# Documentation of an OPC-UA Integrated Multi-Robot Manufacturing System


Contact information:
umut.aygor@student.uantwerpen.be

# Overview:

This project implements a multi robot control system using OPC UA communication and Visual Components simulation software.

The system manages multiple mobile robots in a manufacturing environment, handling tasks such as product pickup and delivery between conveyors.

# First-Time Setup Guide

When opening the project for the first time, it's essential to prepare certain files and configurations to ensure the system functions correctly.

### Step 1: Prepare/Modify JSON Configuration Files

The system relies on JSON files to define various properties and configurations, such as **robot initial positions, pathway properties, idle properties,input and output conveyor properties**. You need to create or modify these JSON files according to your simulation environment.

**Example: Pathway_info.json file:**


```
{
    "Name": "Pathway Area #2",
    "X": 2931.874,
    "Y": 412.917,
    "Rz": 90,
    "AreaLength": 12012.9040906581,
    "AreaWidth": 3284.87814662502
}
```


These properties are necessary to create the pathways for the robots to move. This whole JSON file will be sent to the simulation. Then the python script in the simulation parses this information and applies these properties for each created pathway.

Notice that the JSON file starts with the **second pathway** because the first Pathway Area must already be added beforehand in the simulation environment for clonning process.

### Step 2: Set Up the OPC-UA Server

Ensure all dependencies, including Eclipse Milo and JADE, are correctly configured in your pom.xml.

**IMPORTANT:** Jade 4.5.0 is not always automatically installed via Maven. You are provided a **Jade.jar** file which you can install the dependency manually. If you are using Intellij Idea(recommended) just navigate to File > Project Structure > Modules > Dependencies, and then you can add this jar file(which is located in the opcua folder) into this section.

Execute the server application(Server.java). The server should start and listen at opc.tcp://localhost:4840.

### Step 3: Connecting Visual Components to the OPC-UA Server

Open Visual Components from the .vcmx file provided to you. Open **File** tab and click **Options,** then click **Add-ons** to enable the **Connectivity** add-on.

Now, you are able to reach Connectivity tab on Visual Components.

If the OPC-UA Server is not already added on your Visual Components, right-click OPC-UA on your Connectivity Configuration section and add the server we created on Java side.

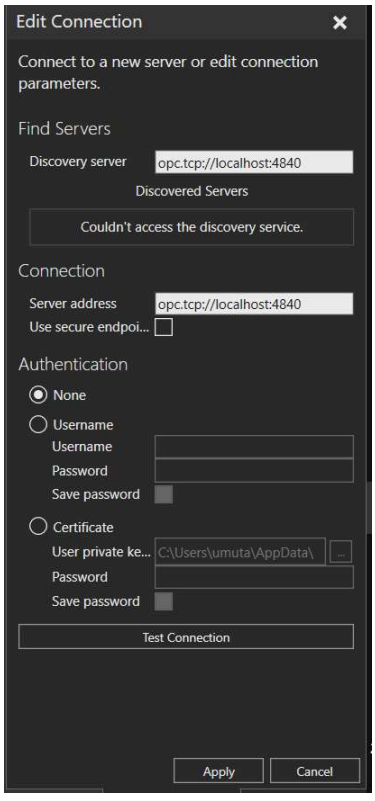



*Figure 1: OPC-UA Server Connection Properties*

When the configurations for the OPC-UA server are set, activate the connection(make sure the OPC-UA server is running and listening on the Java side).

You will be notified from the Visual Component's output that the connection is successful.

Finally, run the simulation from the Play button on top of the environment. You can increase the simulation speed but please do not go for high values because the variables might not be updated simultaneously.

# Purpose of the Provided Example Warehouse

The example implements an automated warehouse solution using multiple mobile robots to handle product transportation between conveyors. The main purposes are:

- Automate product movement between input and output conveyors
- Optimize robot path planning and collision avoidance
- Provide real-time monitoring and control of robots and conveyors

**Simulation Environment:**

- **Robots**: 8 mobile robots navigating pathways.
- **Conveyors**: 4 input and 4 output conveyors.
- **Pathway Areas**: 51 Pathway Areas for robot navigation.
- **Idle Locations**: 8 idle locations
- **Component**: A basic box shaped component to simulate products arriving on conveyors.

This is an example scenario that was previously created. You can adjust the quantities, locations, or any other logic from either GUI or code.

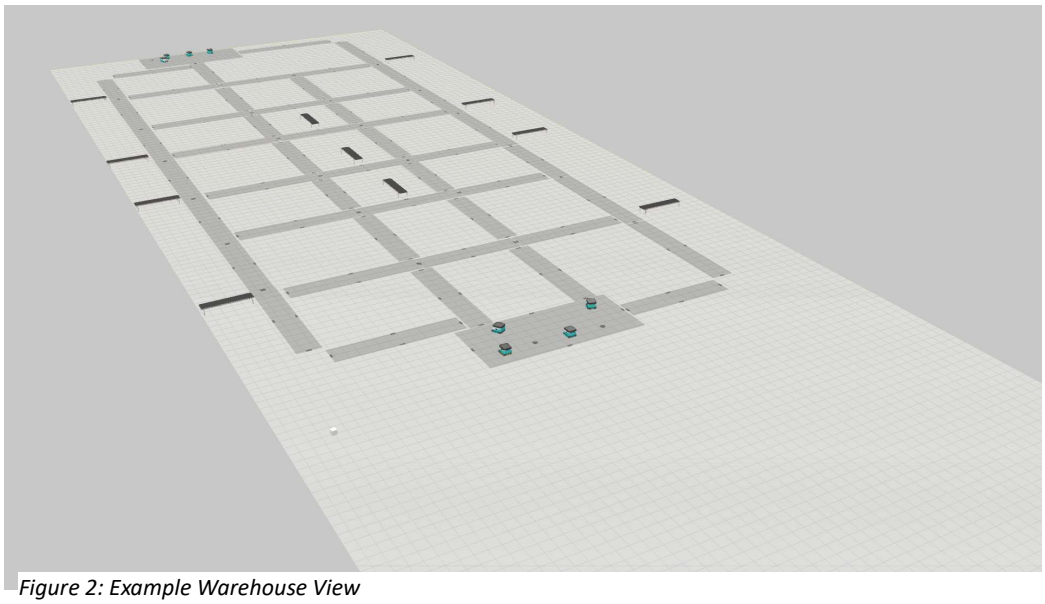


*Figure 2: Example Warehouse View*

# How Does the Project Work?

The system operates through OPC UA communication where variables(e.g. pathway length) are paired between Visual Components and Java.

The Java server creates the **same variables** and modifies/monitors these variables (robot locations, battery levels, conveyor states, etc.). While this can also be done in Visual Components we become dependent to it and it has limitations.

## Explanation of the Java side(OPC-UA and Jade Agent)

### 1-)Server.java File:

- **OPC-UA Server Initialization**: Configures and starts the OPC-UA server with specified endpoints, security policies, and application details.
- **Namespace Registration**: Registers the custom namespace (CustomNamespace.java file) with the server to make the simulation's data model available to clients.
- **Client Integration and GUI Launch**: Establishes a connection with an OPC-UA client and launches a graphical user interface (RobotControlUI.java file) to interact with the simulation.
- **Multi-Agent System Startup**: Initiates a JADE container to host agents that will control the robots within the simulation.

### 2-)Container.java File:

Initializes the JADE agent container that manages the robot agents.

- **JADE Initialization:** Sets up the JADE runtime environment and creates a main container for hosting agents.
- **Agent Deployment:** Creates and starts an instance of RobotAgent within the JADE container, initiating the multi-agent system that controls robot behaviors in the simulation.

### 3-)CustomNamespace.java File:

Defines a custom namespace for the OPC-UA server and manages the variables.

- **Variable Registration:** Creates and registers OPC-UA variables for robots and conveyors, including properties like location, battery level, product status, and pathway information.
- **Initial Locations Loading for Components:** Reads configuration data from JSON files (e.g., PathwayInfo.json, IdleInfo.json) to initialize pathway and conveyor properties within the OPC-UA namespace.

- **Battery Level Simulation:** Includes a scheduled task to simulate battery level reduction in robots over time, updating the corresponding OPC-UA variables.

### 4-)RobotTemplate.java File:

Represents the template for robot variables and contains methods for accessing and modifying robot properties. Also includes collision detection logic shared among robots.

- **Property Management:**
    - Defines properties for each robot, such as location, next location, stop status, battery level, and priority.
    - Provides getter and setter methods for these properties to facilitate state management.
- **Collision Detection**:
    - Implements a static method checkForCollisions() that checks for potential collisions between robots.
    - Uses robots' current and next locations to detect conflicts.
    - Sets appropriate flags (CollisionDetected, UnavailablePathway, Stop) to prevent collisions.
- **Priority Handling:**
    - Determines which robot should yield during a collision based on their priorities.
    - Implements logic to handle equal priorities, ensuring consistent behavior.

### 6-)RobotTemplate.java File:

Defines a JADE agent that manages robot behaviors, including task assignments, collision detection, and updating robot states based on conveyor statuses.

- **Agent Initialization**:
    - Sets up agent behaviors using ParallelBehaviour and TickerBehaviour to perform periodic checks.
- **Task Assignment**:
    - Monitors conveyor production statuses.

    - Assigns robots to pick up products from conveyors that have produced items.
    - Ensures that only one robot is assigned per conveyor to avoid conflicts.
- **Collision Management**:
    - Integrates collision detection by invoking RobotTemplate.checkForCollisions().
    - Resumes robot movement when paths become clear.
- **State Updates**:
    - Checks if robots have picked up products and updates their CarryingProduct status.
    - Assigns drop-off targets to robots carrying products.
    - Directs robots back to idle locations after completing tasks.
- **Dynamic Targeting**:
    - Adjusts robot targets based on current tasks and conveyor statuses.
    - Randomly selects drop-off locations from available options.

**7-)RobotControlUI.java File:**

The purpose of this user interface is to monitor the robot, conveyor and carried products. In addition, you can also dynamically modify some part of the system while the simulation is running.

## Explanation of the Visual Components Side

### Introduction:

Each component in V.C., such as robots and conveyors, has **properties** and associated **Python scripts** to define their state and behavior within the simulation.

To see the Properties and Behaviors of a component, click the component and navigate to the **Modeling** tab on Visual Components.

**Properties**: These are attributes that store state information and configuration settings for each component(like a robot's size or coordinates). We create the same properties in OPC-UA(CustomNamespace.java) to modify or monitor them. That is why we are calling the OPC-UA as the brain of this system.

**For example**, robots have properties like Location, Target, BatteryLevel, and Priority, while conveyors might have properties like Produced status(when the product is produced on that conveyor) or ProductType. Properties enable dynamic interaction with components during the simulation.

**Python Scripts**: Each component has a Python script that controls its behavior. The scripts utilize essential functions like:

- **OnRun**: Called when the simulation starts. It contains the main logic for the component's operation, such as initializing robots, handling movements, or starting processes.
- **OnReset**: Triggered when the simulation is reset. It cleans up the component's state, deletes cloned components, and resets properties to their initial values.
- **OnSimulationUpdate**: Executes at each simulation timestep. It updates the component's state in response to simulation dynamics, such as moving robots, updating positions, or checking conditions.

These functions allow components to respond to simulation events and maintain proper operation throughout the simulation lifecycle.

**Synchronization with OPC-UA Server**: The properties of the V.C. components correspond directly to the variables defined in CustomNamespace.java on the OPC-UA server side. This design ensures that both the simulation environment and the OPC-UA server maintain consistent and synchronized state information.

- For instance, when a robot's Location property changes in the simulation, the corresponding OPC-UA variable is updated. This synchronization allows external systems or clients connected via OPC-UA to access and interact with the latest state of the robots and conveyors.

**Why Components Have Properties and Scripts**: By combining properties and scripts, each component becomes a self-contained unit that can:

- Store its state and configuration.
- Define its behavior and interactions within the simulation.
- Communicate with external systems through synchronized OPC-UA variables(actually the whole point of the project).

## Environment Creation:

When the simulation is not running, template components(robots, pathways, conveyors) are placed in the environment. In PythonScripts of these components there are clonning processes and they clone these components based on the OPC-UA Server’s requests, for example, when a user specifies for 8 robots in the user interface, 7 robots are cloned(the original robot is already in the layout).

Initial coordinates for these components(X, Y,  Rz) are sent from a JSON file that we read in CustomNamespace.java in OPC-UA side.

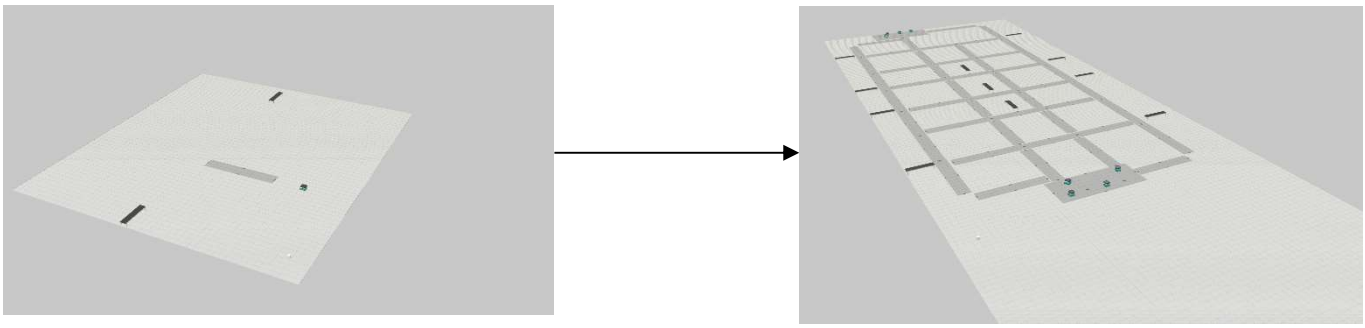


*Figure 3: Environment creation before and after running the simulation*

**Explanation of the Python Script of Each Component:**

**1-)Mobile Robot Resource Python Script:**

As previously mentioned, in order to see the PythonScript of a component, navigate to the Modeling tab in Visual Components while the component is clicked. Not to be confused with Resource Script as the robot doesn't use any logic from there, every logic for the robot is implemented in Python Script.

**1.1 Initialization and Cloning of Robots**

- **Robot Quantity and Initial Positions**:
    - The script begins by retrieving the RobotQuantity property, which determines the number of robots to be simulated.
    - An InitialPositions property is also used, which contains the starting positions and orientations for each robot, typically formatted as a JSON-like string.
- **Cloning Robots**:
    - The original robot component (Mobile Robot Resource) serves as a template.
    - The script clones this component to create the required number of robots, naming them uniquely (e.g., Mobile Robot Resource #2, Mobile Robot Resource #3, etc.).
    - Each cloned robot is positioned according to the data from InitialPositions. If insufficient positions are provided, default positions or offsets are applied.

**1.2 Dynamic Property Creation for Individual Robots**

- **Purpose of Properties**:
    - Properties are used to store and control the state of each robot individually, such as their targets, current locations, and statuses.
    - Having unique properties for each robot allows for independent control and monitoring within the simulation.

- **Creating Unique Properties**:
    - A list of property names is defined, including properties like Target, Stop, CarryingProduct, BatteryLevel, Location, NextLocation, among others.
    - For each robot, the script appends the robot's index to each property name (e.g., Target1, Stop2, BatteryLevel3) to ensure uniqueness.
    - These properties are created dynamically if they do not already exist, using the appropriate data types (e.g., VC_STRING, VC_BOOLEAN, VC_INTEGER).

### 1.3. Pathfinding and Navigation Using A* Algorithm

- **Collecting Pathway Data**:
    - The script gathers all pathway components in the simulation (e.g., areas named Pathway Area, Idle Location) to define possible routes.
    - It constructs a list of pathways with their positions and dimensions, which is used by the pathfinding algorithm.
- **Implementing A* Algorithm**:
    - When a robot needs to move to a target, the script invokes the A* pathfinding algorithm to compute the shortest path between its current location and the destination.
    - The algorithm considers distances between pathways and avoids any pathways marked as unavailable (e.g., due to collisions or other robots occupying them).
- **Assigning Movement Instructions**:
    - The resulting path is translated into movement instructions for the robot.
    - Control points along the path are calculated, and the robot's vehicle behavior is configured to follow these points smoothly.

### 1.4. Collision Detection and Avoidance

- **Proximity Checks**:
    - The script regularly checks the distance between robots to detect potential collisions.
    - A safe_distance is defined, and if two robots come within this distance, collision avoidance procedures are initiated.
- **Priority-Based Collision Handling**:
    - Each robot has a Priority property.

    - In case of a potential collision, the robot with the lower priority stops to let the higher-priority robot proceed.
    - If priorities are equal, default rules (e.g., the robot with the lower index stops) are applied.
- **Updating Collision Properties**:
    - Collision-related properties (CollisionDetected, UnavailablePathway, Stop) are updated accordingly to manage the robots' responses.

### 1.5. Product Handling and Interaction with Conveyors

- **Product Pickup from Conveyors**:
    - Robots are tasked with picking up products from input conveyors (e.g., InputConveyor, InputConveyor #2).
    - When a product is available (indicated by the conveyor's Produced property), the robot attaches the product to itself.
    - The script includes functions to attach the product to the robot's physical structure within the simulation.
- **Product Drop-off at Output Conveyors**:
    - Robots transport the products to designated output conveyors or drop-off locations (e.g., Conveyor3, Conveyor4).
    - Upon arrival, the robot detaches the product and places it onto the conveyor.
    - The script manages the detachment and proper positioning of the product on the conveyor.
- **Updating Carrying Status**:
    - Robot properties such as CarryingProduct and CarriedProduct are updated to reflect whether the robot is currently transporting a product.
    - These properties are used to make decisions about the robot's next actions.

### 1.6. Integration with Simulation Events

- **OnRun Function**:
    - The OnRun function is executed when the simulation starts.
    - It initializes robots, clones them as needed, sets up initial positions, and prepares pathway data for pathfinding.
- **OnSimulationUpdate Function**:

- This function is called on each simulation update.
- It updates robot positions, checks for collisions, and manages the incremental movement of robots along their paths.
- It also updates properties like PositionX, PositionY, and MaxSpeed based on the robot's behavior.

- **OnReset Function**:
  - Handles the resetting of the simulation.
  - Deletes cloned robots, resets properties to default values, and ensures that the simulation can be run again cleanly.

### 1.7. Utility Functions and Helpers

- **Vector and Matrix Operations**:
  - The script uses vector mathematics to handle positioning, movement, and orientation.
  - Functions for vector addition, subtraction, normalization, and length calculation are defined to support movement calculations.
- **Custom Parsing Functions**:
  - Since external libraries are not imported, custom functions are used to parse strings that contain position data (e.g., parsing the InitialPositions property).

## 2-)Input Conveyor Python Script

### 2.1 Cloning and Positioning Input Conveyors

- **Reading Conveyor Locations**:
  - The script accesses a property named Input_Conveyor_Location from the original conveyor component. This property contains a JSON-formatted list of dictionaries, each specifying the X, Y, and Rz (rotation around the Z-axis) for each input conveyor.
  - The locations are parsed using the json module, resulting in a list of conveyor position data.
- **Determining Conveyor Quantity**:
  - It retrieves or creates a property called InputConveyorQuantity on the original conveyor component, which represents the number of input conveyors to be created.

    - The property is set to match the number of locations provided in Input_Conveyor_Location, ensuring that each location corresponds to a conveyor.

- **Cloning Conveyors**:
    - The original conveyor serves as a template for cloning.
    - For each conveyor beyond the first (since the original represents the first conveyor), the script clones the original conveyor component.
    - Each cloned conveyor is renamed uniquely (e.g., InputConveyor #2, InputConveyor #3).
- **Positioning Conveyors**:
    - Cloned conveyors are positioned based on their specified X, Y, and Rz values from the Input_Conveyor_Location data.
    - The script uses transformation matrices (vcMatrix) to apply rotations and translations to each conveyor, accurately placing them in the simulation environment.

**2.2 Dynamic Property Management for Each Conveyor**

- **Creating Individual Properties**:
    - The script dynamically creates or retrieves properties for each conveyor to allow individual configurations. These properties include:
        - **ProductType#**: Specifies the type of product that the conveyor will produce.
        - **CloneTimeInterval#**: Sets the time interval at which the conveyor will clone new products.
        - **CloneCount#**: Tracks the number of products cloned by the conveyor.
        - **Produced#**: A boolean that indicates whether the conveyor has produced a product.
- **Assigning Properties to Conveyors**:
    - The function set_conveyor_properties assigns these properties to each conveyor.
    - For the original conveyor and each cloned conveyor, the properties are set based on the index, ensuring that each conveyor has its own set of properties for independent operation.

**2.3. Scheduled Product Cloning onto Conveyors**

- **Timing and Scheduling**:

    - The OnRun function contains an infinite loop that processes each conveyor at regular intervals.
    - It calls process_conveyor(conveyor) for each conveyor to check if it's time to produce a new product based on the CloneTimeInterval.

- **Processing Each Conveyor**:
    - The process_conveyor function compares the current simulation time with the conveyor's LastCloneTime.
    - If the elapsed time since the last clone exceeds the CloneTimeInterval, the conveyor is scheduled to clone a new product.

- **Cloning Products (Components)**:
    - The clone_component function is responsible for cloning the product and placing it onto the conveyor.
    - It clones a base component (e.g., Component1), assigns it a unique name incorporating
      the ProductType and CloneCount (e.g., Component1_1, Component2_2), and sets the ProductType property.
    - The cloned component is positioned correctly on the conveyor using transformation matrices, taking into account the conveyor's position and height.
    - The conveyor's Produced property is set to True to indicate that a product is available on the conveyor.

**3-)Pathway Area and Output Conveyor Python Scripts**

Both the **Output Conveyor** script and the **Pathway Area** scripts in the Visual Components simulation environment utilize similar code logic centered around cloning and placing components based on configuration data from an OPC-UA JSON file.**Key Similarities:**

- **Dynamic Component Cloning**: Both scripts read position and orientation data from a JSON file provided via OPC-UA. This data specifies how many components need to be created and where they should be placed in the simulation.
- **Placement Based on Configuration**: The scripts parse the JSON data to extract properties like X, Y, and Rz (rotation around the Z-axis). They then clone the original component (e.g., a conveyor or pathway area) and position each clone according to these properties.

**How They Work:**

- **Reading JSON Data**: Each script connects to the OPC-UA server and retrieves a JSON string representing the properties of the components to be created.
- **Parsing and Applying Configuration**: The scripts parse this JSON string to obtain a list of configurations. They iterate through this list to clone the required number of components.

- **Cloning Components**: Using functions like component.clone(), the scripts create new instances of the original component for each configuration entry.
- **Positioning Cloned Components**: They apply transformations to each cloned component's position matrix, setting the position and orientation based on the extracted X, Y, and Rz values.

# Connecting the variables between V.C and OPC-UA Server

Connecting variables between Visual Components and the OPC-UA server enables real-time synchronization and communication between the simulation and external systems. This allows for two types of information transfer:

- **Simulation to Server**: Variables updated within the simulation (e.g., robot positions, statuses) are sent to the OPC-UA server, allowing external systems to monitor the simulation state.
- **Server to Simulation**: Variables set or modified on the OPC-UA server can control or influence the simulation (e.g., setting targets for robots), enabling external systems to direct simulation behavior.

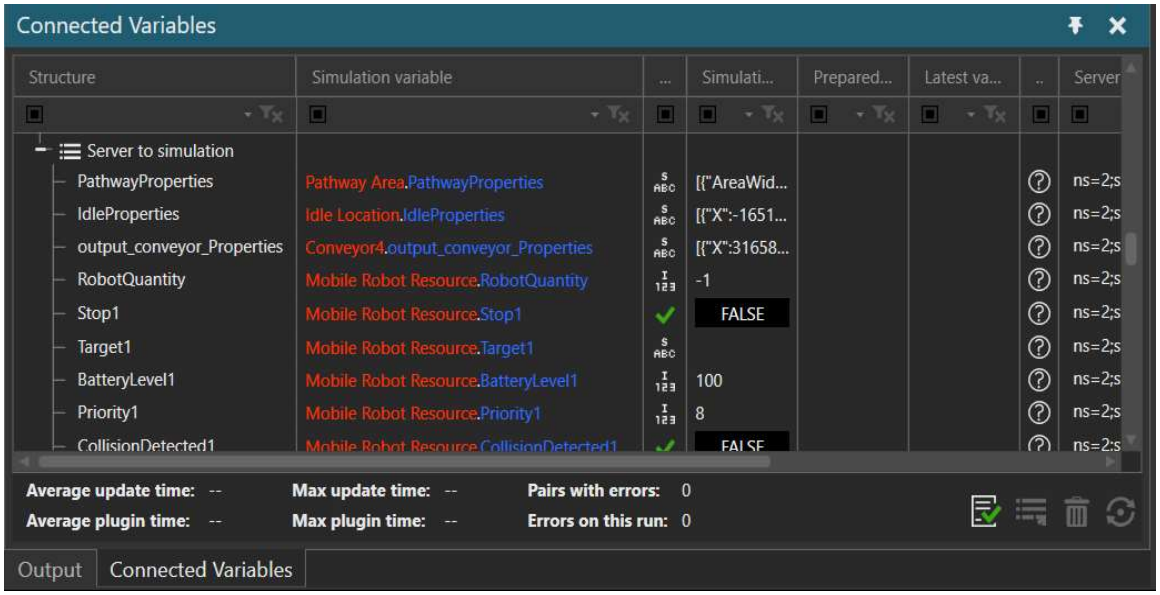
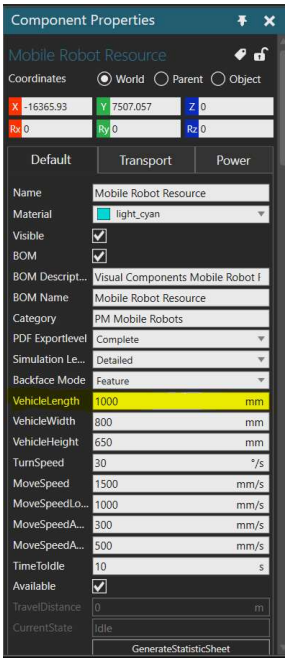



*Figure 4: Sending variables from Server to Simulation*

This bidirectional data transformation allows us to create a task flow. For instance, when input conveyors produced an item, they send a boolean signal to the OPC-UA server that the *Produced* value has become true(Simulation to Server). Then, OPC-UA checks which robot is closest to that conveyor and sends the Target information to the closest robot(Server to Simulation). As the robot move along pathways it sends its current location information to the OPC-UA server for monitoring and also for the **collision detection.**

## Example of how to connect variables between V.C and OPC-UA Server

Assume that you would like to read/set Vehicle Length of a robot from the OPC-UA server. As default property, the robots already have Vehicle Length property in their **Properties Section** in **Modeling** tab.




By default the value is set to 1000mm. Let's print that value by creating the same variable in CustomNamespace.java(where we defined all the variables). Then connect the variable in connectivity tab. Since we will not update and only monitor this value from OPC-UA, the connection type will be Simulation to Server.

To create a VehicleLength variable in CustomNamespace.java and make it available on the OPC UA server:

**1-)Class level declaration:** In CustomNamespace.java, add this variable declaration at the class level:

```
static UaVariableNode vehicleLength;
```

**2-)Declare the Variable:** In the registerItems() method, add the following code to create the VehicleLength variable:

```
UaVariableNode vehicleLength = createUaVariableNode(
    newNodeId("vehicle-length-unique-id"),    // Unique identifier
    AccessLevel.READ_WRITE,                   // Access levels
    AccessLevel.READ_WRITE,
    Identifiers.Int32,                  // Data type (e.g., Integer(Int32))
    "Vehicle Length",                   // Qualified name
    "Updating Vehicle Length",                // Display name
    "Get Vehicle Length"                // Description
);
```

**3-)Add the Variable to the Server**: Add the vehicleLength node to the folder and context so it becomes part of the OPC-UA address space:

*addNodeToFolderAndContext(folder, context, vehicleLength);*

That's it! The variable will now be visible on the OPC-UA server. You can now connect the variable from Visual Components side.

Navigate to the Connectivity tab again and right click Simulation to Server connection type:

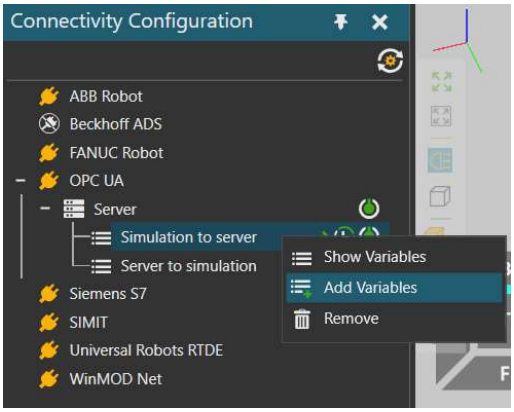



*Figure 6: Selecting the connection type for the variable in Visual Components*

After selecting the connection type, this screen will show up to find and pair your value between V.C and OPC-UA server.

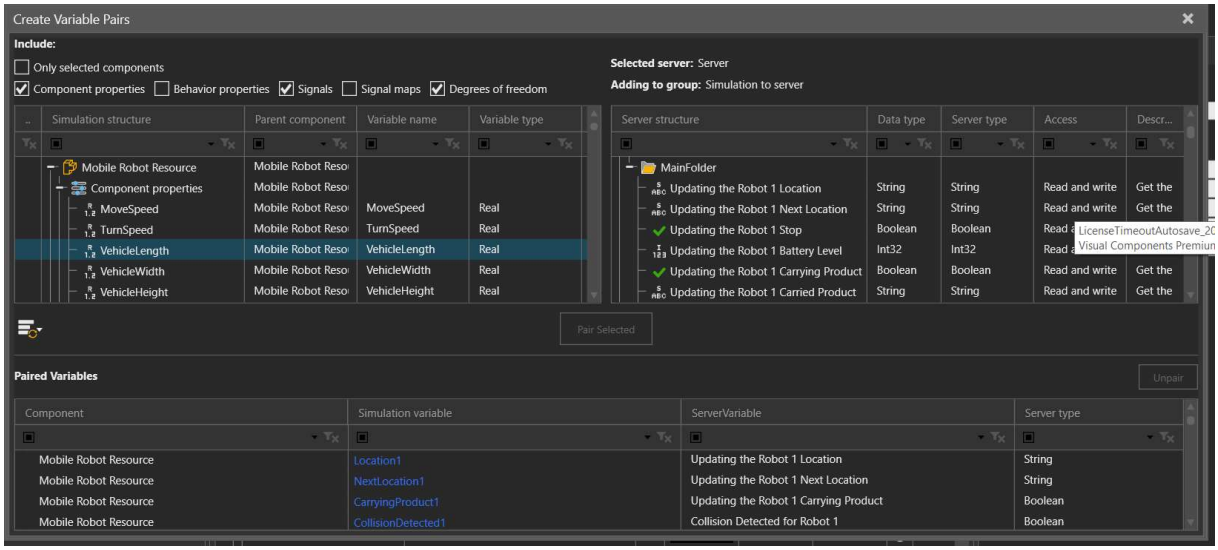



*Figure 7: Pairing the variables in Visual Components*

The list opened on the left side is your simulation environment, and the list on the right is your OPC-UA variables(that we created in CustomNamespace.java).

Find the corresponding variables on both sides, which is Vehicle Length, and click **Pair Selected.**

Finally, the connection is made and now you will be able to print out the Vehicle Length's value dynamically on OPC-UA server.

Note, if you would like to change this value from OPC-UA server, you need to connect the same variable but this time the connection type must be **Server to Simulation.**

## Final Notes

1-)The example provided to you is only a specific example for manufacturing environment. As the components in the layout are parametric, you can alter the logic(work flow) and create your own system with it.

2-)The current implementation of the project supports creating (cloning) up to 8 robots. If your system requires more robots, you will need to manually create and connect additional variables on both the Visual Components (VC) side and the OPC-UA server side to accommodate them. This limitation exists because Visual Components has a built-in connection for remote servers, and altering this system via code is not currently possible.

3-)If you notice any bugs or would like to implement a different system out of it please contact me.

4-)Make sure to not delete the template components on the layout as they have python scripts attached for making them parametric.

**Helpful resources:**

**Comprehensive API Documentation**: Visual Components offers extensive API documentation. If you are working on process modeling, you can access the documentation by pressing the F1 key within Visual Components.

**Visual Components Academy:** There is an academy that provides tutorials and resources for learning how to use the software effectively. You can visit it at https://academy.visualcomponents.com.