Final Project

Group 1

Building a Manufacturing Robotic Software System

Students:

Ankur Chavan

Datta Lohith Gannavarapu

Shail Kiritkumar Shah

Vinay Krishna Bukka

Vishnu Mandala

Instructors:
Z. Kootbally, C. Schnellof

Semester:
Spring 2024

Course code:
ENPM663

June 13, 2024

Contents

1 Conventions

In this document, the following conventions are used:

- *File* is used to denote files.
- t *Topic* is used to denote topics.
- p *Parameter* is used to denote parameters.
- s *Service* is used to denote service name used.
- *Function* is used to denote functions or methods.
- *Package* is used to denote packages.
- *Folder* is used to denote folders used inside packages.
- *Attribute* is used to denote attributes.

is used to denote the bullet points.

is also used to denote the bullet points.

2 Introduction

The $ARIAC(AgileRoboticsforIndustrialAutomationCompetition)$, hosted by the National Institute of Standards and Technology (NIST), presents a rigorous platform for testing the agility and adaptability of robotic systems in dynamic manufacturing environments. This annual competition challenges participants to develop innovative solutions capable of performing a series of tasks, including pick-and-place operations, assembly, and kitting, within a simulated warehouse setting.

The essence of agility in ARIAC extends beyond the mere physical capabilities of robots; it encompasses their ability to efficiently navigate through uncertainties such as component failures, sensor glitches, and the presence of human operators. A successful solution must demonstrate robustness in handling such dynamic conditions while ensuring task completion within stipulated constraints.

Our project explores and addresses the core challenges posed by ARIAC, particularly focusing on the kitting task. Kitting involves the systematic gathering and arrangement of parts required for assembly processes. In our approach, we aim to develop a Competitor Control System (CCS) that performs the entire kitting process with precision and adaptability.

The simulated environment provided by ARIAC mimics an assembly line shop floor, featuring floor and ceiling robots equipped with UR10e arms. These robots serve as versatile tools for executing various tasks, including kitting. The kitting process involves handling four types of parts - Battery, Pump, Sensor, and Regulator - each available in five different colors. Kitting trays, distinguished by unique ARUCO fiducial markers, facilitate the organized arrangement of parts, ensuring efficient retrieval during assembly operations.

Our project addresses specific agility challenges inherent in the ARIAC competition:

1. **High-Priority Order**
2. **Insufficient Parts**
3. **Correct Gripper**
4. **Faulty Parts**
5. **Faulty Gripper**

By developing a robust robotics software system, our project aims to streamline the kitting process, ensuring seamless task execution while tackling the agility challenges in dynamic manufacturing environments. Through this endeavor, we seek to contribute to the advancement of agile robotics solutions, with potential applications across diverse industrial settings.

Dependencies: The following dependencies are used to complete the project and run the simulation:

- **VS Code**: VS Code is used as the IDE.
- **Python**: Python 3.8 is used as the programming language.
- **Ubuntu**: Ubuntu 20.04 LTS is used as the operating system.
- **ROS**: ROS Galactic is used as the middleware.
- **Gazebo**: Gazebo 11.0.0 is used as the simulator.

3 ARIAC Simulation Environment Overview

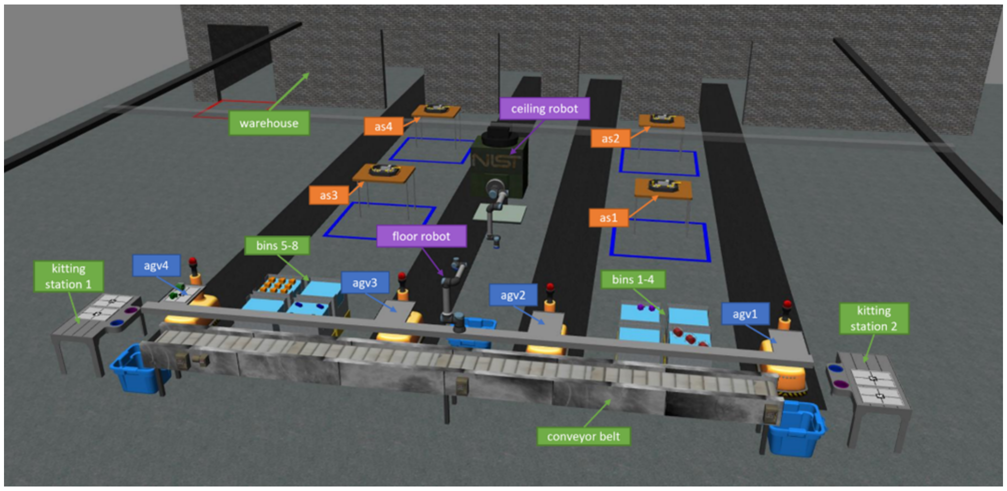


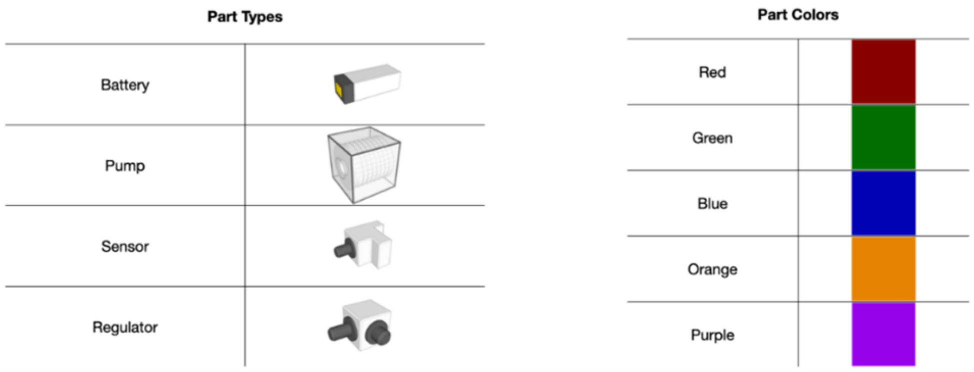
Figure 1: ARIAC Simulated Environment

The ARIAC competition's simulation environment as shown in *Fig*.1 emulates an order fulfillment work-cell, featuring:

- **Parts:** Four types with five colors, used for kitting and assembly.
- **Kit Trays:** Organize parts with quadrants and fiducial tags.
- **Robots:** Two UR10e robots with vacuum grippers for kitting and assembly.
- **AGVs:** Four automated guided vehicles for part transportation.
- **Part Locations:** Bins, conveyor belts, and AGVs.
- **Kit Tray/Tool Changer Station:** Facilitates gripper tool type changes.
- **Warehouse:** Processing area for AGVs.

- **Part Disposal Bins:** Allows disposal of faulty parts without penalty.

This dynamic environment challenges competitors to develop adaptable robotic systems for efficient task completion.



Part Types

Battery	
Pump	
Sensor	
Regulator	

Part Colors

Red	🟥
Green	🟩
Blue	🟦
Orange	🟧
Purple	🟪

Figure 2: Part Identification code

4 Architecture Overview

Our robotic system for the ARIAC competition adopts a Hybrid architecture, combining hierarchical and reactive approaches to efficiently execute kitting tasks in the simulated environment. This architecture ensures a structured flow of order execution while dynamically addressing agility challenges as they arise.

- **Perception Module:** Utilizes camera data to detect and identify parts, employing computer vision algorithms for accurate part recognition based on type, color, and location within the environment.
- **High Priority Order Module:** Formulates plans for kitting tasks based on received orders and available parts. It employs task prioritization algorithms to handle high-priority orders, ensuring their timely execution.
- **Insufficient Parts Module:** Identifies available parts to fulfill orders and submits orders accordingly, ensuring proactive handling of insufficient parts scenarios.
- **Gripper Tool Changer:** Manages gripper tool transitions for the floor robot, enabling it to pick up trays and parts seamlessly during kitting operations.
- **Fault Detection and Handling:** Detects faulty parts within orders and replaces them with suitable alternatives to complete the task successfully, ensuring robustness in the face of component failures.
- **Faulty Gripper Detection:** Identifies instances where parts are dropped due to faulty grippers during kitting tasks and compensates by utilizing available similar parts to complete the order.

By integrating following interconnected components as shown in $Fig.3$ within our robotic system, we aim to achieve efficient and agile execution of kitting tasks, addressing both planned order flow and unforeseen challenges that may arise during execution.

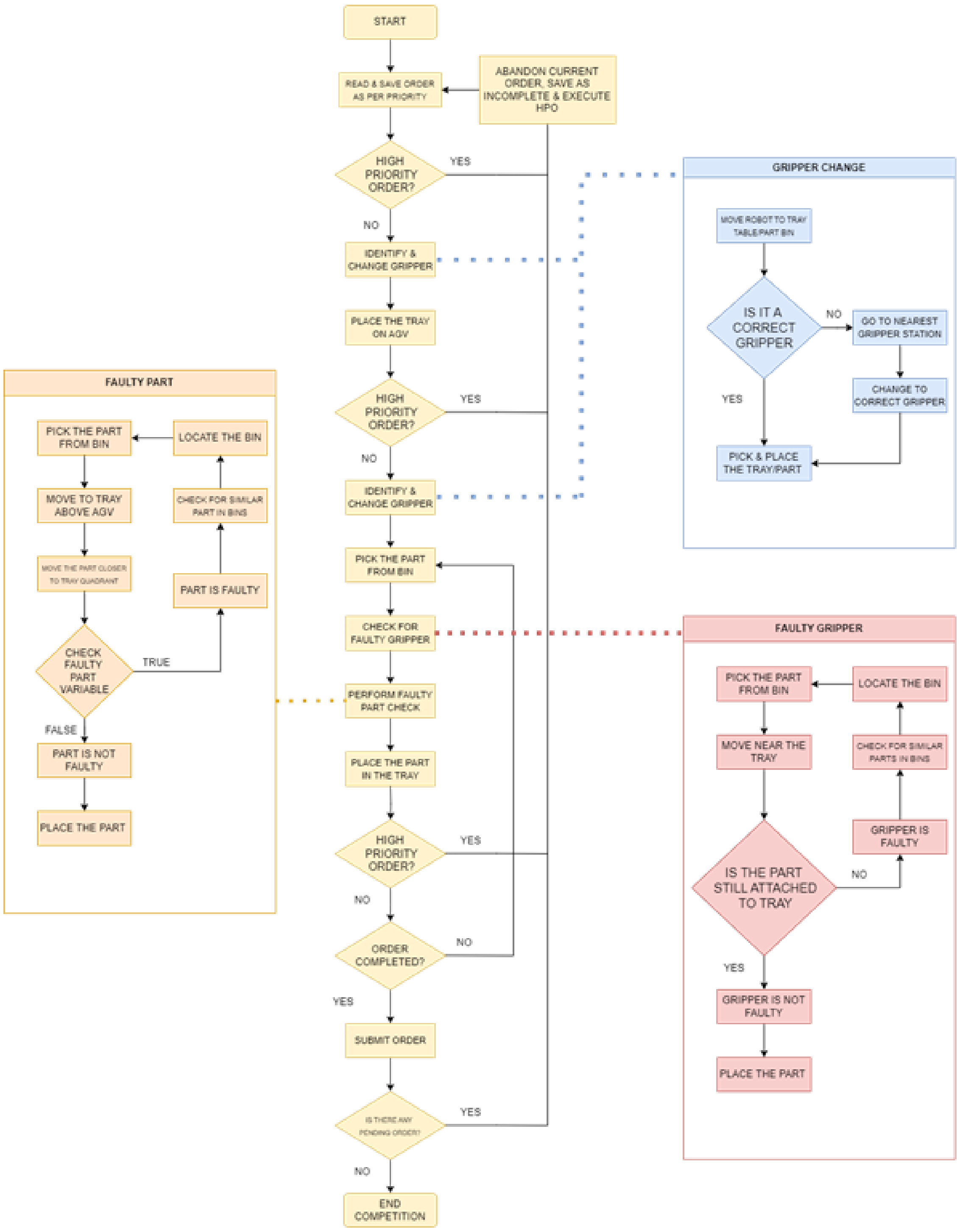


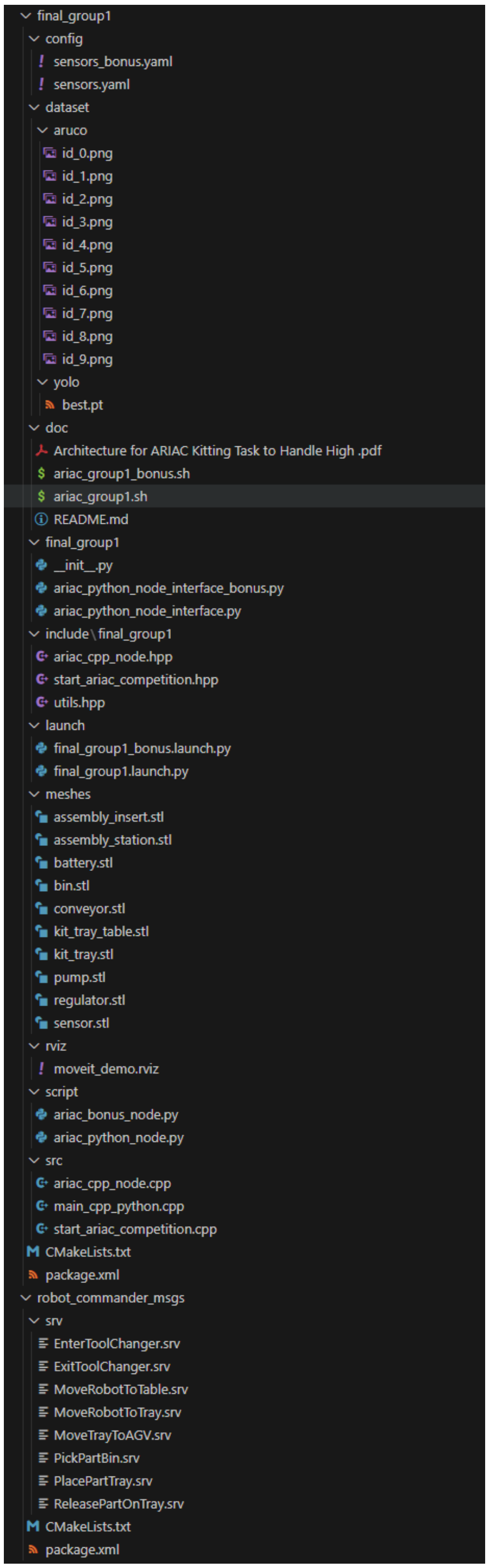
Figure 3: Architecture Flow Chart

5 Package Structure

Our software development for the ARIAC competition utilizes both Python and C++, encapsulated within a ROS2 package structure. This structure ensures organized management of code and resources for efficient development and deployment. The package also includes robot_commander_msgs which contains the custom services created to have efficient communication between C++ and Python nodes.

- config: Stores YAML files containing sensor information to be spawned at the start of the ARIAC simulation.
- final_group1: This folder contains all the python and C++ code for ARIAC.
- include: Contains C++ header files for the CCS (Competitor Control System) and OrderData class.
- launch: Hosts the launch file used to initiate the package.
- src: Houses C++ scripts responsible for implementing the functionality of the CCS.
- script: Stores Python scripts for the CCS.
- rviz: Holds the Rviz setup configuration, facilitating visualization if started from the launch file.
- CMakeLists.txt Configuration file specifying build instructions for the package.
- package.xml Metadata file providing information about the package and its dependencies.

The structured shown in $Fig.4$ organization enables seamless integration of Python and C++ components within our ROS2 package, streamlining development and ensuring robust functionality for executing kitting tasks in the ARIAC simulation environment.



```
final_group1
  config
    sensors_bonus.yaml
    sensors.yaml
  dataset
    aruco
      id_0.png
      id_1.png
      id_2.png
      id_3.png
      id_4.png
      id_5.png
      id_6.png
      id_7.png
      id_8.png
      id_9.png
    yolo
      best.pt
  doc
    Architecture for ARIAC Kitting Task to Handle High .pdf
    ariac_group1_bonus.sh
    ariac_group1.sh
    README.md
  final_group1
    __init__.py
    ariac_python_node_interface_bonus.py
    ariac_python_node_interface.py
  include\final_group1
    ariac_cpp_node.hpp
    start_ariac_competition.hpp
    utils.hpp
  launch
    final_group1_bonus.launch.py
    final_group1.launch.py
  meshes
    assembly_insert.stl
    assembly_station.stl
    battery.stl
    bin.stl
    conveyor.stl
    kit_tray_table.stl
    kit_tray.stl
    pump.stl
    regulator.stl
    sensor.stl
  rviz
    moveit_demo.rviz
  script
    ariac_bonus_node.py
    ariac_python_node.py
  src
    ariac_cpp_node.cpp
    main_cpp_python.cpp
    start_ariac_competition.cpp
  CMakeLists.txt
  package.xml
robot_commander_msgs
  srv
    EnterToolChanger.srv
    ExitToolChanger.srv
    MoveRobotToTable.srv
    MoveRobotToTray.srv
    MoveTrayToAGV.srv
    PickPartBin.srv
    PlacePartTray.srv
    ReleasePartOnTray.srv
  CMakeLists.txt
  package.xml
```

Figure 4: Package Structure

6 UML Diagram

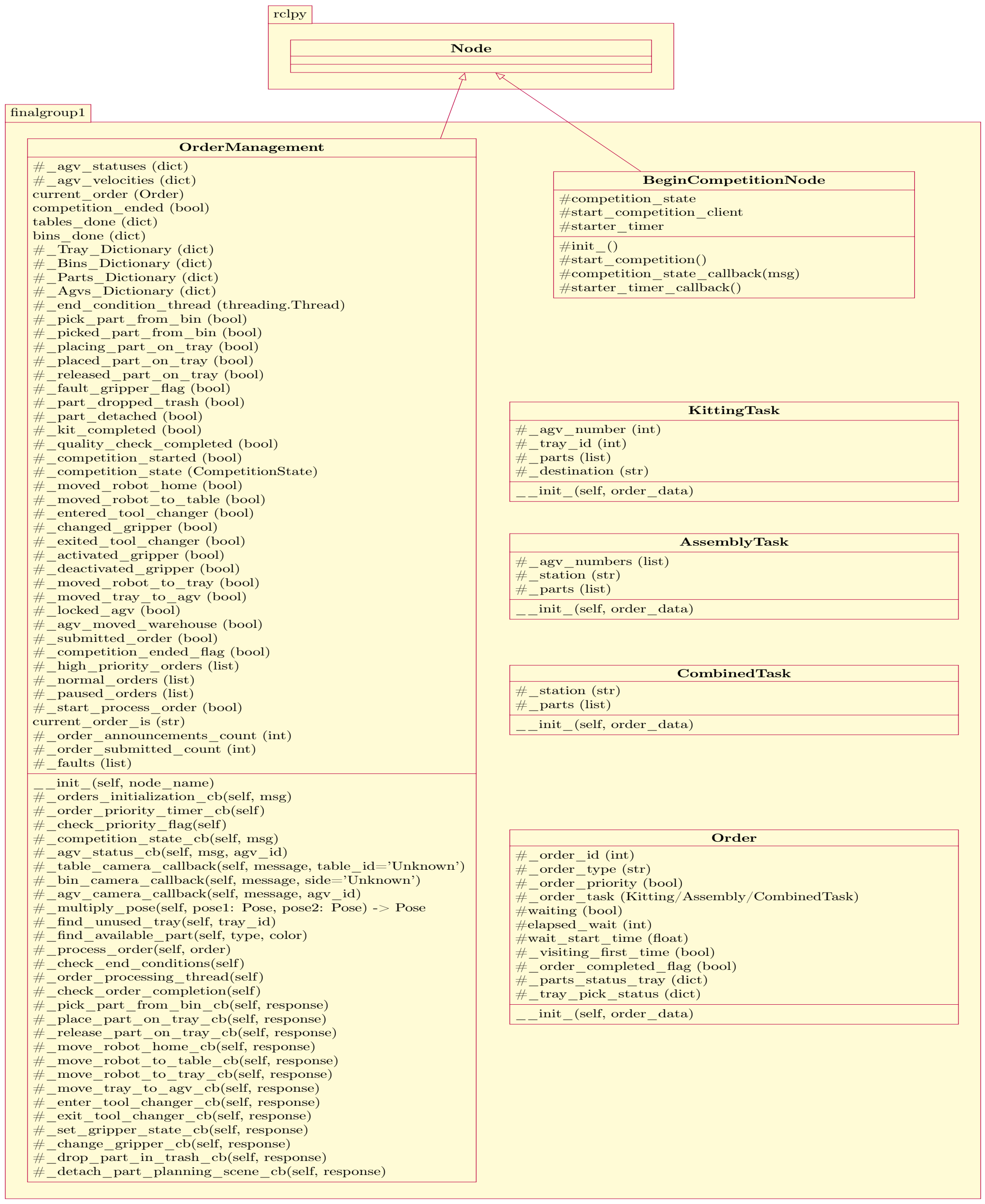

rclpy
Node
finalgroup1
OrderManagement
#_agv_statuses (dict)
#_agv_velocities (dict)
current_order (Order)
competition_ended (bool)
tables_done (dict)
bins_done (dict)
#_Tray_Dictionary (dict)
#_Bins_Dictionary (dict)
#_Parts_Dictionary (dict)
#_Agvs_Dictionary (dict)
#_end_condition_thread (threading.Thread)
#_pick_part_from_bin (bool)
#_picked_part_from_bin (bool)
#_placing_part_on_tray (bool)
#_placed_part_on_tray (bool)
#_released_part_on_tray (bool)
#_fault_gripper_flag (bool)
#_part_dropped_trash (bool)
#_part_detached (bool)
#_kit_completed (bool)
#_quality_check_completed (bool)
#_competition_started (bool)
#_competition_state (CompetitionState)
#_moved_robot_home (bool)
#_moved_robot_to_table (bool)
#_entered_tool_changer (bool)
#_changed_gripper (bool)
#_exited_tool_changer (bool)
#_activated_gripper (bool)
#_deactivated_gripper (bool)
#_moved_robot_to_tray (bool)
#_moved_tray_to_agv (bool)
#_locked_agv (bool)
#_agv_moved_warehouse (bool)
#_submitted_order (bool)
#_competition_ended_flag (bool)
#_high_priority_orders (list)
#_normal_orders (list)
#_paused_orders (list)
#_start_process_order (bool)
current_order_is (str)
#_order_announcements_count (int)
#_order_submitted_count (int)
#_faults (list)
__init_(self, node_name)
#_orders_initialization_cb(self, msg)
#_order_priority_timer_cb(self)
#_check_priority_flag(self)
#_competition_state_cb(self, msg)
#_agv_status_cb(self, msg, agv_id)
#_table_camera_callback(self, message, table_id='Unknown')
#_bin_camera_callback(self, message, side='Unknown')
#_agv_camera_callback(self, message, agv_id)
#_multiply_pose(self, pose1: Pose, pose2: Pose) -> Pose
#_find_unused_tray(self, tray_id)
#_find_available_part(self, type, color)
#_process_order(self, order)
#_check_end_conditions(self)
#_order_processing_thread(self)
#_check_order_completion(self)
#_pick_part_from_bin_cb(self, response)
#_place_part_on_tray_cb(self, response)
#_release_part_on_tray_cb(self, response)
#_move_robot_home_cb(self, response)
#_move_robot_to_table_cb(self, response)
#_move_robot_to_tray_cb(self, response)
#_move_tray_to_agv_cb(self, response)
#_enter_tool_changer_cb(self, response)
#_exit_tool_changer_cb(self, response)
#_set_gripper_state_cb(self, response)
#_change_gripper_cb(self, response)
#_drop_part_in_trash_cb(self, response)
#_detach_part_planning_scene_cb(self, response)
BeginCompetitionNode
#competition_state
#start_competition_client
#starter_timer
#init_()
#start_competition()
#competition_state_callback(msg)
#starter_timer_callback()
KittingTask
#_agv_number (int)
#_tray_id (int)
#_parts (list)
#_destination (str)
__init_(self, order_data)
AssemblyTask
#_agv_numbers (list)
#_station (str)
#_parts (list)
__init_(self, order_data)
CombinedTask
#_station (str)
#_parts (list)
__init_(self, order_data)
Order
#_order_id (int)
#_order_type (str)
#_order_priority (bool)
#_order_task (Kitting/Assembly/CombinedTask)
#waiting (bool)
#elapsed_wait (int)
#wait_start_time (float)
#_visiting_first_time (bool)
#_order_completed_flag (bool)
#_parts_status_tray (dict)
#_tray_pick_status (dict)
__init_(self, order_data)

Figure 5: Python UML Diagram

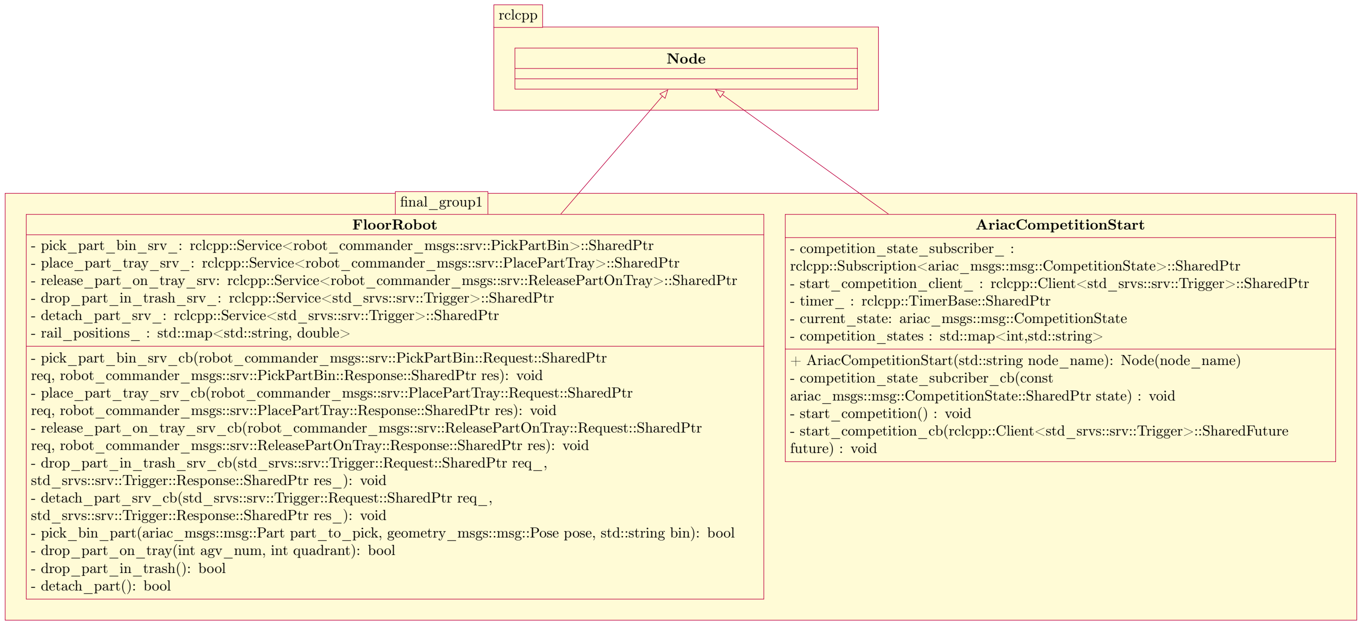

rclcpp
Node
final_group1
FloorRobot
- pick_part_bin_srv_: rclcpp::Service<robot_commander_msgs::srv::PickPartBin>::SharedPtr
- place_part_tray_srv_: rclcpp::Service<robot_commander_msgs::srv::PlacePartTray>::SharedPtr
- release_part_on_tray_srv: rclcpp::Service<robot_commander_msgs::srv::ReleasePartOnTray>::SharedPtr
- drop_part_in_trash_srv_: rclcpp::Service<std_srvs::srv::Trigger>::SharedPtr
- detach_part_srv_: rclcpp::Service<std_srvs::srv::Trigger>::SharedPtr
- rail_positions_ : std::map<std::string, double>
- pick_part_bin_srv_cb(robot_commander_msgs::srv::PickPartBin::Request::SharedPtr req, robot_commander_msgs::srv::PickPartBin::Response::SharedPtr res): void
- place_part_tray_srv_cb(robot_commander_msgs::srv::PlacePartTray::Request::SharedPtr req, robot_commander_msgs::srv::PlacePartTray::Response::SharedPtr res): void
- release_part_on_tray_srv_cb(robot_commander_msgs::srv::ReleasePartOnTray::Request::SharedPtr req, robot_commander_msgs::srv::ReleasePartOnTray::Response::SharedPtr res): void
- drop_part_in_trash_srv_cb(std_srvs::srv::Trigger::Request::SharedPtr req_, std_srvs::srv::Trigger::Response::SharedPtr res_): void
- detach_part_srv_cb(std_srvs::srv::Trigger::Request::SharedPtr req_, std_srvs::srv::Trigger::Response::SharedPtr res_): void
- pick_bin_part(ariac_msgs::msg::Part part_to_pick, geometry_msgs::msg::Pose pose, std::string bin): bool
- drop_part_on_tray(int agv_num, int quadrant): bool
- drop_part_in_trash(): bool
- detach_part(): bool
AriacCompetitionStart
- competition_state_subscriber_ : rclcpp::Subscription<ariac_msgs::msg::CompetitionState>::SharedPtr
- start_competition_client_ : rclcpp::Client<std_srvs::srv::Trigger>::SharedPtr
- timer_ : rclcpp::TimerBase::SharedPtr
- current_state: ariac_msgs::msg::CompetitionState
- competition_states : std::map<int,std::string>
+ AriacCompetitionStart(std::string node_name): Node(node_name)
- competition_state_subcriber_cb(const ariac_msgs::msg::CompetitionState::SharedPtr state) : void
- start_competition() : void
- start_competition_cb(rclcpp::Client<std_srvs::srv::Trigger>::SharedFuture future) : void

Figure 6: C++ UML Diagram

7 Order Handling

In our approach, the efficient handling of orders plays a pivotal role in executing the kitting tasks within the ARIAC competition framework. When an order is received via the ROS2 topic t /ariac/orders, it undergoes processing within the Order Handling module.

Upon reception, the Orders class captures essential order details, including the order's ID, type, and priority. These details are important for subsequent decision-making processes within our robotic system.

To optimize order execution, we categorize orders into low-priority and high-priority lists. This categorization enables us to streamline the execution flow, ensuring that high-priority orders receive prompt attention while maintaining a structured workflow for low-priority tasks.

Simultaneously, we initiate an order processing thread to handle the execution of queued orders efficiently. This threaded approach enhances system responsiveness and throughput, enabling swift and coordinated execution of kitting tasks based on order priorities. As soon as we initiate the processing of an order, we store every detail of that order within the same order object. This includes information like part pose, orientation, and the status of the tray, as well as part placements. This approach enhances the robustness of our programming and ensures that all details of a resumed order are retained at any point during execution.

By implementing this robust order handling mechanism, our system effectively manages incoming orders, prioritizes tasks based on urgency, and ensures timely and efficient execution of kitting operations within the ARIAC simulation environment.

8 Perceiving the Environment

8.1 Sensor Placement

In our robotic system, achieving an effective perception of the environment relies on utilizing appropriate sensors to gather important information essential for efficient task execution and adaptability to dynamic environmental changes. To fulfill this requirement, we have strategically deployed a combination of sensors tailored to the specific needs of our system.

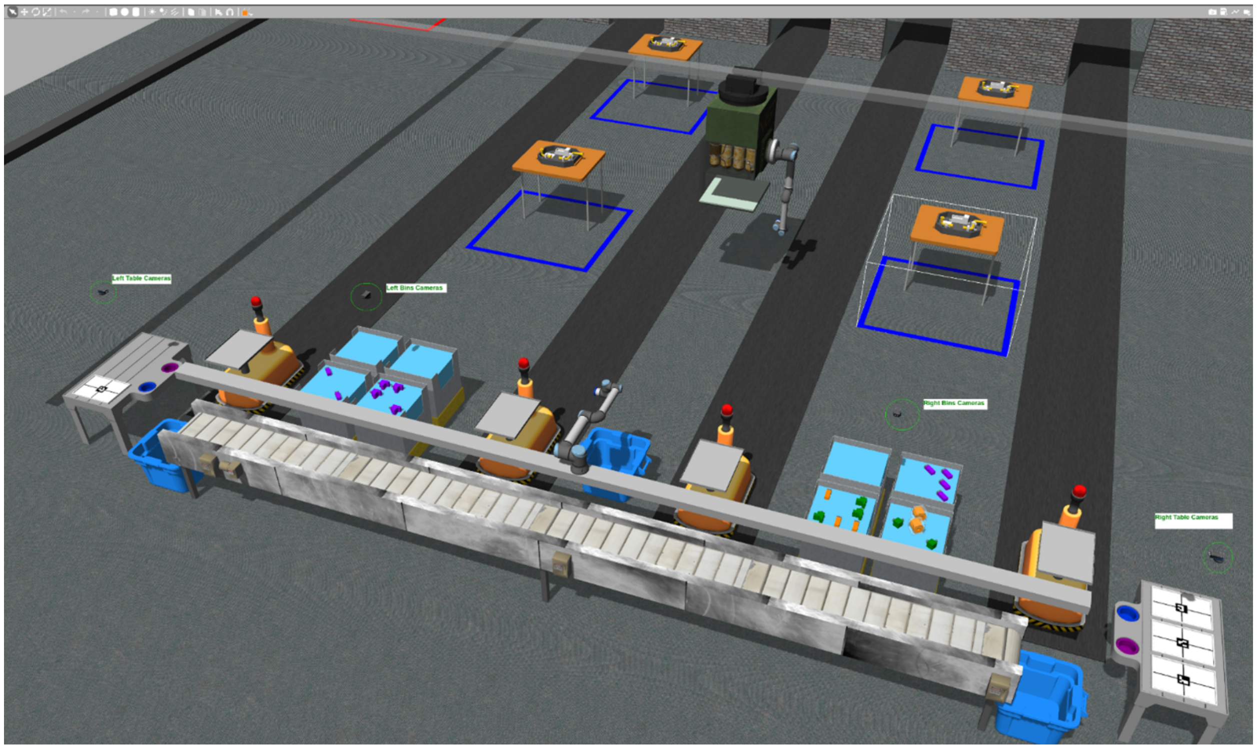


Figure 7: Sensor placement in simulated environment

Our sensor setup consists of four basic logic cameras (BLC) and four RGB cameras, strategically positioned to capture comprehensive environmental data. Specifically, we have deployed:

- **BLC and RGB Cameras Above Kitting Stations:** Two sets of BLC and RGB cameras positioned above the kitting stations. These cameras are tasked with locating and identifying trays and ARUCO markers, crucial for facilitating efficient tray manipulation and identification within the kitting process.
- **BLC and RGB Cameras Above Bins:** Additionally, we have deployed two sets of BLC and RGB cameras positioned above the bins. These cameras are instrumental in locating and identifying parts, enabling accurate part retrieval and manipulation during task execution.

By strategically placing these cameras in key areas of the environment, our system can effectively perceive and interpret critical information necessary for seamless task execution. This sensor arrangement enhances our system's adaptability and responsiveness, empowering it to efficiently navigate and operate within the dynamic ARIAC competition environment.

8.2 Identifying Parts Location, Orientation, and Type

In our system, we have developed a robust pipeline dedicated to locating parts, determining their orientation, and identifying their type and color. Leveraging a combination of Basic Logical Cameras (BLC) and RGB cameras, our pipeline ensures accurate and comprehensive part identification.

Using BLC, we extract precise pose and orientation information of the parts within the environment. This data serves as a crucial foundation for subsequent processing steps.

To identify part type and color, we employ a YOLO (You Only Look Once) model trained specifically for this purpose. By analyzing RGB camera images, the YOLO model accurately identifies the type and color of each part present in the environment. The integration of BLC-derived pose and orientation data with YOLO-based part identification results in a comprehensive part information format, facilitating efficient part handling and manipulation.

- **Training:**
  - **Image Collection:** The initial step involved capturing a series of images showcasing various parts. Each part was photographed in multiple orientations and positions to ensure a comprehensive dataset representing real-world scenarios.
  - **Edge Conversion:** To enhance the model's ability to focus on the shape and outline of the parts rather than textural details, each image was converted to an edge representation. This was accomplished using edge detection techniques which emphasize boundaries and contours of objects within an image.
  - **Annotation:** Utilizing Roboflow, an online tool for image annotation, each edge-enhanced image was manually annotated to label the parts. This process involved drawing bounding boxes around each part and assigning a label that identifies the type of part.
  - **Training the Model:** YOLOv8 was chosen for its efficiency and accuracy in real-time object detection tasks. It is particularly well-suited for scenarios requiring the detection of small to medium-sized objects, such as machine parts.
- **Recognition:**
  - **Part Recognition:** The trained YOLOv8 model's weights file was loaded into the detection system. This system uses RGB cameras to capture images of the environment where parts need to be identified. For each new image captured, the model predicts bounding boxes and class labels indicating the presence and type of parts observed.
  - **Color Detection:** Once a part is detected, the corresponding area within the bounding box is cropped from the original RGB image. The cropped image is converted into the HSV color space, which is more effective for color segmentation compared to RGB. Specific HSV ranges predefined for each part type ($Fig.2$) are used to determine the color of the part.
  - **Sorting and Matching** The detected parts are then sorted based on their spatial coordinates and matched against a list of part poses received from a Basic Logical Camera message. This integration ensures that the parts are not only recognized but also correctly aligned with their expected positions within the ARIAC Environment.

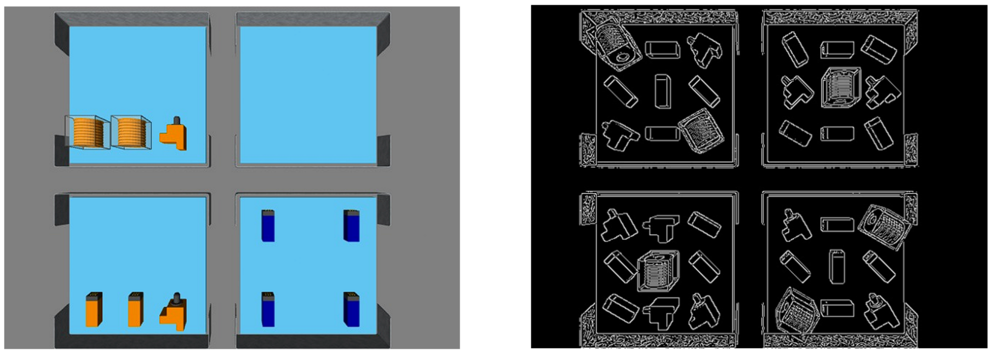


Figure 8: Part Identification using YOLO(Left - RGB Image, Right - Image used for Training)

8.3 Tray Detection

Trays play a pivotal role in fulfilling kitting orders, hence accurate identification of the correct tray for each order is important. To achieve this, we utilize RGB cameras in conjunction with ARUCO markers.

By leveraging RGB camera data and ARUCO markers, our system accurately detects and identifies trays within the environment. This approach ensures precise tray detection, enabling seamless integration into the kitting process for efficient order fulfillment.

- **Image Acquisition and Preprocessing:** Upon receiving an image message from the camera, the image is converted from its ROS message format to a OpenCV-compatible format using the CvBridge library. Any errors occurring during this conversion process are logged for debugging purposes.

  The acquired image is then cropped to focus on the region of interest where the ArUco markers are expected to appear. This cropping enhances efficiency by reducing the area of the image that needs to be processed. Additionally, the cropped image is converted from color (BGR) to grayscale, which simplifies subsequent edge detection operations.

- **Edge Detection and Template Matching:** Edge detection is performed on the grayscale image using the Canny edge detection algorithm. Template matching is employed to search for ArUco markers within the edge-enhanced image. This technique involves comparing a template image of an ArUco marker with subregions of the edge image to identify potential matches. The template image is resized to match the expected size of ArUco markers in the scene and also undergoes edge detection for consistency.

  For each template image representing a unique ArUco marker, the OpenCV function `matchTemplate()` computes a similarity score indicating the degree of resemblance between the template and the image region. A predefined threshold is used to determine whether a match is significant enough to be considered valid.

- **Sorting and Matching** Detected ArUco markers are visualized on the original image by drawing bounding boxes around them. Additionally, the IDs of the detected markers along with their spatial coordinates are stored in a set for further processing. These IDs provide unique identifiers for differentiating between ArUco markers and are crucial for subsequent tasks such as pose estimation.

  Finally, the detected ArUco markers are sorted and are matched with the tray poses which we get from BLC.

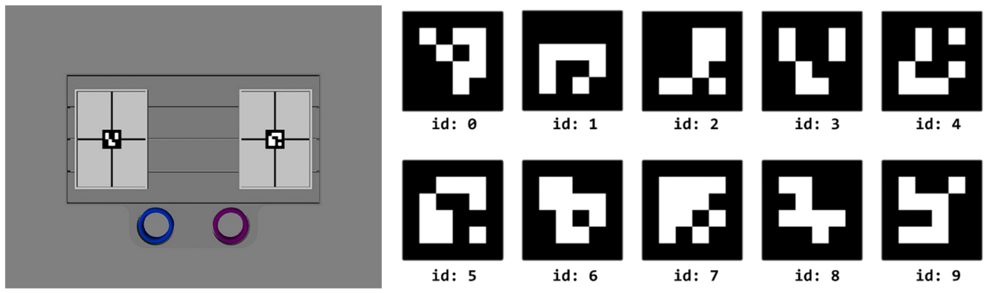


Figure 9: Tray Detection with corresponding ARUCO markers

9 Agility Challenges

In the context of the ARIAC competition, agility refers to a robot's ability to swiftly and effectively adapt to changing conditions and requirements within a dynamic manufacturing environment. It encompasses several key aspects:

- **Adaptability:** Robots must be able to adjust their behavior and decision-making processes in response to unforeseen events or changes in the environment. This includes adapting to variations in part availability, unexpected obstacles, or alterations in task priorities.

- **Efficiency:** Agility also involves maximizing the efficiency of robot operations, ensuring tasks are completed promptly with minimal resource usage. This includes optimizing task scheduling, motion planning, and resource allocation to achieve the desired outcomes efficiently.

- **Autonomy:** Robots should exhibit a high degree of autonomy, capable of making independent decisions and taking appropriate actions to fulfill assigned tasks. This autonomy enables robots to operate effectively in dynamic environments without constant human intervention.

Our Scope: Kitting Task

Our project scope was limited to developing a software system dedicated to completing the kitting task within the ARIAC simulation environment. This task involves organizing and gathering parts required for assembly or production processes, with a focus on efficiency and accuracy.

Agility Challenges to Handle: As part of our project, we addressed the following agility challenges set forth by ARIAC:

- **High-Priority Order:** Ensuring timely execution of high-priority orders before lower-priority tasks, optimizing task scheduling and resource allocation accordingly.
- **Insufficient Parts:** Proactively identifying and managing scenarios where required parts are insufficient to complete orders, submitting orders with available parts while notifying relevant stakeholders.
- **Correct Gripper:** Ensuring the correct gripper tool is selected for handling trays and parts during the kitting process, facilitating seamless manipulation and placement.
- **Faulty Parts:** Detecting and managing faulty parts within orders, replacing them with functional alternatives to ensure successful task completion.
- **Faulty Gripper:** Identifying instances of gripper malfunction during task execution and implementing corrective measures to mitigate the impact on order fulfillment.

By effectively addressing these agility challenges within our software system, we aimed to demonstrate the adaptability and efficiency of our solution in navigating the complexities of the ARIAC competition environment.

9.1 High-Priority Order

The High Priority Order challenge assesses the Competitor Control System's (CCS) capability to prioritize high-priority orders over regular-priority ones. The objective is to seamlessly switch task execution when a high-priority order is received, ensuring efficient order fulfillment.This is also illustrated in flowchart [10]

In our approach to this challenge, we devised an architecture that facilitates the execution of normal orders while monitoring incoming orders and their respective priorities. Upon receiving an order, its information, including priority, is stored for future reference. This priority information serves as a crucial determinant for task prioritization.

To effectively handle high-priority orders, our architecture incorporates a dynamic task-switching mechanism. When a high-priority order is detected, the ongoing task is paused, allowing the high-priority order to be executed promptly. Once the high-priority task is completed, the system seamlessly resumes the previously paused task, ensuring uninterrupted progress in order fulfillment.

Note: By effectively leveraging programming techniques such as classes and objects, we've been able to store order details along with their status. This approach has enabled us to seamlessly transition between orders without relying heavily on global lists or dictionaries. It enhances the modularity and readability of our code while also improving its maintainability and flexibility.

Throughout the order execution process, we implement three distinct checkpoints to verify the presence of high-priority orders:

- **Upon Order Reception:** We immediately check for high-priority orders upon receiving and saving the order information.

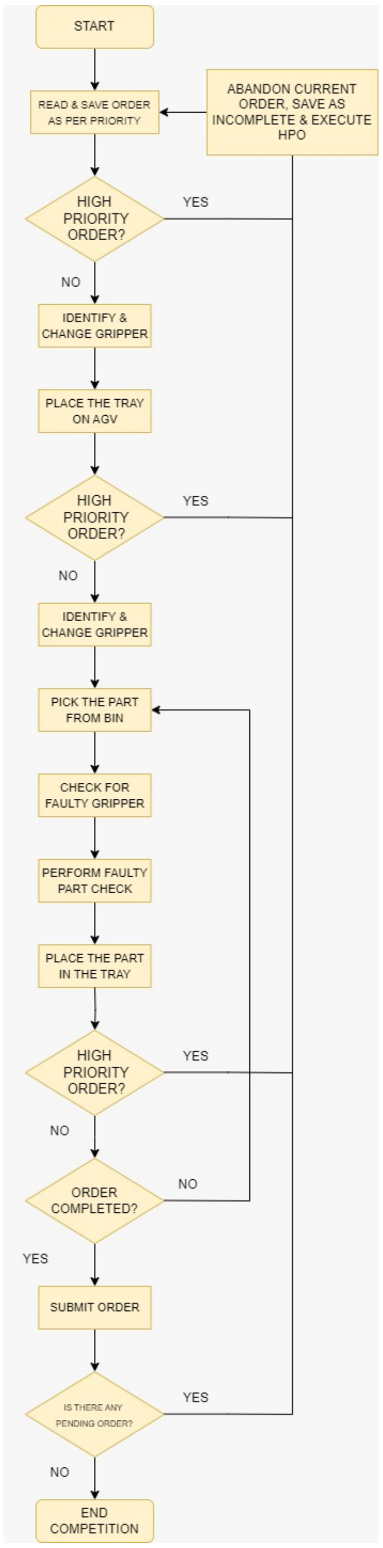

START
READ & SAVE ORDER AS PER PRIORITY
ABANDON CURRENT ORDER, SAVE AS INCOMPLETE & EXECUTE HPO
HIGH PRIORITY ORDER?
YES
NO
IDENTIFY & CHANGE GRIPPER
PLACE THE TRAY ON AGV
HIGH PRIORITY ORDER?
YES
NO
IDENTIFY & CHANGE GRIPPER
PICK THE PART FROM BIN
CHECK FOR FAULTY GRIPPER
PERFORM FAULTY PART CHECK
PLACE THE PART IN THE TRAY
HIGH PRIORITY ORDER?
YES
NO
ORDER COMPLETED?
NO
YES
SUBMIT ORDER
IS THERE ANY PENDING ORDER?
YES
NO
END COMPETITION

Figure 10: High Priority Order Flow Chart

- **After Tray Placement on AGV:** Following the placement of trays on AGVs, we conduct another check to ascertain if any new high-priority orders have been received.
- **After Part Placement and Quality Check:** After each part is placed in the tray and subjected to quality checks for faulty parts, we perform a final check for high-priority orders.

By incorporating these three checkpoints, our system ensures timely detection and execution of high-priority orders while maintaining the integrity of task prioritization. This approach enables efficient handling of high-priority orders without compromising on the execution of regular-priority tasks, thereby demonstrating the agility and adaptability of our system within the ARIAC competition framework.

9.2 Insufficient Parts

The Insufficient Parts challenge simulates scenarios where the work cell lacks the necessary parts to complete one or multiple orders, presenting a test of the Competitor Control System's (CCS) ability to identify and address such situations.

In our system, upon receiving an order, we record part information, including location, type, and color. Each order specifies the required quantity and type of parts needed for fulfillment. Before order processing, we conduct a check of our bins dictionary to assess the availability of required parts. If all parts are present, we proceed to execute the complete order and submit it accordingly. However, if any required parts are missing, we proceed to place the available parts in the tray and submit the incomplete order.

Insufficient parts may arise due to various factors, including faulty parts or gripper malfunctions. To address these potential issues, we implement robust error-handling mechanisms during part placement. Upon placing parts in the tray, we perform quality checks to identify and discard any faulty parts. Subsequently, we update our bins dictionary to reflect the availability of remaining parts and check for similar parts to complete the order. If suitable replacements are found, we proceed to complete and submit the order. In cases where replacements are unavailable, we submit the incomplete order accordingly.

Similarly, issues stemming from faulty grippers are addressed during the pick-and-place process. If the gripper malfunctions and drops parts, we promptly update our bins dictionary, search for available replacements, and proceed to complete the order if feasible.

By implementing these comprehensive measures, our system effectively manages scenarios of insufficient parts, ensuring timely identification, and appropriate handling to maintain the integrity of order fulfillment within the ARIAC competition environment.

9.3 Gripper Change

While gripper change itself may not be classified as an agility challenge, its implementation is crucial for seamless task execution within our system. In our kitting task workflow, the need to handle different types of objects—trays and parts—necessitates the utilization of distinct grippers tailored to each object type. As part of our system's logic, we developed a robust mechanism to facilitate gripper changes at key stages of the workflow.This is also illustrated in flowchart [11]

Our kitting task workflow involves two primary instances where gripper change is required:

1. Tray pick and place on the AGV.
2. Part pick and place from the bins onto the tray.

To address gripper changes effectively, we implemented a logic based on the status of the floor robot gripper **t** /ariac/floor_robot_gripper_state. This logic ensures that the appropriate gripper is used for each pick and place action.

Before initiating any pick and place operation, our system checks the status of the floor robot gripper to determine if the correct gripper is available for the upcoming task. If the required gripper is already in place, the system proceeds with the pick and place action as planned.

However, if the correct gripper is not available, the system identifies the nearest gripper station and initiates the gripper change process. Once the gripper change is completed, the system proceeds to perform the pick and place action using the newly equipped gripper.

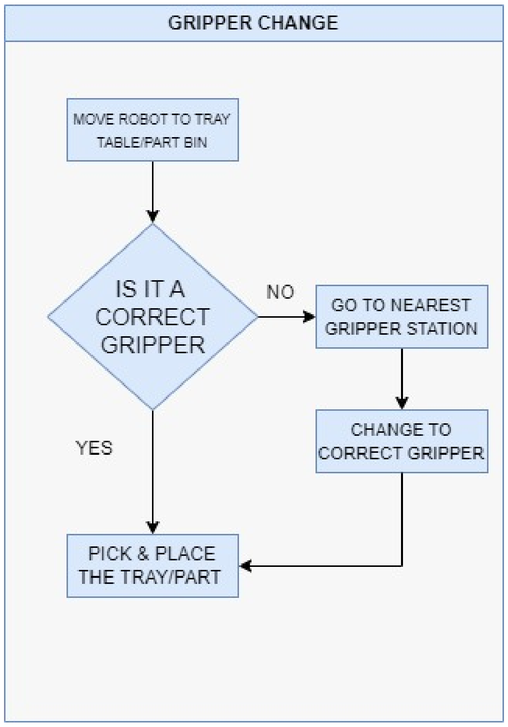

GRIPPER CHANGE
MOVE ROBOT TO TRAY TABLE/PART BIN
IS IT A CORRECT GRIPPER
NO
GO TO NEAREST GRIPPER STATION
CHANGE TO CORRECT GRIPPER
YES
PICK & PLACE THE TRAY/PART

Figure 11: Gripper Change Flow Chart

By utilizing the t /ariac/floor_robot_gripper_state status as a guide, our system effectively manages gripper changes during the kitting task workflow, ensuring seamless execution of pick and place operations for trays and parts. This approach enhances the adaptability and efficiency of our system, enabling it to navigate dynamic manufacturing environments with agility and precision.

9.4 Faulty Parts

The Faulty Parts challenge assesses the Competitor Control System's (CCS) capability to detect and replace faulty parts during task execution, ensuring the delivery of orders free from defective components.This is also illustrated in flowchart [12]

During the execution of the kitting task, our system encounters scenarios where the parts picked for order fulfillment may be faulty. To address this, our system incorporates a quality check mechanism to identify and rectify faulty parts.

Upon picking a part for placement on the tray, our system initiates the quality check process. The part is positioned above the respective tray on the AGV and moved closer to the designated quadrant within the tray. We then invoke the S /ariac/perform_quality_check to perform the quality check. The quality control sensor returns a all_passed status as true only when all conditions are met:

- All parts in the kitting tray are not faulty.
- All parts are present in the kitting tray (no missing parts).
- All parts have the correct orientation (no flipped parts).

- All parts are of the correct type.
- All parts are of the correct color.

Upon detecting a faulty part, our system promptly discards the defective component and proceeds to replace it with a new part of the same type and color if available in the bins dictionary. If a suitable replacement is found, the order is completed and submitted. However, if replacements are unavailable, the system submits the incomplete order accordingly.

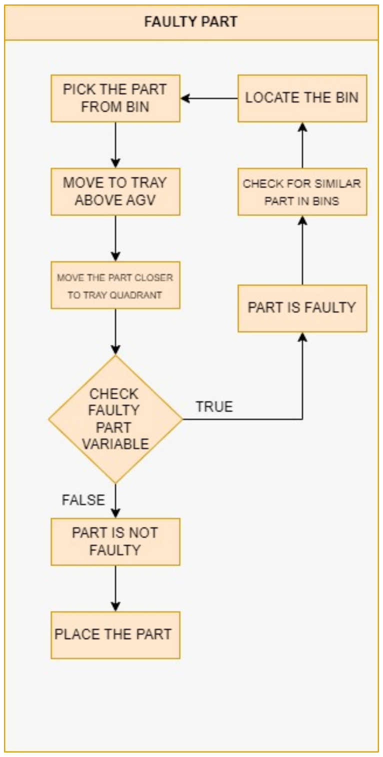


Figure 12: Faulty Parts Flow Chart

This process is repeated for each part within the order, ensuring that orders are fulfilled with high-quality components, free from defects. By effectively handling faulty parts challenges, our system upholds the integrity of order fulfillment within the ARIAC competition environment, showcasing its adaptability and reliability in dynamic manufacturing scenarios.

9.5 Faulty Gripper

The Faulty Gripper challenge evaluates the Competitor Control System's (CCS) ability to detect instances where a part has been dropped from the gripper during task execution and to subsequently pick and place a replacement part of the same type and color.

During the execution of the kitting task, our system may encounter scenarios where a faulty gripper causes the robot to inadvertently drop parts during pick-and-place actions. To address this challenge, we implemented a mechanism to detect dropped parts and seamlessly replace them with suitable alternatives.

Our approach to handling the Faulty Gripper challenge involves monitoring the gripper status to determine if a part has been dropped. As the robot picks a part from the bin and plans its path for placement on the tray, we subscribe to the *t* /ariac/floor_robot_gripper_state topic to check the status of the gripper. This is also illustrated in flowchart [13]

If the gripper is no longer holding the part, indicating that it has been dropped, our system promptly initiates a process to replace the dropped part. We search for a similar part in the bin and perform the faulty gripper check again to ensure the replacement part is securely held by the gripper.

Conversely, if the gripper is still holding the part, we proceed to place the part on the tray as planned.

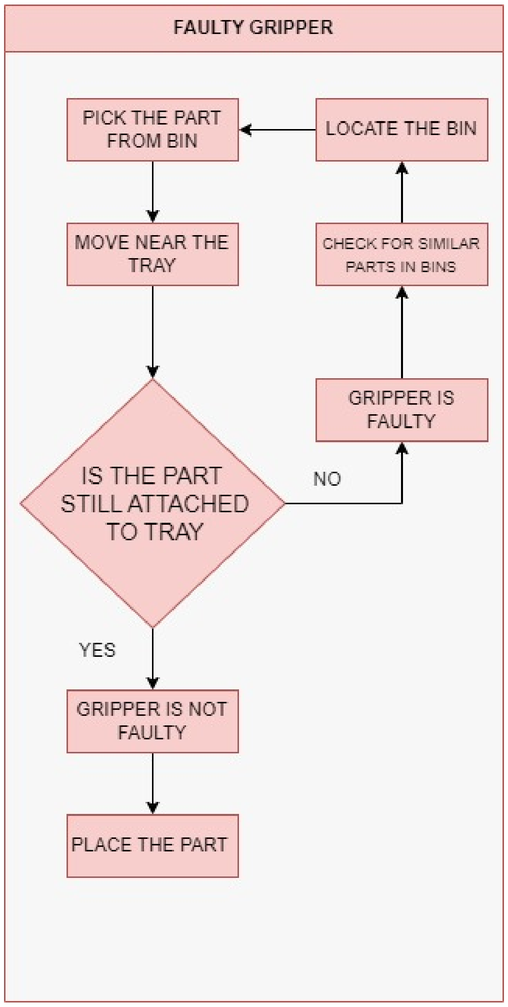


Figure 13: Faulty Gripper Flow Chart

By continuously monitoring the gripper status and implementing a loop to handle dropped parts, our system effectively manages instances of faulty gripper behavior during the kitting task. This ensures the accurate placement of parts on the tray and prevents the submission of incomplete orders, thereby demonstrating the adaptability and reliability of our system within the ARIAC competition environment.

10 Difficulties Faced

Throughout the development and implementation of our robotic system for the ARIAC competition, we encountered various challenges that required innovative solutions to overcome. While these difficulties presented formidable obstacles, our team successfully navigated through them, ultimately delivering a robust system capable of performing the kitting task while effectively handling agility challenges. Some of the key difficulties we faced include:

1. **Parts Crashing with Tray on AGV:** We encountered issues with parts crashing into the tray on

the AGV due to inconsistent drop heights. To resolve this, we adjusted the drop heights based on the type of part being handled, ensuring smoother operation and preventing collisions. Additionally, we implemented a mechanism to lock the tray as soon as it is placed on the AGV to ensure stability during part placement.

2. **Real-Time Factor Variation:** Discrepancies in real-time factors across different systems posed challenges such as part crashes and timely reading of parts from cameras. To address this, we enhanced the robustness of our code by incorporating conditions and parallel processing at each stage of the process, ensuring accurate timing synchronization and action execution.

3. **Proper Use of Callback Groups:** Implementing and managing callback groups to enable concurrent execution of different functions and prevent thread-locks proved challenging. Through careful optimization and structuring of callback functions, along with the utilization of native threads in Python, we achieved efficient concurrent operation without compromising system stability.

4. **Floor Robot Path Planning Crashes:** Instances of floor robot crashes during path planning between waypoints disrupted task execution. This was resolved by preventing modifications to the plan during execution, clearing waypoints after successful plan execution, and removing unwanted models from the planning scene, ensuring smooth functioning of the floor robot.

5. **Part Order Identification from RGBD Camera Images:** Obtaining accurate part order information from RGBD camera images posed challenges due to variations in lighting and image quality. By refining image processing algorithms and optimizing object detection techniques, we improved the accuracy of part order identification, enhancing the overall efficiency of our system.

6. **Seamless Communication Between C++ and Python Nodes:** Integrating C++ and Python nodes presented challenges in establishing seamless communication and data exchange. Leveraging ROS communication protocols, creating additional ROS services, and optimizing message-passing mechanisms facilitated efficient inter-node communication and collaboration.

7. **Incorrect Floor Robot Planning in the Presence of Attached Parts:** Inaccuracies in floor robot path planning occurred when parts were not properly detached from the planning scene, resulting in erroneous trajectories. Detaching the part from the floor robot and clearing it from the planning scene resolved this issue.

8. **AGV Status Publication Errors:** Discrepancies in AGV status publication, particularly when all AGVs reached the warehouse simultaneously, posed challenges. While this issue is to be addressed by ARIAC, we integrated additional conditions to avoid modifications to AGV status once it reaches the warehouse internally, ensuring more reliable operation.

9. **Pose Transformations:** Calculating and applying pose transformations accurately using the PyKDL library proved challenging, especially when dealing with multiple coordinate frames and orientations. Thorough testing and debugging were conducted to ensure correct robot movements and object positioning.

10. **Devising a Robust Flow for Priority Handling:** Initially, managing order priority using multiple threads resulted in thread locks or incorrect execution. Transitioning to a streamlined approach enabled us to handle both priority and non-priority orders effectively, ensuring smoother execution without encountering thread locks or incorrect behavior.

11. **Gripper Attach and Detach:** Issues with gripper attachment and detachment were addressed by adjusting attachment time and gripper positioning for different part types, improving the reliability of the gripper.

12. **Testing and Debugging:** Thorough testing and debugging were essential to address variability in system configurations and real-time factors. Comprehensive testing across various scenarios helped identify and address potential issues early on, leading to a more robust and reliable system.

11 Course Feedback

The course structure, content, and assignments provided a comprehensive learning experience in ROS2, ensuring a well-rounded understanding of robotics concepts. The quizzes served as effective revision sessions, reinforcing the theoretical knowledge gained in class. Additionally, the Real-World Assignments (RWAs) offered practical applications of concepts, fostering analytical thinking and problem-solving skills.

The highlight of the course was the ARIAC project, which provided a unique opportunity to develop a complete robotic system. This project enabled us to apply learned concepts in a practical setting, honing our logic-building and system architecture skills. The hands-on experience gained from developing a complete system is invaluable, particularly for future job applications.

The accessibility of the professor and TA throughout the semester was commendable. Their availability to address doubts and provide guidance greatly enhanced the learning experience.

A suggestion for improvement would be to invite industry professionals from companies utilizing ROS2 in their products. This would offer students insights into industry practices and requirements, aligning their learning with real-world applications of ROS2 technology.

Overall, the course provided a well-structured and enriching learning experience, equipping students with the knowledge and skills necessary for success in the field of robotics and automation.

12 Resources

- https://pages.nist.gov/ARIAC_docs/en/latest/index.html
- ENPM663 Course content
- https://docs.ros.org/en/galactic/index.html
- https://docs.ultralytics.com