UNIVERSITÉ DE BOURGOGNE

MASTER OF COMPUTER VISION - ROBOTIC PROJECT

KURAbot
Kura-kura Restaurant Assistant Robot

Authors:

Dennis CHRISTIE

Osama MAZHAR

Omid MORADTALAB

Supervisor:

Ralph SEULIN

Cansen JIANG

Raphael DUVERNE

Robotic Research Group

Centre Universitaire Condorcet - uB, Le Creusot

Dec 2015

UNIVERSITÉ DE BOURGOGNE

Abstract

MSCV

Centre Universitaire Condorcet - uB, Le Creusot

MSCV - ROBOTIC Project

KURAbot - Kura-kura Restaurant Assistant Robot

by Authors

The purpose of this report is to introduce frame works of basic theoretical and practical parts of Turtlebot by using Linux, ROS. Here we will have complete consideration of robot navigation, mapping, localization, control and computer vision using ROS. To achieve this purpose we will consider many details of mentioned robot and a plan will be introduced to robot to do the scenario. We will consider this robot as Kitchen Assistance to deliver food from kitchen to customer. In this report we will pay to discuss more detail about how we implement the project and knowledge behind every steps from making map and localization to detect the goal and deliver the task.

Contents

List of Figures

Chapter 1

Introduction

Robotic technology is playing fundamental role in future life. Here, we want to consider knowledge behind of robotic system based on ROS which is theoretical and practical work on Robotic Lab. To achieve this purpose, we implement special plan and scenarios on the robot to do predefined tasks. The Robot Operating System (ROS) is a framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms [1].

We named the Turtlebot as KURAbot, that name came from Indonesian word for turtle, kura-kura. KURAbot is an abbreviation of Kura-Kura Restaurant Assistant and its task is to deliver food from the kitchen to the the customers. It can be used in automatic restaurants and such places which someone needs to give the costumers order.

In this practical work we used different references like ROSwiki [1] tutorials and ROS by Examples under hydro distribution [7]. After learning the Professor Lectures and tutorials of basic practical of ROS, Linux and Turtlebot details we worked on implementation step by step after testing during the lab sessions and then we did Robotic Project Lab under our professor supervision and thanks a lot to helpful lectures and supervision of our professors.

The first Idea for our robot task came from automatic resturant idea to make a robot assistant in the kitchen to deliver food for costumers. And then we start to make a plan to do such tasks by KURAbot. After careful

consideration and many examples and tests, the idea finally became the reality.

The abilities of KURAbot are obstacle avoidance, Motion control, automatic navigation and localization, tag recognition, automatic noise removal, voice indication according to each scenarios and recognize the step to take order for costumer. So, it will introduce itself and start running to arrive goal. Then, it will start speaking to take order and if it is successful, it will come back to kitchen to bring the food and so on. When it faced with noisy environment, we suggest new scenarios to make clear the noise during. Each of AR Tags contains an id, so KURAbot can differentiate them. Since we have 6 points as customer table, and another 1 AR tag for indication of the table, we used 7 different AR Tags. We have two checkpoint location and when robot detect that the map is noisy it will come back to these points and by rotation it will clear the noisy map (by using laser scan).

In next step we will consider more details of robot working and overall structure of implementation, mapping, navigation, recognition, state machine and so on. We will consider all things behind algorithms from creating the map to detection the goal and finish the robot mission. Also, we will suggest some methods to make robot work better. The following chapters will be considered for our report:

- Chapter 1: Introduction to the basic topic
- Chapter 2: General Concept of ROS
- Chapter 3: Setting Up
- Chapter 4: Implementation
- Chapter 5: Problems Faced
- Chapter 6: Conclusion

Chapter 2

General Concept of ROS

2.1 What is ROS

The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms [1]. ROS system is comprised of a number of independent nodes, each of which communicates with the other nodes using a publish/subscribe messaging model [7]. For example, a particular sensor's driver might be implemented as a node, which publishes sensor data in a stream of messages. These messages could be consumed by any number of other nodes, including filters, loggers, and also higher-level systems such as guidance, path finding, etc.

2.2 Why ROS

Creating truly robust general-purpose robot software is hard. From the robot's perspective, problems that seem trivial to humans often vary wildly between instances of tasks and environments. Dealing with these variations is so hard that no single individual, laboratory, or institution can hope to do it on their own. As a result, ROS was built from the ground up to encourage collaborative robotics software development [1].

Moreover, with the great concept of nodes, the tasks can be done in different system. ROS nodes allow user to communicate with any sensor drivers even if it is not in the same system, as long as they are connected in

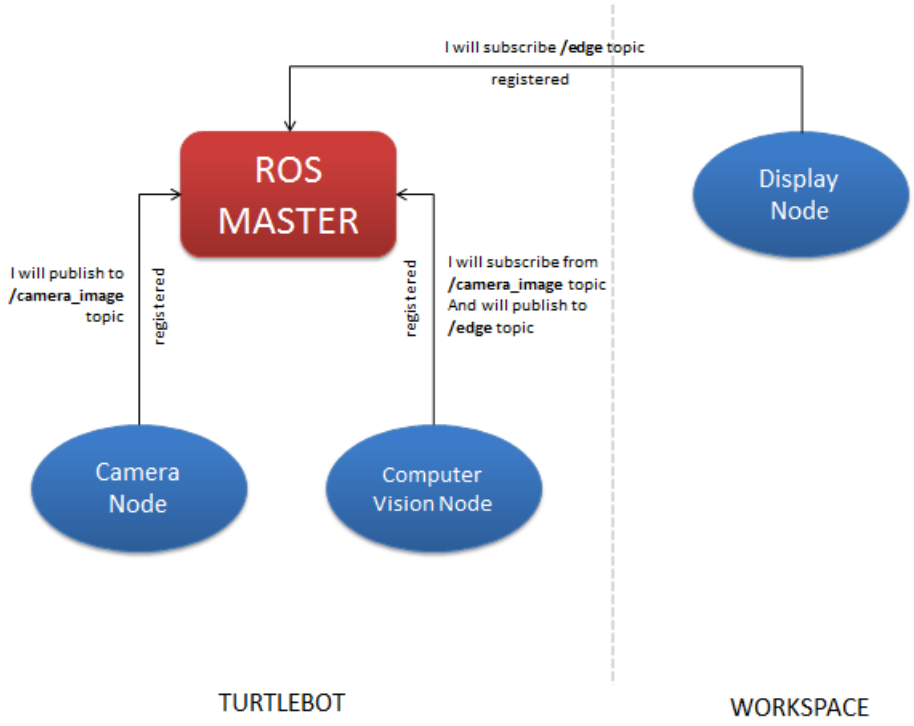

FIGURE 2.1: Register Nodes to ROS Master

the same wireless connection. This makes ROS really flexible and adaptable to the needs of the users.

2.3 How ROS Works

ROS Start with ROS Master. The ROS Master provides naming and registration services to the rest of the nodes in the ROS system. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer [7]. To make easier understanding, just consider ROS Master has somekind of lookup table where all the nodes go to find where exactly to send messages.

For example, in Figure 2.1, we create three nodes, there are Camera Node, Computer Vision Node, and Display Node. Camera Node will communicate with camera, Computer Vision node will process the image that come from Camera Node, Display Node will display image that has been processed by Computer Vision Node

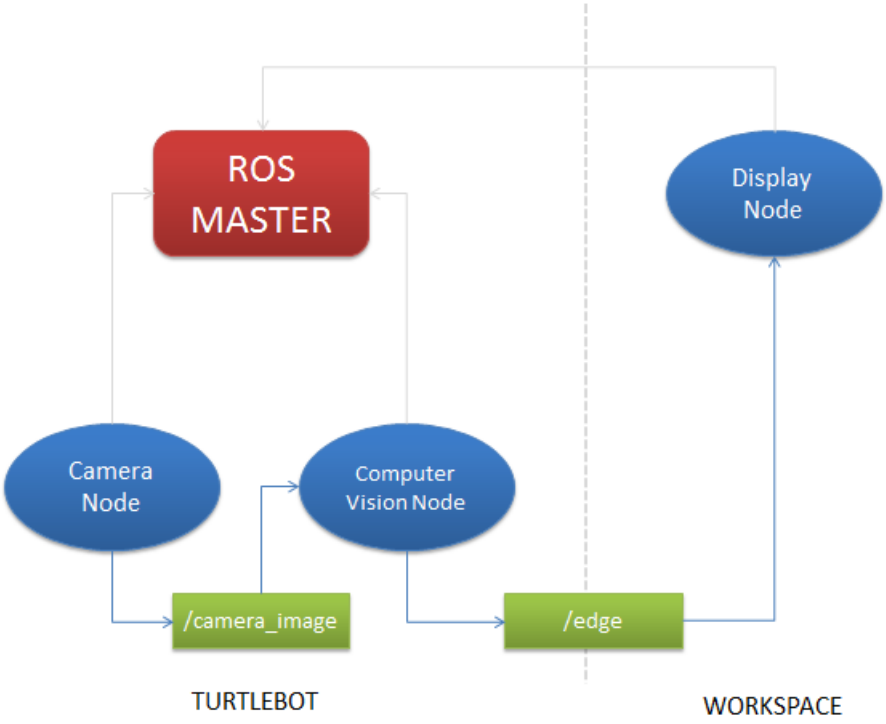

FIGURE 2.2: Publisher and Subscriber Nodes

All the nodes registered with ROS Master also with its behavior for being publisher or subscriber (Figure 2.2). Thus, once the Camera Node receives some data from the Camera, Camera Node will publish a camera stream data message called /camera_image, Computer Vision Node will subscribe that message and will publish a /edge message, Display Node will subscribe /edge message, and do stuffs whatever user want to do (through what is essentially TCP/IP).

Chapter 3

Setting Up

3.1 Motion Control

Motion Control are the core of this project. KURAbot must has a procedure to make it move as we want. There are a lot of ways to make the KURAbot move from one place to another. In this section several ways are introduced, and in the end of this section the solution provided.

3.1.1 Motion Only with Linear and Angular Velocity

ROS using twist message type for publishing motion commands to be used by the base controller (it is usually called /cmd_vel). From ROS documentation, twist message contains two submessages that is Vector3. Each of sub message contains three element: x, y, and z.

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

LISTING 3.1: Twist Message

Twist message basically contains information about linear velocity and angular velocity in x, y, and z axis of turtlebot. Since turtlebot doesn't have omni-direction wheel, we can only use x axis for linear velocity. And also,

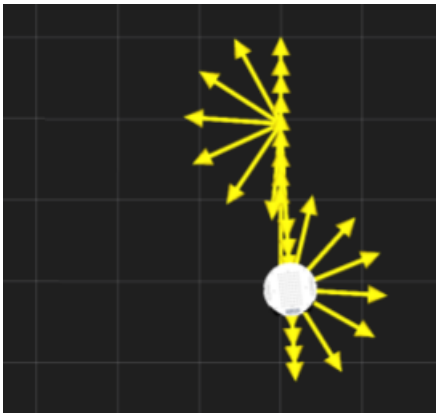

FIGURE 3.1:
Motion RVIZ

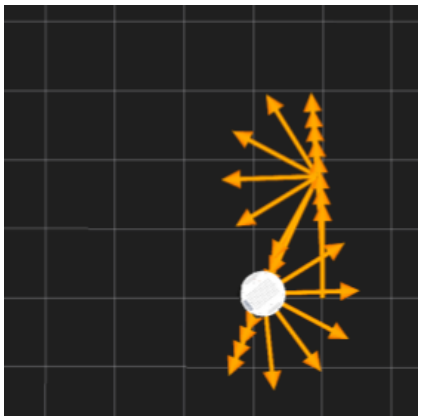

FIGURE 3.2:
Motion Real

since turtlebot is a static ground robot (not a flying robot), we can only use z axis for angular velocity.

In the first motion command, we can simply do a basic calculation to make the turtlebot move from one point to another. With specified distance and linear velocity or specified angle and angular velocity, we can obtain amount of time we should publish this message. Turtlebot will move or rotate within this amount of time.

This way of motion control do not guarantee that the robot will go 1 meter forward, nor 180 rotation. Due to external condition, (such as the wheel, the area plane, and another disturbance) turtlebot sometimes not really going at the speed that we already specified. Since this way of motion control only depends on time, there is no guarantee that the turtlebot will go to the place we desired.

If we command the turtlebot to take a simple navigation by executing odom_out_and_back.py from rbx1_nav package that provided in ROS by Example Tutorials, the result shown in Figure 3.1 and Figure 3.2. Figure 3.1 shows how motion was really good in simulation with RVIZ, but when implemented in turtlebot, it is completely different, Figure 3.2 shows turtlebot can not return to its original position.

3.1.2 Motion Control with Odometry

Odometry is the use of data from motion sensors to estimate change in position over time. Odometry is used by some robots to estimate (not determine) their position relative to a starting location. From ROS documentation, odometry message contains a Header, a string identifying the child_frame_id, and two sub-messages, one for PoseWithCovariance and one for TwistWithCovariance.

```
Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

LISTING 3.2: Odometry Message

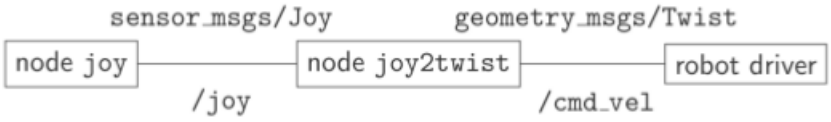

FIGURE 3.3: Joy2twist Node Graph [4]

The PoseWithCovariance sub-message records the position and orientation of the robot while the TwistWithCovariance component gives us the linear and angular speeds as we have already seen. Both the pose and twist can be supplemented with a covariance matrix which measures the uncertainty in the various measurements [5].

Before, we publish Twist message to /cmd_vel topic by only using time as reference (it is just like guessing distances and angles based on time and speed). To make a better result of turtlebot's translation or rotation, we use our previous pose and compare it to the pose we desired, we will keep publish twist message to /cmd_vel topic until our previous pose reach our desired position.

This way of motion control only can be performed by using odometry. If we command the turtlebot to take a simple navigation by executing nav_square.py from rbx1_nav package that provided in ROS by Example Tutorials, the turtlebot will move in a square shape and return to original position (which is almost impossible to do if we don't use odometry)

3.1.3 Motion Control with JoyStick

There is also another way to control turtlebot motion by manually. Manually here means user can control turtlebot in runtime by using another device, such as joystick. Figure 3.3 shows the general idea to do this motion control. We have to provide a node that will handle communication with joystick and publish a message in respect to linear and angular velocity message if there is some event from the joystick (when user press the button).

```
1 ...
2 def __init__(self):
```

```
  rospy.init_node('joy2twist_node', anonymous=False)
  self.cmd_vel = rospy.Publisher('/cmd_vel', Twist)
  rospy.Subscriber("/joy", Joy, self.callback_function)
  rospy.spin()

def callback_function(self, data):
  msg = Twist()

  # left analog button for forward-backward
  msg.linear.x = data.axes[1]
  # left analog button for left-right
  msg.angular.z = data.axes[0]

  self.cmd_vel.publish(msg)
...
```

LISTING 3.3: Joy2Twist

In order to do that, we created a node that will subscribe to /joy topic, convert joystick message to twist message, and publish it to /cmd_vel topic. After we run this program, turtlebot will start moving if user move the left analog button.

3.1.4 Motion Control with Path Planning (move_base)

Subsection 3.1.1, subsection 3.1.2, and subsection 3.1.3 basically can control the motion of turtlebot, but none of them can handle an obstacle. It is mandatory to consider about obstacle, because in real life, walls, tables, chairs, are an obstacle that our robot should not hit one of them.

ROS provides a much more elegant way to do the same thing using the move_base package [5]. Instead of using distance or angle as reference to do a motion, point to point movement was achieved by using the move_base package. This package helps in moving the turtlebot to any desired position only by specifying coordinate with (0,0) for its origin position. Since this motion control based on coordinates, it will helps users a lot, because we will not concern anymore about the path, move_base package itself will handle this, which is called the Path Planning.

The move_base package uses the MoveBaseActionGoal message type for specifying the goal [5]. From ROS Documentation [8], MoveBaseActionGoal message contains a lot of things to fill, but in practice, we only need few of the fields like the position and orientation.

```
Header header
  uint32 seq
  time stamp
  string frame_id
actionlib_msgs/GoalID goal_id
  time stamp
  string id
move_base_msgs/MoveBaseGoal goal
  geometry_msgs/PoseStamped target_pose
    Header header
      uint32 seq
      time stamp
      string frame_id
    geometry_msgs/Pose pose
      geometry_msgs/Point position
        float64 x
        float64 y
        float64 z
      geometry_msgs/Quaternion orientation
        float64 x
        float64 y
        float64 z
        float64 w
```

LISTING 3.4: Joy2Twist

The move_base node requires four configuration files before it can be run. We need to specify a number related to how move_base will behave such as radius of robot, tolerance of goal, maximum and minimum of velocity value, and others. We will discuss this in section 3.4

3.1.5 Conclution of Motion Control

There are 4 ways to do a motion control, Motion Control only with Linear and Angular Velocity (subsection 3.1.1), Motion Control with Odometry (subsection 3.1.2), Motion Control with Joystick (subsection 3.1.3) and Motion Control with Path Planning or move_base (subsection 3.1.4). Every motion control basically can be used depends on the situation, if we want a full control of the turtlebot, Motion Control with Joystick will be used, if we want to move by usind coordinates, Motion Control with Path Planning will be used, if we want to make an extraordinary moves (not like navigating) Motion Control with Odometry will be used. Every motion control except the first one, because the error is beyond the tolerance.

3.2 Map Building

Turtlebot will run in indoor environment. This situation leads that turtlebot supposed to able to avoid obstacles (such as walls). By motion control with move_base, now turtlebot is able to avoid obstacles, the problem arise because turtlebot do not know what are the obstacles unless we specifiy it. Turtlebot have to know first what is around so that it can aware to the obstacles. To make the turtlebot know the environment around, we have to give an information about the map.

There are two solution to do this. First, we can create the map manually by looking at the real scene and sketch it in any graphics program by ourself, and second, we let the turtlebot to build its own map. Since the turtlebot already equipped with a laser scan, it is better to let the turtlebot create its own map. It will be more accurate than we create it by ourself by looking towards the real scene (if the users are not measuring the map appropriately, it will leads to miscalculation obstacles by the turtlebot).

Building the map can be perform by using gmapping package that provided by ROS. Gmapping package is a third party package that developed by Brian Gerkey [4]. The gmapping package provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called

FIGURE 3.4: Arena that built by MSCV6

slam_gmapping. This node does the work of combining the data from laser scans and odometry into an occupancy map. GMapping is a highly efficient Rao-Blackwellized particle filer to learn grid maps from laser range data [9]. There are several procedure to perform the map building

1. First we put our robot in the map. The place of the robot will be the origin of the map (coordinate 0,0), and also turtlebot initialization point that later use of the map. We can change the initial pose if we need it

2. Launch every launch files and node needed, there are rplidar and gmapping launch file for turtlebot notebook, RVIZ visualization and joy2twist node (subsection 3.1.3) for workstation PC. Appendix A shows every command that executed in terminal. RVIZ will help us to interact with the process result of gmapping.

3. Teleoperate the turtlebot around the area manually by the joystick. Teleoperating needs to be careful, sometimes if the turtlebot move to fast, the map will not generated corectly.

4. After every spot in the area generated into a map, then we save the map by using map_saver node. This will create two files, our map in pgm format, and configuration file of the map in yaml format. These

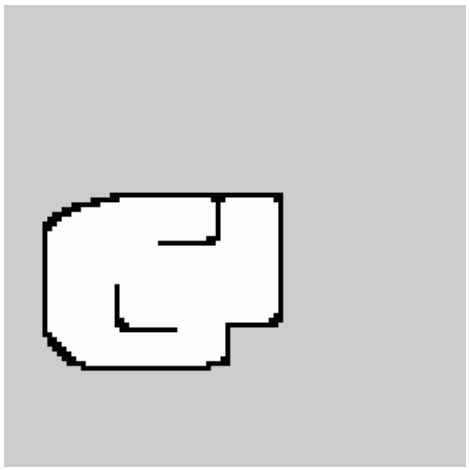

FIGURE 3.5: Generated Map

pairs of files should be together in the same directory, and will be used for Navigation and Localization, that will discussed more detail in section 3.3

5. Sometimes the map contains noises or errors. We used graphics program (such as GIMP), to remove these noises and errors.

Figure 3.4 is the real arena that built by MSCV6 and Figure 3.5 is the generated map that used in this project. As shown in Figure 3.5, the map is shifted, not centered, this is caused by our starting point when we were creating the map. The center of the picture shows where we started.

3.3 Navigation and Localization

Until this point, the turtlebot can move to one point to another, know its position relative to the origin position, avoid obstacles (such as walls) in the map, but turtlebot still can not determine where does its location in the map. This process called localization, knowing where we are in the known map.

In the real world, human can know his/her location in the map by looking a sign in the street, or another feature like houses, interesting statues. Robots does something similar, they use features around them to localize

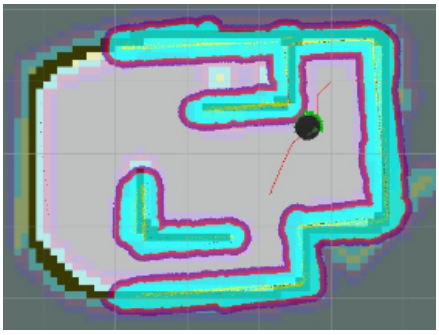

FIGURE 3.6:
Navigation 1

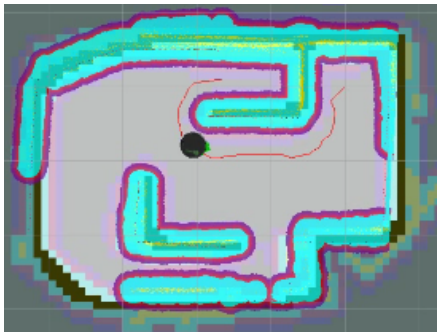

FIGURE 3.7:
Navigation 2

themself in the map [15]. There are two ways to collect this features, we can use depth camera by Kinect or Laser Scan.

AMCL is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach, which uses a particle filter to track the pose of a robot against a known map [2]. Each possible locaion is represented by a particle, and particle that do not match well to map are removed, which is resulting only a group of particles representing the location of the robot in the map. ROS uses the amcl package to localize the robot within an existing map using the current scan data coming from the robot's laser or depth camera [5].

Motion control by move_base and AMCL with generated map are best combination, makes the turtlebot accurately moving from one location to another. Figure 3.6 and Figure 3.7 shows how the turtlebot know its location in the known maps (section 3.2)

3.4 Configurations

As already mentioned in subsection 3.1.4, motion control with move_base requires four configuration files before it can be run. We have to give a spesific value to some of the parameters, so that the turtlebot can bahave as we desired.

3.4.1 base_local_planner_params.yaml

We saw our behavior of our turtlebot. When we commanded it to move to certain pose in the map, sometimes it hardly to got the exact location point. When the turtlebot arrive at the location area that we specified, the turtlebot tried to move forward, backward, rotate, etc. This is due to little tolerance specified. Turtlebot tried to make an exact orientation and an exact position point.

In order to remove this behavior, we increase parameter of the tolerances. yaw_goal_tolerance and xy_goal_tolerance from 0.1 into 0.25 (around 15 degrees of yaw tolerance, and around 25 cm of coordinate tolerance). Since we increase the tolerance, that means bigger possibility that the turtlebot is not going to the exact position and orientation, but 15 degrees of yaw tolerance and 25 cm of coordinate tolerance is not big deal for our project purpose, so we choose this value for the tolerances.

3.4.2 costmap_common_params.yaml

Another behavior of our turtlebot is, sometimes it can not go to certain pose in the map due to radius of the obstacle. Figure 3.8 shows sometimes amcl leaves noisy particles in certain area. This situation will make our turtlebot can not go the the yellow circled area, due to radius of the obstacle.

We tried to decrease inflation_radius parameter a little bit from 3.0 to 2.5, but it is not giving any good result. Then we tried to decrease raytrace_range parameter from 3.0 into 2.0, it will decrease the range of laser scan as shown in Figure 3.9, sometimes the range smaller than in the picture. At the beginning we thought this solution is good because we will not have noisy map in the far distance, then we realize that if we decrease the raytrace, that means we will decrease the particles obtained, and it will leads to miscalculation of localization by amcl. So at last we revert back to default value, but we solve this problem in another way, that will discussed more detail in Chapter 4.

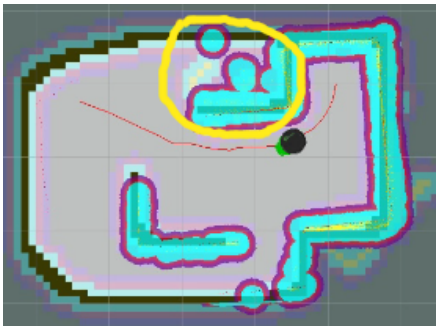

FIGURE 3.8:
Noisy Map

FIGURE 3.9:
Low Raytrace

3.4.3 global and local_costmap_params.yaml

We leave every value in this configuration files as default values. We did multiple time of testing and thought that every value that specified in this files already good enough.

Chapter 4

Implementation

4.1 Scenario

As stated earlier, KURAbot is designed such that it helps the restaurant representatives to deliver food to their customers. The scenario for the operation of KURAbot mainly includes a predefined map that is already scanned by the LiDAR laser sensor and considered "known" to the robot. The serving tables for the customers are assumed to be placed in fixed positions namely table1, table2 until table6 in our case.

The robot starts from the middle of the map, initialize its position and orientation with respect the known map and align itself with the data coming from LiDAR while it goes to the predefined position near the chefś counter. The robot waits for the order until chef puts the food on the tray

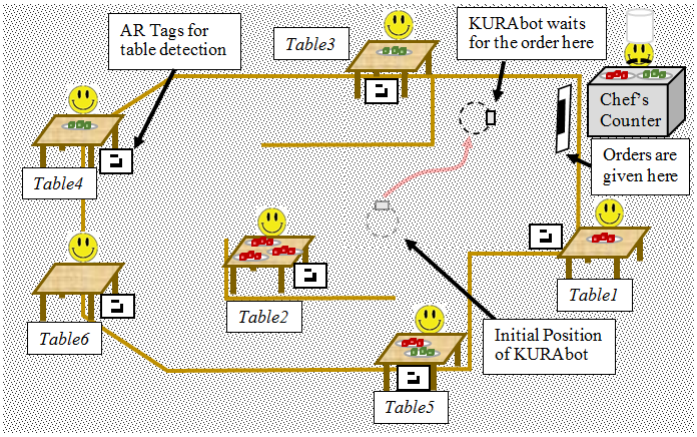

FIGURE 4.1: Overall Scenario of KURAbot Design

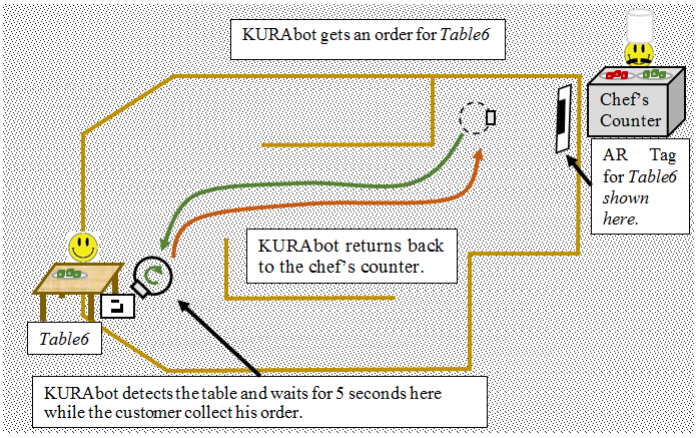

FIGURE 4.2: Order Delivery Task with No Temporary Obstacles

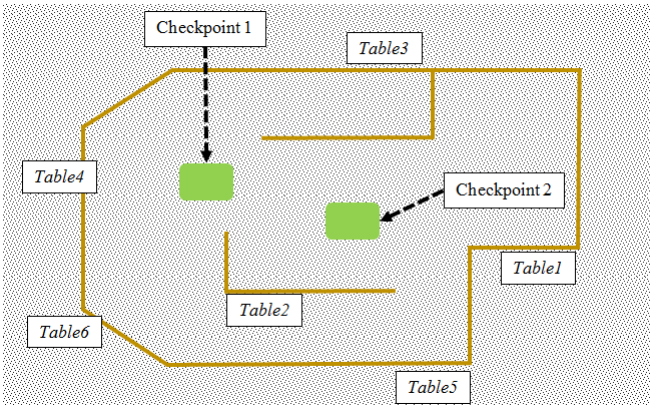

FIGURE 4.3: Position of Checkpoints

over the robot and assert a command by showing a specified AR Tag ranging from one to six corresponding to the table number. KURAbot detects the tag and starts moving towards the corresponding table. The pose for each table (waypoint) is already defined with the specified position and orientation. As soon as the robot reach the table, it starts rotating until it detects a table with another unique AR Tag or the rotation exceeds 360 degrees.

When the table is detected the robot beeps a sound to indicate the detection, wait until 5 seconds so that the customers pick their order, thanks the customer with good wish and returns to the kitchen/chefś counter again. If the robot could not detect the table, it goes back to the chef's counter without any delay.

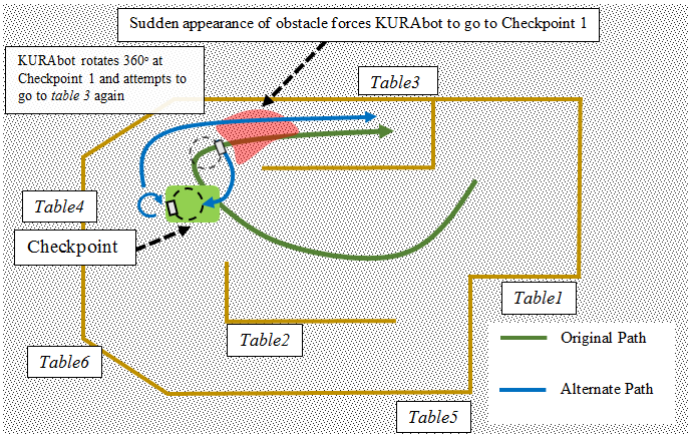

FIGURE 4.4: Order Delivery Task with Temporary Obstacles

The design of KURAbot inherently takes care of the fixed obstacles (walls in the arena) while estimating its trajectory to move towards the specified waypoint/table. However if the robot detects any other obstacle on its way to the service, it is still capable to deal with them. For this purpose two checkpoints are defined in the map as shown in Figure 4.1.

If the robot detects an obstacle it adopts another route towards one of the nearest checkpoint, rotates for 360 degree to scan and wait until the obstacle has moved, it goes back to its desired destination to serve the order. If the obstacle do not move while the robot has come for the second time, the robot will move back to the chef's counter while raising an unsuccessful flag signal.

In order to make the provision of service robust, a series of events are defined in the algorithm. KURAbot moves from one checkpoint to the other and back to the first checkpoint while it continuously scans for the obstacles to make sure that the service is successfully completed before it returns to the chef.

4.2 Robot Tasks

In order to become a good restaurant assistant, KURAbot supposed to be able to do, at least, three general task. These task are, Navigation, Tags Detection, and Voice Indication. There are a lot more details when performing

each task. These details will explained as follows

4.2.1 Navigates

First main task of restaurant assistant is able to deliver food from the kitchen to the the customers. It is mandatory for KURAbot to be able to navigates to points in the map. With map provided (section 3.2), motion control with move_base (section 3.1.4), localization with AMCL (section 3.3), and tuned configuration files (section 3.4), navigation task become an easy task to do. With the existing scenario, KURAbot will do the navigation task perfectly as we desired.

It has been mentioned in section 4.1, we already specified there are one kitchen room and six table of the costumers, and two checkpoint location. These two checkpoint location were made in order to solve the problem that arise when there are noises in the map (subsection 3.4.2). In order to avoid cancellation of navigation to certain point that caused by noises in the map, some addition points called checkpoints are used, so that KURAbot has higher possibility to arrive at the point we desired from those checkpoints rather than from the origin of the position, kitchen.

All the navigation task considered as a state, and being called by a state machine. The details of navigation with state machine will be discussed more detail in subsection 4.3.4.

4.2.2 AR Tags Detection

Second main task of restaurant assistant is able to recognize the delivery order and adjust its position towards the customer table. After the chef done with the food, he will put the food on the top of KURAbot and give a command to go to the table of the customer who order that food. When KURAbot arrive to the table, it supposed to be able to face the customers for better service.

AR Tags can be used as delivery order and table indicator. Each of AR Tags contains an id, so KURAbot can differentiate them. Since we have 6

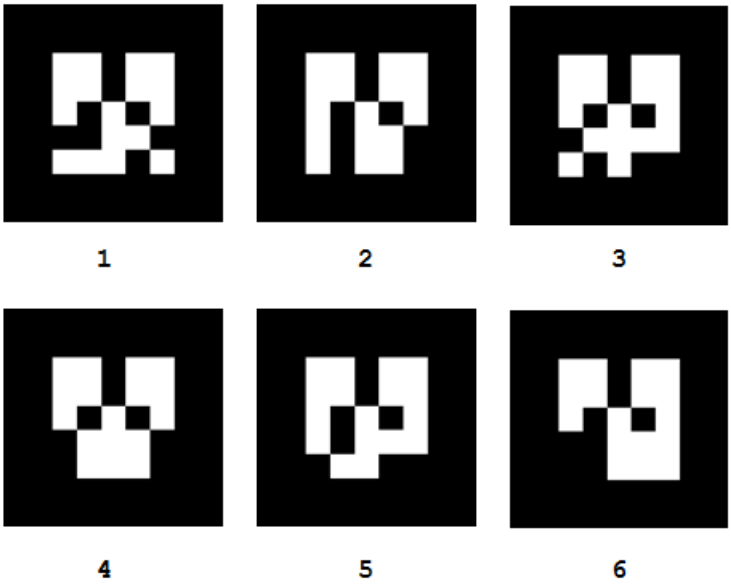

FIGURE 4.5: Marker from 1 until 6

points as customer table, and another 1 AR tag for indication of the table, we used 7 different AR Tags.

ROS provide ar_track_alvar package that will handle AR Tags detection. ar_track_alvar has several functionalities, two of them are generating AR tags of varying size, resolution, and data/ID encoding and identifying and tracking the pose of individual AR tags [3]. ar_track_alvar package can generate an AR Tags which has id from 0 until 65535.

AR Tags for Delivery Order

Since we only have six customer table, we generated 6 AR Tags from 1 until 6. Appendix A.2 shows command that we used to generate the AR Tags. We generate an AR Tags which has an ID number 0 that has a size 30 cm x 30 cm. Larger tags are useful for navigation and localization since they are more easily recognized from a greater distance. Smaller tags can be used to label objects that will be viewed at a closer range [5]. Figure are the tags that considered to become a delivery order from table 1 to table 6

AR Tags for Table Indicator

Similar to AR Tags for delivery order, we generated one more tags, which has ID number 7, and will consider this last AR tag as a Table. We attached this AR tag in each of the table location. So we have 6 AR tags which has ID number 7.

There is minor problem when we tried to implement this scenario. Let's consider there is delivery order to table number 1, KURAbot will navigate itself to location number one. Once it is arrived, now it tried to face itself to the table (AR tag number 7), problem arise when sometimes it can detect the table in another table location, for example location table 2, since we also use AR tag number 7 as table indicator number 2. KURAbot will end up facing that direction instead of facing to table number 1.

We discussed to solve this problem. The easiest solution is generating another AR Tags, so that each position has its own table indicator. The drawbacks of this solution is we will have a lot of AR Tags IDs. It is better if we just use 1 AR tags for table indicator.

Fortunately, ar_track_alvar node publishes AlvarMarkers message that contains not only the ID, but also the pose of each detected marker on the /ar_pose_maker topic. From ROS Documentation [10], details of AlvarMarkers message listed as follows.

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 id
uint32 confidence
geometry_msgs/PoseStamped pose
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
```

```
15      float64 y
16      float64 z
17    geometry_msgs/Quaternion orientation
18      float64 x
19      float64 y
20      float64 z
21      float64 w
```

LISTING 4.1: AlvarMarkers message

With position of AR Tags provided, we can use this information to differentiate table indicator AR tags from different customer table. We assumed that once KURAbot arrive to the location, the table must be near KURAbot right now. With the information of x position from AlvarMarker message (which is the distance of the marker relative to the camera) we can set the treshold, so that AR Tags from far distance will not be considered as the table in this delivery order task.

All the AR detection task considered as a state, and being called by a state machine. Details of how KURAbot manage to detect the delivery order and adjust itself to facing towards the table will discussed more detail in subsection 4.3.5.

4.2.3 Voice Control

The third main task of restaurant assistant is able to give voice message. It is not mandatory, but as a good restaurant assistant, KURAbot suppose to give a voice indicator what tasks that is performed right now.

ROS provide sound_play package that provides a ROS node that translates commands on a ROS topic into sounds. The node supports built-in sounds, playing OGG/WAV files, and doing speech synthesis via festival [11]. To synthesize a voice, we can simply use say function that already provided by sound_play class. It will give a result of (likely terrible) robot sound. We want to make KURAbot become user friendly that has friendly sound, so instead of using sound from the system, we record our own voices.

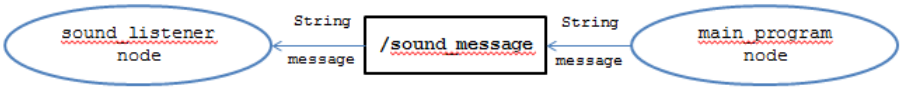

FIGURE 4.6: Voice Control Graph

We made a complete list of voices that will be used by KURAbot, the list is provided in Appendix A.3. Sound files are in WAV format and stored in KURAbot notebook.

To make a voice control, we can use a simple program called talker and listener [12] and modify it a little bit, as described in Figure 4.6. We published some strings from workstation PC to KURAbot notebook depend on task that is performed right now to a topic called /sound_message. In KURAbot side, we created a node that subscribed string message from /sound_message topic, with a condition specified, KURAbot will play sound file correspond to the strings obtained.

4.3 State Machine

Sometimes, when developing a robot, the programs will become complicated and more complicated, especially when robot have do do a difficult task. If the developer still using regular programming methods, the code will become a mess. More complex the task, more complex the code, more difficult to continue the developing process. Therefore, another method of programming supposed to be invented, a method that can manage difficult task, divide them into small-simple-pieces of task. There comes state machine.

SMACH or State Machine usually used in order to help user to perform complicated tasks. SMACH syntax makes it easy to quickly prototype a state machine and start running it. SMACH allows you to design, maintain and debug large, complex hierarchical state machines. SMACH also gives you full introspection in your state machines, state transitions, data flow, etc. [13].

ROS provides SMACH package to make a state machine for our program, and also provides smach_viewer pakage This package let us visualize the current state, all possible state transitions, data passed between the states and many more giving a complete freedom to the user to introspect and debug the system.

4.3.1 Creating a State

SMACH states are Python classes that extend the smach.State class by overriding the execute() method to return one or more possible outcomes [6]. We can build our own class, but there are a number of predefined state types that can save a lot of unnecessary code. In particular, the SimpleActionState class turns a regular ROS action into a SMACH state. Similarly, the MonitorState wraps a ROS topic and the ServiceState handles ROS services.

4.3.2 Welcoming State

KURAbot start from Welcoming State. This state only contain voice control, to make KURAbot speak and welcoming the user. This state also give us indication that everything started correctly.

```
1  class Welcome(State):
2    def __init__(self):
3      State.__init__(self, outcomes=['succeeded'])
4      self.soundstring_publisher = rospy.Publisher('sound_message',
       String, queue_size=10)
5
6    def execute(self, userdata):
7      rospy.sleep(1)
8      rospy.loginfo("Hello, I am kura-kurabot")
9      rospy.loginfo("I will help you to deliver food to your
       customers")
10
11     self.soundstring_publisher.publish("welcome")
12     rospy.sleep(8)
13
```

```
14      return 'succeeded'
```

LISTING 4.2: Welcoming State

4.3.3 Waiting Order State

KURAbot will go to the kitchen and waiting until delivery order comes. This state will subscribe to /ar_pose_maker topic, it will listen to that topic, so every event that happen in that topic, this state will run some function that required to achieve Waiting Order Process.

We will take only the most important point in this state. Basically, a state machine will execute the execute function in a state, and if it reached to returning outcome statement, the state is terminated. We do not want this state to be terminated if it is still not detect any delivery order (AR Tags). So we create a loop and the only condition that can break this loop is when this state detect at least one AR Tag. Since we are using a callback function to manage an event in /ar_pose _maker topic (behavior of callback function is behaving like an interrupt, will be called where ever the line of code being executed), it is possible to make this kind of process.

When the AR Tags detected, it will store its ID, breaks the loop, and right after the loop breaks, the state will know what is the delivery order. State will terminated by returning the outcomes detect1, or until detect6.

```
1  ...
2  # while there is no marker, hold the execution of execute function
3  while(self.n_markers==0):
4
5    if self.preempt_requested():
6      rospy.loginfo("State ComputerVision is being preempted!!!")
7      self.service_preempt()
8      return 'preempted'
9
10   rospy.loginfo("waiting for delivery command...")
11   rospy.sleep(1)
12
13 # right after there is marker return an outcome
14 rospy.loginfo("number detected! it is: " + str(self.my_tagid) )
```

```
15 ...
```

LISTING 4.3: Waiting Order State

4.3.4 Navigation States

After KURAbot knows what is the delivery order, KURAbot has to navigate itself to the table position correspond to the ID of AR Tag detected. We seperate our navigation task into two group, Simple Navigation State (Figure 4.7), and Extended Navigation State (Figure 4.8). From figure 4.1, position of table number 1, number 2, and number 5, are easy to reach. In contrast of those table position, table number 3, number 4, and number 6, are more difficult to reach. This problem is already mentioned in subsection 3.4.2, and the general idea of solution also already mentioned in subsection 4.2.1, which is creating checkpoints.

As we can see from figure 4.1, there are two checkpoints that specified. We tried several positions to become our checkpoints, and these two locations are the chosen locations that considered as the best positions that can reach every area in the map.

Let's take an example of KURAbot is delivering food to table number 4. if KURAbot failed to go to table number 4, it will try to go to checkpoint number 2 (KURAbot will try to go to the farthest checkpoint first). If it is succeeded then proceed to go to table number 4, but if it is failed, it will try to go to checkpoint number 1 (nearest checkpoint). If it is succeeded then go to the next checkpoint and go to table number 4, but if it is still failed, KURAbot will cancel the delivery order. Similar things also applied when KURAbot want to go back to the kitchen after delivering the food.

KURAbot also have Clearing Noise State, it is a state that only contains 360 degrees rotation. In some rare case, noises in the map still not removed when KURAbot go to checkpoints, this state make sure noises removed from the map. Details of this extended navigation state can be seen in Figure 4.8.

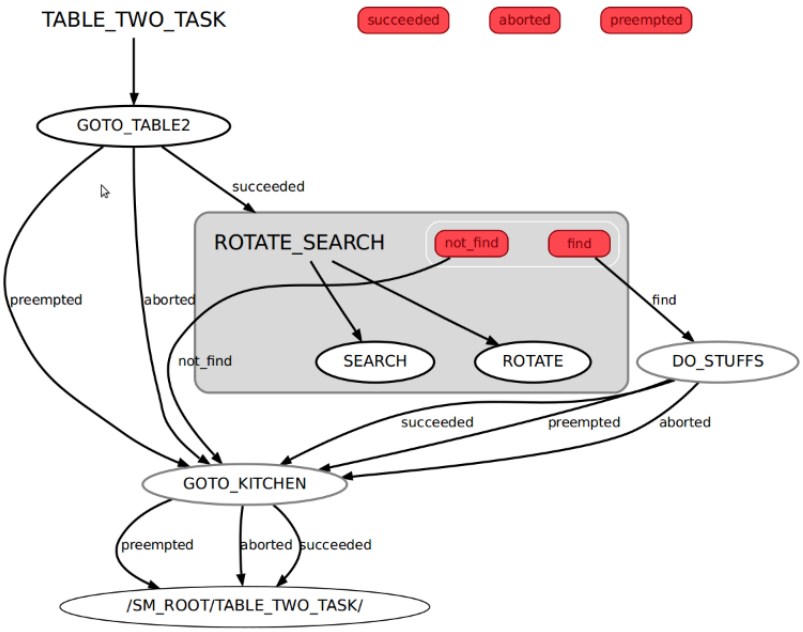

FIGURE 4.7: Simple Navigation State

4.3.5 Searching Table State (with Concurrence Class)

Ater KURAbot arrived to table position, KURAbot has to adjust itself to face towards the table. This state actually contains two states that are running together simultaneously, Rotate State, and Searching Tag State (gray part in the state machine describe by Figure 4.7 and Figure 4.8). This process can be performed by using Concurrence Class, instead of State Machine Class. Basically, Concurrence Class is similar to State Machine Class. In fact, State Class inherit from Concurrence Class [14].

Since Concurrence Class will execute states simultaneously, there will be a minor problem, we will do not know which state will terminate first, that leads to ambiguity of outcomes in Concurrence Class. Concurrence class already provide a way to solve this problem, is to implement two callback functions called child termination callback, and outcome callback. Child termination callback function called every time one of the child states terminates. In the callback function you can decide if the state machine should keep running (return False), or if it should preempt all remaining running states (return True). Outcome callback function is called once when

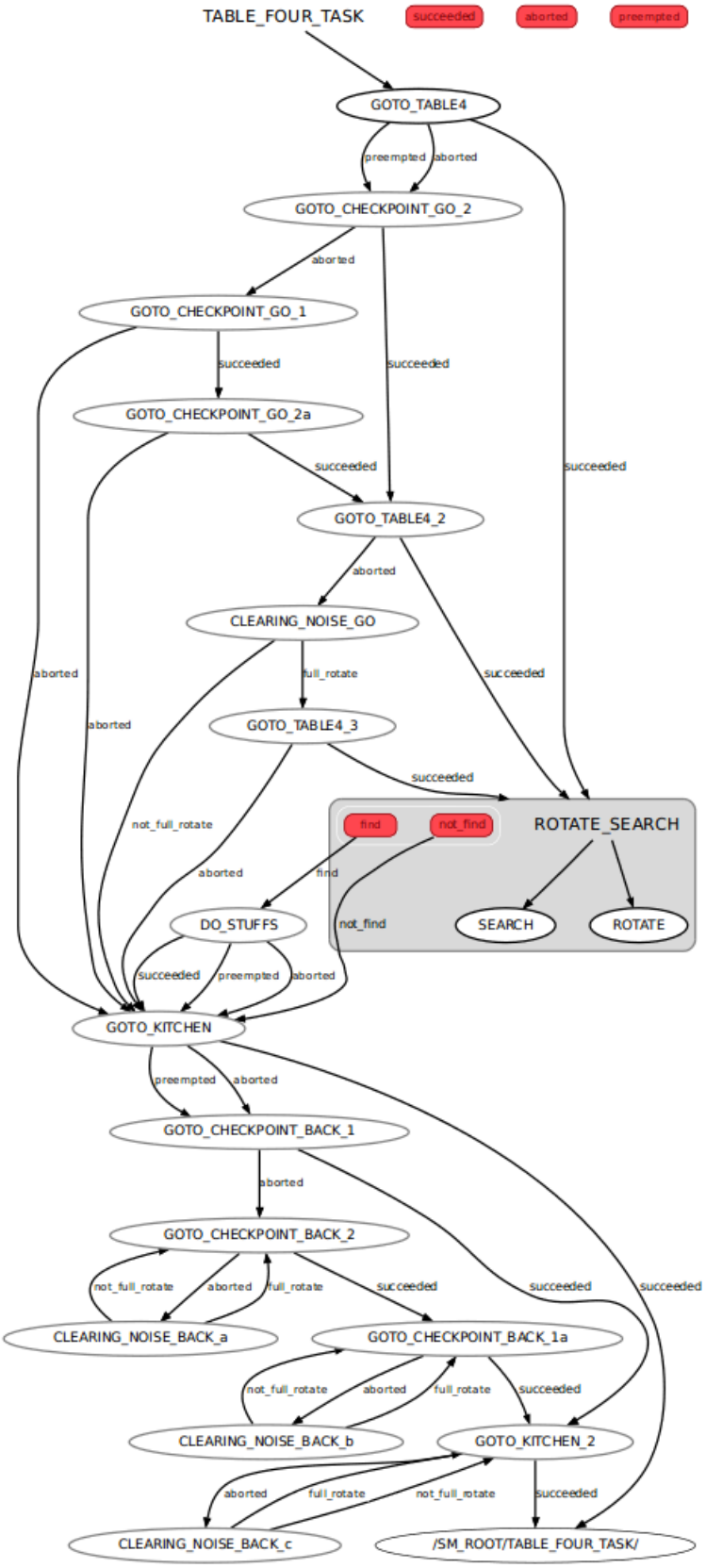

FIGURE 4.8: Extended Navigation State

the last child state terminates. This callback returns the outcome of the concurrence state machine. If we implement these two callback functions, that means we will have full control of Concurrence Class.

Let's take example of KURAbot executing Rotate and Search State simultaneously. KURAbot tried to rotate until 360 degrees, and while it is rotating, it keep searching for AR Tag nearby (below some threshold, mentioned in subsection 4.2.2). Rotate state will give full_rotate outcome if KURAbot finish to rotate in 360 degrees, and if it doesn't finish Rotate state will give not_full_rotate outcome. Search state will give detect outcome if KURAbot detect AR Tag nearby, and if it didn't detect any AR Tag nearby Search state will give not_detect outcome.

If Rotate state terminated first, it means KURAbot finish to rotate in 360 degrees, or we can say KURAbot is not detecting any AR Tag nearby. If Search state terminated first, it means KURAbot successfully detect AR Tag nearby, or we can say KURAbot is not rotating full 360 degrees. We can we can take advantage of these condition, if one of the state terminate, we terminated the other states, because we already know what is the outcomes depends on what state that terminate first. The implementation of the two callback function is written as follows

```
def concurrence_child_termination_callback(self, outcome_map):
  if outcome_map['SEARCH'] == 'detect':
    return True

  elif outcome_map['ROTATE'] == 'full_rotate':
    return True

  else:
  return False

def concurrence_outcome_callback(self, outcome_map):
  if outcome_map['SEARCH'] == 'detect':
    return 'find'

  elif outcome_map['ROTATE'] == 'full_rotate':
    return 'not_find'
```

```
17
18  else:
19  return 'not_find'
```

LISTING 4.4: Concurrence Callback Functions

4.4 Final

Finally, KURAbot is complete. We arranged all of those state machine to become one big state machine. The picture is too big to shown here, we put our graph of state machine in Appendix.

Chapter 5

Problems Faced

This is a chapter of collection of every problem that we faced when we developing KURAbot. Some already mentioned in previous chapters, some are not. We summary everything in sections that contain the desciption and solution of the problem

5.1 Initialization Position Problem

5.1.1 Description

We built the map by running command that written in Appendix A.1. The place of the robot will be the origin of the map (coordinate 0,0), and also KURAbot initialization point. Our starting position of KURAbot when we built the map is in the top right of the arena. Instead of using that position, we want to make center of arena as our initial position.

5.1.2 Solution

We have to publish a PoseWithCovarianceStamped message to /initial_pose topic. We can specify our desired initial location and orientation with that message. Code below is how do we publish PoseWithCovarianceStamped message to /initial_pose topic.

```
#initialize position
initial_publisher = rospy.Publisher('/initialpose',
    PoseWithCovarianceStamped, queue_size=20)
rospy.sleep(3)
pose_msg = PoseWithCovarianceStamped()
```

```
5 pose_msg.pose.pose= self.room_locations['checkpoint1']
6 initial_publisher.publish(pose_msg)
```

LISTING 5.1: Publish message to /initialpose topic

Important note, we put sleep 3 second after we instantiate Publisher class. This is due instantiating Publisher class need some times, if we do not wait a little amount of time, initial_publisher cannot publish anything since it is not well initialize.

5.2 AR Tag for Table Indicator Problem

5.2.1 Description

Let's consider there is delivery order to table number 1, KURAbot will navigate itself to location number one. Once it is arrived, now it tried to face itself to the table (AR tag number 7), problem arise when sometimes it can detect the table in another table location, for example location table 2, since we also use AR tag number 7 as table indicator number 2. KURAbot will end up facing that direction instead of facing to table number 1.

Something that make it worse is, when far distance AR Tag are detected in the center of our camera, it will give us good result of distance, but since in this situation KURAbot is rotating, eventually far distance AR Tag move to the edge of the camera, in this situation (in rarely times), the distance is detected close enough to the camera (sometimes below 0.7), and the result, KURAbot will end up facing that direction instead of facing to another table.

5.2.2 Solution

Since this situation rarely encountered (but quite terrible if it is happening), instead of just using one value of distance, we collect the distances values and take a mean of the distances. If that condition happen, since we collect more than value and most of the value are true distance, it will not change mean of distances much.

```
1 self.sum_x = self.sum_x + tag.pose.pose.position.x
2 self.mean_distance = self.sum_x/self.counter
```

LISTING 5.2: collecting and compute mean of distances

5.3 Noisy Map Problem

5.3.1 Description

KURAbot sometimes can not go to certain pose in the map due to radius of the obstacle. Figure 3.8 shows sometimes amcl leaves noisy particles in certain area. This will leads cancelation of navigation to certain point.

5.3.2 Solution

In order to avoid cancelation of navigation to certain point that caused by noises in the map, some addition points called checkpoints are used, so that KURAbot has higher possibility to arrive at the point we desired from those checkpoints rather than from the origin of the position. Subsection 4.3.4 explained in detail how these checkpoints works.

Chapter 6

Conclusion

In this Lab Project session, we faced with a big practical working to analysis, test and implement algorithms on robot. After finishing final tests, the robot was successful for each predefined tasks.

We considered special scenarios implemented on robot. Tag detection does works well and is good enough to recognize delivery order. After each step, robot speak clearly to inform goal costumer and will be prepared for next steps. We have tested the tag detection with 6 different tags which robot gives correct output with high accuracy. All tasks done successfully. After creating and implementing mapping part, it does work but some times in some special places which the map becomes noisy to make navigation and localization more robust, we implement a scenario that make the noise clear well and will increase robot steady state. It works well also after different testing and experiments which has been implemented in the state machine.

In this lab, we learned many technical methods and applications of computer vision like servo vision, different detection schemes and so on, and also robotic part like mapping, navigation, localization and so on and both of these two parts made a strong robot to do many stuffs in real words. Also, we learned how to implement different tasks on robot by ROS to make it flexible to do any special works.

Appendix A

Environment Needed

A.1 Map Building

In turtlebot notebook

```
$ roslaunch turtlebot_le2i rplidar_minimal.launch
$ roslaunch rbx1_nav rplidar_gmapping_demo.launch
```

In workstation PC

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

A.2 AR Tags

Generate AR Tags

```
$ rosrun ar_track_alvar createMarker -s 30 1
$ rosrun ar_track_alvar createMarker -s 30 2
$ rosrun ar_track_alvar createMarker -s 30 3
$ rosrun ar_track_alvar createMarker -s 30 4
$ rosrun ar_track_alvar createMarker -s 30 5
$ rosrun ar_track_alvar createMarker -s 30 6
$ rosrun ar_track_alvar createMarker -s 30 7
```

A.3 Sounds

`.../new_sound_wav/welcome.wav`

"Hello, I'm KURAbot, I will help you to deliver food to the customers"

`.../new_sound_wav/waiting_order.wav`

"Waiting for the order"

`.../new_sound_wav/O1.wav`

"Order to table one detected"

`.../new_sound_wav/M1.wav`

"Moving to table one"

`.../new_sound_wav/table_search.wav`

"Searching for the table, please wait"

`.../new_sound_wav/thanks.wav`

"Thank you for the order, have a nice meal, bye.."

`.../sound/smb_powerup.wav`

"[mario power up sound (table found)]"

`.../sound/smb_mariodie.wav`

"[mario die sound (table not found)]"

Appendix B

Source Code

B.1 Main Program (delivering_food_10.py)

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

import rospy
import smach
import actionlib
from smach import State, StateMachine, Concurrence
from smach_ros import SimpleActionState, IntrospectionServer
from geometry_msgs.msg import Twist
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from geometry_msgs.msg import Pose, PoseWithCovarianceStamped,
    Point, Quaternion, Twist
from visualization_msgs.msg import Marker
from ar_track_alvar.msg import AlvarMarkers

import easygui
import datetime
from collections import OrderedDict
from tf.transformations import quaternion_from_euler
from math import radians, pi
import random

import tf
from rbx1_nav.transform_utils import quat_to_angle,
    normalize_angle

from custom_state_class.computer_vision_beep import ComputerVision
from custom_state_class.search_table_beep import SearchTable
```

```
26 from custom_state_class.do_stuffs_beep import DoStuffs
27 from custom_state_class.rotate360 import Rotate360
28
29 from std_msgs.msg import String
30
31 class Welcome(State):
32     def __init__(self):
33         State.__init__(self, outcomes=['succeeded'])
34         self.soundstring_publisher = rospy.Publisher('
    sound_message', String, queue_size=10)
35
36     def execute(self, userdata):
37         rospy.sleep(1)
38         rospy.loginfo("Hello, I am kura-kurabot")
39         rospy.loginfo("I will help you to deliver food to your
    customers")
40
41         self.soundstring_publisher.publish("welcome")
42         rospy.sleep(8)
43
44         return 'succeeded'
45
46
47 class MyMainClass():
48     def __init__(self):
49         rospy.init_node('deliver_food', anonymous=False)
50         self.initialize_destination()
51
52         # Track success rate of getting to the goal locations
53         self.n_succeeded = 0
54         self.n_aborted = 0
55         self.n_preempted = 0
56
57         nav_states = {}
58
59         for room in self.room_locations.iterkeys():
60             nav_goal = MoveBaseGoal()
61             nav_goal.target_pose.header.frame_id = 'map'
62             nav_goal.target_pose.pose = self.room_locations[room]
```

```
63          move_base_state = SimpleActionState('move_base',
    MoveBaseAction, goal=nav_goal, result_cb=self.
    move_base_result_cb,
64                                                  exec_timeout=rospy
    .Duration(60.0),
65
    server_wait_timeout=rospy.Duration(10.0))
66          nav_states[room] = move_base_state
67          rospy.loginfo(room + " -> [" + str(round(self.
    room_locations[room].position.x,2)) + ", " + str(round(self.
    room_locations[room].position.y,2)) + "]" )
68
69      sm_rotate_search = Concurrence(outcomes=['find', 'not_find
    '],
70                                     default_outcome='not_find'
    ,
71                                     child_termination_cb=self.
    concurrence_child_termination_callback,
72                                     outcome_cb=self.
    concurrence_outcome_callback)
73
74      with sm_rotate_search:
75          Concurrence.add('ROTATE', Rotate360(0.4, 2*pi) )
76          Concurrence.add('SEARCH', SearchTable() )
77
78      sm_table1 = StateMachine(outcomes=['succeeded','aborted','
    preempted'])
79      with sm_table1:
80          StateMachine.add('GOTO_TABLE1', nav_states['table1'],
    transitions={'succeeded':'ROTATE_SEARCH','aborted':'
    GOTO_KITCHEN','preempted':'GOTO_KITCHEN'})
81          StateMachine.add('ROTATE_SEARCH', sm_rotate_search,
    transitions={'find':'DO_STUFFS','not_find':'GOTO_KITCHEN'})
82          StateMachine.add('DO_STUFFS', DoStuffs(5), transitions
    ={'succeeded':'GOTO_KITCHEN','aborted':'GOTO_KITCHEN','
    preempted':'GOTO_KITCHEN'})
83          StateMachine.add('GOTO_KITCHEN', nav_states['kitchen'
    ], transitions={'succeeded':'','aborted':'','preempted':''})
84
```

```
85        sm_table2 = StateMachine(outcomes=['succeeded','aborted','
    preempted'])
86        with sm_table2:
87            StateMachine.add('GOTO_TABLE2', nav_states['table2'],
    transitions={'succeeded':'ROTATE_SEARCH','aborted':'
    GOTO_KITCHEN','preempted':'GOTO_KITCHEN'})
88            StateMachine.add('ROTATE_SEARCH', sm_rotate_search,
    transitions={'find':'DO_STUFFS','not_find':'GOTO_KITCHEN'})
89            StateMachine.add('DO_STUFFS', DoStuffs(5), transitions
    ={'succeeded':'GOTO_KITCHEN','aborted':'GOTO_KITCHEN','
    preempted':'GOTO_KITCHEN'})
90            StateMachine.add('GOTO_KITCHEN', nav_states['kitchen'
    ], transitions={'succeeded':'','aborted':'','preempted':''})
91
92        sm_table3 = StateMachine(outcomes=['succeeded','aborted','
    preempted'])
93        with sm_table3:
94            StateMachine.add('GOTO_TABLE3', nav_states['table3'],
    transitions={'succeeded':'ROTATE_SEARCH',
95
96                 'aborted':'GOTO_CHECKPOINT_GO_2',
97
98                 'preempted':'GOTO_CHECKPOINT_GO_2'})
99            StateMachine.add('ROTATE_SEARCH', sm_rotate_search,
    transitions={'find':'DO_STUFFS',
100
101              'not_find':'GOTO_KITCHEN'})
102           StateMachine.add('DO_STUFFS', DoStuffs(5), transitions
    ={'succeeded':'GOTO_KITCHEN',
103
104                'aborted':'GOTO_KITCHEN',
105
                   'preempted':'GOTO_KITCHEN'})
106           StateMachine.add('GOTO_KITCHEN', nav_states['kitchen'
    ], transitions={'succeeded':'',
107
108                  'aborted':'GOTO_CHECKPOINT_BACK_1'})
109
110           # if something wrong when we tried to go to the table
```

```
106
107         StateMachine.add('GOTO_CHECKPOINT_GO_2', nav_states['
    checkpoint2'], transitions={'succeeded':'GOTO_TABLE3_2',
108                                                 'aborted'
    :'GOTO_CHECKPOINT_GO_1'} )
109
110         StateMachine.add('GOTO_CHECKPOINT_GO_1', nav_states['
    checkpoint1'], transitions={'succeeded':'GOTO_CHECKPOINT_GO_2a'
    ,
111                                                 'aborted'
    :'GOTO_KITCHEN'} )
112
113         StateMachine.add('GOTO_CHECKPOINT_GO_2a', nav_states['
    checkpoint1'], transitions={'succeeded':'GOTO_TABLE3_2',
114                                                 'aborted':'
    GOTO_KITCHEN'} )
115
116         StateMachine.add('GOTO_TABLE3_2', nav_states['table3'
    ], transitions={'succeeded':'ROTATE_SEARCH',
117                                                 'aborted':'
    CLEARING_NOISE_GO'})
118
119         StateMachine.add('CLEARING_NOISE_GO', Rotate360(0.8,
    2*pi), transitions={'full_rotate':'GOTO_TABLE3_3',
120                                                 'not_full_rotate'
    :'GOTO_KITCHEN'})
121
122         StateMachine.add('GOTO_TABLE3_3', nav_states['table3'
    ], transitions={'succeeded':'ROTATE_SEARCH',
123                                                 'aborted':'
    GOTO_KITCHEN'})
124
125         # if something wrong when we tried to go back to
    kitchen
126
127         StateMachine.add('GOTO_CHECKPOINT_BACK_1', nav_states[
    'checkpoint1'], transitions={'succeeded':'GOTO_KITCHEN_2',
128                                                 'aborted'
    :'GOTO_CHECKPOINT_BACK_2'})
```

```
129
130         StateMachine.add('GOTO_CHECKPOINT_BACK_2', nav_states['checkpoint2'], transitions={'succeeded':'GOTO_CHECKPOINT_BACK_1a',
131                                                                                             'aborted':'CLEARING_NOISE_BACK_a'})
132
133         StateMachine.add('CLEARING_NOISE_BACK_a', Rotate360(0.8, 2*pi), transitions={'full_rotate':'GOTO_CHECKPOINT_BACK_2',
134                                                                                      'not_full_rotate':'GOTO_CHECKPOINT_BACK_2'})
135
136         StateMachine.add('GOTO_CHECKPOINT_BACK_1a', nav_states['checkpoint1'], transitions={'succeeded':'GOTO_KITCHEN_2',
137                                                                                             'aborted':'CLEARING_NOISE_BACK_b'})
138
139         StateMachine.add('CLEARING_NOISE_BACK_b', Rotate360(0.8, 2*pi), transitions={'full_rotate':'GOTO_CHECKPOINT_BACK_1a',
140                                                                                      'not_full_rotate':'GOTO_CHECKPOINT_BACK_1a'})
141
142         StateMachine.add('GOTO_KITCHEN_2', nav_states['kitchen'], transitions={'succeeded':'',
143                                                                                'aborted':'CLEARING_NOISE_BACK_c'})
144
145         StateMachine.add('CLEARING_NOISE_BACK_c', Rotate360(0.8, 2*pi), transitions={'full_rotate':'GOTO_KITCHEN_2',
146                                                                                      'not_full_rotate':'GOTO_KITCHEN_2'})
147
148     sm_table4 = StateMachine(outcomes=['succeeded','aborted','preempted'])
149     with sm_table4:
150         StateMachine.add('GOTO_TABLE4', nav_states['table4'], transitions={'succeeded':'ROTATE_SEARCH',
```

```
151                 'aborted':'GOTO_CHECKPOINT_GO_2',
152                 'preempted':'GOTO_CHECKPOINT_GO_2'})
153         StateMachine.add('ROTATE_SEARCH', sm_rotate_search,
    transitions={'find':'DO_STUFFS',
154                  'not_find':'GOTO_KITCHEN'})
155         StateMachine.add('DO_STUFFS', DoStuffs(5), transitions
    ={'succeeded':'GOTO_KITCHEN',
156                 'aborted':'GOTO_KITCHEN',
157                 'preempted':'GOTO_KITCHEN'})
158         StateMachine.add('GOTO_KITCHEN', nav_states['kitchen'
    ], transitions={'succeeded':'',
159                   'aborted':'GOTO_CHECKPOINT_BACK_1',
160                   'preempted':'GOTO_CHECKPOINT_BACK_1'})
161
162         # if something wrong when we tried to go to the table
163
164         StateMachine.add('GOTO_CHECKPOINT_GO_2', nav_states['
    checkpoint2'], transitions={'succeeded':'GOTO_TABLE4_2',
165                                                 'aborted'
    :'GOTO_CHECKPOINT_GO_1'} )
166
167         StateMachine.add('GOTO_CHECKPOINT_GO_1', nav_states['
    checkpoint1'], transitions={'succeeded':'GOTO_CHECKPOINT_GO_2a'
    ,
168                                                 'aborted'
    :'GOTO_KITCHEN'} )
169
170         StateMachine.add('GOTO_CHECKPOINT_GO_2a', nav_states['
    checkpoint1'], transitions={'succeeded':'GOTO_TABLE4_2',
171                                                   'aborted':'
    GOTO_KITCHEN'} )
172
```

```
173         StateMachine.add('GOTO_TABLE4_2', nav_states['table4'
    ], transitions={'succeeded':'ROTATE_SEARCH',
174                                             'aborted':'
    CLEARING_NOISE_GO'})
175
176         StateMachine.add('CLEARING_NOISE_GO', Rotate360(0.8,
    2*pi), transitions={'full_rotate':'GOTO_TABLE4_3',
177                                               'not_full_rotate'
    :'GOTO_KITCHEN'})
178
179         StateMachine.add('GOTO_TABLE4_3', nav_states['table4'
    ], transitions={'succeeded':'ROTATE_SEARCH',
180                                             'aborted':'
    GOTO_KITCHEN'})
181
182         # if something wrong when we tried to go back to
    kitchen
183
184         StateMachine.add('GOTO_CHECKPOINT_BACK_1', nav_states[
    'checkpoint1'], transitions={'succeeded':'GOTO_KITCHEN_2',
185                                                     'aborted'
    :'GOTO_CHECKPOINT_BACK_2'})
186
187         StateMachine.add('GOTO_CHECKPOINT_BACK_2', nav_states[
    'checkpoint2'], transitions={'succeeded':'
    GOTO_CHECKPOINT_BACK_1a',
188                                                     'aborted'
    :'CLEARING_NOISE_BACK_a'})
189
190         StateMachine.add('CLEARING_NOISE_BACK_a', Rotate360
    (0.8, 2*pi), transitions={'full_rotate':'GOTO_CHECKPOINT_BACK_2
    ',
191                                                       '
    not_full_rotate':'GOTO_CHECKPOINT_BACK_2'})
192
193         StateMachine.add('GOTO_CHECKPOINT_BACK_1a', nav_states
    ['checkpoint1'], transitions={'succeeded':'GOTO_KITCHEN_2',
194                                                      'aborted
    ':'CLEARING_NOISE_BACK_b'})
```

```
195
196         StateMachine.add('CLEARING_NOISE_BACK_b', Rotate360
    (0.8, 2*pi), transitions={'full_rotate':'
    GOTO_CHECKPOINT_BACK_1a',
197                                                 '
    not_full_rotate':'GOTO_CHECKPOINT_BACK_1a'})
198
199         StateMachine.add('GOTO_KITCHEN_2', nav_states['kitchen
    '], transitions={'succeeded':'',
200                                                 'aborted':'
    CLEARING_NOISE_BACK_c'})
201
202         StateMachine.add('CLEARING_NOISE_BACK_c', Rotate360
    (0.8, 2*pi), transitions={'full_rotate':'GOTO_KITCHEN_2',
203                                                 '
    not_full_rotate':'GOTO_KITCHEN_2'})
204
205
206     sm_table5 = StateMachine(outcomes=['succeeded','aborted','
    preempted'])
207     with sm_table5:
208         StateMachine.add('GOTO_TABLE5', nav_states['table5'],
    transitions={'succeeded':'ROTATE_SEARCH','aborted':'
    GOTO_KITCHEN','preempted':'GOTO_KITCHEN'})
209         StateMachine.add('ROTATE_SEARCH', sm_rotate_search,
    transitions={'find':'DO_STUFFS','not_find':'GOTO_KITCHEN'})
210         StateMachine.add('DO_STUFFS', DoStuffs(5), transitions
    ={'succeeded':'GOTO_KITCHEN','aborted':'GOTO_KITCHEN','
    preempted':'GOTO_KITCHEN'})
211         StateMachine.add('GOTO_KITCHEN', nav_states['kitchen'
    ], transitions={'succeeded':'','aborted':'','preempted':''})
212
213     sm_table6 = StateMachine(outcomes=['succeeded','aborted','
    preempted'])
214     with sm_table6:
215         StateMachine.add('GOTO_TABLE6', nav_states['table6'],
    transitions={'succeeded':'ROTATE_SEARCH',
216
                'aborted':'GOTO_CHECKPOINT_GO_2',
```

```
217
                'preempted':'GOTO_CHECKPOINT_GO_2'})
218         StateMachine.add('ROTATE_SEARCH', sm_rotate_search,
    transitions={'find':'DO_STUFFS',
219
              'not_find':'GOTO_KITCHEN'})
220         StateMachine.add('DO_STUFFS', DoStuffs(5), transitions
    ={'succeeded':'GOTO_KITCHEN',
221
               'aborted':'GOTO_KITCHEN',
222
               'preempted':'GOTO_KITCHEN'})
223         StateMachine.add('GOTO_KITCHEN', nav_states['kitchen'
    ], transitions={'succeeded':'',
224
                  'aborted':'GOTO_CHECKPOINT_BACK_1',
225
                  'preempted':'GOTO_CHECKPOINT_BACK_1'})
226
227         # if something wrong when we tried to go to the table
228
229         StateMachine.add('GOTO_CHECKPOINT_GO_2', nav_states['
    checkpoint2'], transitions={'succeeded':'GOTO_TABLE6_2',
230                                                     'aborted'
    :'GOTO_CHECKPOINT_GO_1'} )
231
232         StateMachine.add('GOTO_CHECKPOINT_GO_1', nav_states['
    checkpoint1'], transitions={'succeeded':'GOTO_CHECKPOINT_GO_2a'
    ,
233                                                     'aborted'
    :'GOTO_KITCHEN'} )
234
235         StateMachine.add('GOTO_CHECKPOINT_GO_2a', nav_states['
    checkpoint1'], transitions={'succeeded':'GOTO_TABLE6_2',
236                                                   'aborted':'
    GOTO_KITCHEN'} )
237
238         StateMachine.add('GOTO_TABLE6_2', nav_states['table6'
    ], transitions={'succeeded':'ROTATE_SEARCH',
```

```
239                                                 'aborted':'
    CLEARING_NOISE_GO'})
240
241             StateMachine.add('CLEARING_NOISE_GO', Rotate360(0.8,
    2*pi), transitions={'full_rotate':'GOTO_TABLE6_3',
242                                                   'not_full_rotate'
    :'GOTO_KITCHEN'})
243
244             StateMachine.add('GOTO_TABLE6_3', nav_states['table6'
    ], transitions={'succeeded':'ROTATE_SEARCH',
245                                                 'aborted':'
    GOTO_KITCHEN'})
246
247             # if something wrong when we tried to go back to
    kitchen
248
249             StateMachine.add('GOTO_CHECKPOINT_BACK_1', nav_states[
    'checkpoint1'], transitions={'succeeded':'GOTO_KITCHEN_2',
250                                                             'aborted'
    :'GOTO_CHECKPOINT_BACK_2'})
251
252             StateMachine.add('GOTO_CHECKPOINT_BACK_2', nav_states[
    'checkpoint2'], transitions={'succeeded':'
    GOTO_CHECKPOINT_BACK_1a',
253                                                             'aborted'
    :'CLEARING_NOISE_BACK_a'})
254
255             StateMachine.add('CLEARING_NOISE_BACK_a', Rotate360
    (0.8, 2*pi), transitions={'full_rotate':'GOTO_CHECKPOINT_BACK_2
    ',
256                                                               '
    not_full_rotate':'GOTO_CHECKPOINT_BACK_2'})
257
258             StateMachine.add('GOTO_CHECKPOINT_BACK_1a', nav_states
    ['checkpoint1'], transitions={'succeeded':'GOTO_KITCHEN_2',
259                                                             'aborted
    ':'CLEARING_NOISE_BACK_b'})
260
```

```
261             StateMachine.add('CLEARING_NOISE_BACK_b', Rotate360
    (0.8, 2*pi), transitions={'full_rotate':'
    GOTO_CHECKPOINT_BACK_1a',
262                                                     '
    not_full_rotate':'GOTO_CHECKPOINT_BACK_1a'})
263
264             StateMachine.add('GOTO_KITCHEN_2', nav_states['kitchen
    '], transitions={'succeeded':'',
265                                                 'aborted':'
    CLEARING_NOISE_BACK_c'})
266
267             StateMachine.add('CLEARING_NOISE_BACK_c', Rotate360
    (0.8, 2*pi), transitions={'full_rotate':'GOTO_KITCHEN_2',
268                                                     '
    not_full_rotate':'GOTO_KITCHEN_2'})
269         # let's initialize the overall state machine
270         sm_deliverfood = StateMachine(outcomes=['succeeded','
    aborted','preempted'])
271
272         with sm_deliverfood:
273             StateMachine.add('STARTING_TASK', Welcome(),
    transitions={'succeeded':'GOTO_KITCHEN'})
274             StateMachine.add('COMPUTER_VISION_TASK',
    ComputerVision(), transitions={'detect1':'TABLE_ONE_TASK',
275
                    'detect2':'TABLE_TWO_TASK',
276
                    'detect3':'TABLE_THREE_TASK',
277
                    'detect4':'TABLE_FOUR_TASK',
278
                    'detect5':'TABLE_FIVE_TASK',
279
                    'detect6':'TABLE_SIX_TASK',
280
                    'preempted':''})
281             StateMachine.add('TABLE_ONE_TASK',sm_table1,
    transitions={'succeeded':'COMPUTER_VISION_TASK','aborted':'
    GOTO_KITCHEN','preempted':'GOTO_KITCHEN'})
```

```
282             StateMachine.add('TABLE_TWO_TASK',sm_table2,
    transitions={'succeeded':'COMPUTER_VISION_TASK','aborted':'
    GOTO_KITCHEN','preempted':'GOTO_KITCHEN'})
283             StateMachine.add('TABLE_THREE_TASK',sm_table3,
    transitions={'succeeded':'COMPUTER_VISION_TASK','aborted':'
    GOTO_KITCHEN','preempted':'GOTO_KITCHEN'})
284             StateMachine.add('TABLE_FOUR_TASK',sm_table4,
    transitions={'succeeded':'COMPUTER_VISION_TASK','aborted':'
    GOTO_KITCHEN','preempted':'GOTO_KITCHEN'})
285             StateMachine.add('TABLE_FIVE_TASK',sm_table5,
    transitions={'succeeded':'COMPUTER_VISION_TASK','aborted':'
    GOTO_KITCHEN','preempted':'GOTO_KITCHEN'})
286             StateMachine.add('TABLE_SIX_TASK',sm_table6,
    transitions={'succeeded':'COMPUTER_VISION_TASK','aborted':'
    GOTO_KITCHEN','preempted':'GOTO_KITCHEN'})
287
288             StateMachine.add('GOTO_KITCHEN', nav_states['kitchen'
    ], transitions={'succeeded':'COMPUTER_VISION_TASK','aborted':'
    GOTO_KITCHEN','preempted':'GOTO_KITCHEN'})
289
290         # Create and start the SMACH introspection server
291         intro_server = IntrospectionServer('deliver_food',
    sm_deliverfood, '/SM_ROOT')
292         intro_server.start()
293
294         # Execute the state machine
295         sm_outcome = sm_deliverfood.execute()
296         rospy.on_shutdown(self.shutdown)
297
298     def move_base_result_cb(self, userdata, status, result):
299         if status == actionlib.GoalStatus.SUCCEEDED:
300             self.n_succeeded += 1
301         elif status == actionlib.GoalStatus.ABORTED:
302             self.n_aborted += 1
303         elif status == actionlib.GoalStatus.PREEMPTED:
304             self.n_preempted += 1
305
306
307     def concurrence_child_termination_callback(self, outcome_map):
```

```
308         if outcome_map['SEARCH'] == 'detect':
309           return True
310 
311         elif outcome_map['ROTATE'] == 'full_rotate':
312           return True
313 
314         else:
315           return False
316 
317     def concurrence_outcome_callback(self, outcome_map):
318         if outcome_map['SEARCH'] == 'detect':
319           return 'find'
320 
321         elif outcome_map['ROTATE'] == 'full_rotate':
322           return 'not_find'
323 
324         # lazy to think, just put this
325         else:
326           return 'not_find'
327 
328     def initialize_destination(self):
329         self.move_base_timeout = rospy.get_param("~
    move_base_timeout", 10) # seconds
330 
331         # Subscribe to the move_base action server
332         self.move_base = actionlib.SimpleActionClient("move_base",
     MoveBaseAction)
333         rospy.loginfo("Waiting for move_base action server...")
334         self.move_base.wait_for_server(rospy.Duration(60))
335         rospy.loginfo("Connected to move_base action server")
336 
337         # Orientation list
338         quaternions = list()
339 
340         euler_angles = (0, -pi/2, pi, 0, pi/2, -pi/2, -3*pi/4, pi
    /2, pi/2)
341 
342         for angle in euler_angles:
```

```
343             q_angle = quaternion_from_euler(0, 0, angle, axes='
    sxyz')
344             q = Quaternion(*q_angle)
345             quaternions.append(q)
346 
347         # Position list
348         points = list()
349         points.append(Point(0, 0, 0))
350         points.append(Point(0, -1.55497133732, 0))
351         points.append(Point(-2.21499538422, -1.55497133732, 0))
352         points.append(Point(-1.3093624115, 0.198241680861, 0))
353         points.append(Point(-3.51539182663, 0, 0))
354         points.append(Point(-0.926210045815, -2.36771917343, 0))
355         points.append(Point(-3.55022335052, -1.96715021133, 0))
356         points.append(Point(-1.13520205021, -1.3924216032, 0))
357         points.append(Point(-2.56331586838, -0.289407044649, 0))
358 
359         # Create a list to hold the waypoint poses
360         self.waypoints = list()
361 
362         # Append each of the four waypoints to the list.  Each
    waypoint
363         # is a pose consisting of a position and orientation in
    the map frame.
364         self.waypoints.append( Pose(points[0], quaternions[0]) )
365         self.waypoints.append( Pose(points[1], quaternions[1]) )
366         self.waypoints.append( Pose(points[2], quaternions[2]) )
367         self.waypoints.append( Pose(points[3], quaternions[3]) )
368         self.waypoints.append( Pose(points[4], quaternions[4]) )
369         self.waypoints.append( Pose(points[5], quaternions[5]) )
370         self.waypoints.append( Pose(points[6], quaternions[6]) )
371         self.waypoints.append( Pose(points[7], quaternions[7]) )
372         self.waypoints.append( Pose(points[8], quaternions[8]) )
373 
374         # Create a mapping of room names to waypoint locations
375         room_locations = (('kitchen', self.waypoints[0]),
376                           ('table1', self.waypoints[1]),
377                           ('table2', self.waypoints[2]),
378                           ('table3', self.waypoints[3]),
```

```
379                         ('table4', self.waypoints[4]),
380                         ('table5', self.waypoints[5]),
381                         ('table6', self.waypoints[6]),
382                         ('checkpoint1', self.waypoints[7]),
383                         ('checkpoint2', self.waypoints[8]))
384
385         # Store the mapping as an ordered dictionary so we can
    visit the rooms in sequence
386         self.room_locations = OrderedDict(room_locations)
387
388         #initialize position
389         initial_publisher = rospy.Publisher('/initialpose',
    PoseWithCovarianceStamped, queue_size=20)
390         rospy.sleep(3)
391         posewithcovariance_msg = PoseWithCovarianceStamped()
392         posewithcovariance_msg.pose.pose= self.room_locations['
    checkpoint1']
393         initial_publisher.publish(posewithcovariance_msg)
394
395         # Initialize a marker for the docking station for RViz
396         self.init_waypoint_markers()
397         self.init_docking_station_marker()
398
399         self.init_waypoint_markers()
400         for waypoint in self.waypoints:
401             p = Point()
402             p = waypoint.position
403             self.waypoint_markers.points.append(p)
404
405         # Publisher to manually control the robot (e.g. to stop it
    )
406         self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
407         rospy.loginfo("Starting Tasks")
408
409         # Publish the waypoint markers
410         self.marker_pub.publish(self.waypoint_markers)
411         rospy.sleep(1)
412         self.marker_pub.publish(self.waypoint_markers) # <- this
    is weird
```

```
413
414         # Publish the docking station marker
415         self.docking_station_marker_pub.publish(self.
    docking_station_marker)
416         rospy.sleep(1)
417
418     def init_waypoint_markers(self):# Set up our waypoint markers
419         marker_scale = 0.2
420         marker_lifetime = 0 # 0 is forever
421         marker_ns = 'waypoints'
422         marker_id = 0
423         marker_color = {'r': 1.0, 'g': 0.7, 'b': 1.0, 'a': 1.0}
424
425         # Define a marker publisher.
426         self.marker_pub = rospy.Publisher('waypoint_markers',
    Marker)
427
428         # Initialize the marker points list.
429         self.waypoint_markers = Marker()
430         self.waypoint_markers.ns = marker_ns
431         self.waypoint_markers.id = marker_id
432         self.waypoint_markers.type = Marker.CUBE_LIST
433         self.waypoint_markers.action = Marker.ADD
434         self.waypoint_markers.lifetime = rospy.Duration(
    marker_lifetime)
435         self.waypoint_markers.scale.x = marker_scale
436         self.waypoint_markers.scale.y = marker_scale
437         self.waypoint_markers.color.r = marker_color['r']
438         self.waypoint_markers.color.g = marker_color['g']
439         self.waypoint_markers.color.b = marker_color['b']
440         self.waypoint_markers.color.a = marker_color['a']
441
442         self.waypoint_markers.header.frame_id = 'odom'
443         self.waypoint_markers.header.stamp = rospy.Time.now()
444         self.waypoint_markers.points = list()
445
446     def init_docking_station_marker(self):
447         # Define a marker for the charging station
448         marker_scale = 0.3
```

```
449         marker_lifetime = 0 # 0 is forever
450         marker_ns = 'waypoints'
451         marker_id = 0
452         marker_color = {'r': 0.7, 'g': 0.7, 'b': 0.0, 'a': 1.0}
453 
454         self.docking_station_marker_pub = rospy.Publisher('
    docking_station_marker', Marker)
455 
456         self.docking_station_marker = Marker()
457         self.docking_station_marker.ns = marker_ns
458         self.docking_station_marker.id = marker_id
459         self.docking_station_marker.type = Marker.CYLINDER
460         self.docking_station_marker.action = Marker.ADD
461         self.docking_station_marker.lifetime = rospy.Duration(
    marker_lifetime)
462         self.docking_station_marker.scale.x = marker_scale
463         self.docking_station_marker.scale.y = marker_scale
464         self.docking_station_marker.scale.z = 0.02
465         self.docking_station_marker.color.r = marker_color['r']
466         self.docking_station_marker.color.g = marker_color['g']
467         self.docking_station_marker.color.b = marker_color['b']
468         self.docking_station_marker.color.a = marker_color['a']
469 
470         self.docking_station_marker.header.frame_id = 'odom'
471         self.docking_station_marker.header.stamp = rospy.Time.now
    ()
472         self.docking_station_marker.pose = self.waypoints[0]
473 
474     def shutdown(self):
475         rospy.loginfo("Stopping the robot...")
476         self.sm_deliverfood.request_preempt()
477         self.cmd_vel_pub.publish(Twist())
478         rospy.sleep(1)
479 
480 if __name__ == '__main__':
481     try:
482         MyMainClass()
483     except rospy.ROSInterruptException:
484         rospy.loginfo("SMACH test finished.")
```

B.2 Waiting Order State (computer_vision_beep.py)

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

import rospy
import smach
import actionlib
from smach import State, StateMachine
from smach_ros import SimpleActionState, IntrospectionServer
from geometry_msgs.msg import Twist
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from geometry_msgs.msg import Pose, PoseWithCovarianceStamped,
    Point, Quaternion, Twist
from visualization_msgs.msg import Marker
from ar_track_alvar.msg import AlvarMarkers

import random
from sound_play.msg import SoundRequest
from sound_play.libsoundplay import SoundClient
from std_msgs.msg import String

class ComputerVision(State):
    def __init__(self):
        State.__init__(self, outcomes=['detect1','detect2','
    detect3','detect4','detect5','detect6', 'preempted'])
        self.taskname="doing the computer vision task"

        # initialize a subscriber of ar_pose_marker
        rospy.Subscriber("ar_pose_marker", AlvarMarkers, self.
    my_listener)
        self.tag_ids = [1,2,3,4,5,6]
        self.execute_flag = False

        self.marker_table_dictionary = {1:'detect1', 2:'detect2',
    3:'detect3', 4:'detect4', 5:'detect5', 6:'detect6'}
        self.my_outcome = self.marker_table_dictionary[1]
        self.n_markers = 0
        self.my_tagid = 0

```

```
34          self.soundstring_publisher = rospy.Publisher('
    sound_message', String, queue_size=10)
35
36      def execute(self, userdata):
37          self.execute_flag = True
38          self.soundstring_publisher.publish("waiting_order")
39          rospy.sleep(3);
40
41          # while there is no marker, hold the execution of execute
    function
42          while(self.n_markers==0):
43
44              if self.preempt_requested():
45                  rospy.loginfo("State ComputerVision is being
    preempted!!!")
46                  self.service_preempt()
47                  return 'preempted'
48
49              rospy.loginfo("waiting for delivery command...")
50              rospy.sleep(1)
51
52          # right after there is marker return an outcome
53          rospy.loginfo("number detected! it is: " + str(self.
    my_tagid) )
54
55          if (self.my_tagid==1):
56              self.soundstring_publisher.publish("O1")
57              rospy.sleep(3)
58              self.soundstring_publisher.publish("M1")
59              rospy.sleep(3)
60          elif (self.my_tagid==2):
61              self.soundstring_publisher.publish("O2")
62              rospy.sleep(3)
63              self.soundstring_publisher.publish("M2")
64              rospy.sleep(3)
65          elif (self.my_tagid==3):
66              self.soundstring_publisher.publish("O3")
67              rospy.sleep(3)
68              self.soundstring_publisher.publish("M3")
```

```
69              rospy.sleep(3)
70          elif (self.my_tagid==4):
71              self.soundstring_publisher.publish("O4")
72              rospy.sleep(3)
73              self.soundstring_publisher.publish("M4")
74              rospy.sleep(3)
75          elif (self.my_tagid==5):
76              self.soundstring_publisher.publish("O5")
77              rospy.sleep(3)
78              self.soundstring_publisher.publish("M5")
79              rospy.sleep(3)
80          else:
81              self.soundstring_publisher.publish("O6")
82              rospy.sleep(3)
83              self.soundstring_publisher.publish("M6")
84              rospy.sleep(3)
85
86
87          self.reset_class_attribute()
88          return self.my_outcome
89
90      def my_listener(self, msg):
91
92          if self.execute_flag:
93
94              # get the number of markers
95              self.n_markers = len(msg.markers)
96
97              # if there is no marker stop the listener
98              # at least we already have the number of markers
99              if self.n_markers == 0:
100                 return
101
102             # assuming there are only 1 tag at once
103             for tag in msg.markers:
104
105                 # Skip any tags that are not in our list
106                 if self.tag_ids is not None and not tag.id in self
    .tag_ids:
```

```
107                     continue
108
109                 self.my_tagid   = tag.id
110                 self.my_outcome = self.marker_table_dictionary[tag
    .id]
111
112     # this function will reset every attribute of this class
113     def reset_class_attribute(self):
114         self.execute_flag = False
115         self.n_markers = 0
```

B.3 Rotate State (rotate360.py)

```
1  #!/usr/bin/env python
2
3  import rospy
4  import smach
5  import actionlib
6  from smach import State, StateMachine
7  from geometry_msgs.msg import Twist
8  from geometry_msgs.msg import Point, Quaternion, Twist
9  from ar_track_alvar.msg import AlvarMarkers
10
11 from math import radians, pi
12 from rbx1_nav.transform_utils import quat_to_angle,
       normalize_angle
13 import tf
14
15 from std_msgs.msg import String
16
17 class DoStuffs(State):
18     def __init__(self, timer):
19         State.__init__(self, outcomes=['succeeded','aborted','
       preempted'])
20         self.timer=timer
21         self.soundstring_publisher = rospy.Publisher('
       sound_message', String, queue_size=10)
22
23     def execute(self, userdata):
```

```
24          self.soundstring_publisher.publish("sound_option5")
25          rospy.sleep(2)
26
27          rospy.loginfo("Waiting for customer to take the food")
28          counter = 0
29
30          while (counter<self.timer):
31              counter+=1
32              rospy.loginfo(str(counter))
33              rospy.sleep(1)
34
35          self.soundstring_publisher.publish("thanks")
36          rospy.loginfo("Thank you for the order, have a nice meal..
        ")
37          rospy.sleep(4)
38          rospy.loginfo("Bye! :D")
39          rospy.sleep(1)
40
41          return 'succeeded'
```

B.4 Search State (search_table_beep.py)

```
1  #!/usr/bin/env python
2
3  import rospy
4  import smach
5  import actionlib
6  from smach import State, StateMachine
7  from geometry_msgs.msg import Twist
8  from geometry_msgs.msg import Point, Quaternion, Twist
9  from ar_track_alvar.msg import AlvarMarkers
10
11 from math import radians, pi
12 from rbx1_nav.transform_utils import quat_to_angle,
       normalize_angle
13 import tf
14
15 class Rotate360(State):
16     def __init__(self, speed, goal):
```

```
17          State.__init__(self, outcomes=['full_rotate', '
    not_full_rotate'])
18
19          # Initialize the tf listener
20          self.tf_listener = tf.TransformListener()
21
22          # Give tf some time to fill its buffer
23          rospy.sleep(2)
24
25          # Set the odom frame
26          self.odom_frame = '/odom'
27
28          # Find out if the robot uses /base_link or /base_footprint
29          try:
30              self.tf_listener.waitForTransform(self.odom_frame, '/
    base_footprint', rospy.Time(), rospy.Duration(1.0))
31              self.base_frame = '/base_footprint'
32          except (tf.Exception, tf.ConnectivityException, tf.
    LookupException):
33              try:
34                  self.tf_listener.waitForTransform(self.odom_frame,
     '/base_link', rospy.Time(), rospy.Duration(1.0))
35                  self.base_frame = '/base_link'
36              except (tf.Exception, tf.ConnectivityException, tf.
    LookupException):
37                  rospy.loginfo("Cannot find transform between /odom
     and /base_link or /base_footprint")
38                  rospy.signal_shutdown("tf Exception")
39
40          self.cmd_vel = rospy.Publisher('cmd_vel', Twist)
41
42          self.r = rospy.Rate(20)
43          self.angular_speed = speed
44          self.angular_tolerance = radians(2.5)
45          self.goal_angle = goal
46
47      def execute(self, userdata):
48          rospy.loginfo("Rotating...")
49
```

```
50          # Get the starting position values
51          (position, rotation) = self.get_odom()
52          last_angle = rotation
53          turn_angle = 0
54
55          rotate_command = Twist()
56          rotate_command.angular.z = self.angular_speed
57
58          while abs(turn_angle + self.angular_tolerance) < abs(self.
    goal_angle) and not rospy.is_shutdown():
59              if self.preempt_requested():
60                  rospy.loginfo("State Rotate360 is being preempted
    !!!")
61                  self.service_preempt()
62                  return 'not_full_rotate'
63
64              # Publish the Twist message and sleep 1 cycle
65              self.cmd_vel.publish(rotate_command)
66              self.r.sleep()
67
68              # Get the current rotation
69              (position, rotation) = self.get_odom()
70
71              # Compute the amount of rotation since the last loop
72              delta_angle = normalize_angle(rotation - last_angle)
73
74              # Add to the running total
75              turn_angle += delta_angle
76              last_angle = rotation
77
78          # Stop the robot before the next leg
79          rotate_command = Twist()
80          self.cmd_vel.publish(rotate_command)
81          rospy.sleep(1)
82
83          return 'full_rotate'
84
85      def get_odom(self):
```

```
86          # Get the current transform between the odom and base
    frames
87          try:
88              (trans, rot)  = self.tf_listener.lookupTransform(self.
    odom_frame, self.base_frame, rospy.Time(0))
89          except (tf.Exception, tf.ConnectivityException, tf.
    LookupException):
90              rospy.loginfo("TF Exception")
91              return
92
93          return (Point(*trans), quat_to_angle(Quaternion(*rot)))
```

B.5 Waiting State (do_stuffs_beep.py)

```
1  #!/usr/bin/env python
2
3  import rospy
4  import smach
5  import actionlib
6  from smach import State, StateMachine
7  from smach_ros import SimpleActionState, IntrospectionServer
8  from geometry_msgs.msg import Twist
9  from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
10 from geometry_msgs.msg import Pose, PoseWithCovarianceStamped,
       Point, Quaternion, Twist
11 from visualization_msgs.msg import Marker
12 from ar_track_alvar.msg import AlvarMarkers
13
14 from sound_play.msg import SoundRequest
15 from sound_play.libsoundplay import SoundClient
16 from std_msgs.msg import String
17
18 class SearchTable(State):
19     def __init__(self):
20         State.__init__(self, outcomes=['detect','not_detect'])
21
22         rospy.Subscriber("ar_pose_marker", AlvarMarkers, self.
       my_listener)
23         self.n_markers = 0
```

```
24          self.table_flag = False
25          self.execute_flag = False
26
27          self.sum_x = 0
28          self.counter = 0
29          self.mean_distance = 0
30
31          self.soundstring_publisher = rospy.Publisher('
    sound_message', String, queue_size=10)
32
33      def execute(self, userdata):
34          self.execute_flag = True
35          rospy.loginfo("execute SearchTable class")
36
37          self.soundstring_publisher.publish("table_search")
38          rospy.sleep(4)
39
40          # while( self.n_markers==0 or not(self.my_tagposition_y >
    -0.2 and self.my_tagposition_y < 0.2)):
41          while( not(self.table_flag) or (self.mean_distance > 1) ):
42
43              if self.preempt_requested():
44                  rospy.loginfo("State SearchTable is being
    preempted!!!")
45                  self.service_preempt()
46                  self.reset_class_attribute()
47
48                  message_str = "sound_option4" # mario die sound
49                  self.soundstring_publisher.publish(message_str)
50                  rospy.sleep(3)
51                  return 'not_detect'
52
53              rospy.loginfo("Searching for table...")
54              rospy.sleep(1)
55
56          rospy.loginfo("Table detected!")
57          self.reset_class_attribute()
58          return 'detect'
59
```

```
60      def my_listener(self, msg):
61
62          if self.execute_flag:
63              # get the number of markers
64              self.n_markers = len(msg.markers)
65
66              # if there is no marker stop the listener
67              # at least we already have the number of markers
68              if self.n_markers == 0:
69                  return
70
71              # assuming there are only 1 tag at once
72              for tag in msg.markers:
73
74                  if tag.id is 7:
75                      self.table_flag = True
76
77                      self.counter=self.counter+1
78                      self.sum_x = self.sum_x + tag.pose.pose.
    position.x
79                      self.mean_distance = self.sum_x/self.counter
80
81                      rospy.loginfo("detected: " +str(tag.id) + ",
    it is: "+str(tag.pose.pose.position.x) + "m")
82                      rospy.loginfo("the mean is: "+str(self.
    mean_distance) + "m")
83
84      # this function will reset every attribute of this class
85      def reset_class_attribute(self):
86          self.table_flag = False
87          self.execute_flag = False
88          self.n_markers = 0
89
90          self.sum_x = 0
91          self.counter = 0
92          self.mean_distance = 0
93
94  #done
```

B.6 Workstation PC Launch File

```
<launch>
  <node pkg="image_view" type="image_view" name="my_camera_node">
    <remap from="image" to="/camera/rgb/image_color" />
  </node>

  <!--
  <node pkg="rbx2_tasks" type="delivering_food.py" name="
    deliver_food_node" />
  <node pkg="rbx2_tasks" type="something.py" name="something_node"
     />
  <node name="soundplay_node" pkg="sound_play" type="
    soundplay_node.py"/>
  <include file="$(find rbx2_ar_tags)/launch/ar_indiv_kinect.
    launch" />
  -->

  <include file="$(find turtlebot_rviz_launchers)/launch/
    view_navigation.launch" />

</launch>
```

LISTING B.1: Workspace Launch File

B.7 Turtlebot Notebook Launch File

```
<launch>
  <include file="$(find turtlebot_le2i)/launch/
    remap_rplidar_minimal.launch" />
  <include file="$(find rbx1_nav)/launch/tb_demo_amcl.launch" >
        <arg name="map" value="my_arena.yaml" />
  </include>

  <!--
  <include file="$(find rbx2_vision)/launch/openni_node.launch" />
  -->
  <include file="$(find rbx2_ar_tags)/launch/ar_indiv_kinect.
    launch" />
```

```
11   <include file="$(find turtlebot_bringup)/launch/3dsensor.launch"
       />
12
13   <node name="soundplay_node" pkg="sound_play" type="
       soundplay_node.py" />
14   <node name="soundlistener_node" pkg="rbx2_tasks" type="
       sound_listener.py" />
15
16 </launch>
```

LISTING B.2: Turtlebot Launch File

B.8 Full Smach Graph

The next page shows the full image of our State Machine that provided by Smach Viewer. Details already explained in Chapter 5.

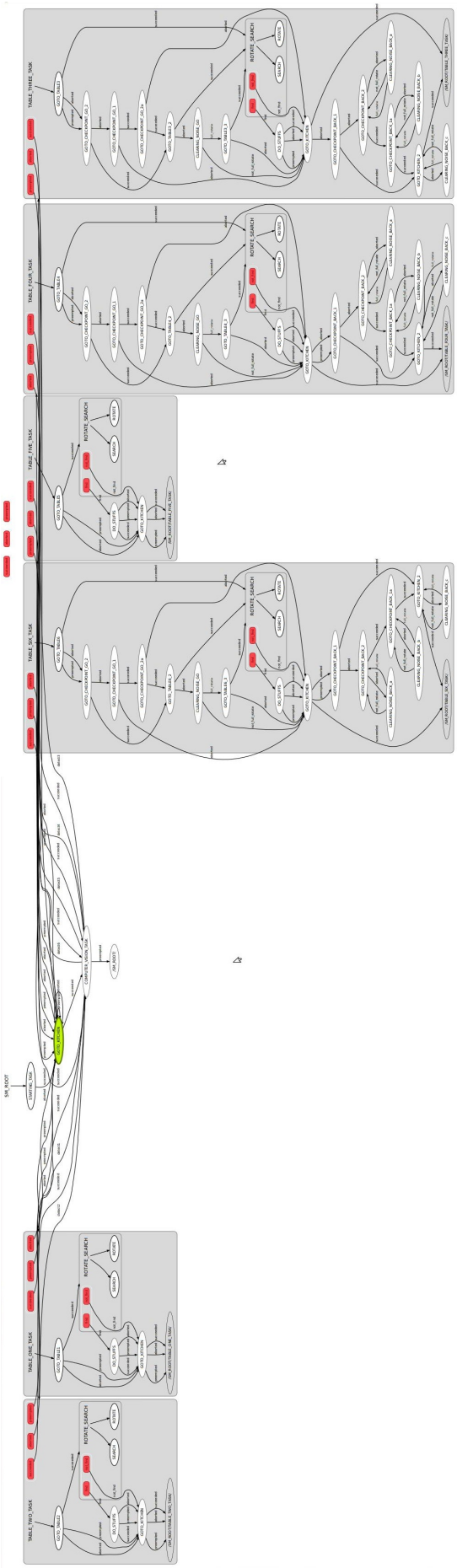

FIGURE B.1: Full Smach Graph from Smach Viewer

Bibliography

[1] *About ROS*. http://www.ros.org/about-ros/. Accessed: December 2015.

[2] *Amcl*. http://wiki.ros.org/amcl. Accessed: December 2015.

[3] *Ar Track Alvar*. http://wiki.ros.org/ar_track_alvar. Accessed: December 2015.

[4] *gmapping*. http://wiki.ros.org/gmapping. Accessed: December 2015.

[5] Patrick Goebel. *ROS By Example*. Ed. by Lulu. Lulu, 2013. URL: \url{http://www.lulu.com/shop/r-patrick-goebel/ros-by-example-hydro-volume-1/ebook/product-21393108.html}.

[6] Patrick Goebel. *ROS By Example*. Ed. by Lulu. Lulu, 2014. URL: \url{http://www.lulu.com/shop/r-patrick-goebel/ros-by-example-volume-2-hydro/ebook/product-21756214.html}.

[7] *Intro to ROS*. http://www.clearpathrobotics.com/guides/ros/Intro_to_the_Robot_Operating_System.html. Accessed: December 2015.

[8] *MoveBaseActionGoal Message*. http://docs.ros.org/fuerte/api/move_base_msgs/html/msg/MoveBaseActionGoal.html. Accessed: December 2015.

[9] *OpenSlam*. http://openslam.org/gmapping.html. Accessed: December 2015.

[10] *Point*. http://docs.ros.org/fuerte/api/geometry_msgs/html/msg/Point.html. Accessed: December 2015.

[11] *Point Message*. http://wiki.ros.org/sound_play. Accessed: December 2015.

[12] Ralph Seulin. *ROS Tutorial Module 4*. 2015.

[13] *Smach*. http://wiki.ros.org/smach. Accessed: December 2015.

[14] *Smach Class*. http://docs.ros.org/diamondback/api/smach/html/python/smach.state_machine.StateMachine-class.html. Accessed: December 2015.

[15] *Turtlebot Navigation*. http://wiki.ros.org/turtlebot_navigation. Accessed: December 2015.