

**School of Electronic  
Engineering  
and Computer Science**

MSc Artificial Intelligence  
Project Report 2019

**Programming by  
Demonstration of  
Human-Robot  
Collaborative Tasks**



Tanushree Chandrakant Mahajan  
180702224

December 2019

# **ACKNOWLEDGEMENT**

I would like to express my sincere gratitude and thanks to my supervisor Dr. Lorenzo Jamone for his support and guidance throughout the project duration. His positive feedback and motivation has helped me a lot during this project and without which it would had not been possible to progress during the course.

This undertaking has helped me in enhancing my knowledge and skills. I would like to thank Queen Mary University of London for giving me this opportunity and every professor who has helped me grow my learning curve and achieve new milestone.

I thank my family and friends for their emotional support during this time period without which it would have been very difficult to accomplish this goal.

## **ABSTRACT**

Human Robot Collaboration is a wide research topic which is worked upon by researchers to enable robots work effectively and economically with humans to complete specific task. The use of robots in low production environment is not cost effective and engaging robots entirely in high volume production environment is a potential risk to the industry. Hence, performing tasks collaboratively with the robot is an efficient and productive way of performing any task. Imitation learning can be referred to as a process wherein a robot can be trained such that it can be possible to handle any difficult or tedious tasks. The robot will accomplish this by simply imitating the human operator and help completing the task. Therefore, making it less tedious and troublesome for the human operator. Programming by demonstration can be viewed as imitation learning which encompasses ways to start with a good solution or eliminate the bad solutions from the set of training solutions. Thus, the main idea is to teach the robot a simple blocksworld task using the Programming by Demonstration technique.

# CONTENTS

<b>ABSTRACT.....</b>	<b>i</b>
<b>1 INTRODUCTION</b>	<b>1</b>
1.1 Background .....	1
1.2 Motivation.....	2
1.3 Objectives.....	3
<b>2 REQUIREMENTS</b>	<b>4</b>
2.1 Software Requirements .....	4
2.1.1 Robot Operating System .....	4
2.1.2 Ubuntu.....	4
2.1.3 Move-it.....	5
2.1.4 Gazebo Simulator.....	6
2.2 Hardware Requirements .....	7
2.2.1 Maria Robot .....	7
<b>3 PLANNING</b>	<b>7</b>
<b>4 TECHNIQUES USED</b>	<b>8</b>
4.1 Relational Reinforcement Learning .....	8
4.2 Markov Decision Process .....	9
4.3 Relational MDP .....	11
4.4 Tree Boosting for Relational Imitation Learning .....	13
4.5 PDDL – Planning Domain Definition Language .....	14
<b>5 IMPLEMENTATION ARCHITECTURE</b>	<b>17</b>

<b>6</b>	<b>IMPLEMENTATION</b>	<b>18</b>
6.1	Blocksworld Domain .....	18
6.2	Implementation of the relational MDP rules .....	19
6.3	Implementation of the Percept .....	21
6.4	Implementation of the decision.....	21
6.5	Implementation of the action.....	23
<b>7</b>	<b>ROBOT PROGRAMMING METHOD</b>	<b>23</b>
7.1	ROS Core .....	23
7.2	ROS- URDF Files .....	25
7.3	ROS- SDF Files .....	26
7.4	ROS configurations .....	28
7.5	Move-it configurations.....	29
7.6	Move-it and Gazebo simulation .....	30
7.7	Python script execution .....	34
7.7.1	Importing the robot configurations .....	34
7.7.2	Obtaining the coordinates of the blocks .....	35
7.7.3	Moving the arm according to the actions .....	35
<b>8</b>	<b>EXPERIMENTS</b>	<b>37</b>
<b>9</b>	<b>DISCUSSIONS</b>	<b>41</b>
<b>10</b>	<b>CONCLUSION AND FUTURE WORK</b>	<b>42</b>
<b>11</b>	<b>REFERENCES</b>	<b>42</b>

## List of Figures

Figure 1: Moveit-Maria Robot.....	5
Figure 2: Maria Robot configured in Gazebo.....	6
Figure 3: Project implementation plan.....	7
Figure 4: Reinforcement Learning.....	9
Figure 5: Example sketch of a transition in blocksworld.....	13
Figure 6: Problem Definition example.....	14
Figure 7: Simple Domain example.....	16
Figure 8: Application architecture.....	17
Figure 9: Get Model State -Perception Node .....	18
Figure 10: Problem file used.....	19
Figure 11: Domain File used.....	20
Figure 12: Domain file - tap_left action.....	20
Figure 13: The three rostopic created for publishing the coordinates.....	21
Figure 14: message file created (.msg).....	24
Figure 15: the rostopics published.....	24
Figure 16: Description of links and joints in a robotic arm.....	25
Figure 17: sdf file of red_box.....	27
Figure 18: model.config file for the red_box.....	28
Figure 19: red_box added to the world file.....	28

Figure 20: moveit installation command.....	29
Figure 21: Moveit - Maria Robot.....	29
Figure 22: Controller definition file.....	30
Figure 23: Maria configuration in moveit and gazebo simulated successfully...	31
Figure 24: Adding the marker and TF in rviz.....	32
Figure 25: After adding the Tf and Marker plugins.....	32
Figure 26: Models loaded successfully in rviz.....	33
Figure 27: Models from gazebo successfully simulated in rviz.....	33
Figure 28: Imported libraries and configurations in python script.....	34
Figure 29: Receiving the block coordinates using the service /gazebo/get_model_state.....	35
Figure 30: Gazebo world as a 3D model. Blue represents Z axis, Green represents Y axis and Red represents X axis.....	36
Figure 31: Implementation of waypoints as a cartesian path.....	36
Figure 32: Initial state of the robot and the blocks.....	38
Figure 33: The goal state where all the boxes are attached to each other.....	38
Figure 34: Initial state of scenario 2 .....	39
Figure 35: Goal state of scenario 2 .....	39
Figure 36: Scenario 3- Already attached .....	40

# 1. INTRODUCTION

## 1.1 Background

Programming by Demonstration (PbD) was considered to be an excellent alternative to manual programming of robots as was realized as a way to reduce the cost invested in the development and maintenance of the robots [1]. PbD can be viewed as Imitation learning, where a robot mimics the human demonstrator and tries to perform a task. It does so by minimising the number of states, that is, search spaces used by the robot during the learning process. It executes this by deciding on good or bad solutions to a state, which can be used by robot to follow either of the two below mentioned approaches:

1. Start to search from a good solution observed.
2. Eliminate from the search space what is called a bad solution, so as to reduce the search space [1]

Since early 1980s it became popular in the field of manufacturing. Initially the techniques involved in PbD were manual control in which the human operator teleoperated the robotic arm and demonstrated the motion which was recorded in the form of positions of the end effector and the forces[1]. These parameters were then used by the robot to imitate the motion [1]. Another approach in the field followed was that human operator demonstrated the task (such as pick and place) for a few times and the robot is expected to adapt to the changing environment and help the human operator to complete the



task [2]. However, the task was demonstrated by the human operator by manually moving the robotic arm. The robot was then expected to perform the task after couple of demonstrations. This also included capability of avoiding the obstacles by the robot using suitable policies[2]. Another aspect to this could be that as humans have a tendency to learn quickly by observing demonstrations, this can be applied to robots as well and can prove an efficient way of communication [16]

## **1.2 Motivation**

Human Robot Collaboration is a research field with a range of applications in fields of medicine, construction, elderly care etc. wherein it is applied to very complex working environments. It is basically team of human and robot who are partners which collaboratively completes a given task or reaches a specific goal. In this case, it is usually human who states that goal, while robot assist human to complete the goal. Programming by demonstration can be referred to as imitation learning which aims at reducing the search space by either starting from the good solution or eliminate the bad solution from the search space. Currently, it is applicable only to teach simple tasks to robots such as performing specific arm movements, but the techniques does not permit the natural movements for teaching. The traditional methods involved in imitation learning included presenting the demonstrations by manually moving the robot, capturing the trajectory and executing which limits the users demonstration capabilities.

### **1.3 Objective**

The objective of this project is to perform tasks with the robot using software modules for natural programming by demonstration of object manipulation tasks. The tasks are performed in simulation using the gazebo simulation tool and Moveit along with ROS (Robot Operating System) and the robot should be able to perform the task as demonstrated by the human operator. In order to perform the tasks, relational domain will be considered which includes multiple objects and relationship between them. The robot used here is “Maria Robot” which comprises of two UR5 arms, left and right arm. Other objectives involve:

- a. Understand the concept of Programming by demonstration (PbD) in relational domain.
- b. Understanding the working and fundamentals of Robot Operating System
- c. Understand the working of simulation tool Gazebo and how to interface it for real world programming.
- d. Understand the Robot manipulation techniques and implement them using ROS.

## **2. REQUIREMENTS**

This section gives a description of the software and technologies that are used during the project.

### **2.1 Software**

#### **2.1.1 Robot Operating System**

Robot Operating System (ROS) is a meta operating system for the robots which enables the user to write robot software using the available tools and libraries. It eases the task of creating complex robot behaviour for a variety of robotic platforms. It helps in collaborative behaviour for the robots [3]. It is a system providing various functionalities such as message passing, package management , etc to create robot applications [4]. The version used in this project is ROS Kinetic Kame which is supported on Ubuntu Xenial (16.04 LTS)

The Maria Robot will be programmed and simulated using ROS kinetic

#### **2.1.2 Ubuntu**

Ubuntu is open source software operating system which manages the basic functionalities such as peripherals handling and schedule the tasks [5]. It is based on Linux distributions. The Ubuntu 16.04 distribution is the version which supports ROS Kinetic Kame.

### 2.1.3 Move-it Package

It is easy to understand software which is used for robot manipulation to develop applications and test robotics software on the joints and links of robots for planning [6]. It has various capabilities such as 3D perception, control trajectories, inverse kinematics, etc. The Maria robot is configured successfully in moveit and will be used along with ROS and simulated in Gazebo. Below is the configuration and visual of Maria Robot in Moveit:

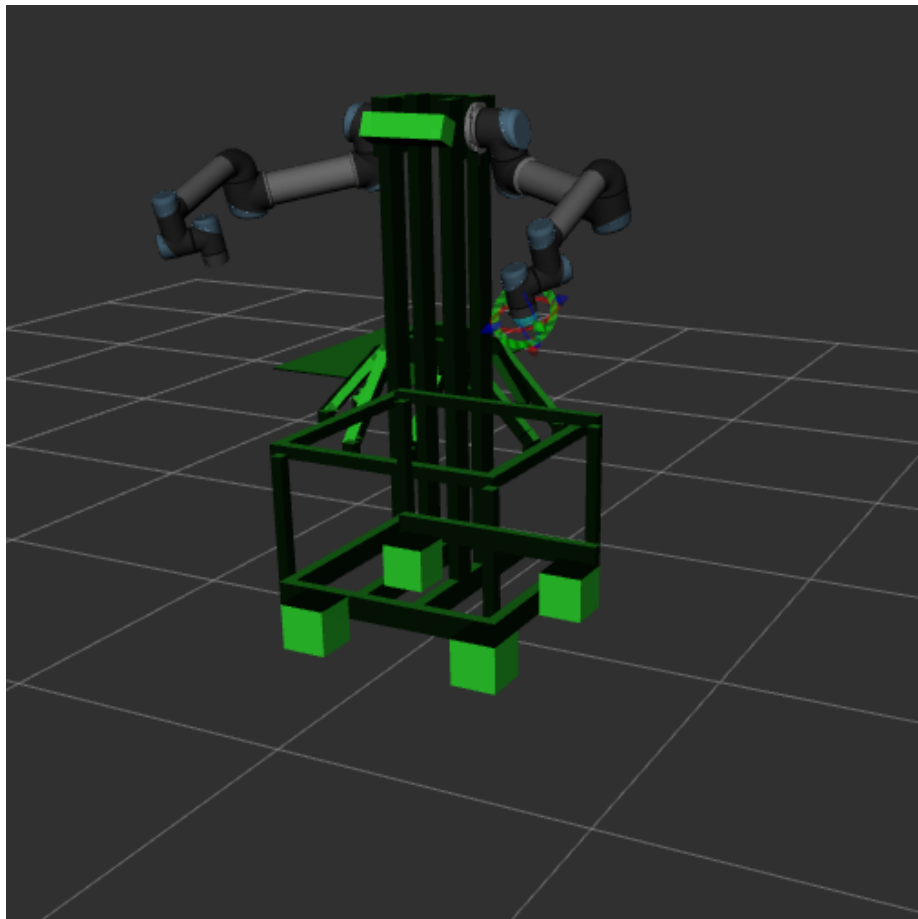


Figure 1: Moveit- Maria Robot

### 2.1.4 Gazebo Simulator

Gazebo simulator is a simulation tool to perform the testing on the algorithms, design robots and test the AI systems using realistic scenarios. It would be used along with Moveit and ROS to perform robot development using objects and dynamics environments and models [7]. Robot programming must be successfully implemented and executed in the simulation before implementing it on the real robot.

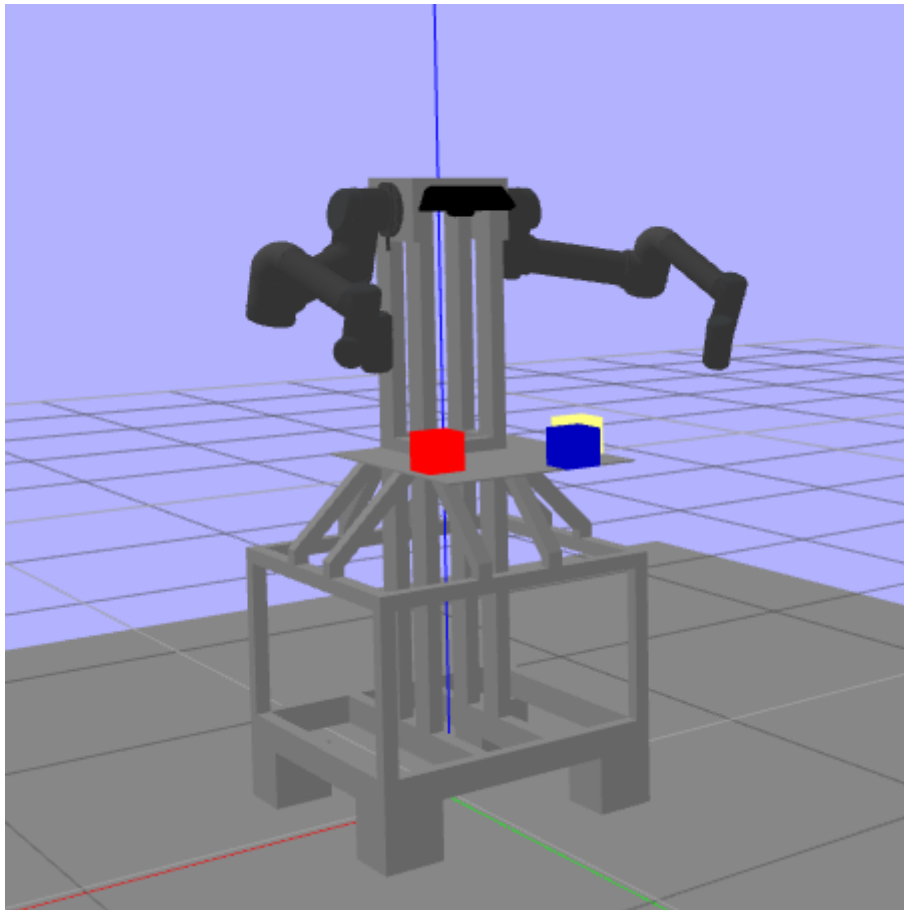


Figure 2: Maria Robot configured in Gazebo

## 2.2 Hardware

### 2.2.1 Maria Robot

The robotic framework which is used is called Maria. It has UR5 robotic arms attached, namely left and right arm

## 3. PLANNING

Phase 1: 8 <sup>th</sup> Oct '19 – 20 <sup>th</sup> Oct '19	Literature review on the keywords: human robot collaboration, programming by demonstration
Phase 2: 21 <sup>st</sup> Oct '19 – 30 <sup>th</sup> Oct '19	Understanding the fundamentals of ROS Move-it and gazebo simulation
Phase 3: 8 <sup>th</sup> Nov '19 – 15 <sup>th</sup> Nov '19	Understand how to visualize the objects in the gazebo world
Phase 4: 16 <sup>th</sup> Nov '19 – 25 <sup>th</sup> Nov '19	Understand how to implement and use the position tracking of blocks
Phase 5: 27 <sup>th</sup> Nov '19 – 4 <sup>th</sup> Dec '19	Learn to move the robotic arm using gazebo and how to perform the actions
Phase 6: 1 <sup>st</sup> Dec '19 – 6 <sup>th</sup> Dec '19	Understand the decision node structure and write the rules for it
Phase 7: 5 <sup>th</sup> Dec '19 – 8 <sup>th</sup> Dec '19	Test if the rules are successfully implemented and the goal state is reached by the robot or not

Figure 3: Project implementation plan

## **4. TECHNIQUES USED**

### **4.1 Relational Reinforcement Learning**

Reinforcement learning can be thought of as a techniques where the agent continuously interacts with the environment to learn as the agent does not have the complete information and needs to learn to act accordingly. The agent improves its performance over period of time [8]. Relational Reinforcement Learning can be referred to as a technique which combines reinforcement learning with relational learning. As relational reinforcement learning provides more expressive representation of objects and relations between them it is used for variety of tasks. As reinforcement learning involves attribute valued representation it puts restrictions in describing even simple planning tasks for blocks world.

As opposed to attribute valued representation, relational representation consists of variables that help describe the task better and used to solve real world problems and can be used to plan the tasks for all objects. Example, if we have to move object a towards object b, the results would be same if we consider moving object b towards object c.

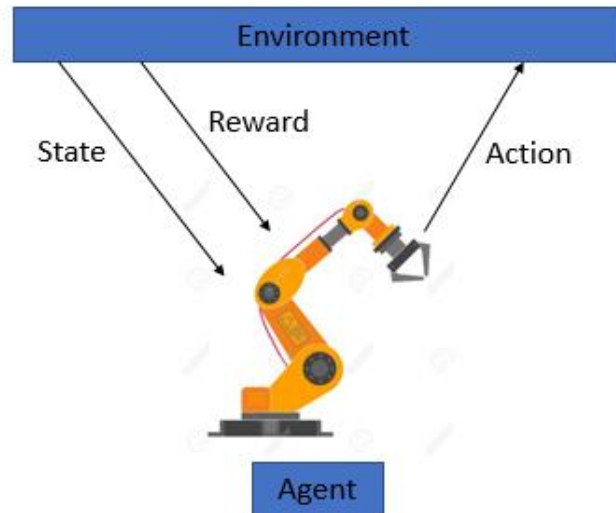


Figure 4: Reinforcement Learning

## 4.2 Markov Decision Process

The approach for PbD relies on Markov Decision Process (MDP) framework. In an MDP, the process being considered should be guided by Markov's underlying chains. In a Markov chain, a process moves from state to state spontaneously in discrete time intervals, and the probability of transition from state to state depends only on the current state and not on the previous state. Furthermore, the system is needed for choosing an action or a control of a number of actions in a subset of States (the set of decision-making States). A policy determines the action to be taken in all states; it maps the actions of one state to another. Under the influence of an action, an immediate reward is earned that may be positive or negative or zero while transitioning from one state to another. The policy's success indicator is usually a feature (the goal variable) of immediate rewards that are obtained when a policy is followed across a predetermined timescale which can be finite or infinite.



It can be defined as a tuple:  $M_R = \{S, A, R, P, \gamma\}$  where,

S is the state space  $\{s_i\}_{1 \leq i \leq N_s}$

A is the action space  $\{a_i\}_{1 \leq i \leq N_A}$

R is the reward function

T is the transition probability

$\gamma$  is the discount factor in the range of 0 to 1

P is the markovian dynamics which gives the probability of going to state  $s'$  from state  $s$  by performing action  $a$

The agent aims to maximize the sum discounted rewards which is given by

$$\sum_{t=0}^{\infty} \gamma^t r_t$$

The policy  $\pi$  is the decision function with the help of which we get the probability to choose an action  $\pi(s, a) = P(s|a)$

The quality function for a given policy expressed as  $Q^\pi_R \in \mathbb{R}^{S \times A}$  is a measure of the performance of this policy. It is expressed for each state-action couple  $(s, a)$  in the form of expected cumulative discounted reward when starting state is  $s$ , performing the action  $a$  and following the policy  $\pi$  afterwards. [10]

Under a policy, the quality function is computed as-

$Q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} P((s, a)) \sum_{a' \in A} \pi(s', a') Q^\pi(s', a')$  that would give the expected sum of discounted future rewards.

### 4.3 Relational MDP

Relational MDPs are a way to generalize the MDPs using high level representations. Using this different objects can be modelled rather than just instances [12]. The main concepts involved here can be explained below:

- Constant- it is a object in real world.
- Variable- can be substituted by a constant. In prolog notation, variables always starts with a capital letter and constant with small letter.
- Term- it is either a constant or a variable
- Predicate- It represents the a relation between n different objects
- Atom- it is an expression of the form  $p(a_1, a_2, \dots, a_n)$  where p is the predicate and a are the terms.
- Formula- it is a set of atoms and negated atoms. Formulas are built using the symbols,  $\wedge$ ,  $\vee$ ,  $\rightarrow$  and  $\sim$  which correspond to “and”, “or”, “if..then” and “not” respectively [12]
- Substitution- Substitute a variable with a term
- Grounding- is a substitution where there are no variables in the mapped terms.

Using the above concepts any state of the environment can be represented. For example, consider a blocksworld domain with 3 blocks and an arm. The blocks can be represented by R, B, Y, together with the arm they form 4 objects represented by R, B, Y and A.

The predicates used are-

- $\text{attach}(R, B)$  which indicates whether they are attached,

- `left(R, B)` whether the object is on the left
- `right(R, B)` whether the object is on the right
- `down(R, B)` whether the object is down
- `up(R, B)` whether the object is up
- `ontable(R)` which states whether the block is on the table
- `near(R, A)` whether the object is near the robotic arm

Hence, the state where the object A and B are not attached, arm is near A block and blocks are on the table will be represented as:

```
{~(attach(A,B)) ontable(A) ontable(B) near(A, C)}
```

Traditionally, the transition function can be represented as a set of rules, that contain 3 parts-

action:

{preconditions}

{effects}

Where,

1. actions are specified, actions are atoms
2. formula that represents a precondition to be satisfied
3. formula that represents the effect of executing the action.

Therefore, in this example, the transition function for the action `tap_left(R, B)` will be represented by rule:

`tap_left(A, B, C):`

`{~(attach(A, B)) left(A, B) ontable(A) ontable(B) near(A, C)}`

`{attach(A, B) ontable(A) ontable(B)}`

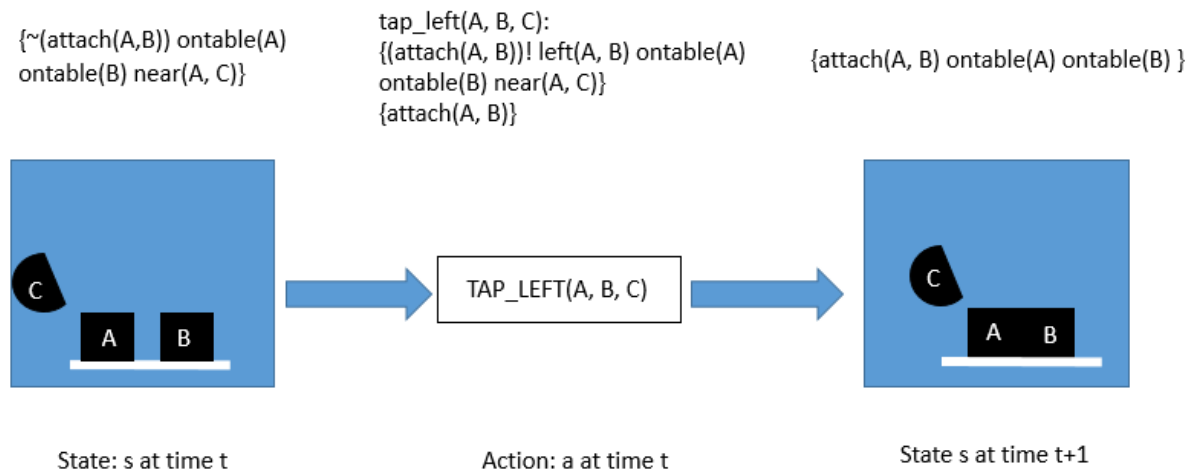


Figure 5: Example sketch of a transition in blocksworld

#### 4.4 TBRIL (Tree Boosting for Relational Imitation Learning)

This is the most widely used policy learning used for relational knowledge using the states and the decisions (actions) provided which is given by :

$$\pi(s,a) = \frac{e^{-\beta q(s,a)}}{\sum_{b \in D(s)} e^{-\beta q(s,b)}}$$

where,  $-\beta q(s, a)$  is the potential function of a given s.

As the joint distribution is unknown finding the function  $q^*$  becomes difficult hence, the gradient is calculated on a subset of the samples (trajectories)

To fit the gradient function at every feature in the training example relational regression trees (RRT) are used. The aim of the tree is to find regression values. It iterates over the tree until it finds a best test for a node according to splitting criteria. Furthermore, the examples are set as success or failure according to the test. The algorithm is repeated for all the splits. The depth limit supplied to the algorithm is set to a minimum value, preferably 3 [13]. The leaves contain the average regression values. Three aggregators count, max and average are used whose values are learned automatically from the data. At each iteration a new set of RRTs is found which try to maximize the likelihood of the distributions [13].

## 4.5 PDDL - Planning Domain Definition Language

PDDL is an encoding language used in problem descriptions and problem domain. In order to plan, there are two files which need to be created, one is problem file and other is domain file, both with extension pddl. Below are the components listed which are key aspects of PDDL:

### PROBLEM DEFINITION:

The problem definition consists of the objects present, initial state and the goal state

The format of a (simple) problem definition is:

```
(define (problem PROBLEM_NAME)
  (:domain DOMAIN_NAME)
  (:objects OBJ1 OBJ2 ... OBJ_N)
  (:init ATOM1 ATOM2 ... ATOM_N)
  (:goal CONDITION_FORMULA)
)
```

Figure 6 : Problem Definition example

The initial state list (:init) consists of all the atoms that are true in the initial state. The goal description is the formula of the same for as declared in the precondition section of the domain file.

## **DOMAIN DEFINITION:**

**The Domain definition consists of :**

- Comments- The comments in pddl start with “;” semicolon and are one liner
- Requirements- This is required as most planners support only a subset. Hence, we need to specify the requirement as one of the following:
  - Strips – it uses only STRIPS
  - Equality – uses the predicate “=” interpreted as equality
  - Typing – means domain uses types
  - Adl – usage of ADL (i.e. disjunctions and quantifiers in goals and preconditions)

It contains the predicates and the operators (also called as actions) Figure 7 is the screenshot of a simple domain definition:

```

(define (domain DOMAIN_NAME)
  (:requirements [:strips] [:equality] [:typing] [:adl])
  (:predicates (PREDICATE_1_NAME ?A1 ?A2 ... ?AN)
               (PREDICATE_2_NAME ?A1 ?A2 ... ?AN)
               ...))

(:action ACTION_1_NAME
  [:parameters (?P1 ?P2 ... ?PN)]
  [:precondition PRECOND_FORMULA]
  [:effect EFFECT_FORMULA]
)

(:action ACTION_2_NAME
  ...)

...)

```

Figure 7: Simple Domain example

The parameters of predicates and actions are distinguished based on the “?” as shown in the above figure. The parameters specified in predicates do not serve any other purpose than specifying the number of arguments a predicate should have. Parameter in the predicates can be zero

- Action definition- The parts of the action definition are all optional however, any action without effect is of no use.
- Precondition formulas-
  - In STRIPS, this can be atomic (PREDICATE\_NAME ARG1...ARGN) or conjunction of atomic formulas (and ATOM1...ATOM2).
  - If the domain is using :equality then it can be of the form (= ARG1 ARG2).
  - ADL Domain- a precondition might be:
    - General negation, conjunction  
(NOT CONDITIONAL\_FORMULA ....  
CONDITIONAL\_FORMULA\_N)

(AND CONDITIONAL\_FORMULA1 ..  
CONDITIONAL\_FORMULA\_N)

- A formula  
(forall (V1? V2? ... ) CONDITIONAL\_FORMULA)  
(exists (V1? V2? ...) CONDITIONAL\_FORMULA)

## 5. IMPLEMENTATION ARCHITECTURE

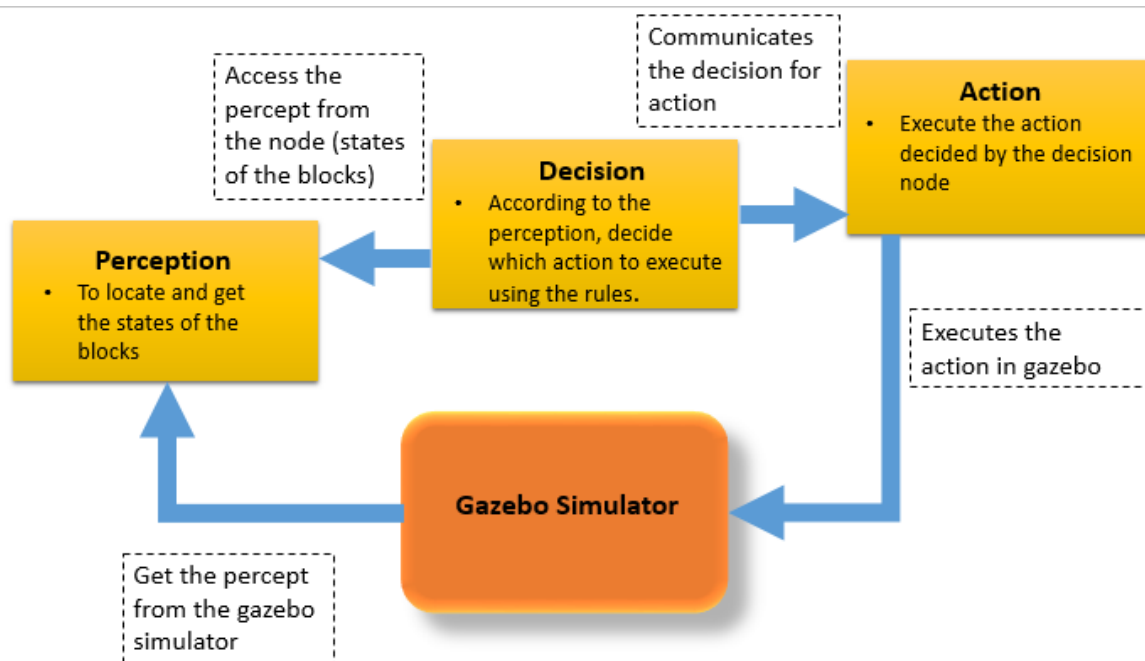


Figure 8 : Application architecture

The figure depicts the overall implementation architecture of the project.

- **The perception node:** This node is responsible for the percept. It gets the states of the blocks placed on the table and publishes it on a ROS topic. Here, the service, /gazebo/get\_model\_state is used to get the position of



each block and publish it so that further decisions can be made. Figure 9 depicts the snippet of the same.

```
model_coordinates = rospy.ServiceProxy('/gazebo/get_model_state', GetModelState)
box_coordinates = model_coordinates(box_name, 'link')
d.x = box_coordinates.pose.position.x
d.y = box_coordinates.pose.position.y
d.z = box_coordinates.pose.position.z
```

Figure 9: Get Model State -Perception Node

- **The decision node:** According to the positions published, this node subscribes them and apply the rules for making a decision on the next move.
- **The action node:** As per the decisions appropriate action is executed namely one of from tap\_left, tap\_right, tap\_up and tap\_down

## 6. IMPLEMENTATION

### 6.1 Blocksworld Domain

The domain which will be used is blocksworld consisting of blocks which is a classical domain with interesting characteristics such as experiment with different kind of tasks with changing number of objects. As we are dealing with relational domain the blockworld can be defined with-

Set of objects- 3 Blocks, Arm

Set of predicates- ontable(X), attach(X, Y), left(X, Y), right(X,Y), isup(X,Y), down(X, Y) and near(X, Z)

Set of actions- tap\_left(X, Y), tap\_right(X, Y), tap\_up(X, Y) and tap\_down(X, Y)

## 6.2 Implementation of the Relational MDP rules

The rules for relational MDP are written as pddl files in PROLOG. PDDL is the Planning Domain Definition Language for problem description.

There are two files primarily created for planning, namely domain and problem. We have used 5 problem files with different scenarios as we are working with 3 boxes.

Figure 10 is the example of one sample problem file used:

```
1(define (problem BLOCKS-3-0)
2 (:domain BLOCKS-typed)
3 (:objects R B Y - block
4          G - arm)
5 (:INIT (NOT (ATTACH R Y)) (NOT (ATTACH R B)) (NOT (ATTACH B Y)) (ONTABLE R) (ONTABLE B) (ONTABLE Y) (LEFT R Y) (LEFT Y B) (RIGHT B Y) (NEAR R G))
6 (:goal (AND (ATTACH R Y)))
7 )
```

Figure 10: Problem file used

The problem file contains the basic elements of any pddl file along with init and goal state. The initial state is according to the predicates and objects we have considered and is for the scenario where any of our boxes are not attached. Our goal state is when all the three boxes are attached. In order to do so, we have 4 actions specified, i.e. tap\_left, tap\_right, tap\_up, tap\_down.

The predicates used here are-

ontable(X) – checks whether the block is on the table

attach(X, Y) – checks whether the two blocks X and Y are attached

left(X, Y) – checks whether the block X is to left of Y

right(X, Y) – checks whether the block X is to the right of Y

up(X, Y) – checks whether the block X is on the upside of Y

down(X,Y) – checks whether the block X is on the down side of Y

near(X, Z) – checks whether the block is near the robotic arm

The domain file contains the requirements and the information about the predicates and objects used. Figure 10 depicts the domain file used here which contains the definition, domain as BLOCKS, predicates. Figure 11 illustrates one of the actions written in the domain file. It consists of rules required to be followed while executing an action. The precondition and effects are part of every action which consists of states (predicates) before executing the action in the form of precondition for the action to be executed and effect of applying the action.

```
1 |;;;;;;;;;;;;;;
2 ;;; 3 Op-blocks world
3 |;;;;;;;;;;;;;;
4
5 (define (domain BLOCKS)
6   (:requirements :strips :typing)
7   (:types block arm)
8   (:predicates (attach ?x - block ?y - block)
9                (ontable ?x - block)
10               (right ?x ?y - block)
11               (left ?x ?y - block)
12               (up x? ?y - block)
13               (down ?x ?y - block)
14               (near ?x - block ?g - arm)
15              )
16
```

Figure 11: Domain File used

```
16
17 (:action tap-left
18   :parameters (?x - block ?y - block ?g - gripper)
19   :precondition (and (left ?x ?y) (ontable ?x) (ontable ?y) (not (attach ?x ?y)) (near ?x ?g))
20   :effect
21     (and (ontable ?x)
22           (ontable ?y)
23           (attach ?x ?y)))
24
```

Figure 12: Domain file - tap\_left action

### 6.3 Implementation of the Percept:

The perception node is responsible for publishing the positions of the blocks in the scene. For this purpose `/gazebo/get_mode_state` service is used which gets the positions of the models we created, in this case the 3 blocks. It then publishes the states of the blocks, namely

Red\_block(x, y, z)

Blue\_block(x, y, z)

Yellow\_block(x, y, z) to the rostopic red\_coordinates, blue\_coordinates, yellow\_coordinates respectively as illustrated in Figure 13

```
def block_states(self):  
    pub_red = rospy.Publisher("red_coordinates", red_block, queue_size=1)  
    pub_yellow = rospy.Publisher("yellow_coordinates", yellow_block, queue_size=1)  
    pub_blue = rospy.Publisher("blue_coordinates", blue_block, queue_size=1)  
    rospy.init_node('Positiontracker', anonymous=True)
```

Figure 13: The three rostopic created for publishing the coordinates

### 6.4 Implementation of the Decision:

The decision node makes decision according to the rules and then passes them to the action node. The node subscribes to the topics and gets the positions. It then uses these positions to decide upon the action and act according to it.

Algorithm:

Function decision()

Step1: Initialize and set the isattached flag to 0

Isattached<- 0

Step2: Repeat the below steps until  $i \leq 3$

2.1: If isattached = 0

2.1.1: If the distance between X coordinates of block[i] and block[i+1] is  $< 0.05$  and  $> 0$

2.1.1.1: If the distance between the Y coordinates of block[i] and block[i+1] is  $> 0.1$

2.1.1.1.1: if the robotic arm is near block[i]

Execute tap\_down()

2.1.1.1.2: else

Execute tap\_up()

2.1.1.2: else they are already attached

isattached  $\leftarrow$  1

2.1.2: else if the distance between Y coordinates of block[i] and block[i+1] is  $< 0.05$  and  $> 0$

2.1.2.1: If the distance between the X coordinates of block[i] and block[i+1] is  $> 0.1$

2.1.2.1.1: if the robotic arm is near block[i]

Execute tap\_left()

2.1.2.1.2: else

Execute tap\_right()

2.1.2.2: else they are already attached

isattached  $\leftarrow$  1

2.2: Else they are attached

## 6.5 Implementation of Action :

The decision is passed from the decision node to action node where the robotic arm execution takes place. According to the actions the arm is moved by creating a Cartesian path. This can be planned using the waypoints and specifying them in the code.

# 7. Robot Programming Method

## 7.1 ROS core

ROS is an open source software for robot programming and can be installed on Ubuntu 16.04. The installation steps are mentioned on the website. Post installation, first step is to create a workspace called “catkin\_make”. This will contain all your package information and the necessary files. The package created inside the workspace contains files such as package.xml which has information about the dependencies, CMakeLists.txt contains the details of how the code should run. The source contains all the python files which would be executed using the launch file [4].

Below are the key features which will be used:

**Nodes:** A node can be viewed as one executing a particular task which can be a part of the process. Several nodes with their individual task make robotic process. The nodes could also be used to send(publish) and receive(subscribe) the data amongst each other, for example a client server executing a simple addition of two numbers amongst which one publishes the data and the other receives the data. In our case we have 3 nodes, namely perception, decision and action which contains the processes and tasks for robot manipulation.

**Messages:** It has the data types used whenever subscribing or publishing the data to a topic. They consists of msg files consisting of structure of the message stored in the message subdirectory of a package. The message contains 2 fields, namely, the data type followed by the data. The data type could be anything, float or a string. We would be using msg for publishing and subscribing the block states for namely, red\_block, blue\_block and yellow\_block which contains the variables x, y and z for the coordinates for each of the different block. All the variables are declared as float 64 type as shown in the figure 14.

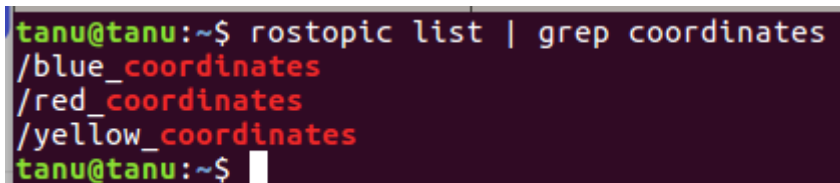
Once the message files are created in msg/ folder, they have to included in the CMakeLists and package.xml files as depicted in the figure:

```
float64 x  # x coordinate in the world
float64 y  # y coordinate in the world
float64 z  # z coordinate in the world
```

Figure 14: message file created (.msg)

**Topics:** They are an entity where nodes can publish the data and can subscribe the data from. It acts like a bus and there can be several publishers and subscribers to one topic[4] .

We have 3 topics created namely for publishing and subscribing to the messages containing the block states. As mentioned in the figure 15, once the perception node publishes the topics they can be seen in the list of rostopics

A terminal window with a dark background and light green text. The prompt is 'tanu@tanu:~\$'. The command 'rostopic list | grep coordinates' has been executed. The output shows three topics: '/blue\_coordinates', '/red\_coordinates', and '/yellow\_coordinates'. The prompt is followed by a cursor.

```
tanu@tanu:~$ rostopic list | grep coordinates
/blue_coordinates
/red_coordinates
/yellow_coordinates
tanu@tanu:~$
```

Figure 15: the rostopics published

## 7.2 ROS – URDF Files

Unified Robot Description format is an XML format which is used for describing model of a robot [17]. The basic components of every URDF files are:

- Robot Name- This identifies the robot which is used in the ROS workspace. ROS will be notified about the robot user is using through this component
- Robot Link- It is a rigid component which connects the joints. The joints are dynamic components which moves between the adjacent links.
- Robot Joints- Joints are responsible for moving the links in a linear, rotational and twisting motion. Every joint provides robot with degree of freedom of motion. Robots are usually identified using the degree of freedom they possess. Joints along with links enable execution of different poses for the robot. The robotic arm can be represented by figure 16 :

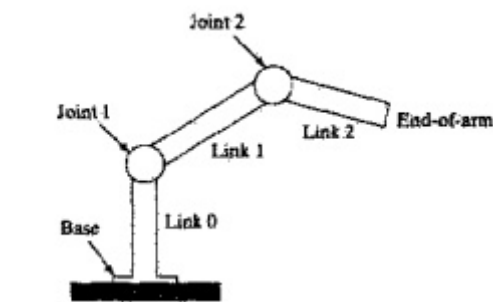


Figure 16: Description of links and joints in a robotic arm

Source: [https://www.brainkart.com/article/Robot-Anatomy-and-Related-Attributes\\_6409/](https://www.brainkart.com/article/Robot-Anatomy-and-Related-Attributes_6409/)



### 7.3 ROS – SDF files

The SDF files are used to specify the models in your environment. SDF models can be used to create simple objects or complex robots in your environment [14]. It is a collective representation of various tags and properties such as links, joints, collision, visual, geometry, pose, etc. The basic components of any SDF files are:

- Links: They contain the physical property of the model. The less links the more stable the model will be. Example: incase of a table with 5 links (4 legs and one top) the model will be unstable. Instead of which only one link with 5 collision elements can be created to ensure stability[14].
  - Collision: It contains the geometry which helps in collision checking
  - Visual: It is used in order to visualize the different links specified and the presence of this tag depends on the link
  - Inertial: This element consists of the dynamic properties of the links, such as mass.
  - Sensor: It is responsible for collecting the data from the world. The link may or not contain any sensor
  - Light: It describes the light source which is attached to the link if required. The light element is optional for the link[14].
- Joints: It connects two links. A parent- child relationship is formed with other specified parameters, for example rotation axis and joint limits [14].
- Plugin: It denotes a shared library which is formed by third party in order to control a model.

In the blocksworld domain, there are 3 models which are created red\_block, blue\_block and yellow\_block. In order to visualize it in gazebo we need to

add the sdf files along with the config files in the model folder. All the models created should be specified here. The sdf file as mentioned above consists of the model name, link and its properties. In this case, the properties include the pose, inertial, box size. Along with the sdf file, any model created should have a config file which specifies the model name and description of the box as shown in figure 18.

Figure 17 includes the sdf file configuration for the red\_block, with the pose and size as <0.1 0.1 0.1>. The box is set to static as false as we want it to be movable by the robot To distinguish between colours the material tags are used and by varying the parameters in the emissive tags:

RGBA(1, 1, 0.5, 1.0) configuration stands for yellow

RGBA(1 0 0 1) configuration appears as red

RGBA(0 0 0.75 1) makes the object as dark blue.

```

1 <?xml version='1.0'?>
2 <sdf version="1.4">
3   <model name="red_block">
4     <static>false</static>
5     <link name="link">
6       <pose>0.1 0.167 1 0 0 0</pose>
7       <inertial>
8         <mass>1.0</mass>
9         <inertia>
10          <ixx>0.083</ixx>      <!-- for a box: ixx = 0.083 * mass * (y*y + z*z) -->
11          <ixy>0.0</ixy>        <!-- for a box: ixy = 0 -->
12          <ixz>0.0</ixz>        <!-- for a box: ixz = 0 -->
13          <iyy>0.083</iyy>      <!-- for a box: iyy = 0.083 * mass * (x*x + z*z) -->
14          <iyz>0.0</iyz>        <!-- for a box: iyz = 0 -->
15          <izz>0.083</izz>      <!-- for a box: izz = 0.083 * mass * (x*x + y*y) -->
16        </inertia>
17      </inertial>
18      <collision name="collision">
19        <geometry>
20          <box>
21            <size>0.1 0.1 0.1</size>
22          </box>
23        </geometry>
24      </collision>
25      <visual name="visual">
26        <material> <!-- Wheel material -->
27          <ambient>0 0 0 1</ambient>
28          <diffuse>0 0 0 1</diffuse>
29          <specular>0 0 0</specular>
30          <emissive>1 0 0 1</emissive>
31        </material>
32        <geometry>
33          <box>
34            <size>0.1 0.1 0.1</size>
35          </box>
36        </geometry>
37      </visual>
38    </link>
39  </model>
40 </sdf>

```

Figure 17: sdf file of red\_block

```

1 <?xml version="1.0"?>
2 <model>
3   <name>red_box</name>
4   <version>1.0</version>
5   <sdf version="1.0">model.sdf</sdf>
6
7   <author>
8     <name>Tanushree</name>
9     <email>t.c.mahajan@se18.qmul.ac.uk</email>
10  </author>
11
12  <description>
13    A small red Box
14  </description>
15 </model>

```

Figure 18: model.config file for the red\_box

In order to visualize the model in the world created, these models should be added to the world file as shown in the figure 19.

As the world file is included in the world file for compilation, it loads the objects in to the world. The world file includes a static tag for the objects which is set to false as our task is to move the object and not make it static.

```

37 <!-- red box -->
38 <include>
39   <uri>model://red_box</uri>
40   <static>false</static>
41   <name>red_box</name>
42   <pose>0.1 0.167 1 0 0 0</pose>
43 </include>

```

Figure 19: red\_box added to the world file

## 7.4 ROS Configuration

A workspace named catkin\_make was created and necessary packages were created in it using “catkin\_create\_pkg”. The necessary python code was written in the src folder. The required dependencies were edited in the CMakeLists.txt file and package.xml files. The package which contains all the files for moveit and gazebo were created in simulation\_ws and the nodes were created in the

package arq\_gazebo further under the src folder

## 7.5 Move-it Configuration

Move-it was installed using the following command from pre-built libraries:

```
sudo apt install ros-kinetic-moveit
```

Figure 20: moveit installation command

By default it had the panda robot configured but the Maria robot was configured successfully using the setup files. The figure 21 illustrates the successful setup of Maria robot with the links set up on the left hand side in the image.

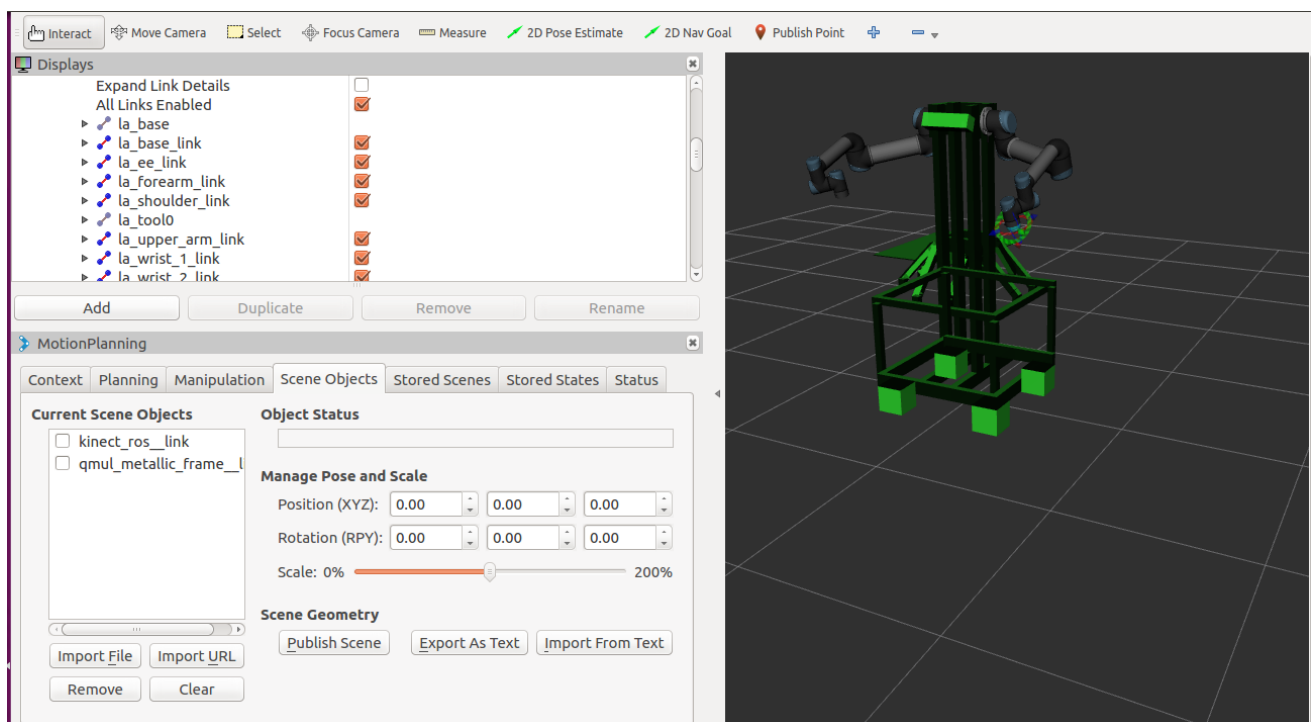


Figure 21: Moveit - Maria Robot

## 7.6 Move it and Gazebo Simulation

The Gazebo and Move-it were configured with each other so that the robot testing can be performed on gazebo along with moveit with real-time objects and scenarios successfully.

The controllers.yaml file consists information about the joints of the maria robot which is located in the move-it directory illustrated in figure 22

```
1 controller_list:
2   - name: fake_arm_controller_right
3     joints:
4       - ra_shoulder_pan_joint
5       - ra_shoulder_lift_joint
6       - ra_elbow_joint
7       - ra_wrist_1_joint
8       - ra_wrist_2_joint
9       - ra_wrist_3_joint
10  - name: fake_arm_controller_left
11    joints:
12      - la_shoulder_pan_joint
13      - la_shoulder_lift_joint
14      - la_elbow_joint
15      - la_wrist_1_joint
16      - la_wrist_2_joint
17      - la_wrist_3_joint
```

Figure 22: Controller definition file

The launch file included all the necessary commands and configurations to start the moveit and gazebo which compiles all the files in the package. This enables the Rviz simulation for moveit to visualize the robot and perform the manipulations.

Figure 23 is the screenshot of the Maria Robot configured in Moveit and Gazebo.

On the left is the moveit configuration of Maria robot and on the right is the gazebo simulation of the same. The gazebo simulation will also send the joint values of the angles for a more controlled manipulation

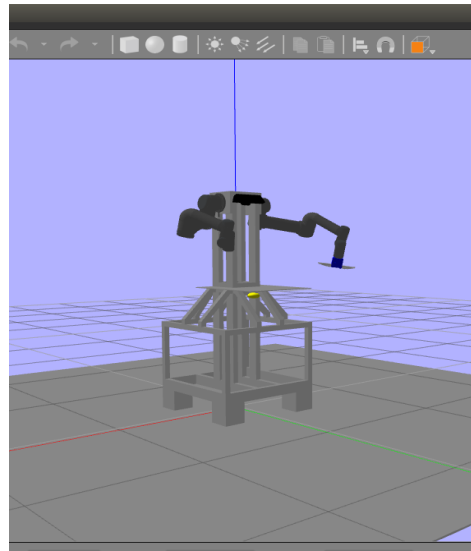
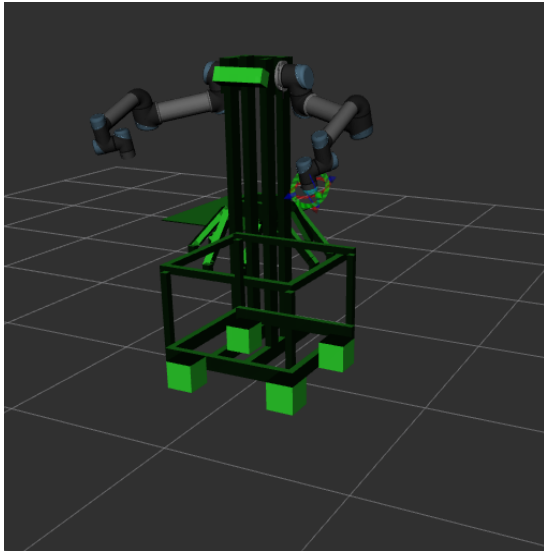


Figure 23: Maria configuration in moveit and gazebo simulated successfully

In order to simulate the objects created in gazebo using the sdf file, gazebo2rviz package is used which automatically simulates the models created in gazebo into rviz via the TF and Marker plugin.

It consists of python nodes which retrieves the models from gazebo/model\_states, post which the sdf files are parsed and the visualization\_msgs/Marker messages are sent for each link. There is one node which is responsible for loading the particular SDF model and publishing it as rviz Marker. [15]

Steps to use the package:

1. Launch gazebo and rviz successfully
2. In rviz, add the TF and Marker plugin as shown in figure 24. After loading the marker and the TF, the resultant robot should be displayed along with the Transform trees and the markers as depicted in figure 25
3. Execute the roslaunch command:

roslaunch gazebo2rviz gazebo2rviz.launch

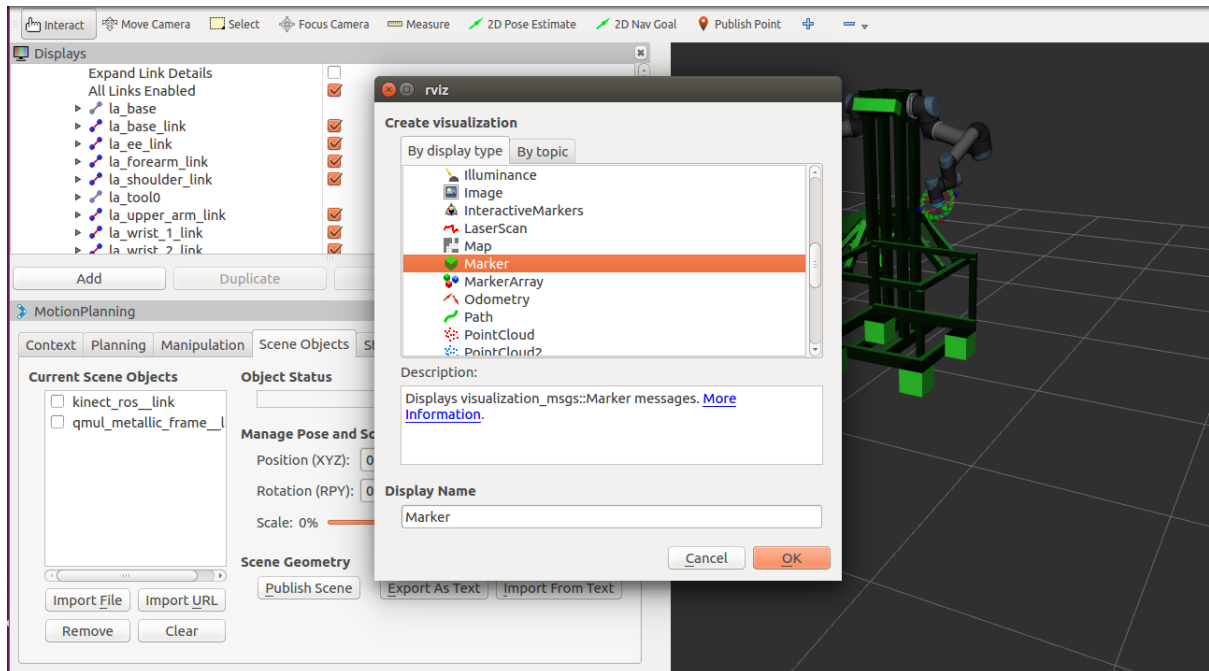


Figure 24: Adding the marker and TF in rviz

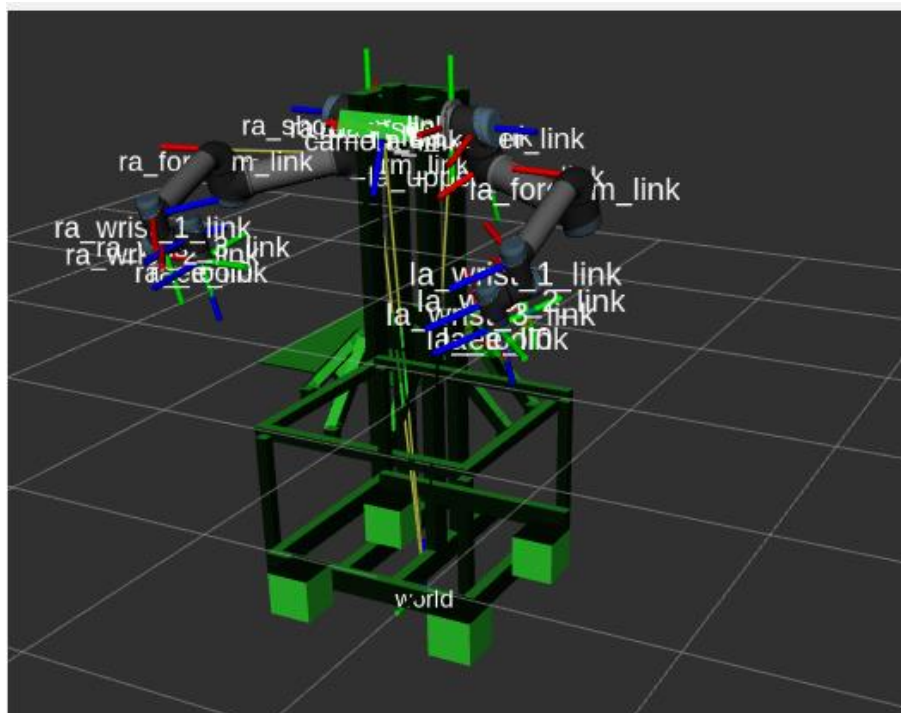


Figure 25: After adding the Tf and Marker plugins

Once the roslaunch is executed, the models loaded successfully message should be populated on the terminal as illustrated in figure 26. This confirms the execution of the script.

The objects created in gazebo will now be visible in rviz and any changes in the model in gazebo will be replicated in rviz as well. Figure 27 illustrates the same

```
[INFO] [1575929309.270148, 3781.537000]: Spinning
[INFO] [1575929309.837719, 3781.671000]: Loaded model: ground
[INFO] [1575929309.886173, 3781.671000]: Loaded model: qmul_metallic_frame
[INFO] [1575929309.892076, 3781.683000]: Loaded model: blue_box
[INFO] [1575929309.900271, 3781.684000]: Loaded model: red_box
[INFO] [1575929309.906628, 3781.684000]: Loaded model: yellow_box
[INFO] [1575929309.915818, 3781.684000]: Loaded model: kinect_ros
```

Figure 26: Models loaded successfully in rviz

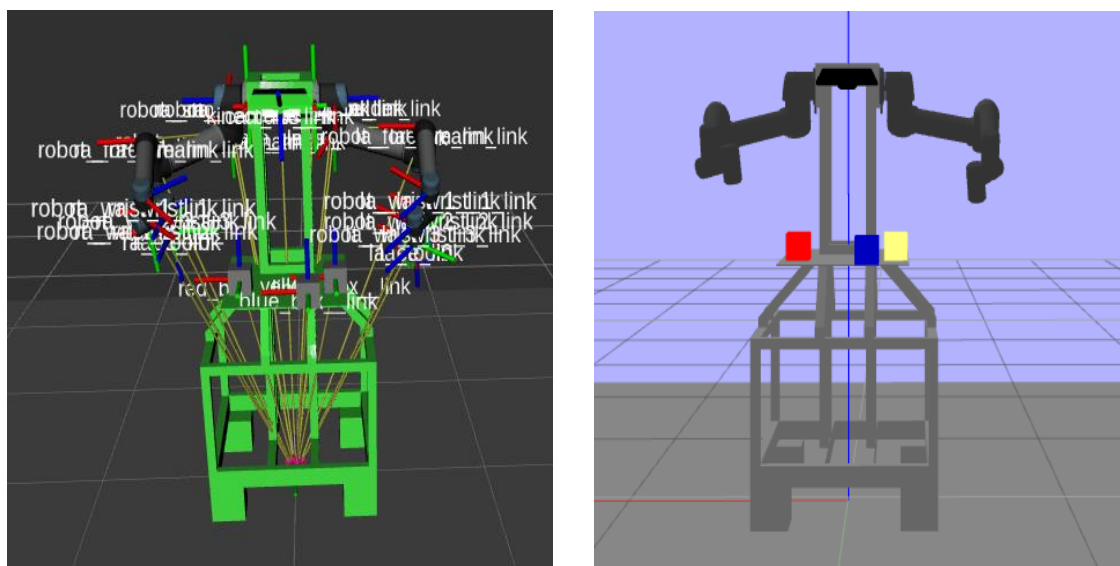


Figure 27: Models from gazebo successfully simulated in rviz



## 7.7 Python Script Execution

Motion planning and execution are to be performed via a python script which contains the configurations in order to plan in moveit and ROS. The nodes created for the motion of the robot should be supported by the essential libraries and packages in order to visualize the robot motion in gazebo and moveit.

### 7.7.1 Importing the robot configurations

In order to get the robot configurations, necessary import files should be specified which enables the movement of the robot via a python script. Figure 28 shows the files that need to be imported, which consists of Rospy which is a library in order to communicate with the rostopics, services, messages and functions. Moveit-commander creates a object for the robot which acts as an interface. Moveit planning scene interface helps in interacting with the world surrounding. Move group commander is an interface to one of the joints in the model. Here, it is left\_arm as we are using the left arm of the robot. This is used for planning and executing the motion on the Maria robot.

```
1#!/usr/bin/env python
2import sys
3import rospy
4import copy
5from copy import deepcopy
6import array as arr
7import numpy
8from std_msgs.msg import String
9from arq_gazebo.msg import *
10import moveit_commander
11import moveit_msgs.msg
12import geometry_msgs.msg
13
14moveit_commander.roscpp_initialize(sys.argv)
15robot = moveit_commander.RobotCommander()
16scene = moveit_commander.PlanningSceneInterface()
17grp_right_arm="right_arm"
18grp_left_arm="left_arm"
19grp_l=moveit_commander.MoveGroupCommander(grp_left_arm)
```

Figure 28: Imported libraries and configurations in python script

### 7.7.2 Obtaining the coordinates of the blocks

The coordinates are obtained via a service in the gazebo called `/gazebo/get_model_state`. The service is called and the coordinates are obtained and published on the respective rostopics created for 3 different blocks. Figure 29 is the snippet for the same.

```
model_coordinates = rospy.ServiceProxy('/gazebo/get_model_state', GetModelState)
box_coordinates = model_coordinates(box_name, 'link')
d.x = box_coordinates.pose.position.x
d.y = box_coordinates.pose.position.y
d.z = box_coordinates.pose.position.z
return d
```

Figure 29: Receiving the block coordinates using the service  
`/gazebo/get_model_state`

### 7.7.3 Moving the arm according to the actions

The actions executed based on the decisions made in the decision node were included in the `action.py` file and the motion was planned accordingly using the box and arm coordinates.

The gazebo world is visualized as a 3D space with x, y, z coordinates respectively as depicted in the figure 30 taking our robot as an example

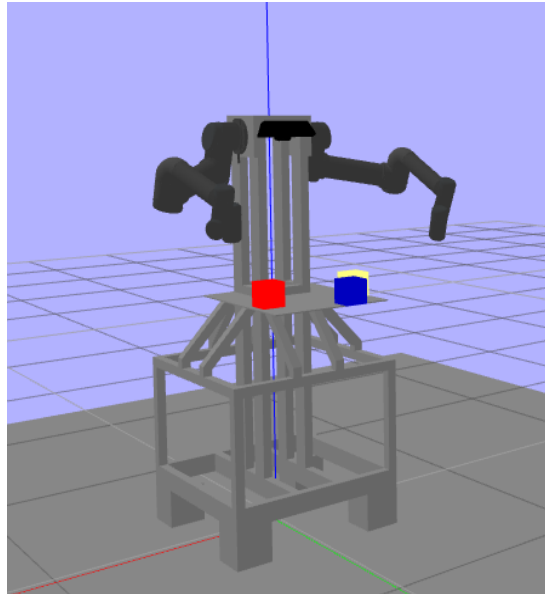


Figure 30: Gazebo world as a 3D model. Blue represents Z axis, Green represents Y axis and Red represents X axis

The implementation of the action in the script is illustrated in the figure 31

The coordinates were supplied to plan a Cartesian path by directly specifying the waypoints for the arm . They were then interpolated with a resolution of 1cm, hence, the eef\_step is coded as 0.01 in the compute\_cartesian\_path. The “execute” is responsible for executing the path which is planned in “plan”.

```

66 def TAP_LEFT(pos, init_pose, grp_joint, factor):
67     waypoints= []
68
69     init_pose.orientation.w = 1.0
70
71     init_pose.position.z = 0.05
72     waypoints.append(copy.deepcopy(init_pose))
73
74     init_pose.position.x += factor #move towards left
75     waypoints.append(copy.deepcopy(init_pose))
76
77     (plan, fraction) = grp_joint.compute_cartesian_path(
78         waypoints, # waypoints to follow
79         0.01, # eef_step
80         0.0) # jump_threshold
81
82     grp_joint.execute(plan, wait=True)
83     return pos, init_pose

```

Figure 31: Implementation of waypoints as a cartesian path

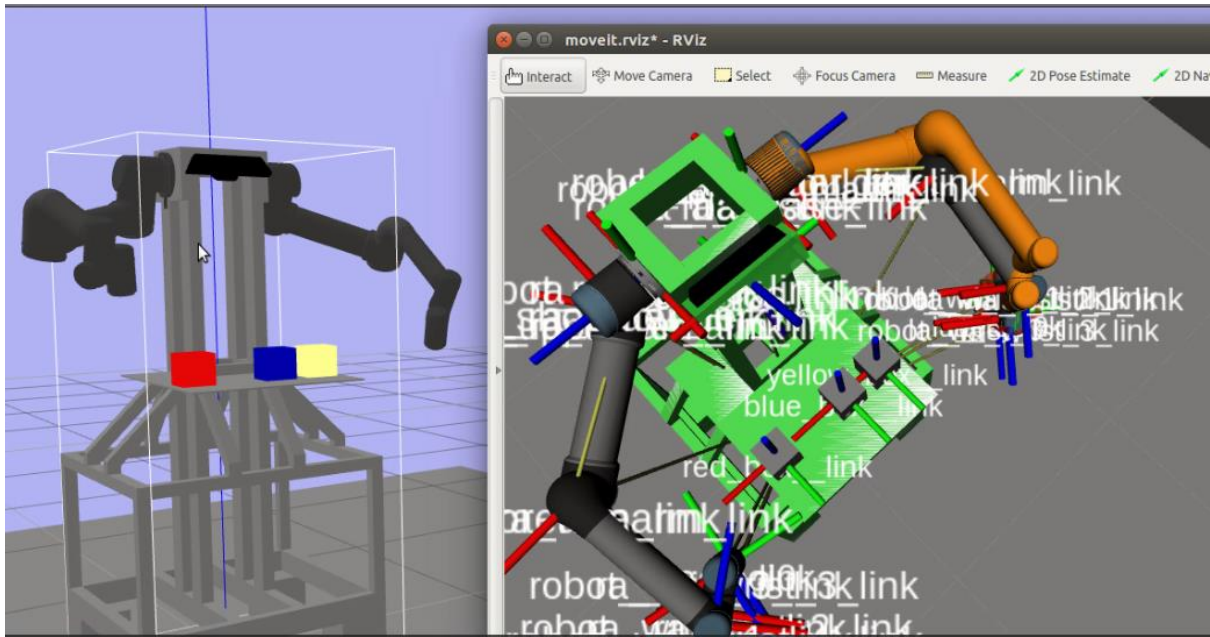
## 8. Experiments

Experiment was performed on the blockworld domain with different configurations of the box. The initial configuration of the blocks were successfully recorded, published by the perception node and subscribed on the rostopic. The coordinates subscribed are used to take the appropriate decision for performing the action. In figure, as the blocks are in a horizontal line, the robot identifies and takes decision to tap\_left. As tap\_left action is performed all the blocks are attached together.

1. States of the 3 blocks mentioned as the pose are considered as -
  - a. Red\_block- 0.1 0.167 1 0 0 0
  - b. Yellow\_block- -0.11 0.167 1 0 0 0
  - c. Blue\_block- -0.05 0.167 1 0 0 0

The pose of the blocks consists of 6 parameters but we are interested only the first 3 which are x, y and z coordinates respectively

The states of the blocks are used to decide the action to be executed according to the rules. In this case, the tap left action is executed by the robot. Figure 32 illustrates the initial state



The final state is represented in the figure 33 where all the three boxes are attached

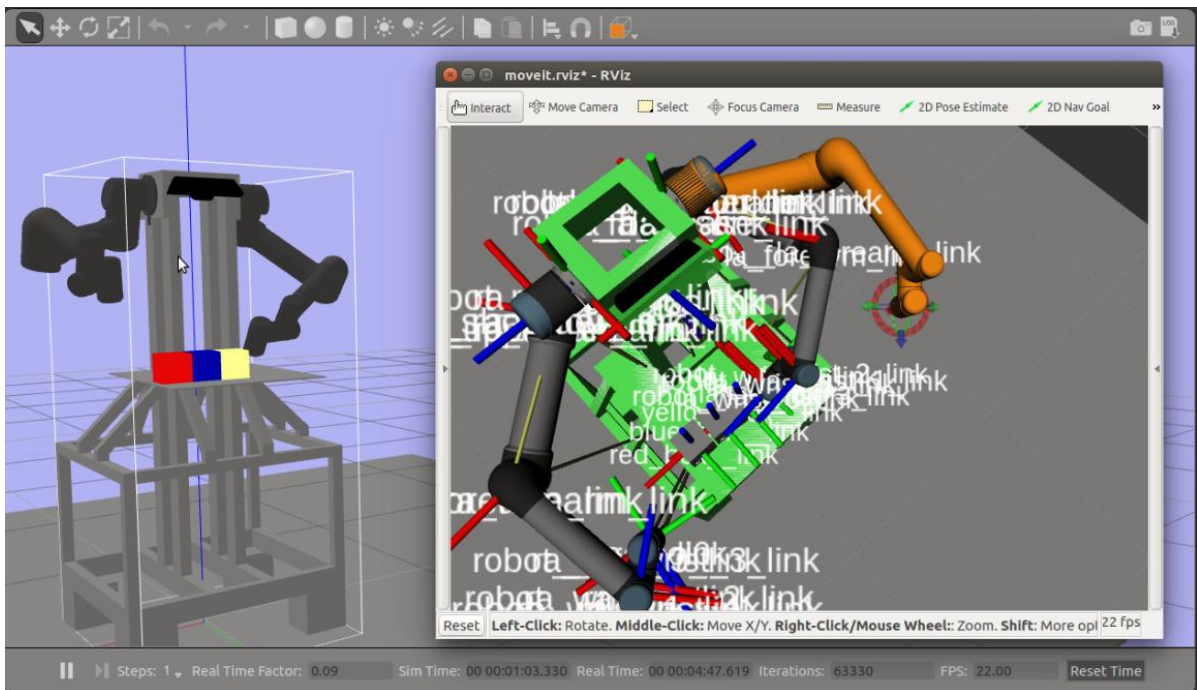


Figure 33: The goal state where all the boxes are attached to each other

2. The second scenario on which testing was performed can be viewed as if two boxes are already attached and the third one is yet to be attached. Figure 34 illustrates this initial state. In this case the robot taps in the correct direction and attaches the third box. The goal state configuration can be viewed in figure 35

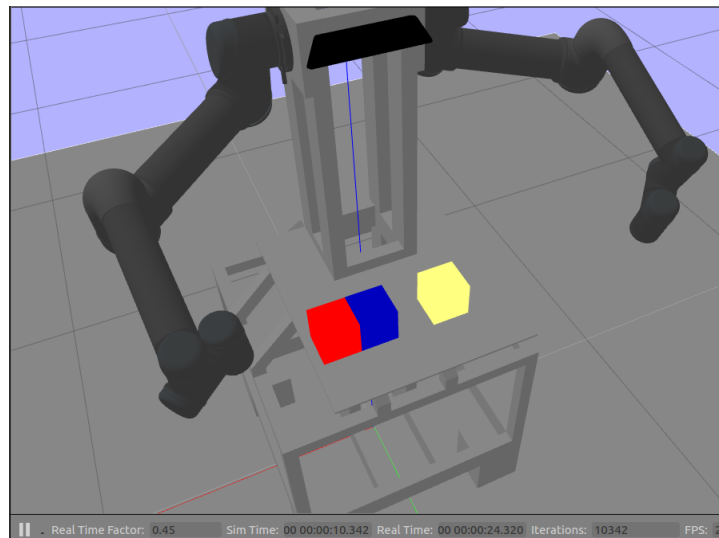


Figure 34: Initial state of scenario 2

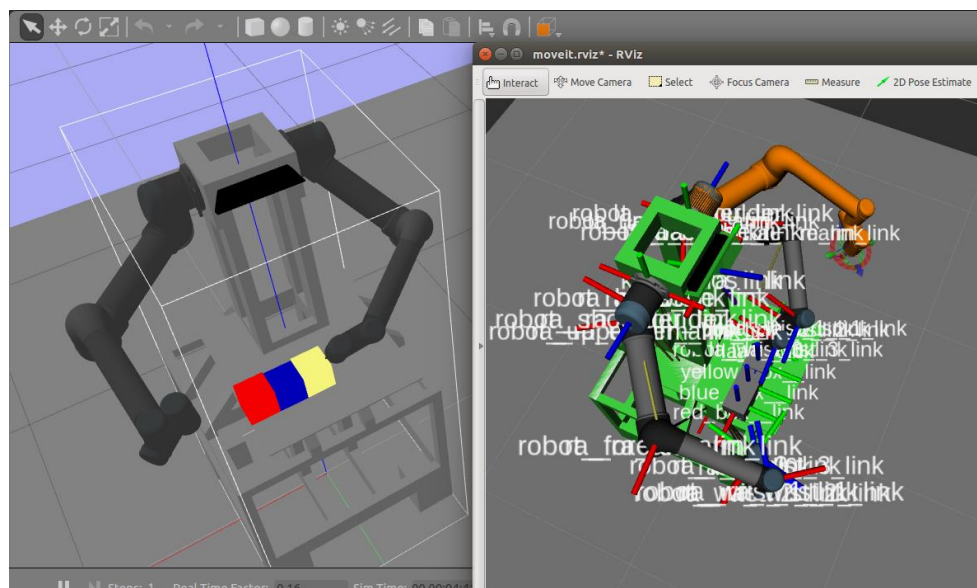


Figure 35: Goal state of scenario 2

3. If we consider the goal state from scenario 3 as the initial state for scenario 3, the arm does not move as the blocks are attached and stops. It displays a message as to already attached. Figure 36 illustrates the same

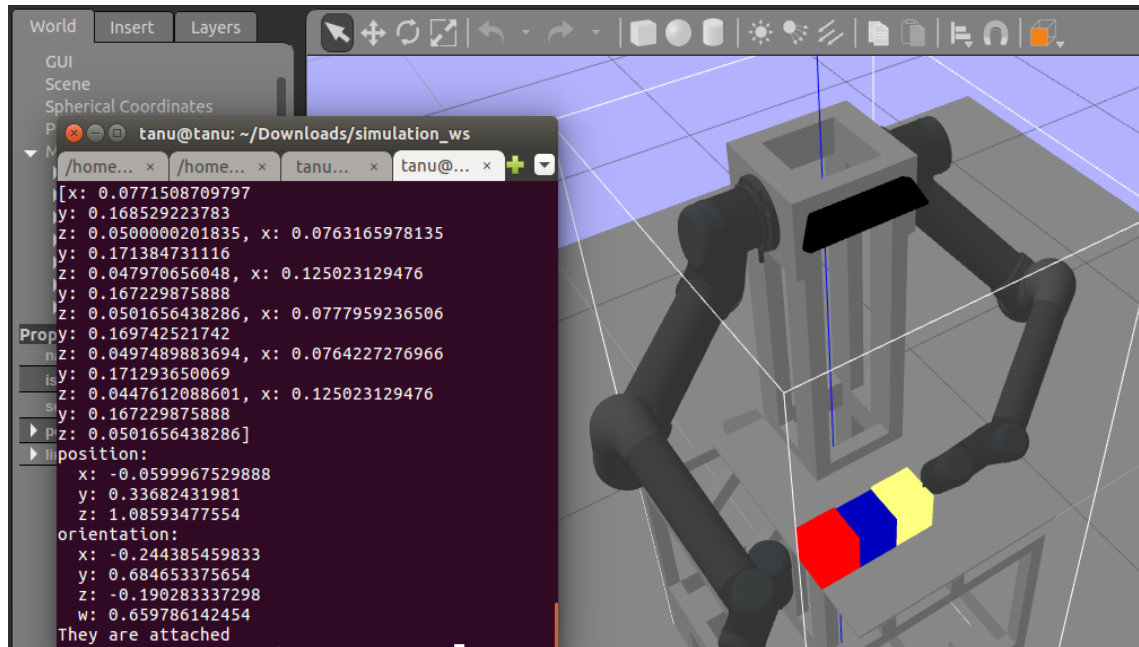


Figure 36: Scenario 3- Already attached

## 9. Discussions

The project aimed at providing human demonstrations and robot executing it. This was planned with gazebo and the Maria robot. The problems faced included that the perception node was executing and publishing the coordinates at a definite rate and sometimes the decision node was in an idle state (waiting) for the coordinates to be subscribed and further take a decision based on the block states. Hence, there was a delay in the execution of the trajectory. The robot was able to follow the rules which were written as a part of the decision node, and were tested on various scenarios except that the robot was not able to execute the backward movement. Whenever a robot had to tap backward the goal state was not reached.

The project also aimed to implement the human demonstrations as relational MDP with the help of which the robot can learn the tasks and execute on its own which was partially accomplished. The task of relational representation was completed wherein all the rules along with the predicates and actions were implemented. A key challenge was to implement this in a limited duration of time in which the robot was successfully able to execute the rules specified.



## 10. Conclusion and Future work

The robot was able to execute the rules and solve the blocksworld successfully with some limitations. The robot was able to detect the object and its position and make decision whether to tap or no (in case of they are already attached) and if yes, depending on the direction of the blocks it was successfully able to tap on the appropriate block. Hence, the goal state of attaching all the blocks was successfully achieved. However, as a part of future work I would implement the human demonstrations as relational MDP and educate the robot to make decisions with the demonstrations as this was not achieved due to limited time scale.

As a part of future work this technique could be implemented using a gripper which can be extended to more wide domains such as manufacturing and hospitality.

## 11. REFERENCES

- [1]. A. Billard, S. Calinon, R. Dillman and S. Schaal, "Robot Programming by Demonstration," in Springer Handbook of Robotics, Springer, Berlin, Heidelberg, 2008, pp. 1371-1394.
- [2]. A. Ghalamzan and M. Ragaglia, "Robot learning from demonstrations: Emulation learning in environments with moving obstacles," Robotics and Autonomous Systems, vol. 101, pp. 45-56, 2018.
- [3]. A. Dattalo, "ROS Introduction," ROS, 8 August 2018. [Online]. Available: <https://www.ros.org/about-ros/>. [Accessed 4 December 2018].
- [4]. "Documentation", ROS, [Online]: Available

<http://wiki.ros.org/>

[5]. V. Beal, “operating system - OS,” webopedia, [Online]. Available: <https://www.techopedia.com/definition/3515/operating-system-os> [Accessed 12 03 2019].

[6]. “Moving robots into the future,” Moveit!, [Online]. Available: <https://moveit.ros.org/>. [Accessed 12 03 2019].

[7]. “Overview,” Gazebo, [Online]. Available: [http://gazebo.org/tutorials?cat=guided\\_b&tut=guided\\_b1](http://gazebo.org/tutorials?cat=guided_b&tut=guided_b1). [Accessed 14 April 2019].

[8]. Abhijit Gosavi, “Reinforcement Learning: A Tutorial Survey and Recent Advances”

[9]. Saso Džeroski, Luc De Raedt, and Kurt Driessens, “Relational reinforcement learning” Machine learning, 43(1-2):7–52, 2001.

[10]. Thibaut Munzer, Bilal Piot, Matthieu Geist, Olivier Pietquin, Manuel Lopes “Inverse Reinforcement Learning in Relational Domains”, In IJCAI, 2015

[11]. Abhijit Gosavi , “Reinforcement Learning: A Tutorial Survey and Recent Advances”, in INFORMS Journal

[12]. Thibaut Munzer, Marc Toussaint, Manuel Lopes “Efficient behavior learning in human–robot collaboration”, Springer Science+Business Media, LLC 2017

[13]. Sriraam Natarajan, Saket Joshi, Prasad Tadepalli, Kristian Kersting and Jude Shavlik “Imitation Learning in Relational Domains: A Functional-Gradient Boosting Approach”, in IJCAI 2011

[14] "Overview," Gazebo, [Online]. Available:

[http://gazebosim.org/tutorials?tut=build\\_model](http://gazebosim.org/tutorials?tut=build_model)

[15] "gazebo2rviz", ROS, [Online]. Available:

<http://wiki.ros.org/gazebo2rviz>

[16]. B. D. Argall, S. Chernova, M. Veloso and B. Browning, "A survey of robot learning by demonstration," *Robotics and Autonomous Systems*, vol. 57, pp. 469-483, 2009.

[17]. "URDF"- ROS, [Online], Available

<http://wiki.ros.org/urdf>