Spring 2023

Final Project

Building a Manufacturing Robotic Software System

May 15, 2023

Students:
Ishan Tamrakar (ishantja)
Krishna Hundekari (krishnah)
Pranav Shinde (pshinde)
Vineet Singh (vsingh03)

Instructors:
Z. Kootbally, C. Schlenoff

Course code:
ENPM663

Contents

List of Figures

1 Introduction

The Agile Robotics for Industrial Automation Competition (ARIAC) is a competition in the field of robotics and automation, organized by the National Institute of Standards and Technology (NIST). The simulation-based competition focuses on the development of agile robotics systems for industrial automation. The aim of this challenge is to test the ability of a robot to perform a series of tasks in a dynamic environment. The tasks are designed to test the robot's ability to perform pick-and-place operations, assembly, and kitting. The agility challenges include various unplanned circumstances including robot failures, sensor failures, humans in the environment, etc., and the software system should be robust enough to manage all these uncertainties and still be able to complete the task. //

The primary goal of this competition is to encourage robotics students to come up with new and innovative approaches to solve challenges related to robotics and automation in manufacturing environments. The competition environment is a simulation of an Assembly line shop floor that uses a floor robot located on a rail parallel to the conveyor belt and a ceiling robot that is able to access the complete shop floor. Both the robots have a UR10e arm and can be used to complete any of the tasks. Four types of parts such as Battery, Pump, Sensor, and Regulator with five different colors will be used in the tasks. Kitting trays are used to place kiting parts of a task. These trays have four quadrants on which the parts are to be placed as per the order information. Each tray is marked with a unique ARUCO standard fiducial marker. Automated guided vehicles (AGV) are used to transport kitted parts to their destination. The Parts are located in the 8 bins or spawned on the conveyor belt at the start of the competition. The Kit Tray/Tool Changer Stations have robot gripper changers and accommodate up to 3 kitting trays on each station. The Part disposal bins are located at two ends and near the center of the conveyor belt. These bins are used to dispose of faulty parts.
The challenges in the competition which were dealt with as part of this project are as follows:

1. **Insufficiency challenge** The insufficient parts challenge requires us to identify whether the required parts to complete the given orders are available in the environment. In case, the parts are not available, we have to submit the order with insufficient parts.
2. **Priority order challenge** This challenge checks for the agility of a robot system to complete the high-priority orders before the other orders. The Competitor Control System (**CCS**) is expected to complete higher-priority tasks as soon as it is announced. Once the priority order is completed, it should then return to the abandoned task and complete it.
3. **Faulty parts challenge** This challenge simulates a realistic manufacturing environment, where few parts are faulty. These faulty parts should be identified and discarded by the CCS and replaced with a new part if available. In any case, the faulty parts cannot be used to complete orders as they do not count for the points in the competition.

Through this project, the aim is to complete all the essential activities of robot control and motion and address the above agility challenges of ARIAC.

2 Architecture

The ARIAC Framework requires an autonomous robot system that is intelligent enough to make decisions in uncertain conditions. Hierarchical architecture is used for task-level planning and reactive architecture is used to handle various agility challenges. The complete architecture diagram is included in **Appendix 1**. A brief explanation of the architecture is done below.

Kitting Task

- The framework consists of the following flow. Order details are read and Parts required for the order are identified in the environment followed by identifying, locating, and moving the tray on AGV.
- If the required parts are not available in the environment, the Insufficiency flag is raised. It is ensured that the part gripper is attached to the floor robot and part locations are read from the camera and are moved from bins to the tray.
- The check for Priority orders is carried out after placing the tray on AGV and after placing a part on the tray.
- The Faulty part check is carried out just before placing the part on the tray and the part is disposed of if faulty.
- When all the parts are placed in the tray, the tray is locked. In case of Insufficient parts, an order is submitted with insufficient parts, and the AGV is moved to the destination.

Assembly Task

- This is another part of the framework with Hierarchical architecture. The AGV needed for the Assembly task is read from the order details.
- The tray is locked on the respective AGV and moved to the Destination Assembly station. A Priority order check is carried out to make sure that no priority order is pending else the current task is abandoned and the priority order is completed and then the abandoned task is completed.
- The Advanced Logical Cameras are used to locate the parts on the tray, and the ceiling robot then picks the parts from the tray and places them in the inserts on the Assembly station.
- A priority check is then carried out to check for any priority order published. If not, all the parts are assembled one after the other.
- Once all parts are assembled, the order is submitted to the Competitor Control System.

Combined Task

- The Combined task requires the system to carry out the kitting as well as the assembly task in order. The parts required for the task and the station for assembly are identified.
- Accordingly, a suitable AGV is selected, for example - Assembly Stations 1 and 2 would require either AGV 1 or 2, and Stations 3 and 4 would require AGV 3 or 4.
- Tray with tray id 0 is selected for the kitting by default. If it is not available, any other tray will be selected for the kitting part of the combined task.
- The floor robot picks the tray and places it on the designated AGV and a priority check is performed.
- If there is no priority order, the floor robot then changes the tray gripper to a part gripper and places all the parts from the bins into the tray while checking for the faulty parts challenge.
- The priority check is done after placing every part in the required quadrant. After placing all the parts in the tray, the tray is locked on the AGV and moved to the Assembly station.
- Priority Check is performed before using the ceiling robot to place the parts from the tray in the assembly inserts in the correct orientation. A priority check is performed after placing each part.
- Once all parts are assembled, the order is submitted by the CCS.

3 Package Structure

The structure of the ROS2 package is shown in Figure 1 and a brief description of each section is given below:

- **config :** stores the *yaml* file which stores the sensor information to be spawned at the start of ARIAC.
- **etc :** stores the complete control architecture of *CCS* and the instructions to run the package.
- **include :** contains the header files for *CCS* and *OrderData* class
- **launch :** contains the launch file to run the package.
- **src :** contains the scripts for CCS.
- **rviz :** contains the Rviz start setup if started from the launch file.

The "group 7" package contains only one node (competition control system) that is defined in the competition state subscriber hpp file. This node has all the topic subscriptions, callbacks, and service client initialization, functions for performing the various tasks and challenges, maps, flags, maps for storing and compartmentalizing the data, and other flags and variables to implement the functions. The function CompleteOrders() is where all the functions are being called to complete the ARIAC challenge.

```
group7
├─ .vscode
│  ├─ c_cpp_properties.json
│  └─ settings.json
├─ CMakeLists.txt
├─ README.md
├─ config
│  └─ group7_sensors.yaml
├─ etc
│  ├─ RWA2 Group7 Control Diagram.jpg
│  └─ instructions.txt
├─ include
│  └─ group7
│     ├─ competition_state_subscriber.hpp
│     └─ order_class.hpp
├─ launch
│  └─ group7.launch.py
├─ package.xml
├─ rviz
│  └─ group7.rviz
└─ src
   ├─ .vscode
   │  ├─ c_cpp_properties.json
   │  └─ settings.json
   └─ competition_state_subscriber.cpp
```

Figure 1: Package Structure

The detailed class diagram for the complete CCS is included in **Appendix 2**.

4 Storing orders

The storing of orders is handled by a class data structure. When the orders arrive at the *ariac/orders* topic, the subscriber callback grabs all the content of the message being published and creates a shared pointer to it. This shared pointer is passed as an argument to the constructor of the *OrderData* class. This is a main class that consists of attributes corresponding to the contents of the first level of the message type being published. Then the subsequent levels are stored as attributes that are classes themselves. The levels keep descending as attributes that are classes till the final level is reached. This way the original hierarchy of the *Orders.msg* is preserved and all the information is contained in a single instance of the *OrderData* class. Once an instance is created, the orders are checked for priority status. If it is a priority order, it is pushed to a vector of *OrderData* objects called a priority order vector. Otherwise, the order is pushed into a normal order vector.

A flow chart demonstrating the order storing process is shown in Figure 2 and the class structure of *OrderData* class is shown in Figure 3

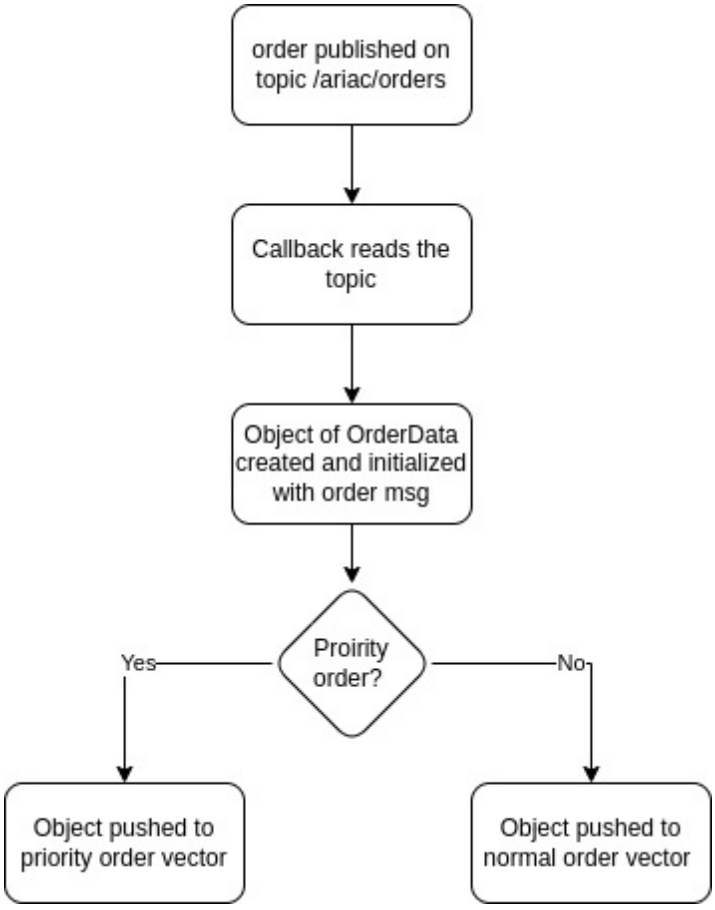

Figure 2: Flowchart showing storing of new orders

OrderData
+id: string
+type: int
+priority: bool
+kitting: **KittingInfo**
+assembly: **AssemblyInfo**
+combined: **CombinedInfo**
+importance_flag: bool
+abandoned_: bool
+agv_part_poses_extracted: bool
+combined_tray_placed_on_agv: bool
+combined_agv_selected: int
+combined_tray_selected: int
+kitting_part_details_orderdata: map
+status_of_assembly: array
<<constructor>>+OrderData(msg: shared_ptr Order

Figure 3: Structure of *OrderData* class

5 Sensors and Cameras

A total of five advanced logical cameras and two break beam sensors are used for this project. Two advanced logical cameras are placed over the two Kit tray stations, two are placed on top of each of the four parts bins grouping, and the fifth is placed on top of the starting point of the conveyor belt where the parts are spawned. The two break beam sensors are placed strategically on two locations on the belt to facilitate part pick up from two locations and allow the collection of most parts. The advanced logical cameras are the most information-dense and it contains all the necessary information about the part already processed such as part type, color, and pose. The advanced logical camera is selected as it provides all the necessary information needed to process a part during orders. Figure 4 shows the coverage of each sensor in the ARIAC environment.

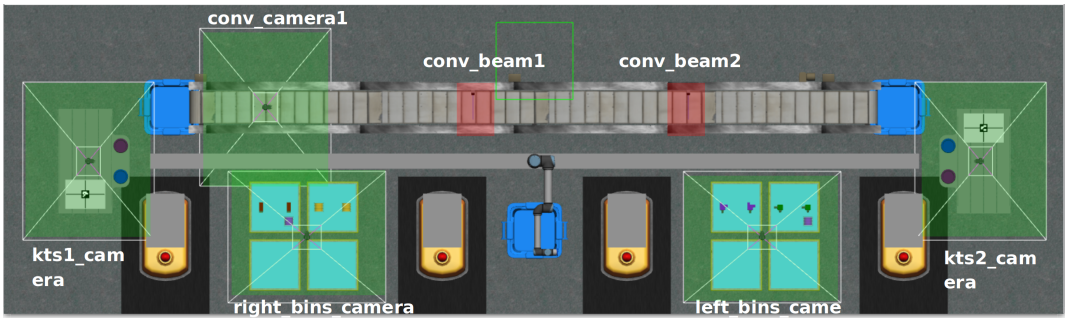

Figure 4: Sensors installation and coverage

6 Parts Pickup from Conveyor

Parts spawned on Conveyor were a major challenge to pick up as the spawn rate and the offset of the parts were variable. To address this challenge, an advanced logical camera is used along with 2 sets of breakbeam sensors. Break-beam sensor 1 corresponds to the first pickup location and break-beam sensor 2 corresponds to the second pickup location.

The logic used for this challenge is shown in Figure 5.

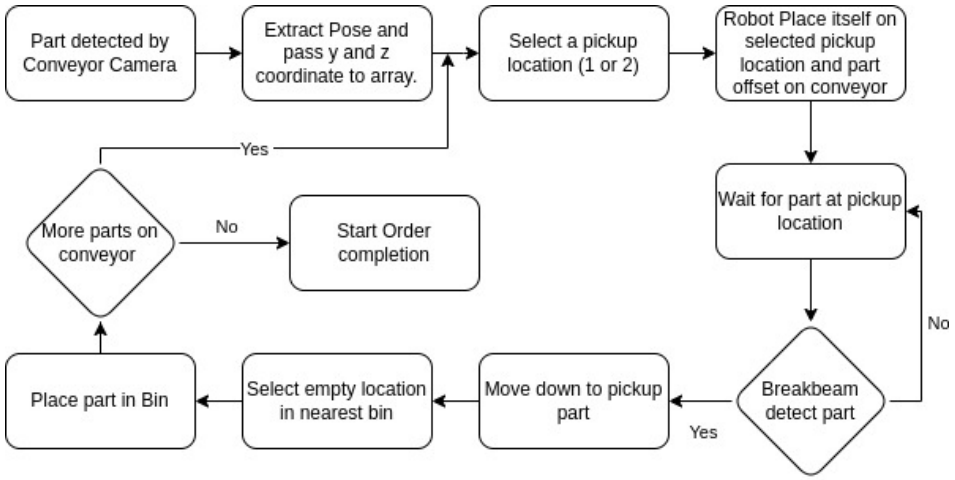

Figure 5: Conveyor parts pickup flowchart

The conveyor camera reads the pose of the spawned part and stores the pose in a vector. Based on the floor robot's current state, a pickup location is decided from where the floor robot will pick up the part. The robot positions itself in the pickup location selected and aligns with the pose of the part and then waits for the part to arrive at the location. As soon as the part is detected by the corresponding breakbeam, the robot moves down by a very small amount with its gripper activated and the part is attached to its gripper.

Once the part is attached, the CCS selects a random empty location in the nearest bins (1,2,5 or 6) and places the part in the assigned location. Once the part is placed, the CCS checks if there are more parts on the conveyor. If yes, then the robot moves again to pick up the next part. This process continues until all the parts have been picked up from the Conveyor. The assumption for the parts pickup from the conveyor is that for all the test cases the conveyor speed will be fixed at 0.2 m/s.

6.1 Challenges faced and solutions implemented

Keeping counters for the breakbeam sensors. This was a challenge as the breakbeam keeps on returning the object detected status as true until the object fully crosses the breakbeam. This leads to the single object being added multiple times to the vector if the filter is not done for the detected object. To address this, a double-switch counter is used, which enables the counter to count a single object only once. Code in Listing 1 shows the implementation, where as soon as the part is detected, $detected_first_breakbeam$ is set to true. This turns false only when the topic of breakbeam stops returning the part detected status as false. This ensures that one part is detected only once.

Listing 1: Breakbeam part counter logic

```
------------------------
void breakbeam_start_counter_cb(const
```

```
    ariac_msgs::msg::BreakBeamStatus::ConstSharedPtr msg){
    if((msg->object_detected == true) &&
    (detected_first_breakbeam == false)){
        breakbeam_one_counter += 1;
        detected_first_breakbeam = true;
    }
    else if((msg->object_detected == false) &&
    (detected_first_breakbeam == true)){
        detected_first_breakbeam = false;
    }
}
------------------------
```

This logic is implemented for both the breakbeam sensors. A similar logic is implemented for the conveyor camera counter but it uses a single-switch counter instead of a double-switch.

Another challenge was to find the empty spaces in the bins to place the picked-up parts. To address this, two sets of arrays namely *left_bin* and *right_bin* are made and these arrays store the total locations (18 each) from the bins next to the conveyor. The locations in these arrays are then checked if the location is empty or not by comparing the data from the respective bins camera. This gives a list of available locations from which a random location is selected. If space is not available in the nearest bins, then the farther bins are checked for availability.

To get the complete poses of bin slots, all the bin slots were spawned with parts, and then a callback read the bin camera status and stored the poses in a text file. The code used is as given in Listing 2 where *msg* is a pointer of type *ariac_msgs :: msg :: AdvancedLogicalCameraImage ::*, and the code is stored in the file named *left_bins.txt*.

Listing 2: Breakbeam part counter logic

```
------------------------
    std::ofstream outfile("left_bins.txt");
    for(auto i:msg->part_poses){
        outfile << i.pose.position.x <<", "<< i.pose.position.y
        <<", "<<i.pose.position.z << std::endl;
    }
    outfile.close();
------------------------
```

7 Agility Challenges

Agility challenges are the main theme of ARIAC to analyze how robust the software control system is and how well can it handle the uncertainties in the environment. Through this project, two agility challenges of Faulty Parts and High Priority Orders are addressed. Along with this an additional challenge of Insufficient available parts is also addressed where sufficient parts are not available to complete the order, and thus insufficient order must be submitted.

The below subsections discussed more on these challenges and how each of the problems is solved.

7.1 Insufficient Orders

The insufficient parts challenge can only happen for kitting and combined tasks, as during the assembly task all the required parts will be spawned on the AGV.

For the combined task, the Insufficient part challenge is addressed in the same way as done for kitting part. For the Kitting task, the information of all the parts is stored on a map (Figure 6), and this map is used to complete the task.

```
//! Kitting Order map
/*!
  Data structure to store kitting task plan // (type,color), bin
*/
std::map<uint8_t, std::pair<std::pair<uint8_t, uint8_t>, uint8_t>> kitting_part_details = {
{1 , std::make_pair(std::make_pair(0, 0), 0)},
{2 , std::make_pair(std::make_pair(0, 0), 0)},
{3 , std::make_pair(std::make_pair(0, 0), 0)},
{4 , std::make_pair(std::make_pair(0, 0), 0)}};
```

Figure 6: Kitting map

The map has key-value pairs, where the key is a quadrant number, and the value is a pair. The first part of the pair contains information about the part type and color that goes into the given quadrant number and the second part of the pair is essential for the challenge. This second part of the pair can have three values:
0: Insufficiency check
1: Part is available for kitting
2: Part is already placed in the respective quadrant

The Insufficiency check and map update happens in the below 2 steps.

1. Updating the quadrant along with the part type and color is done by parsing through the order details that are received at the order announcement.

2. Updating the availability of the part by checking the available parts in bins and conveyors. If all the parts are available, then the order can be completed else the insufficient flag is raised for the order and the kitting order is submitted with whatever parts are available.

To do this second part of updating the availability of the parts, data structures are used that store the parts that are available in the bin and conveyor and update these data structures through a callback to the respective topics. These data structures are parsed part by part to check if enough parts are available to complete the kitting task. If the part required to perform kitting in the given quadrant is available, then the second part of the pair for that corresponding quadrant is updated as "1", and if the part is not available then to "0".

This way when the parsing is completed for the data structures for all four parts, the map has all the required data to perform kitting order. This process is used for combined orders as well.

7.2 High Priority Orders

In order to tackle the challenge of High priority tasks, CCS has to differentiate and sort out received orders into three categories:

1. Priority Orders: The orders with priority flag true

2. Incomplete Orders: The orders that are abandoned during execution to take on other orders of high priority.

3. Normal Orders: Orders that are not priority tasks.

Once the orders are sorted into different vectors, the CCS can decide which task to undertake based on the importance levels as follows:

1. Priority vector

2. Incomplete vector
3. Normal order vector

Firstly, the priority vector is checked for any priority order. If one exists, the CCS will take up that order for completion. If there is no priority order, the CCS checks the incomplete vector for any incomplete orders, i.e., any orders that were abandoned earlier to take on a higher priority task order. If such an order exists, the CCS will take on that task. If there are no priority or incomplete orders only then the CCS will proceed to take the order from the normal order vector.

To make CCS agile, and take on this challenge of "High Priority Task", additions are done as explained below:

1. **Store the status of the incomplete order.** If CCS receives a new order and CCS should know whether the current order itself is a priority order, for this reason, there is a flag in the *OrderData* Class that saves whether the order is a priority task or not. If the current order is not a priority order, then before taking on the new task it is important to save the status of completion of the current order. For that below attributes were added to the *OrderData* class:

   (a) **Kitting Task:** CCS uses a map to store all the data required to execute the kitting task in a map as explained earlier. This map is updated to store the status of the kitting task as well. The second element of the value(pair) of the key(Quadrant) stores the status as below:
   0: Insufficiency check
   1: Part is available for kitting
   2: Part is already placed in the respective quadrant
   The map used is shown in Figure 7.

```
/**
 * @brief A map containing the details of the kitting parts already placed on the tray before the task was abandoned to take
 * new priority order. This map saves the progress of the kitting task.
 */
std::map<uint8_t, std::pair<std::pair<uint8_t, uint8_t>, uint8_t>> kitting_part_details_orderdata = {
    {1 , std::make_pair(std::make_pair(0, 0), 0)},
    {2 , std::make_pair(std::make_pair(0, 0), 0)},
    {3 , std::make_pair(std::make_pair(0, 0), 0)},
    {4 , std::make_pair(std::make_pair(0, 0), 0)}
    };
```

Figure 7: Map storing kitting details for abandoned orders

   While performing kitting, this second element of the value for every quadrant is updated, and if CCS is going to abandon the current task for a higher priority order, then CCS saves the incomplete order in the incomplete vector along with this updated map as the order attribute. Hence, once these incomplete orders are resumed, it will resume the kitting from the last point. This is the same for Kitting part in the combined task as well.

   (b) **Assembly Task:** To store the status of the assembly task, the CCS needs to save how many parts are assembled in the given order. For that purpose, an array of booleans with four elements is created (Figure 8) where each index position stores the status of the assembled parts as true or false. If the current assembly task is abandoned, the incomplete order is saved in the incomplete vector along with the status of the assembly array as the order attribute. Consequently, when resuming these incomplete orders, the assembly will resume from the last point.

```
/**
 * @brief An array indicating the status of assembly for each part that has been assebled before the assebly task is abandoned to take new
 * priority order. This array saves the progress of the assembly task.
 */
std::array<int, 4> status_of_assembly{{1,1,1,1}};
```

Figure 8: Stores part assembled status

(c) **Combined Task:** The kitting map and assembly status array remain the same for storing the status of the kitting and assembly status of the combined task. However, additional information needs to be stored for the combined task. Firstly, the AGV number and tray ID are saved, which were decided to be used for performing the combined task. Along with this, the status is stored to indicate whether the tray is placed on the AGV before the combined task was abandoned. Moreover, a flag is saved to indicate whether the positions of the parts in the AGV have been extracted. (Figure 9)

```
/**
 * @brief A flag indicating whether the positions of parts in an AGV have been extracted in the incomplete order.
 */
bool agv_part_poses_extracted{false};

/**
 * @brief A flag indicating whether a combined tray has been placed on an AGV in the incomplete order.
 */
bool combined_tray_placed_on_agv{false};

/**
 * @brief The ID of the selected AGV for a combined tray.
 */
int combined_agv_selected = -1;

/**
 * @brief The ID of the selected combined tray.
 */
int combined_tray_selected = 0;
```

Figure 9: Variables used for abandoned combined task

2. **Milestones in every task after which the priority check is performed**. Now that the *OrderData* class and CCS can identify priority tasks and store the status of the tasks before they are abandoned, the priority checks are added at the right stages (milestones) while executing these tasks. When referring to priority check, the following steps are taken: (Figure 10)

   (a) Firstly, CCS checks if the current order itself is a priority order. If it is, the order is continued to be executed without further checks.

   (b) If the current order is not a priority order, the CCS checks if any new priority orders have been announced. If there are, the flag of the current order is raised as abandoned, and the completion status of that order is stored. The current order is then appended to the incomplete order vector.

   (c) If no other priority orders have been announced, the CCS continues executing the current order.

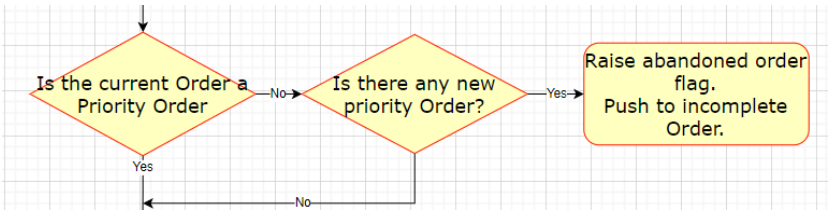

Figure 10: Priority check logic for every section

Below are the milestones for every task:

- Kitting task:
  (a) Tray placement on AGV

(b) After each part is placed in the respective quadrant

- Assembly task:
  (a) Moving the AGV to the stations and the ceiling robot to the initial pose
  (b) After each part is assembled as per the order instructions
- Combined task:
  (a) Milestones same as kitting
  (b) Before switching to the assembly task
  (c) Milestones same as assembly

This way the CCS is agile enough to take care of the High Priority Challenge.

7.3 Faulty Parts detection

Faulty parts challenge includes the detection of parts that are faulty in a given order and can only happen in Kitting or Combined Tasks. Faults parts are checked by using the service */ariac/perform_quality_check*, which returns the faulty part status for all four quadrants on the given agv. So the faulty part detection is done based on the quadrant and not the actual part being faulty. Once the part is detected as faulty, the part should be discarded in the waste bins and then a new part should replace it if available, else the insufficient order must be submitted. The logic used to address this task is shown in the Flowchart in Figure 11.

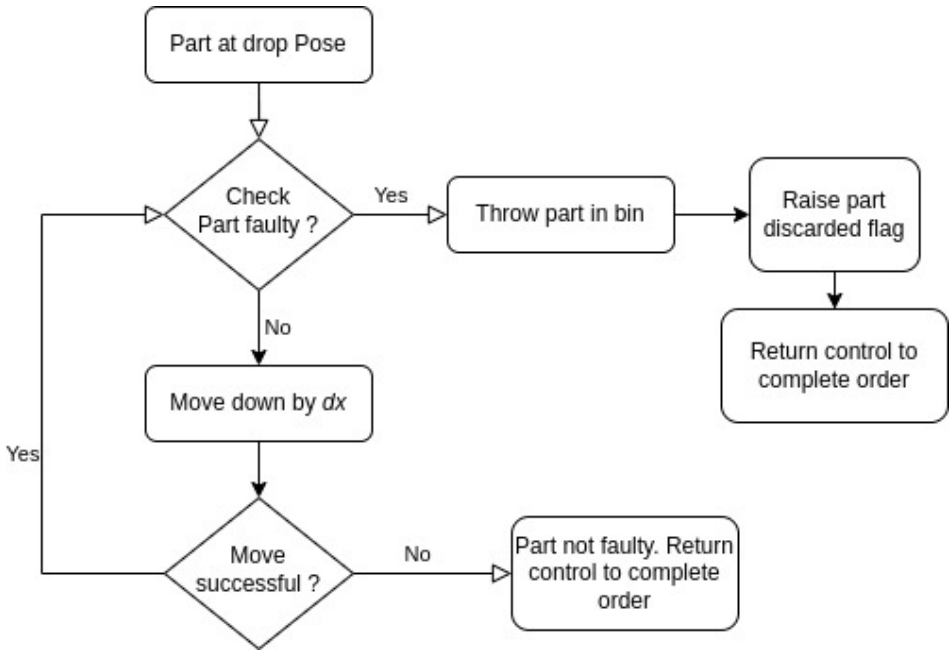

Figure 11: Faulty parts flowchart

When a part is picked up by the floor robot, it takes the part to its drop pose on the agv and calls the quality check. If the part is found faulty the part is discarded and replaced with a new part if available. However, if the part is not found faulty, then the robot moves down by a small distance dx, and again calls the quality check. This is repeated until either the part is found faulty, or there is no further movement possible.

The checking for a faulty part is done repeatedly because the quality check service returns the part status only when the part is close enough/in contact with the agv for a period of time. So multiple checks ensure that no faulty will be missed.

8 Challenges Encountered

Multiple challenges occurred at different stages of this project and are highlighted below in sequence.

1. Control architecture of parallel vs sequential model for robots.
2. Collision with parts while pickup operation, especially pump.
3. Priority Order Switching and storing current order data to resume it later.
4. Ensuring callback variable updates before variables are actually used by the functions. **Sol:** Keeping empty loop checks before variable use.
5. Crashing callbacks and variable updates. **Sol:** Implemented separate callbacks for each variable update.

9 Contributions to the project

The brief distribution of work by the team members is shown in Figure 12.

Tasks	Ishan	Krishna	Pranav	Vineet
Architecture diagram	✔	✔	✔	✔
OrderData class	✔	✔	✔	
Kitting Task		✔		✔
Assembly & Combined Task	✔		✔	✔
Sensor & camera	✔		✔	✔
Conveyor Part		✔		✔
Insufficient task		✔		✔
Priority Order		✔		✔
Faulty Part	✔		✔	✔
Doxygen Comment	✔	✔	✔	✔

Figure 12: Team Members Contribution

10 Resources

- ARIAC Official Page
- MoveIt Tutorials ROS2 Galactic

11 Course Feedback

- Ishan Tamrakar: The course has been a comprehensive introduction to all the important ROS2 concepts as well as architecture formulation. The lectures were succinct and the programming demos were easy to follow. One thing the course could benefit from is the better organization and balance of teams. Some groups had five members, some had four. Some groups had people taking only one class/sem, but some had all members taking three classes/sem. The workload of the assignments is too high if the team is not working properly. So what one gains out of this course is closely tied to the team they are pseudo-randomly assigned to and the personalities involved in the team. It's a hit or a miss situation for someone who may be taking the course independently without forming their own group prior to enrolling in the course. So my recommendation is to either make the assignments lighter and individual or have a prerequisite in the course description that says a solid team of 4-5 is crucial for success in this course. That being said, I think our team did pretty well given that all four of us were taking three classes. I gained enough skills from this course that I can utilize in my future pursuits.
- Krishna Hundekari: The course was intensive and honed my C++ skills along with helping me adapt to ROS2. I enjoyed and learned a great deal from both professors and would recommend my friends to do the course in the coming fall.
- Vineet Singh: The course has been very helpful to enforce the concepts of ROS2, MoveIt, and advance C++. The course was very well structured and RWAs were very relevant and the chronology of the topics was very good and easy to follow. Overall it was a very good experience through this course and got to learn very relevant content for the Robotics Major.

12 APPENDIX 1: Control Architecture

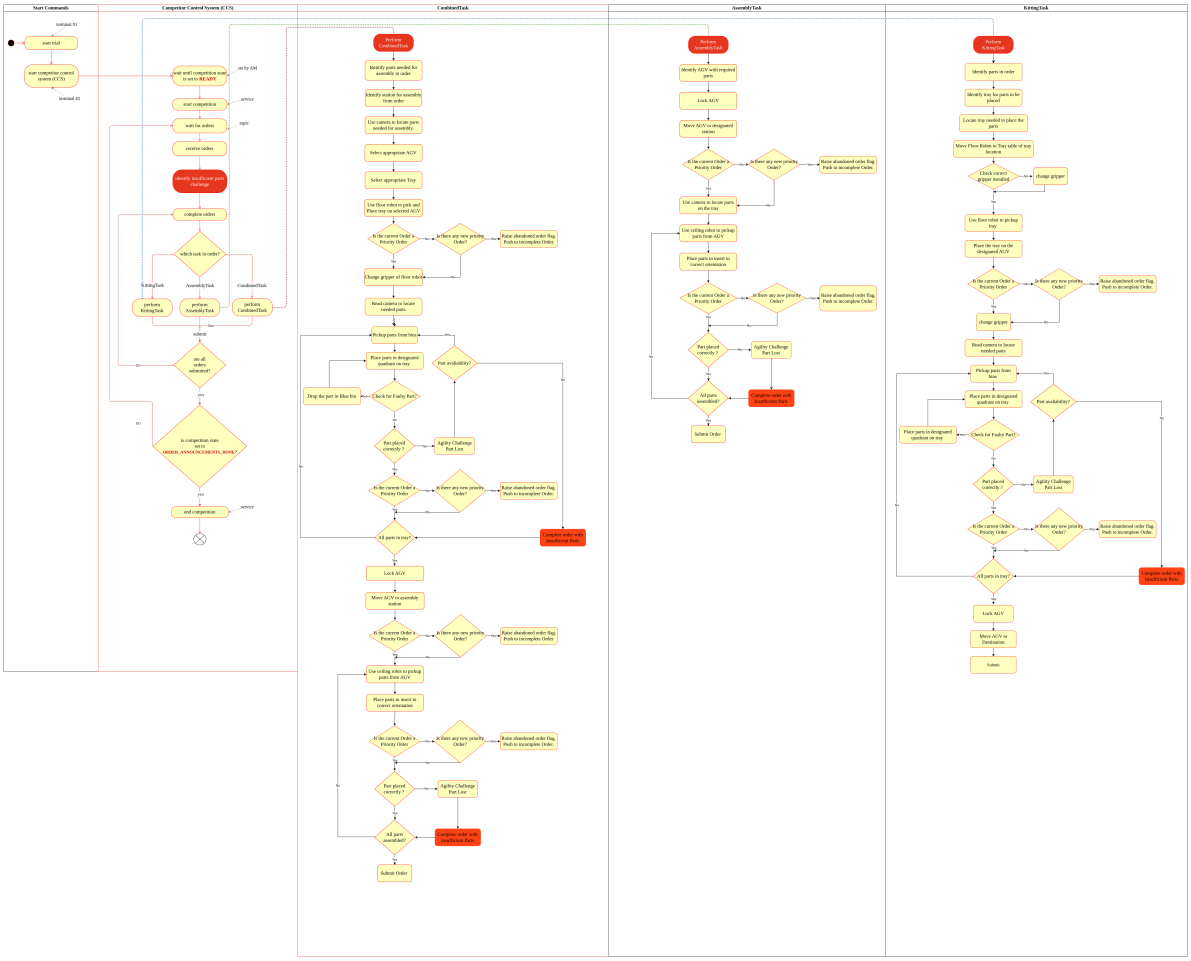

Figure 13: Complete Control Architecture

The complete architecture diagram is accessible through the following link: Architecture Diagram.

13 APPENDIX 2: CCS Class Diagram

The complete Class diagram (Figure 14) of CCS is accessible through the following link: Class Diagram.

Figure 14: CCS Class Diagram