ARIAC Package, Final Report

ENPM 663
Z. Kootbally & C. Schlenoff

Group 1
Robens Cyprien
Manas Desai
Anne-Michelle Lieberson
Hamsaavarthan Ravichandar

May 17, 2025

Table of Contents

Project Overview

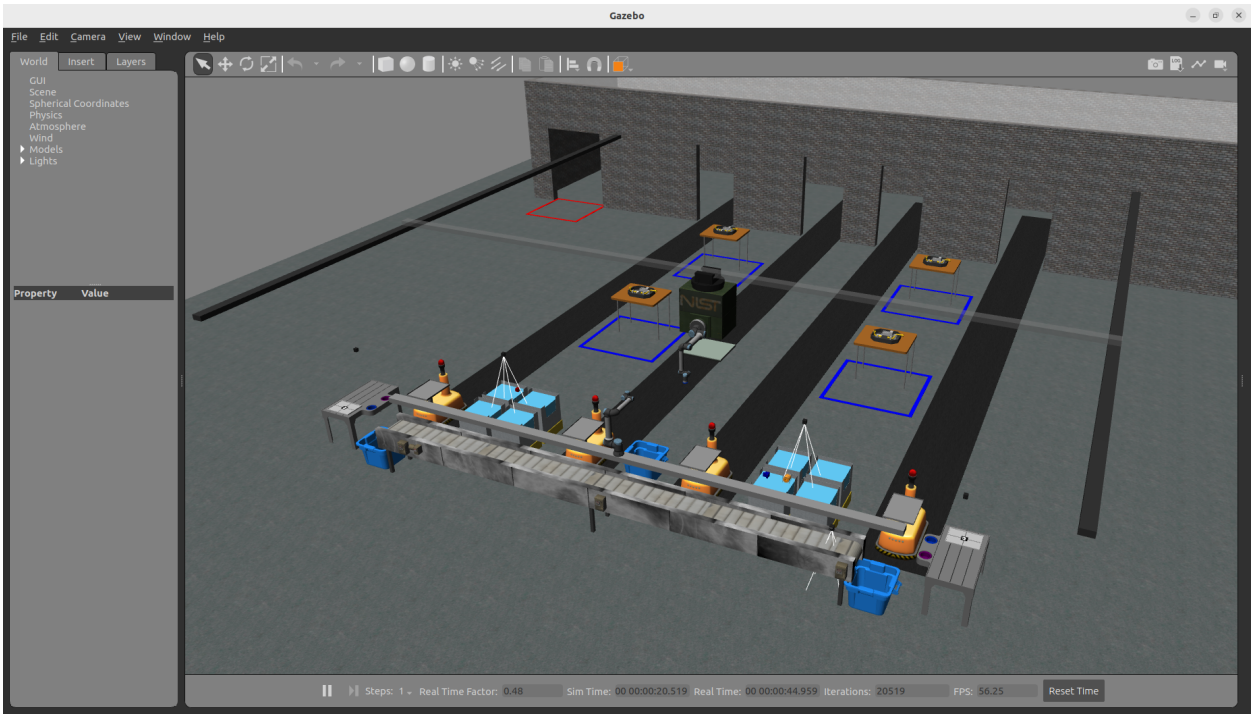

Fig 1.: ARIAC environment with final project trial file

The Agile Robotics for Industrial Automation Competition (ARIAC) is an annual robotics challenge hosted by the National Institute of Standards and Technology (NIST) since June 2017, designed to push the boundaries of industrial robot agility in a simulated manufacturing setting. Participants develop a Competitor Control System (CCS) that interfaces with an ARIAC Manager (AM) within a containerized Docker environment to execute trials, leveraging ROS 2 Iron on Ubuntu 22.04 for seamless integration. The competition unfolds in a dynamic Gazebo simulation where teams must perform pick-and-place, assembly, and kitting tasks, coordinating multiple robot types—including autonomous guided vehicles (AGVs), floor robots, and ceiling gantry systems—to mirror real-world factory floors. ARIAC incorporates a suite of agility challenges—faulty parts, flipped parts, dropped parts, robot malfunctions, sensor blackouts, high-priority orders, and insufficient parts—to emulate the unpredictability of modern manufacturing. Scoring combines performance, efficiency, and cost metrics into a unified framework, making ARIAC a standardized testbed for benchmarking autonomous robotic algorithms and shaping future standards in manufacturing agility.

The specific objectives of the final project were to perform a kitting task using both the ceiling and the floor robot, and then ship both the robots to the warehouse, submit the order, score the order, and then end the competition. The kitting task involved picking up parts from the bin using both the floor and the ceiling robot, placing it on their respective trays, and then picking and placing those trays on the AGVs using the floor robot.

Software Architecture

Once the gazebo simulation is launched using the command *ros2 launch ariac_gazebo ariac.launch.py trial_name:=final_project competitor_pkg:=group1_ariac_final* with the final_project trial and the sensors config file, all the parts appear in the bin along with the sensors configured in order to detect them.

- Then, when we run the start competition service using the command *ros2 run group1_ariac_final check_competition_status.py*, the competition status becomes ready and all the orders are announced. The sensors start to publish data on their respective topics and we can check the camera feed using *rqt_image_view* library.
- In a separate terminal, we run ros2 run group1_ariac_final tray_detector.py. The tray detector node is responsible for detecting all the trays present in the environment and publishing the information like poses.
- In a separate terminal, we run *ros2 run group1_ariac_final bin_detector.py* and the bin detector node is responsible for detecting all the parts present in the bins throughout the environment and publish the part type, color and pose information.
- In a separate terminal, we run the command *ros2 launch group1_ariac_final group1_ariac_final.launch.py program:=python rviz:=true* and this command initializes both the floor and the ceiling robot nodes along with the MoveIt packages. The floor and ceiling robot nodes subscribe to the parts and poses information that the tray detector and bin detector nodes provide and supply the part type and pose information to the execute_pick_and_place_part as well as execute_pick_and_place_tray function in order to execute pick and place operations using the MoveIt library.

Once the kitting operation is complete and the trays are placed on their respective AGVs, the tray lock service is called to lock the trays and then ship the AGV to the warehouse. Fig.1 shows how the entire workflow for fulfilling the orders and shipping them to the warehouse works. Fig.2 also shows how the different nodes communicate with each other.

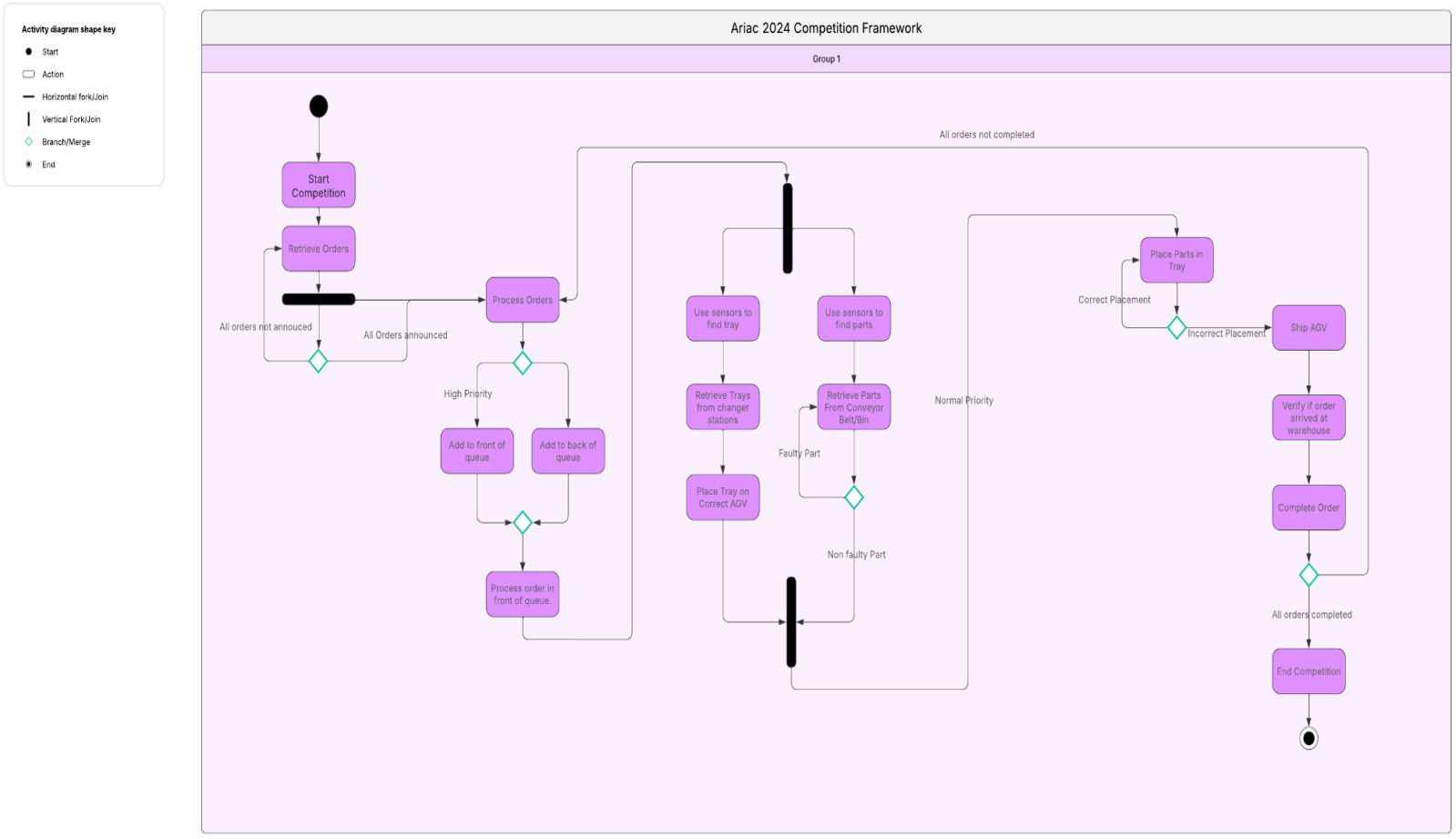

Fig 2. : The workflow of the order fulfillment process

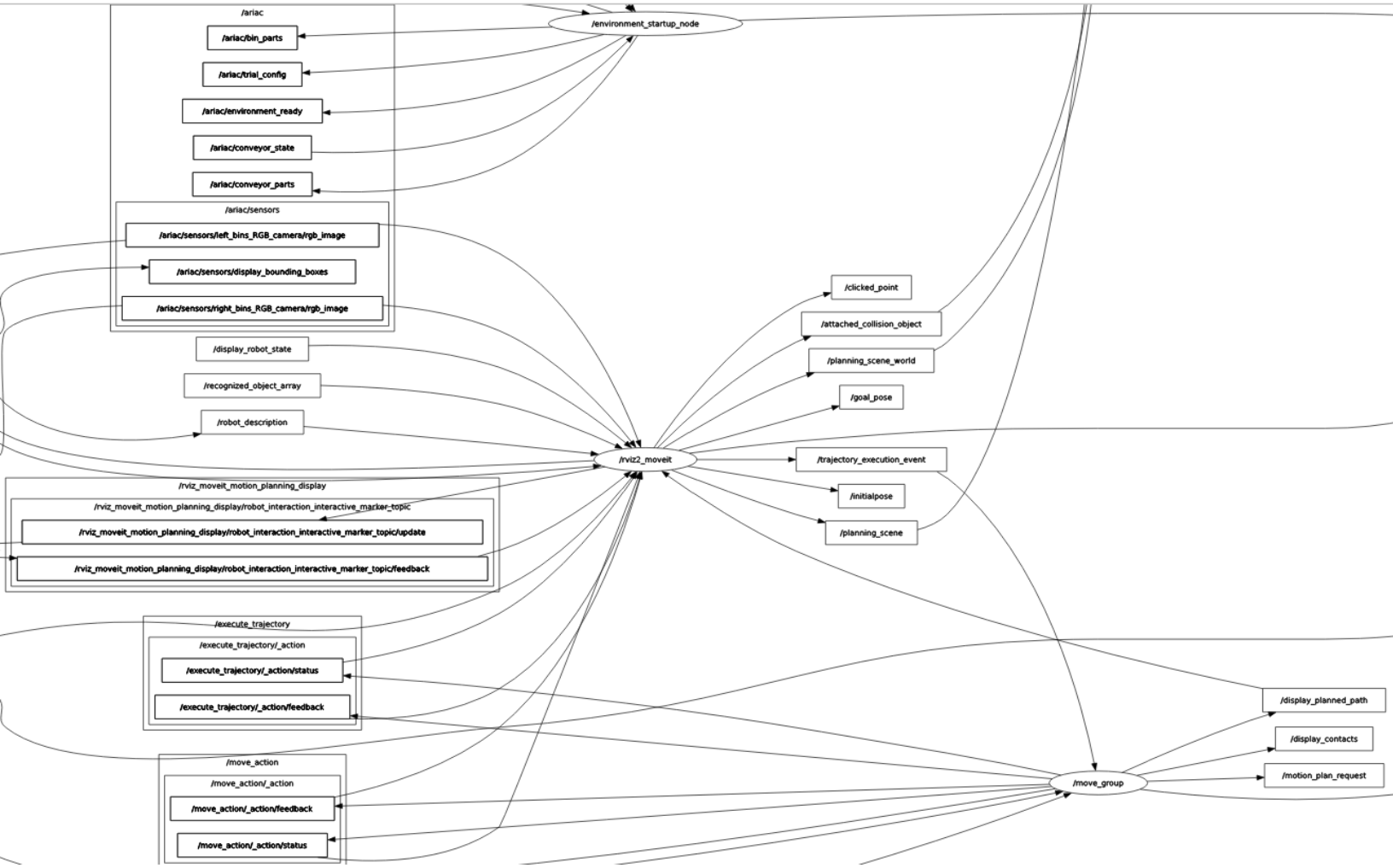

Fig 3. : rqt_graph showing the communication between different ROS nodes

Implementation Details

Perception System

Trays

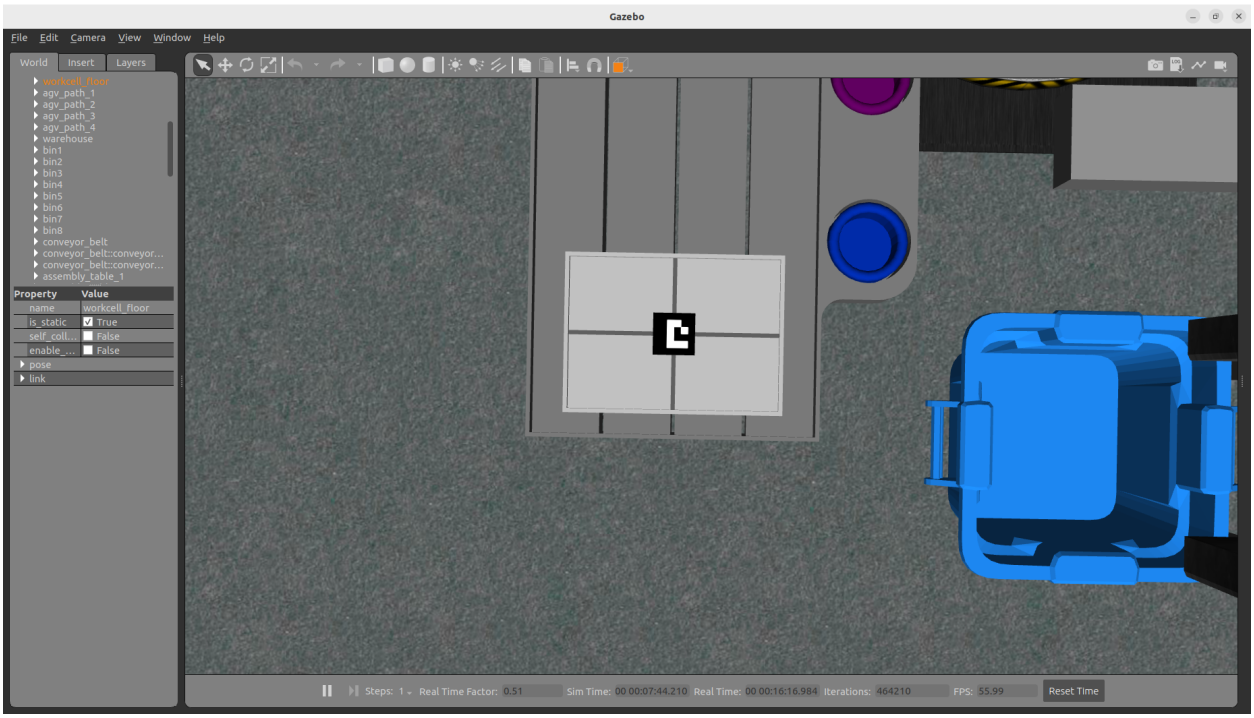

Fig 4. : Tray table with ARUCO marker

Our competitor package detects trays through the tray detector node. In the ARIAC environment, two RGBD cameras are placed above the left and right tray tables so that they can detect all possible trays on the table. In the tray detector node, we read the camera images through the camera sensors. The RGBD cameras also have a depth image that we do not use. Next, the images for the left and right trays are processed using opencv's Aruco package to detect the Aruco image on the tray. The Aruco image gives the tray id, and the pose of the tray in camera frames. Since the camera image was consistently reading the Aruco markers on the tray in the environment, we did not do any processing or filtering to improve detection accuracy. To convert the camera frame detected by the Aruco markers to world frame representation, We used kdl frame multiplication to multiply the camera pose, and the tray pose in the camera frame to obtain the tray pose in the world frame. The resulting pose was close to the actual pose of the tray in the world but buffer values needed to be added to bring it close enough to allow for pickup. We then published the tray pose using a *KitTrayPose* message through the */group1_ariac/kit_tray_poses* topic. The floor robot controller then reads the message and extracts the tray id and pose, and stores them in a dictionary with the tray id as the key and the tray pose as the value. Once an order is announced, the floor robot controller retrieves the tray id of the order. Then it extracts the pose of the tray from the dictionary.

Bins

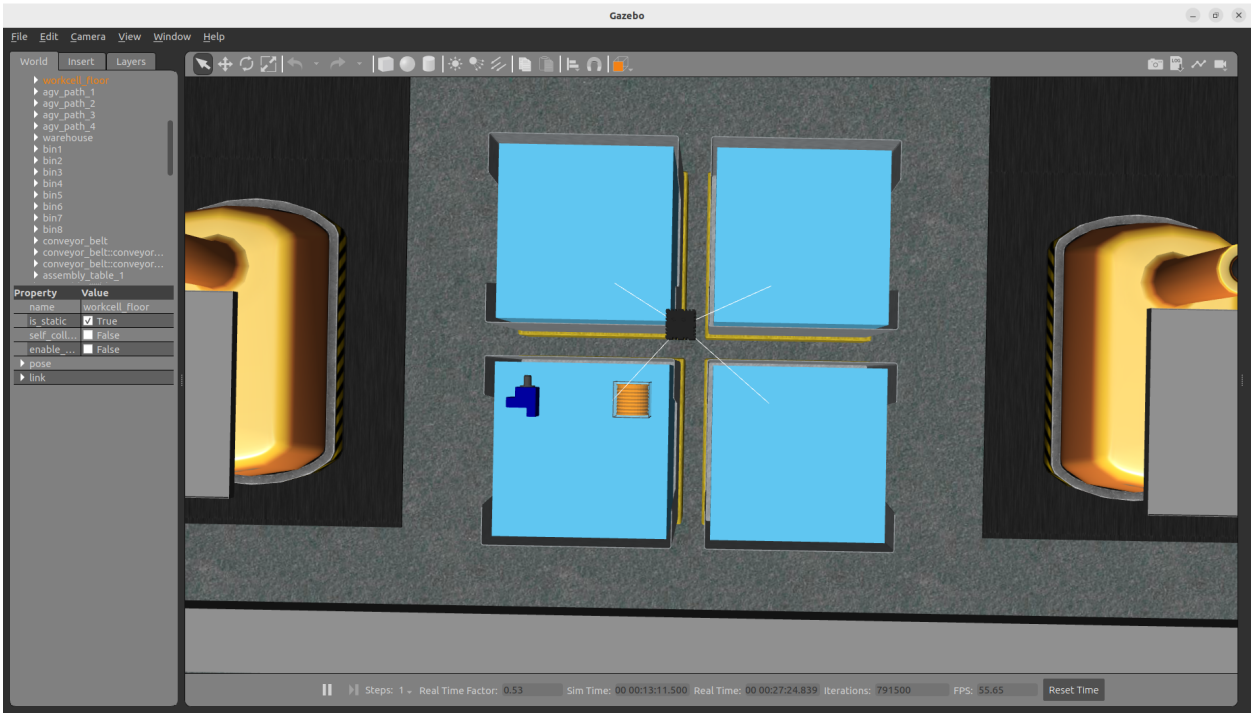

Fig 5. : Bin parts on right bins

The Parts in bins detection System in this ARIAC 2024 project operates as a multi-step pipeline, integrating image processing, template matching, and pose estimation to accurately identify and localize parts within predefined bin locations. The system begins by capturing RGB images from overhead cameras, systematically segmenting the images into regions of interest (ROIs) based on predefined bin positions. Next, an HSV-based color segmentation is applied, filtering pixel values according to predefined color thresholds to isolate different object categories based on their distinct color characteristics. Once color segmentation is performed, the system conducts template matching using preloaded grayscale reference images of various parts. Through OpenCV functions, the detected shapes are validated against known templates, followed by non-max suppression to refine bounding boxes and remove overlapping detections. Once a part has been detected and classified, its bin slot location is determined using predefined coordinate mappings. The system then performs pose transformations, converting the detected object's position from the camera frame to the world frame using transformation matrices derived from roll, pitch, and yaw angle calculations. This ensures accurate reporting of each part's real-world pose, which is then published as ROS2 messages (*BinPartsPoseLot*) for external robotic manipulation or competition-related actions. Additionally, debugging tools allow visualization of detection results via ROS image topics, ensuring real-time monitoring and refinement.

Key Functionalities:

1. Camera integration and Image processing
2. Color-based Object Detection

3. Template matching for part recognition
4. Post estimation and Frame transformation
5. Data logging and Publishing using custom ROS2 interfaces

By combining image recognition, geometric transformations, and structured data logging, the system provides an efficient and scalable method for part identification and localization within dynamic industrial automation environments.

Conveyor

```
group1_ariac_msgs/ConveyorPartsPose[] conveyor_parts
        ariac_msgs/PartPose initial_detection
                ariac_msgs/Part part
                        uint8 RED=0
                        uint8 GREEN=1
                        uint8 BLUE=2
                        uint8 ORANGE=3
                        uint8 PURPLE=4
                        uint8 BATTERY=10
                        uint8 PUMP=11
                        uint8 SENSOR=12
                        uint8 REGULATOR=13
                        uint8 color #
                        uint8 type #
                geometry_msgs/Pose pose
                        Point position
                                float64 x
                                float64 y
                                float64 z
                        Quaternion orientation
                                float64 x 0
                                float64 y 0
                                float64 z 0
                                float64 w 1
        ariac_msgs/PartPose[] predictions
                ariac_msgs/Part part
                        uint8 RED=0
                        uint8 GREEN=1
                        uint8 BLUE=2
                        uint8 ORANGE=3
                        uint8 PURPLE=4
                        uint8 BATTERY=10
                        uint8 PUMP=11
                        uint8 SENSOR=12
                        uint8 REGULATOR=13
                        uint8 color #
                        uint8 type #
                geometry_msgs/Pose pose
                        Point position
                                float64 x
                                float64 y
                                float64 z
                        Quaternion orientation
                                float64 x 0
                                float64 y 0
                                float64 z 0
                                float64 w 1
```

Fig 6. : Message to store conveyor poses

In previous assignments, we had detected moving parts on the conveyor and created a velocity estimator. We had intended to use it here in order to attempt to do the extra credit portion, but were

unable to refine it enough. It is still in the code however (in the *FloorRobotControl* node), and so it will be explained here:

First, we create a subscriber to the *group1_ariac/conveyor_part_poses topic*, which is a custom topic that holds the information of detected parts (the publisher to this topic is not in this code, but was in the previous assignment as *self._part_conveyor_pose_publisher*). In terms of processing camera data, it was necessary to figure out a way to not double-count parts as they are moving. In fact, we do not even need to consider multiple instances of the same part, since we know that we will only have one or a few orders (to make the detection more robust and track multiple instances of the part, additional checks would need to be made to verify, such as using location as a check). With this simplification in mind, we use a dictionary with keys corresponding to a string of the format *part_color + " " + part_type* and create a key based on the detected part. If the dictionary using this key is *None*, we can assume that this is the first time we are detecting this part, otherwise this is a known part and we update its predicted location instead using assumed conveyor velocities.

Motion Planning

```
[floor_grab.py-1] [INFO] [1747249794.643180512] [floor_robot_node]: Successfully added bin6 to planning scene
[floor_grab.py-1] [INFO] [1747249794.644992536] [floor_robot_node]: Adding model bin7 to planning scene
[floor_grab.py-1] [INFO] [1747249794.668709395] [floor_robot_node]: Successfully added bin7 to planning scene
[floor_grab.py-1] [INFO] [1747249794.676927164] [floor_robot_node]: Adding model bin8 to planning scene
[floor_grab.py-1] [INFO] [1747249794.700105446] [floor_robot_node]: Successfully added bin8 to planning scene
[floor_grab.py-1] [INFO] [1747249794.702329595] [floor_robot_node]: Adding model as1 to planning scene
[floor_grab.py-1] [INFO] [1747249794.734971164] [floor_robot_node]: Successfully added as1 to planning scene
[floor_grab.py-1] [INFO] [1747249794.737191368] [floor_robot_node]: Adding model as2 to planning scene
[floor_grab.py-1] [INFO] [1747249794.770928176] [floor_robot_node]: Successfully added as2 to planning scene
[floor_grab.py-1] [INFO] [1747249794.772076609] [floor_robot_node]: Adding model as3 to planning scene
[floor_grab.py-1] [INFO] [1747249794.807640711] [floor_robot_node]: Successfully added as3 to planning scene
[floor_grab.py-1] [INFO] [1747249794.809207649] [floor_robot_node]: Adding model as4 to planning scene
[floor_grab.py-1] [INFO] [1747249794.835830403] [floor_robot_node]: Successfully added as4 to planning scene
[floor_grab.py-1] [INFO] [1747249794.837682917] [floor_robot_node]: Getting transform for frame: as1_insert_frame
[floor_grab.py-1] [INFO] [1747249794.840434442] [floor_robot_node]: Adding model as1_insert to planning scene
[ceiling_grab.py-2] [INFO] [1747249795.075982507] [ceiling_robot_node]: Successfully added as2_insert to planning scene
[ceiling_grab.py-2] [INFO] [1747249795.077451894] [ceiling_robot_node]: Getting transform for frame: as3_insert_frame
[ceiling_grab.py-2] [INFO] [1747249795.079582158] [ceiling_robot_node]: Adding model as3_insert to planning scene
[floor_grab.py-1] [INFO] [1747249795.388051271] [floor_robot_node]: Successfully added as1_insert to planning scene
[floor_grab.py-1] [INFO] [1747249795.389072661] [floor_robot_node]: Getting transform for frame: as2_insert_frame
[floor_grab.py-1] [INFO] [1747249795.390357530] [floor_robot_node]: Adding model as2_insert to planning scene
[ceiling_grab.py-2] [INFO] [1747249795.609899517] [ceiling_robot_node]: Successfully added as3_insert to planning scene
[ceiling_grab.py-2] [INFO] [1747249795.611675243] [ceiling_robot_node]: Getting transform for frame: as4_insert_frame
[ceiling_grab.py-2] [INFO] [1747249795.616756573] [ceiling_robot_node]: Adding model as4_insert to planning scene
[floor_grab.py-1] [INFO] [1747249795.930870715] [floor_robot_node]: Successfully added as2_insert to planning scene
[floor_grab.py-1] [INFO] [1747249795.934503243] [floor_robot_node]: Getting transform for frame: as3_insert_frame
[floor_grab.py-1] [INFO] [1747249795.939559624] [floor_robot_node]: Adding model as3_insert to planning scene
[ceiling_grab.py-2] [INFO] [1747249796.145635447] [ceiling_robot_node]: Successfully added as4_insert to planning scene
[ceiling_grab.py-2] [INFO] [1747249796.146654092] [ceiling_robot_node]: Adding model conveyor to planning scene
[ceiling_grab.py-2] [INFO] [1747249796.164151893] [ceiling_robot_node]: Successfully added conveyor to planning scene
[ceiling_grab.py-2] [INFO] [1747249796.165094473] [ceiling_robot_node]: Adding model kts1_table to planning scene
[ceiling_grab.py-2] [INFO] [1747249796.250496716] [ceiling_robot_node]: Successfully added kts1_table to planning scene
[ceiling_grab.py-2] [INFO] [1747249796.251978880] [ceiling_robot_node]: Adding model kts2_table to planning scene
[ceiling_grab.py-2] [INFO] [1747249796.354281301] [ceiling_robot_node]: Successfully added kts2_table to planning scene
[ceiling_grab.py-2] [INFO] [1747249796.355265899] [ceiling_robot_node]: Planning scene initialization complete
[floor_grab.py-1] [INFO] [1747249796.467246309] [floor_robot_node]: Successfully added as3_insert to planning scene
[floor_grab.py-1] [INFO] [1747249796.468633565] [floor_robot_node]: Getting transform for frame: as4_insert_frame
[floor_grab.py-1] [INFO] [1747249796.471149420] [floor_robot_node]: Adding model as4_insert to planning scene
[floor_grab.py-1] [INFO] [1747249796.989057971] [floor_robot_node]: Successfully added as4_insert to planning scene
[floor_grab.py-1] [INFO] [1747249796.990125665] [floor_robot_node]: Adding model conveyor to planning scene
[floor_grab.py-1] [INFO] [1747249797.020920637] [floor_robot_node]: Successfully added conveyor to planning scene
[floor_grab.py-1] [INFO] [1747249797.027611721] [floor_robot_node]: Adding model kts1_table to planning scene
[floor_grab.py-1] [INFO] [1747249797.109594494] [floor_robot_node]: Successfully added kts1_table to planning scene
[floor_grab.py-1] [INFO] [1747249797.111172261] [floor_robot_node]: Adding model kts2_table to planning scene
[floor_grab.py-1] [INFO] [1747249797.188822349] [floor_robot_node]: Successfully added kts2_table to planning scene
[floor_grab.py-1] [INFO] [1747249797.189647277] [floor_robot_node]: Planning scene initialization complete
[ceiling_grab.py-2] [WARN] [1747249797.388905318] [ceiling_robot_node]: Timeout waiting for planning scene service, trying direct publish instead
[ceiling_grab.py-2] [INFO] [1747249797.610234176] [ceiling_robot_node]: Directly published planning scene with 0 objects
[ceiling_grab.py-2] [INFO] [1747249797.611295547] [ceiling_robot_node]: ---> All objects added
[ceiling_grab.py-2] [INFO] [1747249797.617661074] [ceiling_robot_node]: Competition state is: 1
[floor_grab.py-1] [WARN] [1747249798.210402997] [floor_robot_node]: Timeout waiting for planning scene service, trying direct publish instead
[floor_grab.py-1] [INFO] [1747249798.420149473] [floor_robot_node]: Directly published planning scene with 0 objects
[floor_grab.py-1] [INFO] [1747249798.420799798] [floor_robot_node]: ---> All objects added
[floor_grab.py-1] [INFO] [1747249798.430437774] [floor_robot_node]: Competition state is: 1
```

Fig 7. : Adding MoveIt to planner for floor and ceiling robot.

Both the floor and ceiling robots use MoveIt for motion planning for both robots. The floor robot uses cartesian path planning. This is when there are some waypoints that the robot has to follow to reach its eventual goal. We used cartesian path planning because there were a large amount of obstacles that the floor could encounter like the AGVs, and tray tables. The ceiling robot uses the built-in MoveIt planner. There are fewer obstacles in the path of the ceiling robot so its path does not need to be as constrained as the floor robot. To further prevent collisions for the floor and ceiling robots, we did not move the robots directly to their target poses but instead above the target pose, and then lower down to the target pose. This is because when we set the desired pose as the goal, the robot tends to contact the tray tables, trays, or AGVs before they reach their goal. For some tasks, We had to reach the robot as high as 0.5 meter above the target pose to prevent collision. To improve efficiency, we had both the robots move at the same time so that they did not have to wait for one another.

Task Coordination

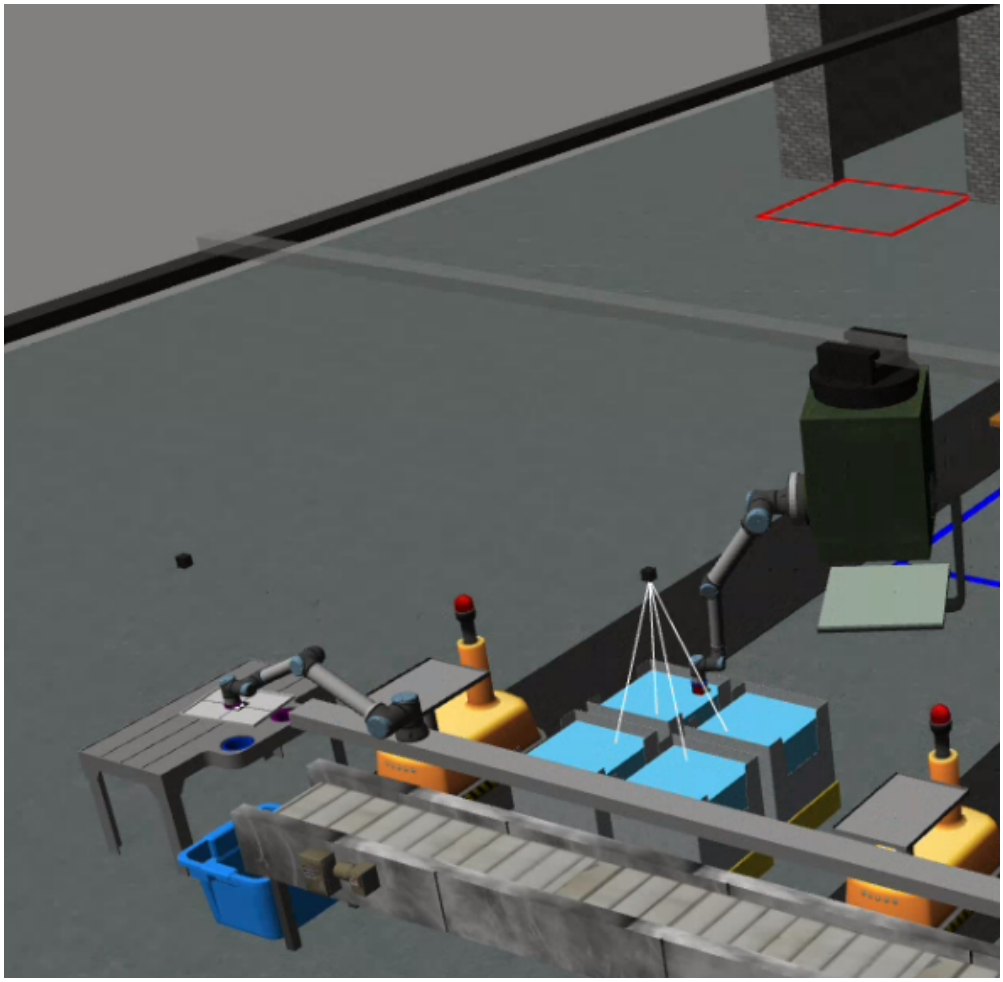

Fig 8. : Floor robot picking up tray and ceiling robot picking up red regulator.

Both of our robots are able to move at the same time. But since the floor robot has to pick up the tray, the ceiling robot has to wait for it to put the tray on the AGV before it can put the red regulator on the tray. The robots do not start moving until the tray poses have been detected by the robot controller nodes. The floor robot is given a 10 seconds head start so that it is able to pick up and place the tray before the ceiling robot needs to place its part on the tray. After the floor robot places the tray on the correct AGV, it moves on to pick up and place the blue sensor, and orange pump while the ceiling robot places the red regulator

For the floor robot, we used the provided error handling and recovery strategies in lecture 9. This includes replanning when the path planning process fails, and returning to a known working position if the robot collides with itself or the environment, or if the robot fails to follow the planned trajectory. The ceiling robot attempts to replan its path three times for recovery.

For the final step, we ship the AGV after the floor robot is done completing its tasks. The ceiling robot has to complete one task while the floor robot has to complete two tasks and has to change its gripper. So it felt safe to assume that once the floor robot is done completing its tasks, all of the tasks have been completed.

Challenges and Solutions

Bin Parts Detection

The primary challenge in parts detection was the difficulty in distinguishing between similar-looking part templates, leading to misclassification. Due to minimal structural differences, false positive values occurred when parts had overlapping features, making conventional template matching techniques insufficient for precise classification. Traditional pixel-based correlation methods struggled to differentiate between nearly identical objects, affecting detection reliability in complex bin arrangements. Hybrid Validation Approach: Feature matching integration improved recognition by allowing scale and rotation invariance. Furthermore, Lowe's ratio test helped remove weak feature correspondences to reduce false matches. Combined template matching with geometric analysis to refine detection and applied contour and shape detection to improve classification of closely related objects. Tuning descriptor thresholds specific to each template and geometric verification improved detection accuracy while minimizing false positives in challenging scenarios.

Tray Camera to World Pose

Opencv's Aruco provides the tray pose in camera frames rather than in world frames. To convert the camera frames to world frames, we had to multiply the pose of the camera in the world frames to the pose of the tray in the camera frame. The issue was that the resulting pose did not match the actual pose of the tray in the world frame. The discrepancy was enough that the floor robot would

not be able to properly place the tray. To fix this problem, we had to insert buffer values to bring them to the expected values. But the buffer values were not universal and needed to be changed between different tray poses. In the future, we would use frame transformations to see if that will reduce errors.

Trajectory Planning

The floor robot was frequently crashing into tray tables when picking the tray and into the AGVs when navigating to place the tray tables. It was also colliding with other parts while attempting to pick up a part. The solution was to raise the initial target pose of the robot arm above the part that it was going to pick up. This was to ensure that when the robot was nearing its target, its gripper was not low enough to hit a tray table or AGV.

Conveyor Part Double-Counting

An issue we had in previous projects was double-counting parts on the conveyor. Because of some of the above perception challenges, a single part registered in the camera view may be detected as multiple instances. In order to address this for this package, we changed our approach to simply only tracking one instance of any given part + color combination by utilizing a dictionary with the key set to this combination. We were able to improve the perception to limit this double-counting, but, due to time, we did not have time to un-implement the modification to achieve better robustness (namely, in the real ARIAC competition, it is very likely that multiple instances of a part can appear on the conveyor and that we might need both or more of them in order to fulfill orders). So, the solution was effective, but only because our test case did not demand that level of robustness.

Intermittent Scoring

Once we ended the competition, sometimes ARIAC didn't detect that the parts were in their correct quadrants and gave us a score of 0 for those tasks. The recommended solution was to ensure that the AGV was at the warehouse and that the order was submitted.

Conclusion

We found the perception portion to be the most challenging overall. Because of this, we had the framework for detecting the parts and their positions by the end, but we did not finish it in time to integrate this framework with the MoveIt. We hardcoded the part positions in MoveIt as a proof-of-concept for testing the MoveIt portion, but this is certainly not ideal and would be something to fix in the future. We learned that integration of subsystems built independently can sometimes be more challenging than expected.

Another area of improvement would be our sensor usage. In the ARIAC competition, using an RGB for each bin cluster and tray area might be expensive in terms of cost. It may be possible to have an RGB sensor at the conveyor (since this is primarily where parts in the ARIAC competition spawn from) and move parts that spawn in the bins to the RGB with the floor robot in order to detect their type and color, for example. Additionally, it may be possible to use an RGB sensor for the trays instead of RGBD.

Overall, we were able to complete the task using MoveIt, the available sensors, and various ROS nodes. One of the main highlights of our implementation is utilizing both robots simultaneously due to utilizing two MoveIt nodes. This project enhanced our ability to break down complex systems into manageable subsections (e.g.: deciding how to partition the floor and ceiling robot tasks,), and also familiarized us with the tools and techniques used in industry for solving these kinds of problems (e.g., MoveIt, system architectures, etc.). While there were a lot of steps and we did not necessarily hit all of our marks perfectly, we believe that, if we were to tackle this problem a second time over, we would perform much better due to the knowledge we have acquired throughout this project and semester.

Bibliography

[1] *ARIAC Documentation — ARIAC Docs 2024.5.0 documentation*. (2024). Nist.gov; NIST. https://pages.nist.gov/ARIAC_docs/en/latest/index.html

[2] Video Demonstration — https://youtu.be/nRMoOkfuviY