in the environment.

Then the first task has been repeated but with monitoring (in the script) the robot's position and orientation using the transform between the **/odom** and **/base_link** frames. So the result has been better. Then, Moving the robot in square task has been performed.

Then the next task has been implemented by moving the robot in square with planning the path of the robot and avoiding obstacles using **/move_base** package. avoiding obstacles while moving to the goal is one of the powerful feature of **/move_base**. Odomotry data from global and local cost map are combined by the **/base_local_planner** when the path is selected. In this task, the configuration parameters has been adjusted to fit later the proposed map. For example the **inflation_radius**, which represents the the radius of the inflated obstacles, has been reduced. The robot's path is recomputed by the local base planner to keep it away from striking obstacles.

2.2 Map Building :

Data from laser scanner and odometry are combined together by **/slam_gmapping** into recorded map. **/Slam_gmapping** is essential node in **/gmapping** package which is a ROS package for creating a map from laser scanned data. The map in ROS is nothing more than a bitmap image which has white pixels which represents the free spaces and black pixels that represents the obstacles.

Usually the robot would be driven around an area and record the laser and odometry data. Then generating the map will be achieved by the previous mentioned node against the recorded data. This node could be run with different parameters to generate different maps for the same recorded data.

The main instructions to integrate the rplidar and to build the map () has been implemented in this project. The turtlebot has been located in the area which is wanted to build its map and start the gmapping package. Then the turtlebot has been moved using keyboard slowly in the desired arena. Then the map has been saved to be used later. The Figure illustrate the final map and the real one.

2.3 Autonomous Navigation and Localization :

using the current laser scan and odometry data, **amcl** package has been used to localize the robot within the map which has been created earlier . **move_base** has been used to move the robot from one position to another one on the map.

As mentioned earlier, configuration parameters of **move_base** has been modified to fit with the environment requirements. For example, the proposed map has narrow wide path. So the **inflation_radius** in the **costmap_common_params.yaml file** has been reduced to allow the robot to pass through this path. However, it has not been reduced so much to avoid collisions with the walls of the map. The frequency of updating the map has been increased to increase the speed of detecting objects.

Four points patrolling scenario has been implemented on the tourtlebot. The starting point has been initialized by MoveBaseGoal action type using RVIZ visualizer. The thing is that the appropriate MoveBaseGoal goal, which includes the desired orientation and position in its POS part, is sent to the move_base action server when the robot moves to new goal.

Next chapter will discuss the computer vision tasks which is implemented in this project.

FIGURE 2.1: Real environment Map.

.....

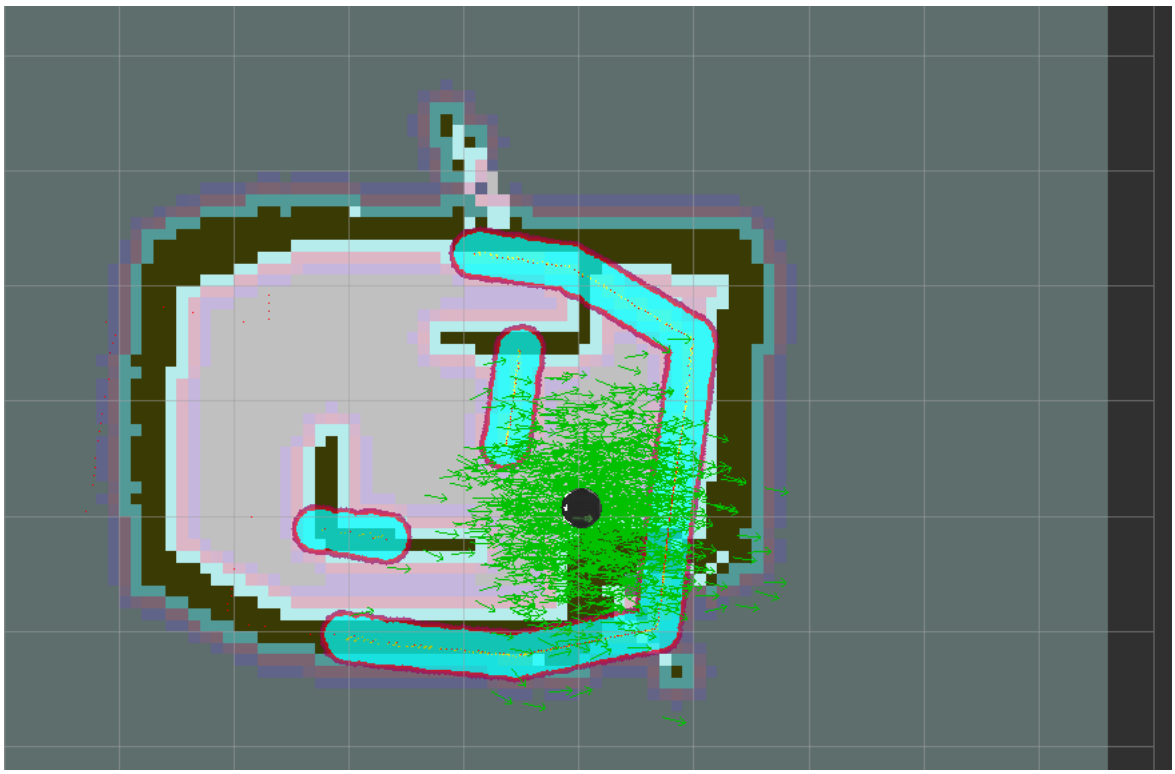

FIGURE 2.2: simulated environment Map.

Chapter 3

Computer Vision Tasks

In this chapter, Object detection, Object tracking,Object following , Object escaping and tag detection are the main three vision tasks which has been implemented in this project and they will be discussed in this chapter.

3.1 Object Detection and tracking :

OpenCV library, which is a library for 2D image processing and machine learning, has been used to implement vision tasks. **cv_bridge** has been used to convert between ROS and OpenCV image format. The thing that gives the ability to process ROS video streams. **ros2opencv.py** Utility from this reference (Goebel, 2013) has been used to subscribe to the raw image topic, to create an interface for **cv_bridge** and to create functions to process normal and depth images. **ros2opencv.py** has been modified to be launched with different region of interest topic names. The default name is **/roi**.

Running a detector to track an Object on the whole image is computationally very expensive and it could be lost at any frame. So the strategy which is followed in SMART project to track Object is :

1- convert the Object template, which is already stored on the workstation, to HSV color space, extract from its color histogram and normalize it.
2- compute probability of belonging to the object histogram for each pixel in each frame.
3- consider the probabilities which are higher than specific threshold.
4- run camshift algorithm to track the object.

Camshift algorithm is build in algorithm in OpenCV which track an object based and move the tracking window towards the area of maximum pixel density. Camshift algorithm spply mean shift first and when it converges it update the size of the tracking window.

This strategy has been implemented in **ObjectMatchDetectTrack.py** to be used later in the main project script in different sub-tasks states.

The second task was to recognize tow objects cone and cylinder. the idea is based on the recognized object the robot will choose its next state in the main project. But classification objects is very require more time and high performance workstation. The attempt to build an object classifier using Haar Feature-based Cascade Classifier has been failed. Training 60 positive

images and 200 negative images takes more than eight hours for ten stages. Training classifier needs at least 20 stages and require a lot of images. So due to the time limits, instead of classifying the two objects(cone and cylinder), detecting one of them has been implemented based on the previous mentioned strategy. if the first object has been detected, the second one will not be considered. This is implemented in **ObjectRecognizition.py**.

Detecting fire on an image has been implemented using the previous mentioned strategy by considering a fire printed image as real fire. This is implemented in **FireMatchDetectTrack.py**.

3.2 Object following and fire escaping:

One of the tasks in this project is to follow an object. The region of interested, which is published on **/ROI** topic, has been used to keep the object centered in the view domain of the robot by rotating the robot to correct the offset of the object from the center.

The depth information has been used to control forward and backward movement of the robot. This movement keeps the robot at fixed distance from the object. The utility object_follower from this reference (Goebel, 2013) has been used implemented to correct the offset and control the forward and backward robot movement.

Escaping fire is also one of the implemented task in SMART project. The depth information of the fire is not detected in the reality. So the depth information has been considered in the implementation. However, printed image fire has been used in this implementation. the region of fire has been published on **/Fire_ROI**. The fire has kept in the center of the view and the robot moves in backward direction far a way from the fire.

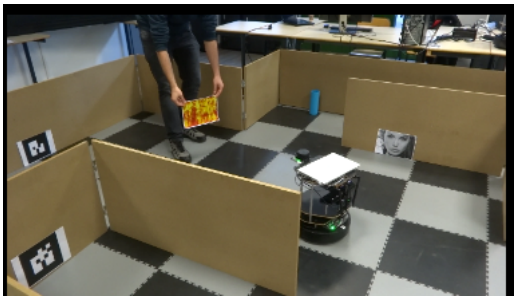

FIGURE 3.1: Fire Escaping.

3.3 Tag Detection:

Recognizing and detecting objects is a great challenge even though the computer vision algorithms are used. The next good solution to recognize objects is to tag them in some way that makes identifying process easier. **ar_track_alvar** package has been used to detect Tags. First, few Markers has been created and stored in the workstation. The **ar_track_alvar** node publishes the ID and pose of each detected marker on the **/ar_pose_maker** topic. In one of the sub-tasks state that the turtlebot goes through, Tags detection has been implemented to detect different markers.

Chapter 4

State Machine

this chapter will discuss the main project scenario which integrate robot navigation and computer vision tasks using SMACH.

Different tasks has previewed in the third chapter. To integrate these tasks to be performed depending on different conditions, the robot behaviors should be controlled. However, when the robot behaviors becomes complected, More advanced approach to organize different complicated tasks of the robot.

One of the advanced methods to integrate complex robot behaviors and to visualize them is SMACH(which is Python library). **SMACH** package has been implemented in this project to create number of state machines to handle the proposed tasks in the different locations of the proposed map. **SMACH_viewer** package has been used to visualize the scenario states.

Different states, their outcomes and how these outcomes will decide the next state of the robot have been defined in this project. Tasks has been broken down into tasks and sub-tasks using the ability of SMACH to nest state machines inside each other. Each one of these tasks, which are defined in different classes could be reused in different sub-tasks.

In SMART project four points(named A,B,C,D) has been chosen in the proposed map. The turtlebot would perform one or more vision task at each point before moving to the next point. First, the robot would move to the start point. Then, it would move to point A and if it succeeded it would detect an object and follow this object there. If not, it would move to the point B. if it finished successfully his task at point A or not, it would move to point B. (Notice that this task is inside sub-task holder and you can add as much tasks you need). Then, if the robot has succeeded to reach point B, it would perform object detection; otherwise it would move to the point C. The Objects are cone and cylinder. if the robot detected cylinder it would go back to point A. if it detected cone, it would continue to the point C. The idea is to make the robot decision based on its realization (what it recognize or detect). The interesting thing is merging the powerful of SMACH with vision algorithms and with ROS capabilities in one implementation and in smart way. Then, At point C, the robot would perform tow tasks; the first detecting one marker. if it succeeded, it would perform the second task which is detecting and following an object(this task has been used again here to preview the strength of SMACH). Finally, the last idea in the project is detecting a fire while the robot is performing its tasks in the proposed

environment. The robot would try to escape from the fire and keep itself in the safe side if it detects a fire. Fire detection task manipulates concurrently with the whole project and publish the fire region of interest on **/Fire_ROI** topic. The main part of the project works concurrently with another task which monitors this topic. If the fire is detected, the system will transit to fire escaping state. After trying escaping the fire, the robot would stop. This is implemented in **smproject.py**.

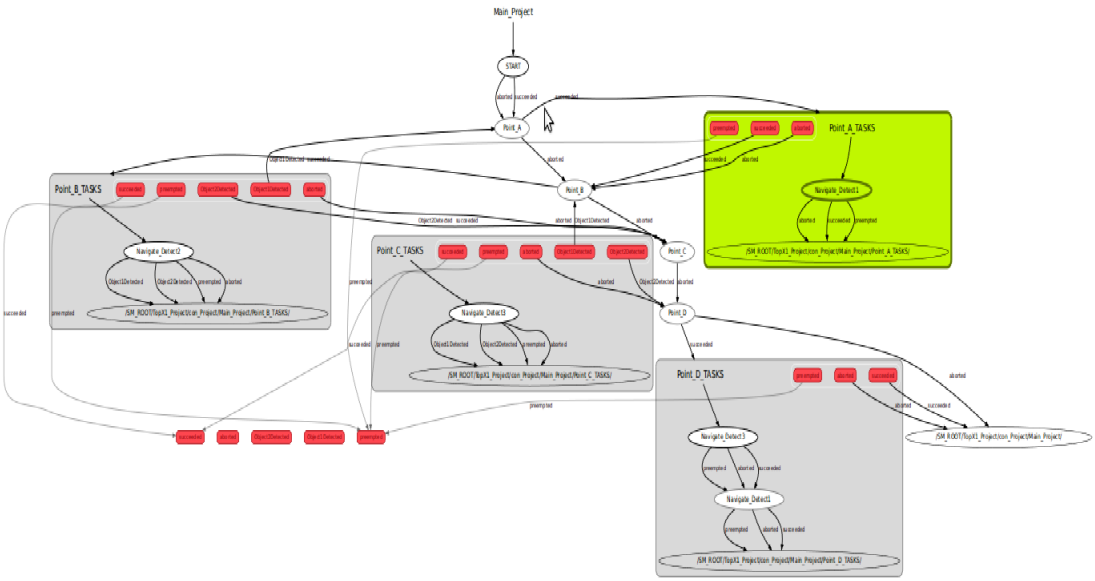

FIGURE 4.1: STATE MACHINE of the Main project.

.....

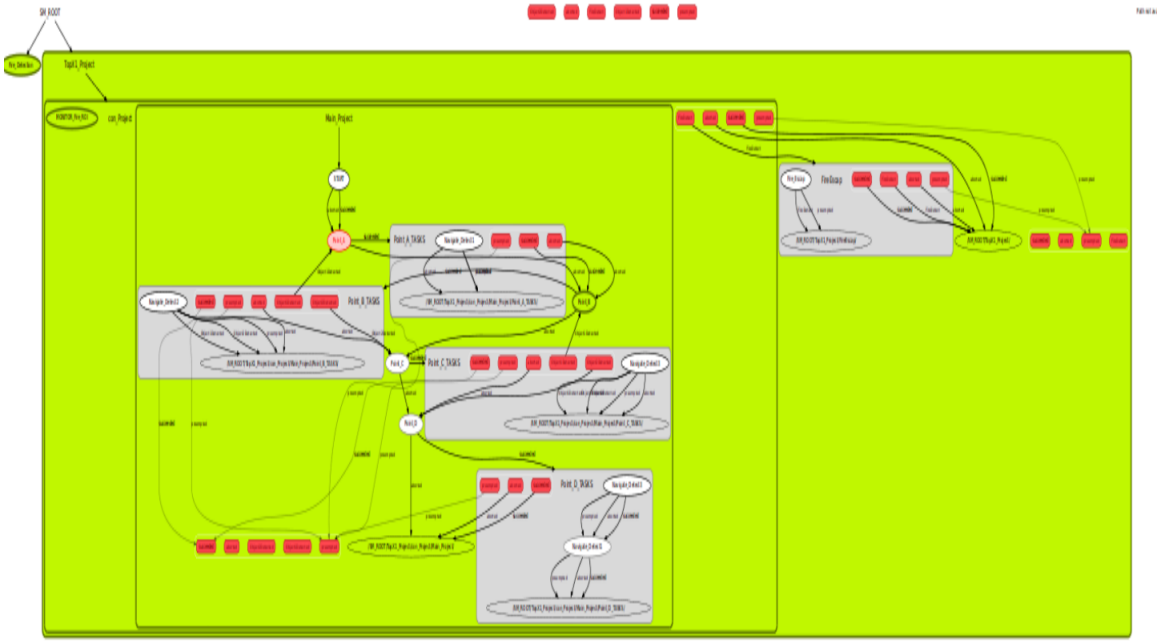

FIGURE 4.2: STATE MACHINE of the Main project with Fire escaping

Chapter 5

Problems Faced and Conclusion

This chapter will discuss some problems has been faced and small conclusion.

5.1 Problems Faced :

The First problem is navigating from one point inside the narrow path inside the map. The robot stuck when it tried to pass this path. The problem is in the obstacles inflation and it has been solved by adjusting the configurations of the move_base node. The second problem the robot stuck sometimes in the same place when it tried to navigate to another symmetric point according to wall obstacle inside the map. I think that the problem is that the wights of the global path and local path are equal in these cases. The problem solved as mentioned in chapter four.

The third problem is by building a classifier to recognize objects and make decision based on recognized object. The classifier, which has been built, failed in performing its mission. Building this classifier took a lot of time because it require high performance machine and a lot of images. This problem could be solved in the future.

The fourth problem is publishing on the same topic from different node. For example, fire detection and object detection nods needs different topics name for publishing. These nodes are inheriting from the same file that initialize the topic name /roi. The problem solved by adding string parameter to the launch file to accept different topic names.

5.2 Conclusion :

Finally, SMART project could be implemented in different environment with adding new vision tasks with, removing old vision tasks and changing the arena(the navigation positions and the map). The interesting thing in this project is not only learning different new things like robot navigation and computer vision algorithms but also integrating them togather in one single peace of art using the power of SMACH to make the entire implementation working in harmonic pretty fantastic way.

Appendix A

Source Code

One Video, which preview some results of this project, has been uploaded on Youtube... https://youtu.be/EUFQ3RmXokQ. The whole code has been uploaded on github as a public repository....https://github.com/anasm87/SMART-project.

```
#!/usr/bin/env python
"""Main ROS Project """
#import some libraries

import rospy
import smach
from smach import State, StateMachine, Concurrence, Container, Us
from smach_ros import MonitorState,SimpleActionState, Introspectio
from geometry_msgs.msg import Twist
from rbx2_tasks.task_setup import *
#import easygui
import datetime
from collections import OrderedDict
from std_msgs.msg import String

import actionlib
from actionlib import GoalStatus
from geometry_msgs.msg import Pose, PoseWithCovarianceStamped,Poi
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal, Move
from tf.transformations import quaternion_from_euler
from math import  pi

from actionlib_msgs.msg import *

#----------------------
import shlex, subprocess, os,signal
#----------------------
from sensor_msgs.msg import Image, RegionOfInterest, CameraInfo

# A list of Main Points and tasks
task_list = {'Point_A':['Navigate_Detect1'], 'Point_B':['Navigate


def setup_task_environment(self):
```

```
    # How long do we have to get to each waypoint?
    self.move_base_timeout = rospy.get_param("~move_base_timeout", 1

    # Initialize the patrol counter
    self.patrol_count = 0

    # Subscribe to the move_base action server
    self.move_base = actionlib.SimpleActionClient("move_base", Movel

    rospy.loginfo("Waiting for move_base action server...")

    # Wait up to 60 seconds for the action server to become availabl
    self.move_base.wait_for_server(rospy.Duration(60))

    rospy.loginfo("Connected to move_base action server")

    # Create a list to hold the target quaternions (orientations)
    quaternions = list()

    # First define the corner orientations as Euler angles
    euler_angles = (pi/2 , 0 , pi , 3*pi/2)

    # Then convert the angles to quaternions
    for angle in euler_angles:
        q_angle = quaternion_from_euler(0, 0, angle, axes='sxyz')
        q = Quaternion(*q_angle)
        quaternions.append(q)

    # Create a list to hold the waypoint poses
    self.waypoints = list()

    # Append each of the four waypoints to the list.
Each waypoint
    # is a pose consisting of a position and orientation in the map

     #   locations['Point1_GoalA'] = Pose(Point(0.685, 1.698, 0.000)
     #   locations['Point2_GoalB'] = Pose(Point(-1.145, 2.028, 0.000
     #   locations['Point3_GoalC'] = Pose(Point(-2.740, -0.241, 0.00
     #   locations['Point4_GoalD'] = Pose(Point(-0.467, -0.373, 0.00

    self.waypoints.append(Pose(Point(0.685, 1.698, 0.000), quaternio
    self.waypoints.append(Pose(Point(-1.145, 2.028, 0.000), quaterni
    self.waypoints.append(Pose(Point(-2.940, 1.000, 0.000), quaterni
    self.waypoints.append(Pose(Point(-0.467, -0.150, 0.000), quatern

    # Create a mapping of room names to waypoint locations
    points_locations = (('Point_A', self.waypoints[0]),
```

```
                                ('Point_B', self.waypoints[3]),
                                ('Point_C', self.waypoints[2]),
                                ('Point_D', self.waypoints[1]))

        # Store the mapping as an ordered dictionary so we can visit
        self.points_locations = OrderedDict(points_locations)


        # Publisher to manually control the robot (e.g. to stop it)
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
        rospy.loginfo("Starting Tasks")

class FireDetection(State):
    def __init__(self):
        State.__init__(self, outcomes=['succeeded','aborted','pree
        self.task = 'Fire_Escap'
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
        self.timer = 300

    def execute(self, userdata):
        rospy.loginfo('Detecting Fire............  ')

        self.FireDetect = subprocess.Popen('roslaunch srproject F

        counter = self.timer
        while counter > 0:
            if self.preempt_requested():
                self.service_preempt()
                break;
            counter -= 1
            rospy.sleep(1)

        message = "Finished Fetecting Fire............!"
        rospy.loginfo(message)
        #easygui.msgbox(message, title="Succeeded")
        os.killpg(self.FireDetect.pid, signal.SIGTERM)
        rospy.sleep(3)
        #update_task_list(self.Point_Location, self.task)

        return 'succeeded'


class FireEscap(State):
    def __init__(self):
        State.__init__(self, outcomes=['FireDetect','succeeded','

        self.task = 'Fire_Escap'
        self.timer = 40
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
```

```
    def execute(self, userdata):
        rospy.loginfo('Escaping Fire............  ')

        self.FireEscaping = subprocess.Popen('roslaunch srproject fi

        counter = self.timer
        while counter > 0:
            if self.preempt_requested():
                self.service_preempt()
                return 'preempted'
            counter -= 1
            rospy.sleep(1)

        message = "Finished Escaping Fire............!"
        rospy.loginfo(message)
        #easygui.msgbox(message, title="Succeeded")
        os.killpg(self.FireEscaping.pid, signal.SIGTERM)
        rospy.sleep(3)
        #update_task_list(self.Point_Location, self.task)

        return 'preempted'

class MatchDetectTrack(State):
    def __init__(self, Point_Location, timer):
        State.__init__(self, outcomes=['succeeded','aborted','preemp

        self.task = 'Navigate_Detect1'
        self.Point_Location = Point_Location
        self.timer = timer
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)

    def execute(self, userdata):
        rospy.loginfo('Detecting and following an Object ' + str(sel

        self.DetectionProcess = subprocess.Popen('roslaunch srprojec
        self.FollowingProcess = subprocess.Popen('roslaunch srprojec

        counter = self.timer
        while counter > 0:
            if self.preempt_requested():
                self.service_preempt()
                return 'preempted'
            counter -= 1
            rospy.sleep(1)

        message = "Finished Detecting an Object  the " + str(self.Po
        rospy.loginfo(message)
        #easygui.msgbox(message, title="Succeeded")
```

```
        os.killpg(self.FollowingProcess.pid, signal.SIGTERM)
        os.killpg(self.DetectionProcess.pid, signal.SIGTERM)
        rospy.sleep(3)
        #update_task_list(self.Point_Location, self.task)

        return 'succeeded'



class ObjectRecognizition(State):
    def __init__(self, Point_Location, timer):
        State.__init__(self, outcomes=['Object1Detected','Object2

        self.task = 'Navigate_Detect2'
        self.Point_Location = Point_Location
        self.timer = timer
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
        self.Object_sub = rospy.Subscriber("/ObjectDetected", Stri
        self.ObjectDetected = "aborted"

    def execute(self, userdata):
        rospy.loginfo('Detecting Cone or Cylinder' + str(self.Poi

        self.ObjectRecognize = subprocess.Popen('roslaunch srproj

        counter = self.timer
        while counter > 0:
            if self.preempt_requested():
                self.service_preempt()
                return 'preempted'
            counter -= 1
            rospy.sleep(1)

        message = "Finished Detecting Cone or Cylinder the " + st
        rospy.loginfo(message)
        rospy.loginfo(self.ObjectDetected)
        #easygui.msgbox(message, title="Succeeded")
        os.killpg(self.ObjectRecognize.pid, signal.SIGTERM)
        rospy.sleep(3)

        return self.ObjectDetected

    def Object_callback(self, data):
        self.ObjectDetected = data.data

class DetectTag(State):
    def __init__(self, Point_Location, timer):
        State.__init__(self, outcomes=['succeeded','aborted','pree
```

```
        self.task = 'Navigate_Detect3'
        self.Point_Location = Point_Location
        self.timer = timer
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)

    def execute(self, userdata):
        rospy.loginfo('Detecting Detecting Tag' + str(self.Point_Loc

        self.DetectTagg = subprocess.Popen('roslaunch srproject ar_i

        counter = self.timer
        while counter > 0:
            if self.preempt_requested():
                self.service_preempt()
                return 'preempted'
            counter -= 1
            rospy.sleep(1)

        message = "Finished Detecting Tag " + str(self.Point_Locatio
        rospy.loginfo(message)
        #easygui.msgbox(message, title="Succeeded")
        os.killpg(self.DetectTagg.pid, signal.SIGTERM)
        rospy.sleep(3)

        return 'succeeded'



def update_task_list(Point_Location, task):
    task_list[Point_Location].remove(task)
    if len(task_list[Point_Location]) == 0:
        del task_list[Point_Location]



class main():
    def __init__(self):
        rospy.init_node('Top_Project', anonymous=False)

        # Set the shutdown function (stop the robot)
        rospy.on_shutdown(self.shutdown)

        # Initialize a number of parameters and variables
        setup_task_environment(self)

        self.last_nav_state = None

        # A variable to hold the initial pose of the robot to be set
        # the user in RViz
```

```
initial_pose = PoseWithCovarianceStamped()
# Get the initial pose from the user
rospy.loginfo("*** Click the 2D Pose Estimate button in R
rospy.wait_for_message('initialpose', PoseWithCovarianceSt
self.last_location = Pose()
rospy.Subscriber('initialpose', PoseWithCovarianceStamped




# Make sure we have the initial pose
while initial_pose.header.stamp == "":
    rospy.sleep(1)

rospy.loginfo("Starting navigation test")


# Turn the locations into SMACH move_base action states
nav_states = {}

for Points in self.points_locations.iterkeys():
    nav_goal = MoveBaseGoal()
    nav_goal.target_pose.header.frame_id = 'map'
    nav_goal.target_pose.pose = self.points_locations[Poi
    move_base_state = SimpleActionState('move_base', Move
                                        exec_timeout=rospy
                                        server_wait_timeo
    nav_states[Points] = move_base_state

''' Create individual state machines for assigning tasks

# Create a state machine for the point_A subtask(s)
sm_Point_A = StateMachine(outcomes=['succeeded','aborted'

# Then add the subtask(s)
with sm_Point_A:
    StateMachine.add('Navigate_Detect1', MatchDetectTrack

# Create a state machine for the point_B subtask(s)
sm_Point_B = StateMachine(outcomes=['Object1Detected','Ob

# Then add the subtask(s)
with sm_Point_B:
    StateMachine.add('Navigate_Detect2', ObjectRecognizit

# Create a state machine for the point_C subtask(s)
sm_Point_C = StateMachine(outcomes=['Object1Detected','Ob
```

```
        # Then add the subtasks
        with sm_Point_C:
            StateMachine.add('Navigate_Detect3', ObjectRecognizition

        # Create a state machine for the point_D subtask(s)
        sm_Point_D = StateMachine(outcomes=['succeeded','aborted','p

        # Then add the subtasks
        with sm_Point_D:
            StateMachine.add('Navigate_Detect3', DetectTag('Point_D'
            StateMachine.add('Navigate_Detect1', MatchDetectTrack('P

        # Initialize the overall state machine
        self.sm_Main_Project = StateMachine(outcomes=['Object1Detect

        # Build the whole project state machine from the nav states
        with self.sm_Main_Project:
            StateMachine.add('START', nav_states['Point_D'], transit

            ''' Add the Point_A subtask(s) '''
            StateMachine.add('Point_A', nav_states['Point_A'], trans

            # When the tasks are done, continue on to the Point_B
            StateMachine.add('Point_A_TASKS', sm_Point_A, transition

            ''' Add the Point_B subtask(s) '''
            StateMachine.add('Point_B', nav_states['Point_B'], trans

            # When the tasks are done, continue on to the Point_C
            StateMachine.add('Point_B_TASKS', sm_Point_B, transition

            ''' Add the Point_C subtask(s) '''
            StateMachine.add('Point_C', nav_states['Point_C'], trans

            # When the tasks are done, return to the Point_D
            StateMachine.add('Point_C_TASKS', sm_Point_C, transition

            ''' Add the hallway subtask(s) '''
            StateMachine.add('Point_D', nav_states['Point_D'], trans

            # When the tasks are done, stop
            StateMachine.add('Point_D_TASKS', sm_Point_D, transition

        #-----------------------------------------------------------
        # Register a callback function to fire on state transitions
        self.sm_Main_Project.register_transition_cb(self.nav_transiti

        # Create  Concurrence container
        sm_con_Project = Concurrence(outcomes=['FireDetect','succeed
```

```
                                        default_outcome='succeedec
                                        child_termination_cb=self
                                        outcome_cb=self.concurrenc

        # Add the sm_nav machine and a battery MonitorState to the
        with sm_con_Project:
            Concurrence.add('Main_Project', self.sm_Main_Project)
            Concurrence.add('MONITOR_Fire_ROI', MonitorState("/Fire

        # Create a state machine for the point_C subtask(s)
        sm_FireEscap = StateMachine(outcomes=['FireDetect','succee

        # Then add the subtasks
        with sm_FireEscap:
            StateMachine.add('Fire_Escap', FireEscap(), transition


        sm_TopX1_Project = StateMachine(outcomes=['succeeded','ab
        with sm_TopX1_Project:
            StateMachine.add('con_Project', sm_con_Project, trans
            StateMachine.add('FireEscap', sm_FireEscap, transition

        sm_Top_Project = Concurrence(outcomes=['FireDetect','Obje
                                        default_outcome='succeedec
                                        child_termination_cb=self
                                        outcome_cb=self.concurrenc
        # Add the sm_nav machine and a battery MonitorState to the
        with sm_Top_Project:
            Concurrence.add('TopX1_Project', sm_TopX1_Project)
            Concurrence.add('Fire_Detection', FireDetection())

        # Create and start the SMACH introspection server
        intro_server = IntrospectionServer('Top_Project', sm_Top_
        intro_server.start()

        # Execute the state machine
        sm_outcome = sm_Top_Project.execute()

        #-----------------------------------------------------------

        # if len(task_list) > 0:
        #     message = "Ooops! Not all locations were completed."
        #     message += "The following Points need to be revisite
        #     message += str(task_list)
        # else:
        #     message = "All locations complete!"

        #rospy.loginfo(message)
        #easygui.msgbox(message, title="Finished tasks")
```

```
        intro_server.stop()
#-----------------------------------------------------------
    # Gets called when ANY child state terminates
    def concurrence_child_termination_cb1(self, outcome_map):
        return True
                # Gets called when ALL child states are terminated
    def concurrence_outcome_cb1(self, outcome_map):
        # If the battery is below threshold, return the 'recharge' o
        return True



    # Gets called when ANY child state terminates
    def concurrence_child_termination_cb(self, outcome_map):
        # If the current navigation task has succeeded, return True
        if outcome_map['Main_Project'] == 'succeeded':
            return True
        # If the MonitorState state returns False (invalid), store t
        if outcome_map['MONITOR_Fire_ROI'] == 'invalid':
            rospy.loginfo("Fire! Need TO Escape...")
            if self.last_nav_state is not None:
                self.sm_Main_Project.set_initial_state(self.last_nav
            return True
        else:
            return False

    # Gets called when ALL child states are terminated
    def concurrence_outcome_cb(self, outcome_map):
        # If the battery is below threshold, return the 'recharge' o
        if outcome_map['MONITOR_Fire_ROI'] == 'invalid':
            return 'FireDetect'


    def FireEscape_cb(self, userdata, msg):
        counter = 5
        counter1=0
        while counter > 0:
            if msg.width <= 30 and msg.height <= 30:
                counter1 +=1
            counter -= 1
            rospy.sleep(1)

        if counter1==5:
            #rospy.loginfo("Hello1....")
            return True
        else:
            #rospy.loginfo("Hello2....")
            return False
```

```
    def nav_transition_cb(self, userdata, active_states, *cb_args
        self.last_nav_state = active_states
#-----------------------------------------------------------------
    def update_initial_pose(self, initial_pose):
        self.initial_pose = initial_pose

    def move_base_result_cb(self, userdata, status, result):
        if status == actionlib.GoalStatus.SUCCEEDED:
            pass

    def cleaning_task_cb(self, userdata):
        rooms_to_clean.remove(userdata.Point_Location)

    def shutdown(self):
        os.killpg(self.FireDetect.pid, signal.SIGTERM)
        rospy.loginfo("Stopping the robot...")
        #sm_nav.request_preempt()
        self.cmd_vel_pub.publish(Twist())
        rospy.sleep(1)



if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        rospy.loginfo("The Main Project test finished.")
```

Bibliography

Goebel, Patrick (2013). "ROS By Example". In: ed. by Lulu. URL: \url{http://www.lulu.com/shop/r-patrick-goebel/ros-by-example-hydro-volume-1/ebook/product-21393108.html}.

Robotics Laboratory, Computer Vision for. In: URL: \url{http://link.aip.org/link/?RSI/62/1/1}.