ENPM663 Building A Manufacturing Robotic System

Project Report

Group 4:

1. **Kiran S. Patil** **UID: 119398364**
2. **Badrinarayanan** **UID: 119215418**
3. **Vyshnav Achuthan** **UID: 119304815**
4. **Aniruddh Balram** **UID: 119206416**

Acknowledgement

We express our heartfelt gratitude to Dr. Zeid Kootbally and Dr. Craig Schlenoff for providing us with the opportunity to work on this magnificent project. We also thank them for their support, guidance and the knowledge imparted through lectures which enabled us to get through this project. We extend our deepest gratitude to Teaching Assistant Mr. Anirudh Komaralingam, who helped us through all the problems and the errors we faced while working on the project and his valuable suggestions to enhance the quality of the project.

1. INTRODUCTION

Agile Robotics for Industrial Automation Competition (ARIAC) is an annual competition conducted by the National Institute of Standards and Technology, wherein participants are tasked with creating robotic systems capable of performing tasks within a simulated factory environment. Each task uses at least one of the following 4 parts, **Sensors, Regulators, Pumps, and Battery.**

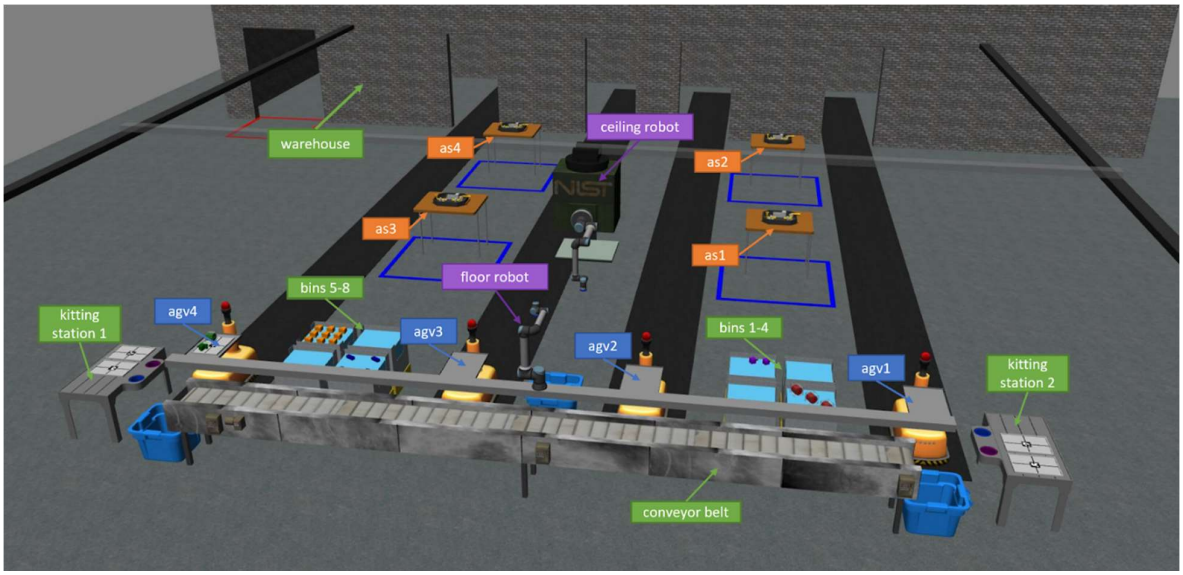

Fig 1. Factory environment

Figure 1 is a visual representation of the factory environment. As can be seen, it consists of a conveyor belt, 2 kitting stations, 4 AGVs, 8 bins, a floor robot, a ceiling robot, 4 assembly stations (as), and finally, the warehouse. Parts necessary to perform different tasks spawn on the conveyor belt, at a constant rate. These parts can be collected and stored in any of the 8 bins that are present near the conveyor belt. The floor robot can be used to collect parts from the belt. Each kitting station consists of trays on which parts necessary for specific tasks can be placed. For the program to identify the trays, there are ARUCO markers on each one of them. The kitting stations also comprise a gripper changing station, where the floor robot can change grippers, either to pick parts or to pick trays. The AGVs are used to transport parts either to the assembly stations or to the warehouse. The ceiling robots are primarily used to perform tasks at assembly stations.

After all the tasks have been completed, the competition comes to a close as the AGVs make their way to the warehouse.

1.1 TASKS:

There are three different tasks that are performed in the competition.

Kitting: Kitting is the process of collecting different parts that are necessary to complete other tasks. If the necessary parts to complete the kitting task are not found in the bins, these parts can be collected from the conveyor. The floor robot is often used to complete this task.

Assembly: Assembly is the process of assembling the different parts to create a respirator. This task takes place at the assembly stations. The Ceiling robot is used to complete this task.

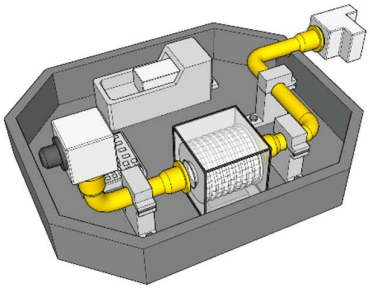

Fig 2. Respirator

Figure 2 is a visual representation of the parts assembled into a respirator. The robot needs to place the parts in a specific manner.

Combined: Combined tasks are those where the robots are asked to perform both kitting and assembly tasks. Hence, the necessary parts need to be collected and transported to an assembly station, and then the parts need to be assembled at the respective station.

1.2 SENSORS:

There are a total of 8 sensors that are at disposal, that can be used to collect information about the environment. Each sensor comes with a cost. Ideally, the competition must be completed with minimum cost.

SENSOR	DESCRIPTION	COST
Break beam	The break beam sensor detects any interruption of its beam caused by any object. It does not provide any information about the distance between the sensor and the object.	100
Proximity sensor	The proximity sensor outputs how far an object is from the sensor.	100
Laser profiler	The laser profiler provides an array of distances to a sensed object.	200
LIDAR	The LIDAR sensor provides a point cloud of detected objects.	300

RGB camera	The RGB camera provides an RGB image.	300
RGBD camera	The RGBD camera provides an RGB image and a depth image.	500
Basic Logical camera	The basic logical camera provides a list of kit tray poses and a list of part poses. The type and color of an object are not reported by this sensor.	500
Advanced logical camera	The advanced logical camera reports the pose, the type, and the color of a detected object.	2000

1.3 CHALLENGES:

In ARIAC, there are multiple challenges that hinder the completion of tasks mentioned previously. These challenges are described as follows:

High Priority Order: In this challenge, as the robots are performing the tasks that were initially received, a new order is announced with a high priority. To earn maximum points, this high-priority order must be started as soon as it has been announced and be completed before other tasks. The challenge here is to halt processing the current order, complete the high-priority order, and then continue with the regular order.

Insufficient Parts: The orders announced are such that, they require a specific type of part with a specific color to complete them. When these two conditions are not met, that implies that there are not enough parts to complete the task at hand. In this situation, the robot must look for another part in the bins, that closely resembles (same type, but different color) the essential part.

Faulty Parts: The parts that are picked from the conveyor or the ones in the bins, might be faulty. That implies that the part cannot be used to complete the order. Hence, it must be discarded accordingly. Additionally, the robot must look for an alternative to complete the order.

2. ARCHITECTURE

There are multiple components that make up the environment, as discussed above. Different components must be controlled simultaneously at different instances of time. The Competitor Control System (CCS) controls these components of the environment. Figure 3 is a visual representation of how the CCS supervises these components. It uses multiple topics and services to control each action each part performs in the environment.

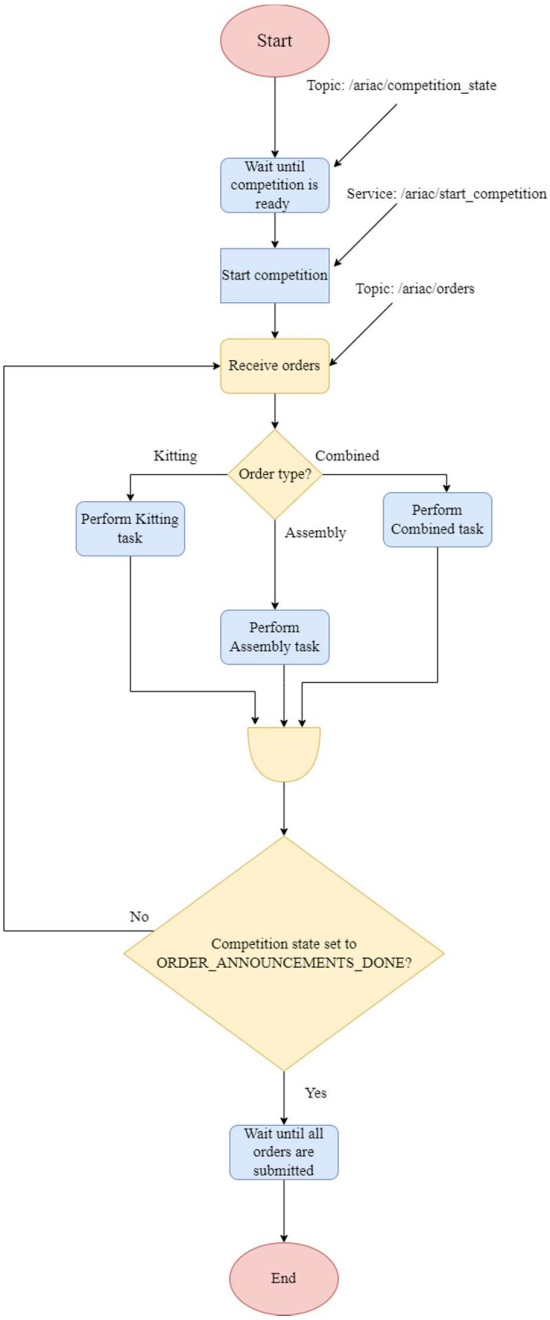

Fig 3. CCS Pipeline

Orders are received only after the competition state is set to "*READY*". The ***/ariac/competition_state*** is used to get the competition state. Once the orders are received, they

must be read through, to know what kind of task needs to be performed. Multiple orders can be received. The appropriate task is completed, and the competition ends, only when the competition state is "*ORDER_ACCOUNCEMENTS_DONE*", after all the orders are submitted.

2.1 KITTING TASK:

The pipeline followed to perform Kitting task has been shown below. The task starts by initially identifying that the given order is a kitting task. This is done by reading the order using the ***/ariac/Orders*** topic. From the order, the necessary parts are identified. These parts can either be found in bins or conveyor.

The ***/ariac/bin_parts*** and the ***/ariac/conveyor_parts*** topics are used to identify what parts are available in the bins and conveyor respectively. If parts are not available either on the conveyor or the bins, it becomes an Insufficient Part Challenge, which has been discussed in later parts of the report. Simultaneously, the necessary trays are also identified using the ARUCO markers present on the trays. After the trays are identified, the robot must change its gripper to pick the tray and place it on the AGV. Before that, the AGV must be near Kitting station.

The ***/ariac/agv(n)_status*** topic is used to check if the nth AGV is near the appropriate Kitting station. To change the gripper, the ROS2 service ***/ariac/(robot)_change_gripper*** has been used. ((robot) could be floor or ceiling). The appropriate tray is picked up and placed on the appropriate AGV after the gripper has been changed. The trays need to be locked to the AGVs. The ***/ariac/agv(n)_lock_tray*** service has been used for this purpose.

To pick parts from the bins, the robot must change the gripper again. This is achieved using the same service. The parts are identified using Advanced Logical Cameras and the floor robot is used to pick the parts from the bins and place them on the kit tray on the AGV. If some parts from the conveyor are needed to complete the kitting task, the first step is to identify the empty slots in each bin. These can be identified using the ***/ariac/bin_parts*** topic.

Once the empty slots are identified, the floor robot has been used to pick up the essential parts and placed on those empty slots. Later, these parts are picked up and placed on kit trays. Finally, to complete the order, the AGV, which has the kit tray and the essential parts on it, must be sent to the warehouse. The ***/ariac/move_agv(n)*** service has been used to achieve that.

The order is then submitted using the ***/ariac/submit_order*** service.

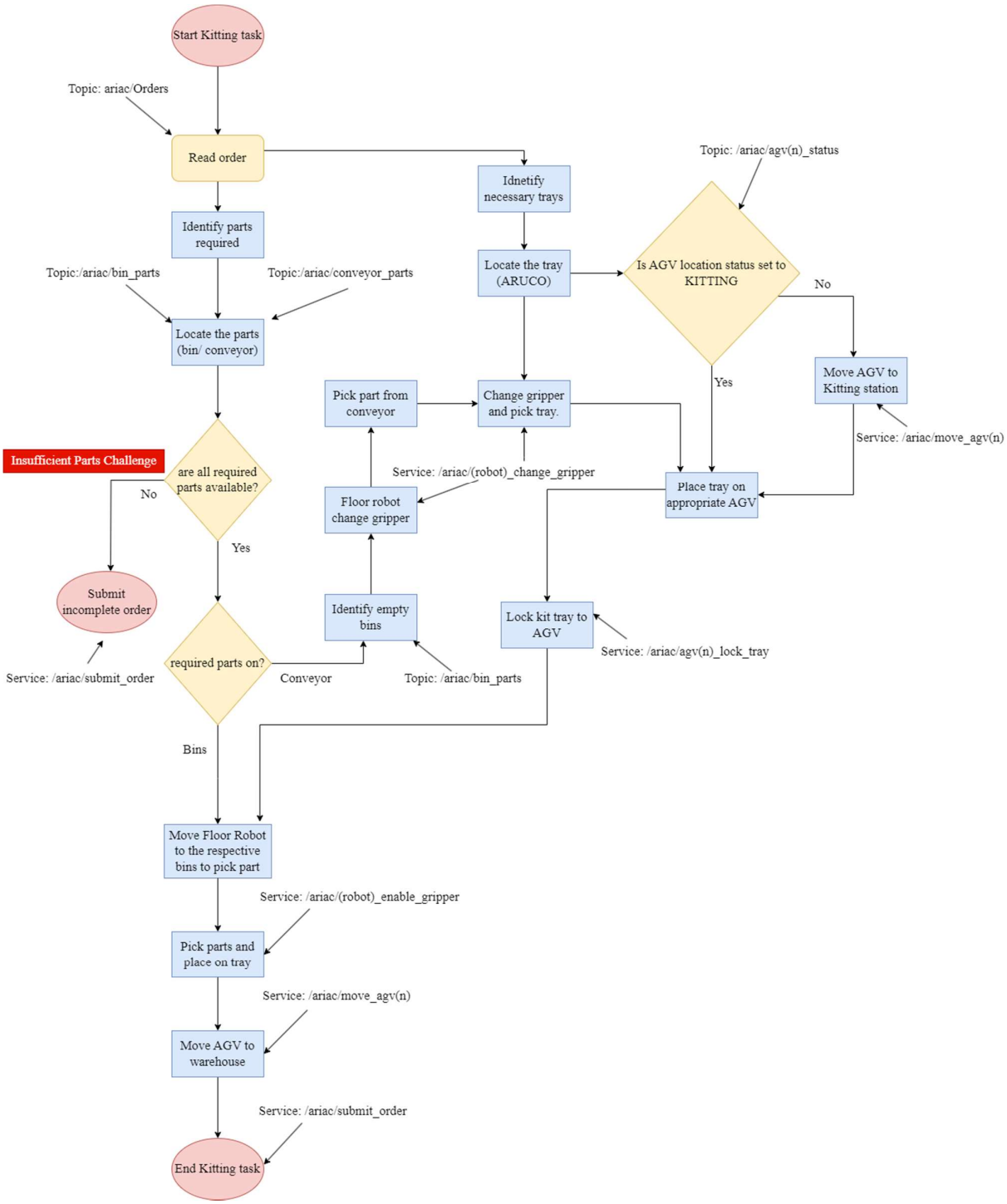

Fig 4. Kitting Task Pipeline

2.2 ASSEMBLY TASK:

The pipeline followed to perform Assembly task has been shown below. The task starts by initially identifying that the given order is a kitting task. This is done by reading the order using the ***/ariac/Orders*** topic. Simultaneously, the Assembly station number must be identified from the order. After getting to know the Assembly station number, the appropriate AGV which holds the essential parts, must be identified. The kit tray must be locked to the AGV. The ***/ariac/agv(n)_lock_tray*** service has been used for this purpose. Finally, the AGV is moved to the right assembly station using the ***/ariac/move_agv(n)***.

To assemble the parts, the ceiling robot must be brought close to the appropriate assembly station. The ***/ariac/get_pre_assembly_poses*** service, is used for that purpose. Finally, the task is completed after the parts are assembled and the order is submitted using the ***/ariac/submit_order.***

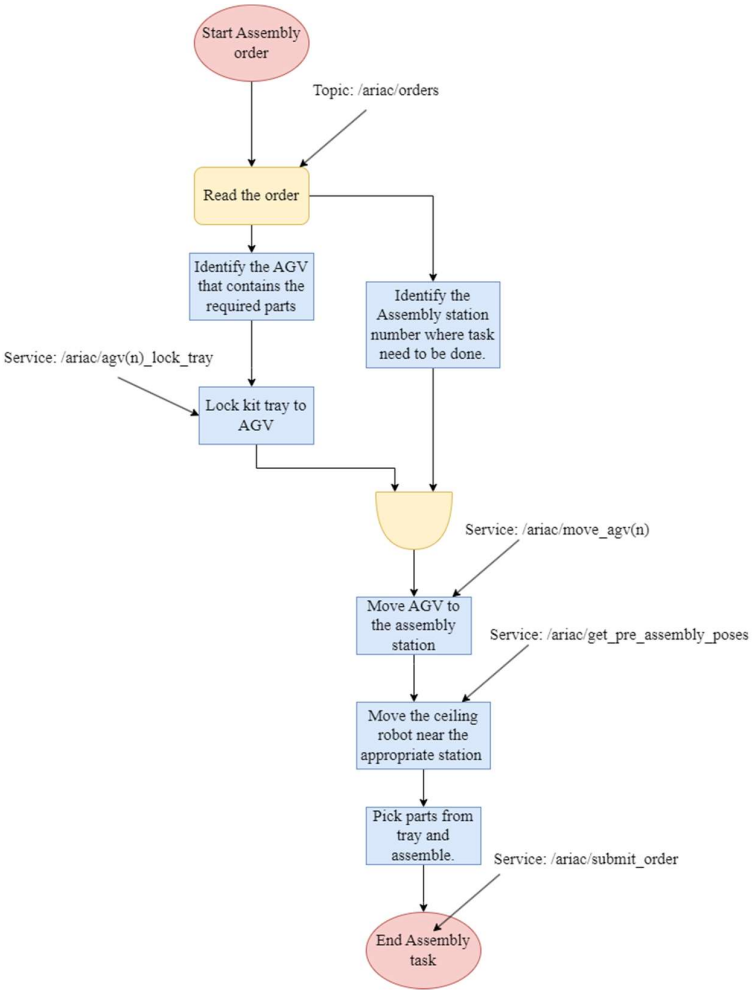

Fig 5. Assembly Task Pipeline

2.3 COMBINED TASK:

The pipeline followed to perform Combined task has been shown below. The task starts by initially identifying that the given order is a combined task. This is done by reading the order using the ***/ariac/Orders*** topic. From the order, the necessary parts are identified. These parts can either be found in bins or conveyor.

The ***/ariac/bin_parts*** and the ***/ariac/conveyor_parts*** topics are used to identify what parts are available in the bins and conveyor respectively. If essential parts are not available either on the conveyor or the bins, it becomes an Insufficient Part Challenge, which has been discussed in later parts of the report. Simultaneously, the necessary trays are also identified using the ARUCO markers present on the trays. After the trays are identified, the robot must change its gripper to pick the tray and place it on the AGV. Before that, the AGV must be near Kitting station. The ***/ariac/agv(n)_status*** topic is used to check if the nth AGV is near the appropriate Kitting station.

To change the gripper, the ROS2 service ***/ariac/(robot)_change_gripper*** has been used. ((robot) could be floor or ceiling). The appropriate tray is picked up and placed on the appropriate AGV after the gripper has been changed. The trays need to be locked to the AGVs. The ***/ariac/agv(n)_lock_tray*** service has been used for this purpose. To pick parts from the bins, the robot must change the gripper again. This is achieved using the same service mentioned previously. The parts are identified using Advanced Logical Cameras and the floor robot has been used to pick the parts from the bins and placed them on the kit tray on the AGV. If some parts from the conveyor are needed to complete the task, the first step is to identify the empty slots in each bin. These can be identified using the ***/ariac/bin_parts*** topic.

Once the empty slots are identified, the floor robot has been used to pick up the essential parts and placed on those empty slots. Later, these parts are picked up and placed on kit trays. The AGV, which has the kit tray and the essential parts on it, must be sent to the appropriate station to perform assembly. The Assembly station number must be identified by reading the order. After getting to know the Assembly station number, the AGV is moved to the right assembly station using the ***/ariac/move_agv(n)***.

To assemble the parts, the ceiling robot must be brought close to the appropriate assembly station. The ***/ariac/get_pre_assembly_poses*** service, is used for that purpose.

Finally, the task is completed after the parts are assembled and the order is submitted using the ***/ariac/submit_order.***

Fig 6. Combined Task Pipeline

3. PACKAGE STRUCTURE

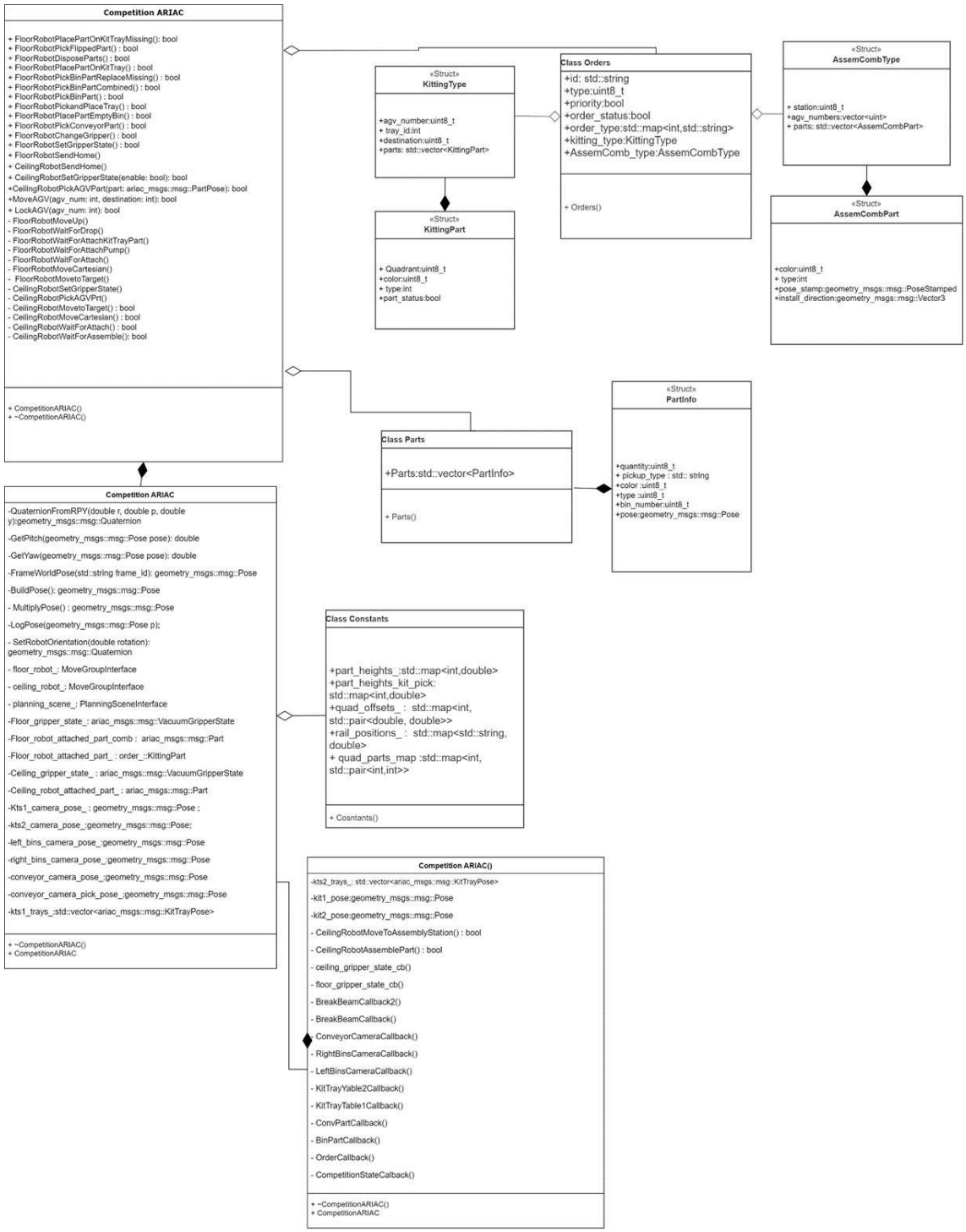

Fig. 7 Package Structure

4. ORDER STORAGE

The order storage task is important to store orders and pass them on to the CCS to process and perform tasks. As the number of orders can vary and can also be announced during the competition, a structural approach is required to store orders dynamically.

A combination of classes and structures are used to store the orders. Order members are stored in vectors, which are easily accessible. Class diagram for order storage is described in Fig. 3

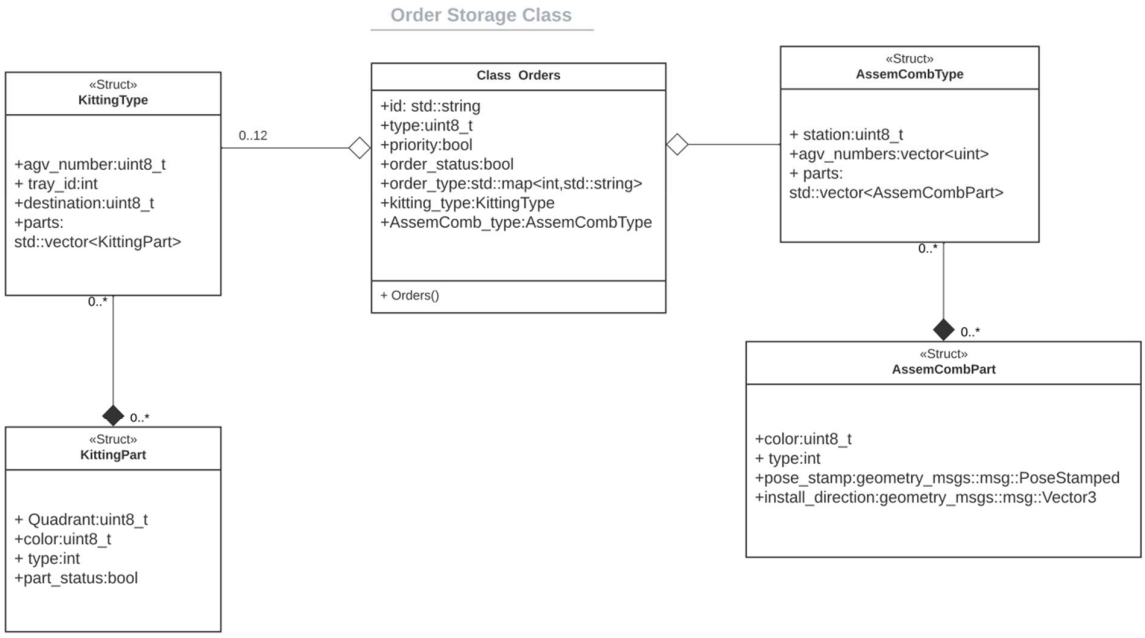

Fig 8. Order Storage flowchart diagram

The order class takes into consideration order variables such as order id, order type, priority and status, which all are part of the ***ariac_msg::msg::Orders***. When the competition starts, the orders are published to the message, and the class can be modeled uniformly to read the message and split the information into the class members mentioned.

The ***CompetitionARIAC*** class has a private member function named ***OrderCallback***, which is invoked whenever a new order message is received. The order data are taken from the received message and assigned to the Orders class elements within this method. The order class consists of the following members:

- **Order. id** - The order id is passed to the competition as a string. This consists of the respective order id. An astute example of this would be '*Order id:MMB30H58*'
- **Order. type** - The order type is an integer variable as per the orders message in the competition. The order. type variable represents the type of task for an order. It is an unsigned 8-bit integer (uint8) with three potential values:
    - *KITTING = 0* denotes that the order is a Kitting task. It denotes that the order entails gathering and assembling certain parts or components into a kit or container.

    - *ASSEMBLY = 1* denotes that the order is an Assembly task. It signifies that the order entails putting together several elements or components to form a bigger product or structure.
    - *COMBINED = 2* denotes that the order is a Combined task, as indicated by this value. It signifies that the order combines Kitting and Assembly tasks.
- **Order. priority** - This is a boolean flag, which checks whether the order passed to the competition is a priority order or not. A priority order must be given preference over the other orders when it comes to accomplishment. So it is quintessential to check for the priority of the order dynamically,
- **Order.order_type** - A simple map that takes in the ***order. type*** integer value and maps it to the string representing the ***order type***. This map converts integer order. type to string, which is used while submitting the order to let the user know about the order in a readable format.

Apart from these basic class members, two more members are used in the class, that act as an instance of the ***KittingType*** struct and ***AssemComdType*** structure as member variables. These structures are created to store detailed information about the order based on the order type. The usage of structures helps with code flexibility and reusability since the orders published are dynamic and it requires proper recall and storage of intrinsic information.

The structures ***KittingType*** and ***AssemCombType*** are composed of two more structures ***KittingPart*** and ***AssemCombPart***. The ***KittingType*** structure stores the order details when the order type is "*KITTING*". The ***KittingType*** structure resembles the ***ariac_msgs::msg::KittingTask***. The structure members consist of:

- **Agv_number** - The ***agv_number*** is of integer type, which provides the information of which ***agv_number*** is used for the kitting task, and where the kitting tray is to be placed.
- **Destination** - This describes the destination of the kitting task. The destination variable is of integer type and has the following locations:
    - *KITTING*: If the destination value is 0, this takes the AGV to the kitting station.
    - *ASSEMBLY_FRONT*: If the destination value is 1, this takes the AGV to the front assembly stations, either as as1 or as3 based on the AGV number.
    - *ASSEMBLY_BACK*: If the destination value is 2, this takes the AGV to the rear assembly stations, either as2 or as4 based on the AGV number.
    - *WAREHOUSE*: If the destination value is 3, this takes the AGV to the warehouse.
- **Parts** - A standard vector created to store the ***ariac_msgs::msg::Parts***, called inside the ***KittingTask*** message. To access individual characteristics of the parts passed by the order, the vector consists of the instance of the ***KittingPart*** structure, which consists of the unique characteristics of the parts such as color and type.

The ***KittingPart*** structure replicates the ***ariac_msgs/KittingPart [] parts*** message. The parts are given as a list of characteristics, and for dynamic allocation purposes, these members are stored inside as a vector inside the ***KittingType*** structure. The ***KittingPart*** struct consists of:

- **Quadrant** - The quadrant is of integer type, which describes the quadrant on the tray where the kitting part needs to be placed.
- **Color** - The color is of integer type, which details the part color that's used for the kitting task. A total of five colors is used in the competition, and the values are assigned to each color.
- **Type** - The type is also of integer type, which describes the type of part that's to be used for the kitting task. In the competition, a total of 4 parts are used namely.
    - 10 - "BATTERY"
    - 11 - "PUMP"
    - 12 - "SENSOR"
    - 13 - "REGULATOR"
- **Part_status** - This is a boolean flag member used in the CCS that describes the status of the parts. If the part status is 0, the parts are not picked and placed on the tray. After the part is placed on the kitting tray, the flag is updated to 1 to signify the completion of the placement of that part on the tray.

Like ***KittingType***, the ***AssemCombType*** structure is used to store the details of the order when the order type is "*ASSEMBLY*" or "*COMBINED*". Since the combined task is a culmination of both kitting and assembly tasks, the structures are designed to accommodate both the required members of the task. The ***AssemCombType*** structure resembles the functioning of messages ***ariac_msgs/msg/AssemblyTask*** and ***ariac_msgs/msg/CombinedTask***.The structure members consist of:

- **Station** - The assembly station is of integer type, which provides the information of which station AGV needs to take the parts, and where the parts are needed to be assembled.
- **Agv_numbers**- For assembly tasks, more than one AGV can be used as part of the competition. The number of AGVs used depends on the order, which is required to be stored as a list of AGVs. A vector is created for storing all the AGV numbers passed in order.
- **Parts** - A standard vector is created to store the ***ariac_msgs::msg::Parts***, which is called inside the ***AssemblyTask*** and the ***CombinedTask*** message. To access individual characteristics of the parts, a vector is created to store all the characteristics of the part. The vector consists of the instance of the ***AssemCombPart*** structure, which consists of the individual characteristics of the parts such as color, type, pose and assembly direction.

The ***AssemCombPart*** structure is created to replicate the ***ariac_msgs/AssemblyPart [] parts*** message. The combined task also uses the same message to obtain part details for assembly and hence it's not mandatory to create two structures for two different tasks. The ***AssemCombPart*** struct consists of:

- **Color** - The color is of integer type, which details the part color that's used for the assembly task. There are totally five colors in the competition and the values are assigned to each color. The competition consists of 5 colors in total namely:
    - *0 - "RED"*

- *1- "GREEN"*
- *2- "BLUE"*
- *3- "ORANGE"*
- *4- "PURPLE"*

- **Type** - The type is also of integer type, which describes the type of part that's to be used for the assembly task. In the competition, a total of 4 parts are used. For a combined task, the parts can be stored in the same variable as the combined task also consists of assembly.
- **Pose_stamp** - The pose stamp variable is of type ***geometry_msgs/PoseStamp*** , which takes in the pose of the part with respect to the coordinate frame along with the timestamp. The variable stores the position and quaternion orientation of the part in the assembly station.

- **Install_direction** - The install direction is of type ***geometry_msgs/msg/Vector3***, which takes in the vector of direction in which the part is required to be assembled in the assembly station. The variable consists of values in float64 type describing direction in x,y and z coordinates.

The individual structs are all associated with the main order class, and the part structures are composed of the Type structs. The composition property indicates a struct holds objects of other structures as members and the enclosing class controls the lifetime of the contained items. In this case, the ***KittingType*** structure and the ***AssemCombType*** structure holds the objects of ***KittingPart*** and ***AssemCombPart*** structures. By using composition, the Orders class can represent complex order structures with different types of tasks and associated data, providing a comprehensive and organized representation of an order and its components.

In a large ARIAC codebase, employing composition gives modularity, code reusability, simplicity of maintenance, increased readability, and flexibility. It helps in the organization of complicated order structures and accompanying data, making the codebase more understandable, maintainable, and extendable.

5. SENSOR DESCRIPTION

For the whole order processing system: 5 Advanced Logical Cameras and 2 break beams were used. Description of the sensors are given below:

- **Advanced Logical Camera (ALC):** It reports pose, type and the color of the object detected. To receive data from the ALC, a subscriber is written listening to the topic ***'/ariac/sensors/<camera_name>/image'*** which returns class "*AdvancedLogicalCameraImage*" which stores part type, part pose, part color and methods to transform part pose in world frame. In detailed description is as follows:
  1. **Right Bin Camera:**

a. **Camera Position:** Its coordinates and orientation with the world frame in gazebo are:
Coordinates: *(x, y, z) = (-2.286, 2.96, 1.8)* in meters,
Orientation: *(roll, pitch, yaw) = (pi, pi/2, 0)* in radians
b. **Usage:** Topic for subscription from this camera is ***'/ariac/sensors/right_bins_camera/image'***. It is positioned above the right bins and provides type, pose and color information of all the parts in both the right bins.

2. **Left Bin Camera:**
   a. **Camera Position:** Its coordinates and orientation with the world frame in gazebo are:
   Coordinates: *(x, y, z) = (-2.286, -2.96, 1.8)* in meters
   Orientation: *(roll, pitch, yaw) = (pi, pi/2, 0)* in radians
   b. **Usage:** Topic for subscription from this camera is ***'/ariac/sensors/left_bins_camera/image'***. It is positioned above the left bins and provides type, pose and color information of all the parts in both the left bins.
3. **Tray Table 1 Camera:**
   a. **Camera Position:** Its coordinates and orientation with the world frame in gazebo are:
   Coordinates: *(x, y, z) = (-1.3, -5.8, 1.8)* in meters
   Orientation: *(roll, pitch, yaw) = (pi, pi/2, pi/2)* in radians
   b. **Usage:** Topic for subscription from this camera is ***'/ariac/sensors/kts1_camera/image'***. It is positioned above the tray table on the left side and provides information about the position of the trays.
4. **Tray Table 2 Camera:**
   a. **Camera Position:** Its coordinates and orientation with the world frame in gazebo are:
   Coordinates: *(x, y, z) = (-1.3, 5.8, 1.8)* in meters
   Orientation: *(roll, pitch, yaw) = (pi, pi/2, -pi/2)* in radians
   b. **Usage:** Topic for subscription from this camera is ***'/ariac/sensors/kts2_camera/image'***. It is positioned above the tray table on the right side and provides information about the position of the trays.
5. **Conveyor Camera:**
   a. **Camera Position:** Its coordinates and orientation with the world frame in gazebo are:
   Coordinates: *(x, y, z) = (-0.65, 3.7, 1.2)* in meters
   Orientation: *(roll, pitch, yaw) = (pi, pi/2, 0)* in radians
   b. **Usage:** Topic for subscription from this camera is ***'/ariac/sensors/conveyor_camera/image'***. It is positioned above the conveyor belt and provides type, pose and color information of all the parts that spawn on the conveyor.

- **Break Beam (BB):** Break beam returns a value if the beam is broken by an object. To receive data from BB, a subscriber is written listening to the topic

'/ariac/sensors/<break_beam_name>/status' which returns a ***'/ariac/msgs/BreakBeamMessage'*** - a bool notifying if an object is detected. In detailed description is as follows:

1. **Break Beam at Start of Conveyor:**
    a. **Break Beam Position:** Its coordinates and orientation with the world frame in gazebo are:
    Coordinates*: (x, y, z) = (-0.403512, 3.871046, 0.880000)* in meters
    Orientation*: (roll, pitch, yaw) = (0, 0, pi)* in radians
    b. **Usage:** Topic of subscription for this break beam is ***'/ariac/sensors/breakbream_0/status***'. It is positioned at the start of the conveyor to detect the incoming parts. Based on the status change, the callback function picks the part and places it on the bin closest to it
2. **Break Beam at End of Conveyor:**
    a. **Break Beam Position:** Its coordinates and orientation with the world frame in gazebo are:
    Coordinates: *(x, y, z) = (-0.409806, -4.152406, 0.88)* in meters
    Orientation*: (roll, pitch, yaw) = (0, 0, pi)* in radians
    b. **Usage:** Topic of subscription for this break beam is ***'/ariac/sensors/breakbream_1/status'***. It is positioned at the end of the conveyor to detect the outgoing parts. Due to the slow speed of operation of the Robot, some parts may be missing. This break beam is used to keep the total count of the missed conveyor parts. The order execution begins only after all the parts from the conveyor have spawned. The callback function just updates the conveyor part count.

6. CONVEYOR PART PICKING TASK:

Two sensors are primarily used for completing this task; a break beam sensor to detect when a part arrives on the conveyor belt, and an advanced logical camera to obtain that part's type, color, and pose. The data the sensor reads is published to the ***/ariac/sensors/*** topic of that sensor, for example, data read by the conveyor camera is published to the topic ***/ariac/sensors/conveyor_camera/image***. Fig. 9 and 10 show an example message published on the topics by the break beam sensor and advanced logical camera when no part is present.

```
~
❯ ros2 topic echo /ariac/sensors/breakbeam_0/status
header:
  stamp:
    sec: 68
    nanosec: 117000000
  frame_id: breakbeam_0_frame
object_detected: false
---
```

Fig. 9

```
~
❯ ros2 topic echo /ariac/sensors/conveyor_camera/image
part_poses: []
tray_poses: []
sensor_pose:
  position:
    x: -0.65
    y: 3.7
    z: 1.2
  orientation:
    x: -0.7071045443232116
    y: -1.2248082623837582e-07
    z: 0.7071090180427863
    w: -1.2246817995071293e-07
---
```

Fig. 10

Note: The sensor names ***breakbeam_0*** and ***conveyor_camera*** are the names that are set in the sensors.yaml file.

The break beam sensor reports when a beam is broken by an object i.e. whenever a part crosses it, the object_detected field is set to true, thus the presence of a part is determined.

A sample message published is shown in Fig. 11, where it can be seen that the object_detected field switches from false to true as soon as a part crosses the beam. But, the break beam sensor does not provide distance information, so the offset/position of the part can't be identified.

```
---
header:
  stamp:
    sec: 202
    nanosec: 18000000
  frame_id: breakbeam_0_frame
object_detected: false
---
header:
  stamp:
    sec: 202
    nanosec: 52000000
  frame_id: breakbeam_0_frame
object_detected: true
---
```

Fig. 11

Whenever a part is spawned onto the conveyor belt, it's spawned at a particular offset that ranges from -1 to 1. Depending on the pose (part_pose.pose.position.z) of the part that is obtained from the conveyor camera, the offset is determined, which is further used to set the joint state values of the floor robot, resulting in it moving to a desired picking point.

An example of a conveyor camera sensor message is shown below. Fig. 12(a) and 12(b) show a part spawned on the conveyor belt at offset -1 and 1 respectively, whereas Fig. 13(a) and 13(b) is the message published on topic ***/ariac/sensors/conveyor_camera/image*** for both the offsets respectively.

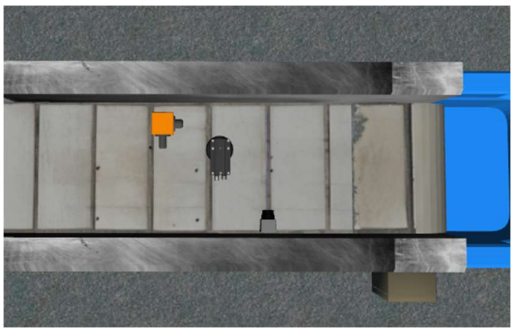

Fig. 12(a)

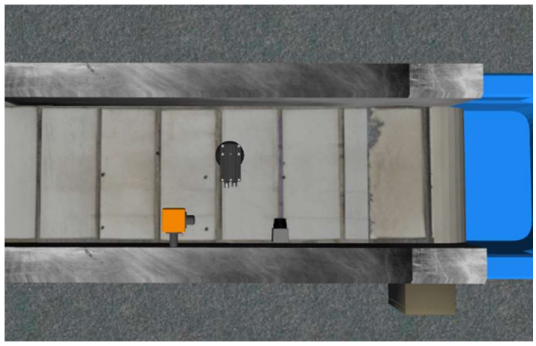

Fig. 12(b)

```
---
part_poses:
- part:
    color: 3
    type: 13
  pose:
    position:
      x: 0.32500381004874745
      y: 0.09964904592862522
      z: 0.08999542808738446
    orientation:
      x: 0.4999742969387746
      y: -0.5000489404463334
      z: -0.500024036641087
      w: 0.4999527201048417
tray_poses: []
sensor_pose:
  position:
    x: -0.65
    y: 3.7
    z: 1.2
  orientation:
    x: -0.7071045443232116
    y: -1.2248082623837582e-07
    z: 0.7071090180427863
    w: -1.2246817995071293e-07
---
```

Fig. 13(a)

```
---
part_poses:
- part:
    color: 3
    type: 13
  pose:
    position:
      x: 0.3249933404631235
      y: 0.31982964457630997
      z: -0.19000589191311007
    orientation:
      x: 0.4999632674327942
      y: -0.5000405951247587
      z: -0.5000350647880354
      w: 0.4999610669118415
tray_poses: []
sensor_pose:
  position:
    x: -0.65
    y: 3.7
    z: 1.2
  orientation:
    x: -0.7071045443232116
    y: -1.2248082623837582e-07
    z: 0.7071090180427863
    w: -1.2246817995071293e-07
---
```

Fig. 13(b)

The width of the conveyor-belt is divided into 9 lanes and depending on the offset value of the part it is designated to one of the lanes, as given in the below table.

Lane No.	z position value of part	offset in yaml
1	0.0725 or more	-1.00
2	0.0375 to 0.075	-0.75
3	0.0025 to 0.0375	-0.50
4	-0.0325 to 0.0025	-0.25
5	-0.0675 to -0.0325	0.00
6	-0.1025 to -0.0675	0.25
7	-0.1375 to -0.1025	0.50
8	-0.1725 to -0.1375	0.75
9	-0.1725 or less	1.00

Note: The z value range given in table z, only correlates to their specific offsets when the advanced logical camera is placed at the mentioned position and orientation.

Fig. 14 contains a video link that shows the conveyor part picking task completed successfully.

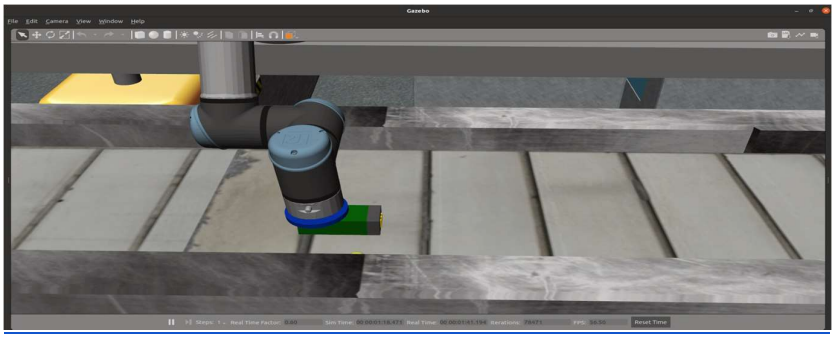

Fig. 14 (Link: https://youtu.be/dyEim3fUaGk)

Once the part is picked from the conveyor belt, it needs to be placed in an empty bin. To achieve this, first, an empty bin has to be determined, and it is done by discarding all the bins that contain parts from the available bins vector. For identifying the bins that contain parts, the bin numbers are extracted from the message obtained by subscribing to the topic ***/ariac/bin_parts***, which has the part information of each bin at the program start-up.

After determining the empty bin, the floor robot needs to move to the bin and drop the part in the desired slot. Once the part is dropped into the desired slot, the slot counter is incremented. For reaching the desired slot, the joint state values for that specific slot of that particular bin are fed to the floor robot. These joint state values were found by keeping track of ***/floor_robot_controller/status.***

The overall task of picking part from the conveyor belt and placing it on to an empty bin is described in the pseudocode 1.

```
BreakBeamCallback
BEGIN
    REPEAT
        IF object_detected
            CALL FloorRobotPickConveyorPart with part type, color and offset
            CALL FloorRobotPlacePartEmptyBin  with part_type, bin_number and slot_number
            INCREMENT slot_number
    UNTIL all_parts_spawned
END

FloorRobotPickConveyorPart with part type, color and offset
BEGIN
    CASE lane
        CASE part_type
            Condition 1:
                SET floor_robot.joint_states_values
                CALL FloorRobotMovetoTarget
                CALL FloorRobotSetGripperState with true
            ....
END

FloorRobotPlacePartEmptyBin  with part_type, bin_number and slot_number
BEGIN
    CASE bin_number
        CASE slot_number
            Condition 1:
                SET floor_robot.joint_states_values
                CALL FloorRobotMovetoTarget
                CALL FloorRobotSetGripperState with false
            ....
END
```

Pseudocode: 1

7. ORDER ACCOMPLISHMENT:

Once the task related to the conveyor belt is done, next, the CCS moves on to the task of accomplishing the orders. The OrderAccomplishment function is the heart of the CCS, it's here, where the type of the order is determined, and depending on the type, the following sub-function are called to accomplish that specific order. The OrderAccomplishment function in a sort of layman's terms can be viewed as described in the below pseudocode.

```
OrderAccomplishment:
BEGIN
    SELECT first order from orders list
    IF order is of high priority
    SELECT high priority order from orders list
    ENDIF
    IF a high priority order is announced in between this function
        CALL OrderAccomplishment RECURSIVELY till high priority orders are finished
    ENDIF
    IF the order type is kitting ...
        [PERFORM KITTING TASK]
    ENDIF

    IF the order type is assembly ...
        [PERFORM ASSEMBLY TASK]
    ENDIF

    IF order type is combined ...
        [PERFORM COMBINED TASK]
    ENDIF
END
```

Pseudocode: 2

7.1 KITTING TASK:

The perform Kitting Task block from the above Pseudocode2 is elaborated in the Pseudocode3

```
IF the order type is kitting
    SET agvNumber for order
    SET destination for order
    SET trayId
    CALL FloorRobotPickAndPlaceTray with trayId and agvNumber
    FOR each part in kitting order
        IF high priority flag is triggered
            CALL OrderAccomplishment
        ENDIF
        IF order part status is "not picked"
            FOR each part on bin
                IF part on bin matches order part
                    CALL FloorRobotPickBinPart with order part
                    CALL FloorRobotPlacePartOnKitTray with agvNumber and   order part quadrant
                ENDIF
            ENDFOR
            DECREMENT quantity of parts on bin
            SET order part status to "picked"
            BREAK
        ENDIF
    ENDFOR
    SET orderStatus to true
    FOR each part in kitting order
        IF order part status is "not picked"
            SET orderStatus to false
        ENDIF
    ENDFOR
    REQUEST for PerformQualityCheck service
    SET result as the response from PerformQualityCheck service
    IF all quality checks are not passed ...
        [RESOLVE QUALITY ISSUES]
    ENDIF
    IF all checks are passed or if all quality checks are resolved
        CALL MoveAGV with agvNumber and destination
    ENDIF
    REMOVE order from order list
ENDIF
```

Pseudocode: 3

The kitting task can be realized as a task with three sub-tasks, which are

1. Pick and place the required kit tray onto the desired agv.
2. Pick the necessary parts from the bins.
3. Place these parts in the appropriate quadrant of the kit tray.

Once all these three tasks are completed, a quality check service is run to check whether all the parts placed are desirable and don't consist of any issues. If there are any issues detected with the part on a specific quadrant, a set of instructions have to be performed to resolve it depending on the type of issue. The resolve quality issue block is discussed in section 8 After, resolving all the issues, if any, the kit tray is locked onto the agv, and the agv moves to its desired destination. An in-detail workflow of the kitting task is explained in section 2.1. Additionally, Fig. 15 below contains a video that shows a sample kitting task in action.

Fig. 15 (Link: https://youtu.be/poNPgrQA3vA)

The three sub-tasks forming the kitting task can be summarized by the pseudocodes given below.

```
FloorRobotPickAndPlaceTray with trayId and agvNumber RETURNS BOOL
BEGIN
    IF trayId found in station 1
        STORE tray_pose
        SET trayFound as true
    ELSE IF trayId found in station 2
        STORE tray_pose
        SET trayFound as true
    ENDIF
    IF trayFound is false
        STORE any available tray_pose
        SET trayFound as true
    ENDIF
    IF trayFound is true
        SET waypoints to tray_pose
        SET gripper to tray_gripper
        CALL FloorRobotMoveToTarget with waypoints
        SET agvWaypoints
        CALL FloorRobotMoveToTarget with agvWaypoints
        CALL FloorRobotSetGripperState with false
    ELSE
        tray not found
    ENDIF
END
```

Pseudocode: 4 Pick and place the required kit tray onto the desired agv.

```
FloorRobotPickBinPart with part_to_pick
BEGIN
    IF part_to_pick in right_bins THEN
        STORE part_pose
    ELSE IF part_to_pick in left_bins THEN
        STORE part_pose
    ELSE IF part_type in right_bins THEN
        STORE part_pose
    ELSE IF part_type in left_bins THEN
        STORE part_pose
    ELSE
        part_to_pick not found
    ENDIF

    SET waypoints with part_pose
    CALL FloorRobotMoveCartesian with waypoints
    CALL FloorRobotSetGripperState with true
    CALL AddModelToPlanningScene with part_to_pick
END
```

Pseudocode:5 Pick the necessary parts from the bins

```
FloorRobotPlacePartOnKitTray with agvNumber and quadrant RETURNS BOOL
BEGIN
    IF no parts attached
        RETURN false
    ENDIF
    SET agvWaypoints from agvNumber
    SET agvTrayPose from agvNumber
    CALL FloorRobotMoveToTarget with agvWaypoints
    SET waypoints for part from agvTrayPose and quadrant
    CALL FloorRobotMoveCartestian with waypoints
    CALL FloorRobotSetGripperState with false to drop part in quadrant
    SET partName from part attached on gripper
    CALL DetachObject with partName on quadrant
    CALL FloorRobotMoveCartesian with default waypoints to move robot to default position
    RETURN true
END
```

Pseudocode: 6 Place these parts in the appropriate quadrant of the kit tray

7.2 ASSEMBLY TASK:

The perform Assembly Task block from the Pseudocode 2 is elaborated in the Pseudocode 7

```
IF the order type is assembly
    FOR each agv
        IF a high priority order is announced in between this function
            CALL OrderAccomplishment RECURSIVELY till high priority orders are finished
        ENDIF
        SET destination based on station number
        CALL MoveAGV with agvNumber and destination
    ENDFOR
    CALL CeilingRobotMoveToAssemblyStation with station number
    CALL GetPreAssemblyPoses service to obtain part poses and store response
    IF response is valid
        SET agvPartPoses
    ENDIF
    FOR each part in assembly order
        SET PartToPick color and type as order part color and type
        IF order part is present in agv
            SET pose of PartToPick as agvPose obtained from service call
        ENDIF
        CALL CeilingRobotPickAGVPart with PartToPick
        SET assembly parameters
        CALL CeilingRobotMoveToAssemblyStation with station number
        CALL CeilingRobotAssemblePart with station number and assembly parameters
        CALL CeilingRobotMoveToAssemblyStation with station number
    ENDFOR
    REMOVE order from order list

ENDIF
```

Pseudocode: 7

For the assembly task, the first step is to move the given AGV to the desired assembly station. This is achieved by passing the AGV number and destination i.e. the assembly station number to the MoveAGV function. The complete assembly task is taken care of by the ceiling robot. Once the AGV reaches the assembly station the ceiling robot has to pick the parts from the

appropriate quadrants from the kit tray that's placed on the AGV and install those parts into the designed compartments to form an installed respirator.
An in-detail workflow of the assembly task is explained in section 2.2. Additionally, Fig. 16 below contains a video that shows a sample assembly task in action.

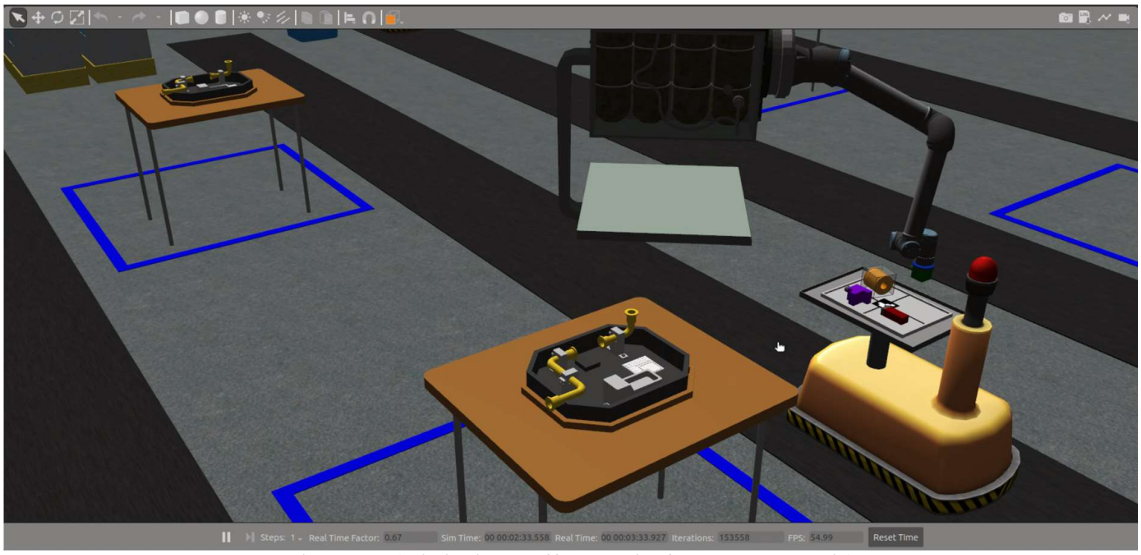

Fig. 16 (Link: https://youtu.be/cPTuMaUZplI)

The main sub-task for the assembly task is described in the pseudocode 8 given below.

```
CeilingRobotAssemblePart with station and part RETURNS BOOL
BEGIN
    IF part is not attached
        RETURN false
    ENDIF
    IF wrong part is attached
        RETURN false
    ENDIF
    SET assembly station frame based on station
    INSERT assembly station frame to world frame
    SET installDirection vector based on part properties
    SET waypoints to part
    CALL CeilingRobotMoveCartesian with waypoints
    CALL CeilingRobotWaitForAssemble with station and part
    CALL CeilingRobotSetGripperState with false to drop the part
    CALL CeilingRobotMoveCartesian with default waypoints to move to default position
    RETURN true
END
```

Pseudocode: 8

7.3 COMBINED TASK:

The Combined Task can be viewed as a combination of a kitting task followed by an assembly task. The one major difference between the kitting task that has to be performed within the combined task to that of the independent kitting task, is that there is no information given regarding the tray_id and AGV number. Thus, to perform the kitting task, first, the tray_id and

the AGV number have to be determined. This is done by taking the assembly station number into consideration, for example, if the desired assembly station number is 1, the AGV number is set to 1. And, for the tray_id, the lowest available id is selected.

The perform Combined Task block from the pseudocode 2 is elaborated in the pseudocode 9.

```
IF order type is combined
    SET agvNumber based on station mentioned in order
    SET trayId based on available trays
    CALL FloorRobotPickAndPlaceTray with trayId and agvNumber
    FOR each kitting part in combined order
        CALL FloorRobotPlacePartOnKitTray with part information
    ENDFOR
    REQUEST for PerformQualityCheck service
    SET result as the response from PerformQualityCheck service
    IF all quality checks are not passed ...
        [RESOLVE QUALITY ISSUES]
    ENDIF
    IF all checks are passed or if all quality checks are resolved
        SET destination based on station number
    ENDIF
    CALL MoveAGV with agvNumber and destination
    CALL CeilingRobotMoveToAssemblyStation with station number obtained from order
    CALL GetPreAssemblyPoses service to obtain assembly part poses and store response
    IF response is valid
        SET agvPartPoses
    ENDIF
    FOR each assembly part in combined order
        SET PartToPick color and type as order part color and type
        IF order part is present in agv
            SET pose of PartToPick as agvPose obtained from service call
        ENDIF
        CALL CeilingRobotPickAGVPart with PartToPick
        SET assembly parameters
        CALL CeilingRobotMoveToAssemblyStation with station number
        CALL CeilingRobotAssemblePart with station number and assembly parameters
        CALL CeilingRobotMoveToAssemblyStation with station number
    ENDFOR
    REMOVE order from order list
ENDIF
```

Pseudocode 9

Once all the kitting task is completed, similar to the independent kitting task, a quality check service is run to check whether all the parts placed are desirable and don't consist of any issues. If there are any issues detected with the part on a specific quadrant, a set of instructions have to be performed to resolve it depending on the type of issue. The resolve quality issue block is discussed in **section 8.** After, resolving all the issues, if any, the kit tray is locked onto the agv, and the agv moves to its desired assembly station. After, the AGV containing the required parts for assembly reaches the assembly station, the ceiling robot also moves to that assembly station to perform the assembly task. The assembly task within the combined task is identical to the independent assembly task.
An in-detail workflow of the combined task is explained in section 2.3. Additionally, Fig. 17 below contains a video that shows a sample combined task in action.

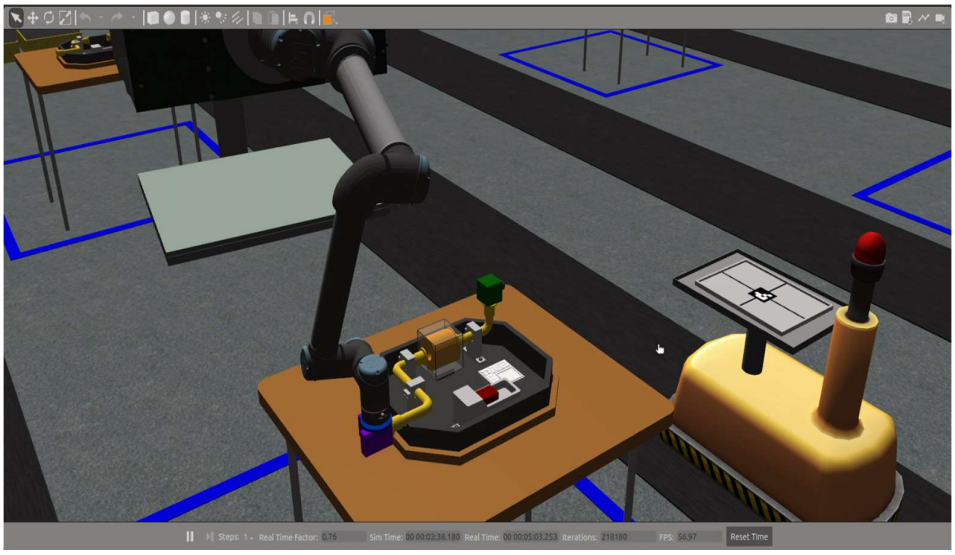

Fig. 17 (Link: https://youtu.be/cPTuMaUZplI)

8. AGILITY CHALLENGES:

1. **High-Priority Task:** As mentioned earlier in pseudocode Y, the high-priority task is solved using the concept of recursion. This is done by using a flag to store the priority of all incoming orders. In the OrderAccomplishment function, which is the heart of the CCS, a track is kept of this priority flag, and as soon as an order is received with a high-priority this flag sets true, and the whole OrderAccomplishment function goes into recursion. Thus, the CCS starts performing tasks to complete the new order and since the concept of recursion is used, once this high-priority order is completed successfully, the CCS goes back to the old order which it was dealing with.

   Fig. 18 below contains a video that shows a high-priority challenge being implemented successfully.

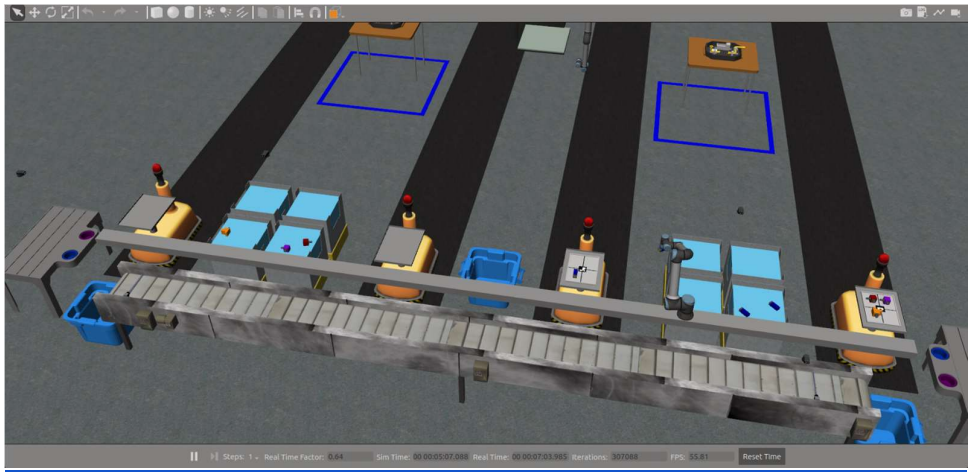

Fig. 18 (Link: https://youtu.be/E-aw8lfrsy4)

2. Quality Issues:

The remaining agility challenges are detected by the Quality check service, which is run after all the required parts of a kitting/combined task are placed on the kit tray. The quality resolve block comes into play when the Quality Check Service returns false to the *all_passed* condition.

Below is the pseudocode for the quality resolve block.

```
REQUEST for PerformQualityCheck service
SET result as the response from PerformQualityCheck service
IF all quality checks are not passed
    CASE quadrant
        Condition 1
            IF incorrect_tray
                PRINT "Incorrect Tray"
            ENDIF
            IF faulty_part
                CALL FloorRobotDisposeParts with agvNumber
                CALL FloorRobotPickBinPartReplaceMissing with part_to_pick
                CALL FloorRobotPlacePartOnKitTrayMissing with agvNumber
            ENDIF
            IF flipped_part
                CALL FloorRobotPickFlippedPart with agvNumber
                CALL CeilingRobotPlaceFlippedPartOnKitTray with agvNumber
            ENDIF
            IF  incorrect_part_color
                PRINT "Incorrect Part Color"
            ENDIF
            IF incorrect_part_type
                CALL FloorRobotDisposeParts with agvNumber
                CALL FloorRobotPickBinPartReplaceMissing with part_to_pick
                CALL FloorRobotPlacePartOnKitTrayMissing with agvNumber
            ENDIF
            IF missing part
                CALL FloorRobotPickBinPartReplaceMissing with part_to_pick
                CALL FloorRobotPlacePartOnKitTrayMissing with agvNumber
            ENDIF
        ....
ENDIF
```

Pseudocode 10

2.1 Insufficient/Missing Part Challenge:

This condition is met whenever a required part is not present on the specific quadrant of the kitting tray. There are multiple possibilities for this case to occur, such as, the required part was not available in the environment from the beginning, or if it was available it was dropped/misplaced by the robot while performing the pick and place task.

For resolving this issue, the function FloorRobotPickBinPartReplaceMissing is used. The main job of this function is to replace the missing part with another part, this can be done either by replacing it with the exact same part (same color and same type) if available, or with a part of a similar type irrespective of the color.

The pseudocodes for the two main functions used to solve the missing part challenge are given below.

```
FloorRobotPickBinPartReplaceMissing with part_to_pick
BEGIN
    IF part_to_pick in right_bins THEN
        STORE part_pose
    ELSE IF part_to_pick in left_bins THEN
        STORE part_pose
    ELSE IF part_type in right_bins THEN
        STORE part_pose
    ELSE IF part_type in left_bins THEN
        STORE part_pose
    ELSE
        part_to_pick not found
    ENDIF

    SET waypoints with part_pose
    CALL FloorRobotMoveCartesian with waypoints
    CALL FloorRobotSetGripperState with true
    CALL AddModelToPlanningScene with part_to_pick
END
```

Pseudocode 11

```
FloorRobotPlacePartOnKitTrayMissing with agvNumber, quadrant and type
BEGIN
    SET agvWaypoints from agvNumber
    SET agvTrayPose from agvNumber
    CALL FloorRobotMoveToTarget with agvWaypoints
    SET waypoints for part from agvTrayPose and quadrant
    CALL FloorRobotMoveCartestian with waypoints
    CALL FloorRobotSetGripperState with false
    SET partName from part attached on gripper
    CALL DetachObject with partName on quadrant to drop part in quadrant
    CALL FloorRobotMoveCartesian with default waypoints to move robot to default position
    RETURN true
END
```

Pseudocode 12

Fig. 19 below contains a video that shows the insufficient/missing part challenge being resolved successfully.

Fig. 19 (Link: https://youtu.be/kGTlkeGYpM0)

2.2 Faulty Part Challenge:

The Faulty part condition is not dependent on any mishappenings caused during the run, but it is a pre-run status that is set in the trial yaml file. Once this condition is met the task for the floor robot is to go and pick the faulty part from the appropriate quadrant, and drop the faulty part into the part disposal bin. Once the faulty part is disposed of, it has to be replaced by another similar part, for this once again the function FloorRobotPickBinPartReplaceMissing is used.

One major challenge faced in disposing of the faulty part was that when the floor robot went to pick up the faulty part from the kit tray, the Moveit planner refused to find a trajectory successfully. This problem occurred since the part on the kit tray was already added to the planning scene, and Moveit was considering it as an obstacle. Thus to resolve this issue, the faulty part had to be removed from the planning scene first.

Pseudocode 13 describes the task of disposing the faulty part.

```
FloorRobotDisposeParts with agvNumber and quadrant RETURNS BOOL
BEGIN
    SET agvWaypoints based on agvNumber
    SET agvTrayPose from agvNumber
    CALL FloorRobotMoveToTarget with agvWaypoints
    SET partPose from agvTrayPose and quadrant
    SET waypoints for partPose
    CALL FloorRobotMoveCartesian with waypoints
    CALL FloorRobotSetGripperState with true and pick up faulty part
    SET waypoints to waste bin
    CALL FloorRobotMoveToTarget with waypoints
    CALL FloorRobotSetGripperState with false to drop part in waste bin
    RETURN true
END
```

Pseudocode 13

Fig. 20 below contains a video that shows the insufficient/missing part challenge being resolved successfully.

Fig. 20 (Link: https://youtu.be/kGTlkeGYpM0)

2.3 Flipped Part Challenge:

Like the faulty part challenge, even the flipped part condition is a pre-run status that is set in the trial yaml file wherein the part section of the flipped condition is set as true. Unlike the faulty part challenge, flipped part can also be caused by some mishappening during the run, which caused the part to be placed on the kit tray as flipped.

Once the flipped part condition is detected by the quality check service, the task for the CCS is to un-flip the part and place it back into the same quadrant. For overcoming this challenge, a synergy between both robots is required. First, the floor robot needs to go and pick the flipped part from the appropriate quadrant, and go and wait in an upright direction. Next, the ceiling robot has to move and place its gripper right below that of the floor robot. Once, this is done the part is dropped by the floor robot and caught by the ceiling robot, thus successfully un-flipping the part. Next, the ceiling robot has to place the part into the appropriate quadrant.

Pseudocode 14 describes the task of overcoming the flipped part challenge.

```
FloorRobotPickFlippedPart with agvNumber and quadrant RETURNS BOOL
BEGIN
    SET agvTrayPose based on agvNumber
    SET partDropPose based on agvNumber and quadrant
    SET waypoints to partDropPose
    CALL FloorRobotMoveCartesian with waypoints
    CALL FloorRobotSetGripperState to true to pick up the part
    CALL FloorRobotMoveCartesian to move to a default position
    CALL CeliingRobotMoveToTarget to a preset pick up position
    CALL CeilingRobotSetGripperState to true so that it picks up the part
    CALL FloorRobotSetGripperState to false so that it drops the part onto ceiling robot
END

CeilingRobotPlaceFlippedPartOnKitTray with agvNumber and quadrant RETURNS BOOL
BEGIN
    SET jointValueTarget of ceiling robot to preset values based on agvNumber
    CALL CeilingRobotMovetoTarget with set joint values
    CALL CeilingRobotSetGripperState with false
    CALL CeilingRobotSendHome to send to home position
    RETURN true
END
```

Pseudocode 14

One major problem faced in flipped part challenge apart from setting up the joint state values of both the robots, and maintaining coordination between them, is when a part is flipped, the pose of the part changes drastically. Due, to this change in pose, the standard part-picking functions fail, thus a custom part-picking function is used to pick the flipped part from the kit tray. Another observation was that the part is flipped along the roll axis in

the world frame, so to confirm whether a part is flipped or not, the part pose had to be transformed to world coordinates and then the roll coordinate was extracted.

Fig. 21 below contains a video that shows the insufficient/missing part challenge being resolved successfully.

Fig. 21 (Link: https://youtu.be/Q-_jIaNquL8)

9. COURSE FEEDBACK:

We would like to express our appreciation for the course you taught this semester. It was engaging, informative, and provided a positive learning environment. Your efforts to create a supportive atmosphere were evident throughout the course. The course content was well-structured and relevant to the course objectives. We found the instructional methods to be effective and engaging. The assessments and assignments were clearly communicated, which made it easy to understand what was expected of us. We appreciate the valuable feedback provided on assignment queries and assessments. The use of ROS2 instead of ROS1, for the course certainly helped us to improve our ROS programming skills, given that companies are expecting their employees to have knowledge of ROS2.

While I found the course to be engaging and informative, the assignments could have been structured better. The first 2 RWA's weren't as time-consuming as the other 2 RWA's. The shift from RWA2 to RWA3 and RWA4 was massive and honestly, we didn't expect such a huge shift. It was difficult to manage towards the end, along with other courses. There could have been another RWA between RWA2 and RWA3, where the work could have evenly been divided among each RWAs. Alternatively, RWA1 and RWA2 could have been combined into a single RWA. By doing that, we could have focused more on Agility challenges (which is the crux of the course). As they were given at the very end, it was very difficult to complete, and we couldn't do a great job with that. Overall, we thoroughly enjoyed this course and would highly recommend it to the others.

10. PROBLEMS FACED:

1. There were multiple instances when Gazebo completely stopped working and ARIAC just crashed.
2. At times, Moveit failed to follow waypoints created by itself.
3. Since working of Gazebo depends on the Real-time factor of laptops, it was difficult to reproduce results in multiple devices.
4. The geometry of the pump was an issue all along the course. Nearly 70 % of the time, was spent to rectify that. Yet, it was unsuccessful.

11. CONTRIBUTION

1. Kiran S Patil:

• Floor Robot Bin parts Pickup.

• Faulty Part detection.

• Flipped Part detection.

2. Vyshnav Achuthan:

• Ceiling robot parts assembly.

• Cv2 Image processing Pipeline (Attempted).

3. Aniruddh Balram:

• Dataset creation for Cv2 image pipeline.

• Cv2 Image Processing Pipeline. (Attempted)

• High Priority Order Agility Challenge.

4. Badrinarayanan:

• Floor Robot Conveyor Parts pickup

• Order storage Class.