

ENPM663: BUILDING A MANUFACTURING ROBOTIC SOFTWARE SYSTEM

RWA #2

v1.0

Lecturer: Z. Kootbally




















School: University of Maryland

Semester/Year: Spring/2025



MARYLAND APPLIED
GRADUATE ENGINEERING














Table of Contents

 Guidelines	 Start the Competition
 Prerequisites	 Retrieve Orders
 Environment	 Fulfill Orders
 Competitor Package	 Ship Orders
 Objectives & Skills Practiced	 Complete Orders
 Activity and Class Diagrams	 End the Competition
 Activity Diagram	 Submission
 Class Diagram	 Peer Reviews
 Tasks	 Grading Rubric
	 Deductions

Changelog

© **v1.0**: Original version.

✿ Conventions

bad practice	
best practice	
code syntax	
example	
exercise	
file	
folder	
guideline	
note	
question	
task	
terminology	
warning	
web link	<u>link</u>
package	

ROS node	n /node
ROS topic	t /topic
ROS message	m /message
ROS parameter	p /parameter
ROS frame	f /frame
ROS service	s /service





✧ Guidelines

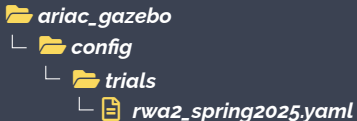
This is a *group assignment*.

Ensure compliance with all specified guidelines, as failure to follow any part of these guidelines will lead to a grade of *zero* for the assignment.

- ⦿ Do not reuse a package obtained from other groups.
- ⦿ Keep your work confidential and refrain from sharing it with other groups.
 - ✍ If you use github, you have to make your repository private.
- ⦿ While discussing assignment challenges is encouraged, refrain from exchanging solutions with peers.
- ⦿ Do not use code generated by AI tools.
 - The potential risks outweigh the benefits.


✿ Prerequisites

- ⦿ Review: Competition Overview
- ⦿ Review: ROS Communication
- ⦿ Review: Orders
- ⦿ Place the trial file  ***rwa2_spring2025.yaml*** in  ***trials***, as shown in the tree below.
- ⦿ Start the environment with the trial file:
 -  `colcon build --packages-select ariac_gazebo`
 - Source the workspace.
 -  `ros2 launch ariac_gazebo ariac.launch.py trial_name:=rwa2_spring2025`
 - ▷ This command starts the Gazebo environment showed in the next slide.





✧ Environment

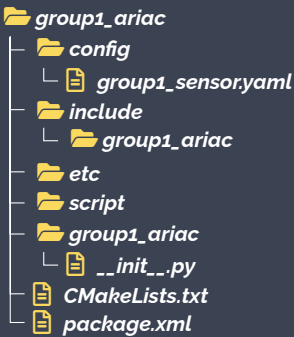
The environment is created based on the contents of  *rwa2_spring2025.yaml*. It includes trays, parts stored in bins, and parts placed in trays on AGVs.

Parts placed in trays on AGVs are used to test some functionalities of your package. In the competition, you will need to:

1. Find trays.
2. Place trays on AGVs.
3. Find parts.
4. Place parts in trays on AGVs.

✧ Competitor Package

Develop a competitor package, which is used to interact with the ARIAC environment. Create this package in the same workspace you cloned ARIAC packages. The structure of a minimal competitor package is shown below.



- ⦿ This package demonstrates the structure for Group #1. Group #2 should maintain the same structure.
- ⦿ **group1_sensor.yaml** consists of your sensors for the competition. For now, use this sensor configuration.
- ⦿ The folder **etc** will be used for extra files, such as diagrams or pictures.
- ⦿ This package must be compressed into a file (zip, tar.gz, ...) and submitted on Canvas.

✧ Objectives

The goal of this assignment is to:

1. Develop UML Activity and Class diagrams for ARIAC. This step requires a good understanding of ARIAC, which can be obtained by reading ARIAC documentation.
2. Create a competitor package to read orders, fulfill orders, and handle agility challenges.



This assignment does not involve sensing, perception, or motion planning. Assume that trays and parts have already been detected and that parts have been placed on AGVs. Future assignments will require you to perform sensing, perception, and motion planning.

✧ Skills Practiced


ROS 2 fundamentals, node development, multi-threaded executors, and service calls.

✱ Activity and Class Diagrams

For this assignment, you will develop UML Activity Diagrams and Class Diagrams for the ARIAC 2024 competition framework. These diagrams will model the workflow, interactions, and system components involved in an ARIAC task, providing a structured representation of how robotic automation is managed within the competition.

- ⦿ Your Activity Diagram should illustrate the flow of actions and decision points within a typical ARIAC scenario.
- ⦿ Your Class Diagram should define the key system components in ARIAC 2024, their attributes, methods, and relationships.



Each diagram (in PDF format) must be included in your competitor package. Place these files in the folder  **etc**

✧ Activity Diagram

- ◎ Start and End Nodes: Represent where the process begins and ends.
- ◎ Key Actions:
 - Starting the competition.
 - Receiving orders.
 - Using sensors to find parts and trays.
 - Retrieving trays from changer stations.
 - Placing trays on AGVs.
 - Retrieving parts from bins or conveyor belt.
 - Placing parts in trays.
 - Shipping AGVs.
 - Completing orders.
 - Ending the competition.
- ◎ Decision points:
 - Detecting if a part is faulty.
 - Detecting if a part is correctly placed.
 - Verifying whether the order is complete.
 - Handling failures (e.g., incorrect placement, missing parts).

✿ Class Diagram

- ⊙ Core Classes and Attributes.
 - **Order** (**order_id**, **parts_list**, **status**, etc)
 - **Part** (attributes: **type**, **location**, **status**, etc)
 - **Tray** (attributes: **tray_id**, **assigned_order**)
 - **AGV** (attributes: **agv_id**, **current_location**, **loaded_tray**, etc)
 - **Robot** (attributes: **robot_id**, **tool_type**, etc)
- ⊙ Methods: **pick_part()**, **place_part()**, **verify_part()**, **transport_tray()**, etc.
- ⊙ Relationships:
 - Association between **Robot** and **Part**.
 - Association between **AGV** and **Tray**.
 - Association between **Order** and **Part**.
 - Association between **Order** and **Tray**.
 - Association between **Tray** and **Part**.

✧ Tasks

Implement some of the key actions identified in your class diagram. Guidance is provided in the following slides.



The ARIAC documentation, including tutorials, provides code examples for most of the tasks required in this assignment. You are encouraged to reuse and adapt code from the ARIAC documentation to implement your solution efficiently.

✧ Start the Competition

The first step of ARIAC is to initiate the competition beforehand. Without starting the competition, sensors, robots, and the conveyor will not be operational. The initiation of the competition is triggered by calling the service `s /ariac/start_competition`. This service should be called only when the state of the competition is **READY**. The current state of the competition is published to the topic

`t /ariac/competition_state`



1. Create a service client to `s /ariac/start_competition` Note that the ARIAC manager starts the server.
2. Create a subscriber to `t /ariac/competition_state`
 - Verify within the subscriber callback whether the received messages indicate that the system is in the **READY** state prior to invoking the service client to initiate the competition.



Make sure to use the constants defined in the message structure of the topic `t /ariac/competition_state`

- ⊙ `>_ ros2 topic type /ariac/competition_state`
- ⊙ `>_ ros2 interface show ariac_msgs/msg/CompetitionState`

✧ Retrieve Orders

Once the competition has started, orders are published to `t /ariac/orders`. These orders contain information about the required parts, their locations, the target trays, and the target AGVs. Your program must retrieve these orders dynamically to ensure the robot executes the correct tasks. Properly handling orders is essential for successful task execution in ARIAC.



1. Create a subscriber to `t /ariac/orders`:
 - Use the appropriate message type `m ariac_msgs/msg/Order` to parse the received data.
 - Ensure your subscriber can handle multiple incoming orders efficiently.
2. Store Orders in Your Program:
 - Maintain a structured data representation to store orders.
 - Each order should retain key information, such as order ID, required parts, and target destinations.
 - Consider handling updates or new orders dynamically to adjust robot behavior accordingly.



- ⦿ Ensure your subscriber runs in a separate thread to process incoming orders without blocking execution.
- ⦿ Orders have two priority levels: **Normal priority** and **High priority**. Consider restructuring your order storage to prioritize **High priority** orders for processing before **Normal priority** orders.

✱ Fulfill Orders

Given our current knowledge limitations in sensing, perception, and motion planning, we are unable to fulfill orders at the moment. Consequently, we will resort to simulating order fulfillment, employing a technique often referred to as *faking*.

We have simulated order fulfillment by pre-loading three AGVs with trays and parts. These AGVs, trays, and parts align with the order configurations published to `t /ariac/orders`



We can bypass the order fulfillment task for this assignment and proceed to the next steps.

✧ Ship Orders

Once a kitting order is completed, the AGV carrying the assembled kit must be shipped to the warehouse. In this assignment, the kits have already been completed, as the environment starts with two AGVs, each carrying a fully assembled kit. Your task is to ship these AGVs in the order their kits were completed.

To ship an AGV to the warehouse, you must make two service calls:

- ⦿ `s /ariac/agv{n}_lock_tray` locks trays and parts on AGV `n` so they do not fall off while the AGV is moving.
- ⦿ `s /ariac/move_agv{n}` moves AGV `n` to a station.



1. Implement a service client for each required service.
2. One of the orders is classified as high priority. The AGV carrying the high-priority order must be shipped first.
3. Both services require an AGV ID as input. The AGV ID must be retrieved dynamically from the order instead of being hardcoded. **Failure to obtain the AGV IDs dynamically will result in point deductions.**

✱ Complete Orders

The order submission process requires invoking a service to evaluate the order. Upon execution, a score will be displayed in the terminal. An order is considered complete **only after** the AGV carrying the kit has successfully arrived at the warehouse.



1. Create a service client for `s /ariac/submit_order`
2. Create a subscriber to `t /ariac/agv{n}_status`. This topic publishes the status of AGV **n**.
 - This topic messages use constants:
 - ▷ `>_ ros2 topic type /ariac/agv1_status`
 - ▷ `>_ ros2 interface show ariac_msgs/msg/AGVStatus`
3. Call the service when the AGV is at the warehouse.

✱ End the Competition







After all orders have been completed (your order storage is empty) and the competition state reaches **ORDER_ANNOUNCEMENTS_DONE**, finalize the competition by calling the service `s /ariac/end_competition`. Similar to `s /ariac/start_competition`, this is a trigger service and must be called with an empty request.



1. Implement a service client for `s /ariac/end_competition`
2. Verify that all orders stored in your system have been successfully submitted.
3. Ensure there are no remaining orders to process. This can be confirmed by checking the competition state, which should be **ORDER_ANNOUNCEMENTS_DONE**.
4. Invoke the `s /ariac/end_competition` service to officially conclude the competition.

✧ Submission

Before submitting your package, make sure the following is addressed.

1. The folder  **etc** contains:
 - Activity Diagram (PDF).
 - Class Diagram (PDF).
 -  **readme.txt** which describes how to run your package.
2.  **package.xml** is updated with a description and maintainer tags.
3. Delete  **log**,  **build**, and  **install**.
4. Rebuild your package.
5. Rerun your node(s).

✧ Peer Reviews

Peer reviews must be submitted on the same day you upload your package to Canvas. Follow the instructions below for peer review submission:

1. Rate each of your teammates on a scale from 1 to 10, where 1 indicates the lowest contribution and 10 represents an excellent contribution.
2. **Do not** rate yourself.
3. Late peer review submissions will result in a penalty.
4. **Email Subject:** "Group 1: Peer Reviews"
5. **Email Content:** Include only the student names and their corresponding ratings.

Example:

John Doe: 10

Jane Doe: 8


Jack Smith: 5



Peer reviews ensure that everyone contributes equally to assignments.

✧ Grading Rubric 30 pts (1/3)

⦿ *Competitor Package Structure (3 pts)*

- **3 pts:** Maintains correct package structure with required files and directories.
- **2 pts:** Minor structural issues, such as missing  *etc* or improper package configuration.
- **0 pts:** Major structural issues or missing required files.

⦿ *Activity & Class Diagrams (5 pts)*

- **5 pts:** Includes well-structured Activity and Class Diagrams in PDF format.
- **3 pts:** Diagrams are included but have minor inaccuracies.
- **0 pts:** Missing diagrams or major inconsistencies.

⦿ *ROS 2 Integration (3 pts)*

- **3 pts:** Checks the competition state is **READY** and calls `s /ariac/start_competition`.
- **2 pt:** One of the steps is missing.
- **0 pts:** Does not call the service.

⦿ *Order Retrieval (2 pts)*

- **2 pts:** Properly subscribes to `t /ariac/orders` and dynamically stores orders.
- **1 pt:** Partially retrieves orders but lacks proper storage or prioritization.
- **0 pts:** No dynamic order retrieval.

✱ Grading Rubric 30 pts (2/3)

⊙ Order Shipping (3 pts)

- **3 pts**: Ships AGVs correctly.
- **2 pts**: Partially implemented but incorrect AGV processing order.
- **0 pts**: No AGV processing or major logic errors.

⊙ Order Submission (2 pts)

- **2 pts**: Successfully submits completed orders and verifies AGV status dynamically.
- **1 pt**: Submits orders but lacks verification of AGV status.
- **0 pts**: No order submission or incorrect implementation.

⊙ End Competition (2 pts)

- **2 pts**: Checks all orders are processed, checks the competition state is **ORDER_ANNOUNCEMENTS_DONE**, and calls `s /ariac/end_competition`.
- **1 pt**: One of the steps is missing.
- **0 pts**: Does not call the end competition service.

⊙ Proper Code Documentation (3 pts)

- **3 pts**: Well-documented functions/classes with appropriate descriptions.
- **2 pts**: Partial documentation with missing details.
- **0 pts**: No meaningful documentation.

✧ Grading Rubric 30 pts (3/3)

⦿ *Programming Conventions (Python & C++) (3 pts)*

- **3 pts**: Follows PEP 8 (Python) and C++ Core Guidelines style guides.
- **2 pts**: Minor violations.
- **0 pts**: Major violations or unreadable code.

⦿ *Submission Requirements (2 pts)*

- **2 pts**: Includes all required files and documentation.
- **1 pt**: Minor submission errors.
- **0 pts**: Major submission errors or missing files.

✿ Deductions

- ⦿ Late submission: *As per syllabus guidelines.*
- ⦿ Use of AI-generated code: Assignment grade of **zero**.
- ⦿ Code sharing/plagiarism: Assignment grade of **zero**.
- ⦿ Non-private GitHub repository: Assignment grade of **zero**.
- ⦿ Classes lack adequate documentation (missing docstrings) or insufficient inline comments: **-2 pts**.
- ⦿ Hardcoding information that should be retrieved dynamically: **-5 pts**.
 - If you are not sure whether some information should be retrieved dynamically, ask me.