ALMA MATER STUDIORUM - UNIVERSITY OF BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

Department of
Electrical, Electronic, and Information Engineering
"Guglielmo Marconi"
DEI

MASTER'S DEGREE PROGRAM

AUTOMATION ENGINEERING

MASTER THESIS
in
DIAGNOSIS AND CONTROL M

Study and implementation of an URCap for coordinated external axes in a Collaborative Robot

CANDIDATE:
Nicola Maiorano

SUPERVISOR:
Prof. Ing. Andrea Tilli

CO-SUPERVISOR:
Ing. Alessandro Bini

Academic Year
2024/2025

Period
V

Outline

Abstract

In modern welding applications, precision and efficiency are crucial to achieve high-quality welds. Many welding processes require complex, multi-axis movements, where external axes and control software play a vital role. External axes are robotic systems that extend the capabilities of a primary welding robot, improving weld quality, reducing defects, and allowing more precise torch movements. Software controlling these external axes also enables dynamic adjustments of welding parameters, improving efficiency, reducing cycle times, and increasing productivity. In hazardous work environments, external axes enhance safety by reducing operator exposure to risks, as the system can stop in case of danger. Additionally, using external axes allows more complex welding trajectories, adapting automatically to surfaces with varying angles and geometries, without the need for manual intervention.

The internship was conducted at Carpano Equipment Srl, a company that collaborates with Universal Robots, a producer of Collaborative Robots (Cobots) for industrial automation. Cobots are appreciated for their innovative features, including reduced machine protections, easier operator access, intrinsic safety features, ease of programming, faster setup, and lower costs compared to traditional robots. However, for welding applications, Cobots need to be taught how to interface with welding machines, external axes, and control software for torch oscillation and arc voltage.

This dissertation focuses on developing an URCap to initialize the welding environment with an intuitive interface for operators and a framework for programmers to collect data and generate a script. The collected data is used by Universal Robots' MotionPlus technology to coordinate robot and external axis movements during welding.

***Keywords**: Collaborative Robots (Cobots), External Axes, MotionPlus, URCap, Welding*

1 Introduction

In modern welding applications, the precision and efficiency of the welding process are crucial to ensure strong, reliable, and high-quality welds. Many welding operations often require complex and multi-axis movements for precise welding. This is where external axes and specialized software for their control come into play.

External axes are robotic systems or actuators that extend the capability of a primary welding robot. These can be used for tasks like positioning the torch in various angles, adjusting its orientation, and working on large components. Therefore, using external axes improves overall weld quality, reducing the risk of defects in the final product. Precise movement and accurate positioning of the welding torch can be assured by developing software to control these external axes.

Welding processes often involve repetitive tasks with the same or similar patterns: thus, the usage of software that integrates and controls external axes can reduce downtime and the need for manual interventions. In addition, the control of external axes through software allows dynamic adjustment of welding parameters ensuring maximum efficiency, shorter cycles times and higher throughput, which are critical factors in high-volume manufacturing environments.

For applications requiring complex welding trajectories, the usage of external axes allows movements with additional degrees of freedom in comparison to the standard linear and rotational axes of the primary welding robot. By using dedicated software to control these movements, the torch can automatically adapt to different surface angles, depths, and geometries without manual repositioning.

In dangerous welding environments, external axes can also increase the safety of the work environment, reducing the exposure of the operator to dangerous conditions. The software controlling the movement of these axes can program robots to safely perform operations in potentially hazardous areas, minimizing the risk of accidents or injury. The software controlling the movement of the external axes can program robots to safely perform operations minimizing the risk of accidents or injury, stopping when sensing a potential hazard.

The internship for the thesis was carried out at Carpano Equipment Srl, a company that specializes in producing portable automation and automation accessories. To improve the welding process, in the past few years Carpano Equipment srl has started working with Universal Robots, a company that produces Collaborative Robots (Cobots) for industrial automation [9]. From the very first contact with this product, the company has appreciated its undoubtedly innovative features, such as the ability to drastically reduce machine protections and allow easier access for operators, thanks

to the numerous intrinsic safety features Collaborative Robots offer. Additionally, the company appreciated the ease and speed of programming, the reduction in setup time and the slightly lower overall costs compared to traditional robots. However, it was noted that in order to use Cobots in the welding industry, they needed to be able to interface with the main welding machine brands, integrate external motorized axes, and deploy a software for torch oscillation and arc voltage (current) control [10].

The focus of this dissertation is the development of an URCap to initialize the welding environment. An URCap (Universal Robots Capabilities) is a a type of software extension or integration developed specifically for Universal Robots (UR) robots. It is essentially an extension that adds new functionalities or capabilities to a robot's control system, allowing users to add custom features, tools, or applications. URCaps are used to integrate external devices, software, or custom tools with Universal Robots' systems [3]. The developed URCap provides the operator with a simple and intuitive interface, easy to understand and modify, while offering the programmer an infrastructure capable of collecting and processing all the data and generating a script at the beginning of the program. The data collected from the URCap is then used by the MotionPlus technology to mimic a welding process. MotionPlus is a software add-on developed by Universal Robots for their collaborative robots (Cobots), designed to improve the coordination and precision of motion between the robot and the external axes [5]. The MotionPlus technology is innovative because it allows to coordinate movement between the robot and external axes, as until now the two parts were controlled separately during welding.

The work was carried out independently, using the resources provided by Universal Robots, which were abundant on the MotionPlus side, but somewhat limited when it came to the guidelines necessary for the actual development of an URCap. Fortunately, Carpano Equipment had previously developed a few on their own, which were used as a starting point to build the final product that will be discussed in this thesis. The code was developed on the Eclipse IDE (Integrated Development Environment) within a virtual machine with a Linux-based system provided by Universal Robots, which already had the tools for compilation and deployment on the Polyscope simulation. The use of the virtual machine significantly simplified debugging sessions and accelerated any downtime that would have been caused by the startup and rebooting of the real robot while reloading the URCap.

The thesis is divided into 3 chapters, each addressing a particular aspect of the project:

- **Overview on tools and technical background** will introduce the theoretical and practical context providing the necessary foundations to understand the tools used.
- **MotionInit URCap Code Structure** will focus on the description of the developed URCap called *MotionInit*.
- **Application of MotionInit URCap for Coordinated Movement** will examine the practical experimentation on the physical robot and the test of the developed URCap to mimic a welding process.

2 Overview on tools and technical background

In paragraph 2.1, general information about Collaborative Robotics are provided. Paragraph 2.2 is dedicated to giving an overview of Polyscope, while in paragraph 2.3 URCaps are described. In paragraph 2.3.1 MotionPlus package is introduced which is one of the URCap provided by Universal Robots installed in Polyscope for the connection to the external axis of the tested robot. Finally in paragraph 2.4 a general overview of EtherCAT is provided.
The Importance of Software for Controlling External Axes in Welding

2.1 Collaborative Robotics

Collaborative robotics is a field that has seen significant development in recent decades, particularly since the 2000s, with advancements in safety technologies, sensors, and robots' computational capabilities. The main innovation brought by collaborative robotics is the idea of working safely alongside humans in a synergistic manner, rather than completely replacing them in production processes [1].

With the introduction of Cobots (Collaborative Robots), the goal was to make robotics more accessible, safe, and versatile. More specifically, Cobots are equipped with sensors and monitoring devices that stop robots in case of contact with a human, minimizing risks, a reconfigurable design easy to program, and the capability of performing a wide range of tasks. Moreover, Cobots are increasingly equipped with intuitive user interfaces, allowing even those without prior experience in robotics to program or quickly adapt to them.

In particular, Universal Robots (UR) has a key role in the Collaborative Robotics Market: being UR a pioneer on Collaborative Robots sector, the company's mission has been to create robots that could be easily integrated into existing work environments, making automation accessible for all companies [2]. These robots are increasingly used in sectors such as manufacturing, electronics, logistics, food production, and any area where automation can improve efficiency without compromising quality.

The reasons why companies, such as Carpano Equipment, choose UR Cobots are:

- **Ease of Programming**

Unlike traditional robots, which require automation experts for programming, UR robots are designed to be programmed without advanced skills. They use simple, intuitive software called Polyscope

(described in paragraph 2.2), which allows users to create movement programs by simply dragging and dropping commands.

- **Portability and Flexibility**

UR robots can be easily moved and adapted to new production lines or different tasks, making them ideal for dynamic environments where needs can change quickly.

- **Adaptability**

Thanks to their modularity and variety of accessories (such as grippers, vision systems, and transport systems), UR robots can be easily configured for a wide range of applications, from assembly to welding, from palletizing to handling delicate materials.

UR provides a range of operating Cobots, such as:

- UR3e: The smallest in the family, with a payload of 3 kg, ideal for applications in confined spaces, such as assembling small and precise components.
- UR5e: With a 5 kg payload and greater reach, versatile and suitable for a variety of tasks, such as automating workstations or integrating with other systems.
- UR10e: With a 12,5 kg payload, perfect for larger tasks, such as handling heavier objects or palletizing operations.

As a matter of fact, UR10e, which is the one represented in Figure 2-1, is the one that has been used for testing the code.

Figure 2-1 UR10e Cobot

2.2 Polyscope

Polyscope is the proprietary software developed by Universal Robots to enable easy programming and control of their collaborative Cobots, such as the UR3e, UR5e, and UR10e models. It is designed to be accessible for both robotics experts and novices, providing a simple, intuitive, and graphical interface that makes programming robots quick and efficient. Polyscope allows users to create automation tasks with minimal programming experience, making it one of the key factors in the adoption of collaborative robotics [7].

Some key Features of Polyscope

- **Graphical User Interface (GUI)**

Polyscope features a touchscreen-based interface that is intuitive and easy to navigate. The interface allows users to interact with the robot in a simple drag-and-drop style. The programming environment displays robot motions, tasks, and actions in a graphical flow, making it easy to visualize and modify the robot's tasks in real-time.

- **Teach Pendant**

The Teach Pendant is the primary control device for interacting with the robot and programming it. Using the teach pendant, an operator can manually guide the robot through the workspace (teaching) and save the robot's movements as waypoints or actions.

Figure 2-2 Teach Pendant

- **Easy Programming with Drag-and-Drop**

Programming the robot is simplified through the drag-and-drop interface. Users can select actions, such as movements, gripper functions, or even logic operations (like loops or conditional statements), and place them into a program sequence.

- **Integrated Tools and Libraries**

Polyscope comes with a rich set of integrated tools and libraries that simplify common automation tasks, such as Path Planning to guide the robot along specific paths while avoiding obstacles, grippers, or other peripherals connected to the robot, I/O functions to interface with external devices like sensors and cameras.

- **Real-Time Monitoring and Visualization**

One of Polyscope's most important features is its ability to monitor and visualize the robot's actions in real time. This is especially useful during the debugging process, allowing operators to track the robot's movements and check for any errors. Users can see a graphical representation of the robot's movement path on the interface of the teach pendant, and Polyscope will provide feedback on errors such as unexpected force readings or incorrect positioning.

- **Multi-Robot Support**

Polyscope allows for easy control of multiple robots from a single teach pendant. Users can manage several UR robots simultaneously, allowing for the automation of more complex tasks involving multiple collaborative robots working together. This feature is especially beneficial in environments that require the integration of multiple cobots, such as assembly lines or automated warehouses.

- **Remote Access and Control**

Polyscope can be connected to a network, enabling remote access and control of the robot via internet-enabled devices. This allows users to remotely monitor and program their robots from any location, granting more flexibility and ease of troubleshooting.

- **Error Handling and Safety Features**

Polyscope includes built-in safety features that ensure that robots work in compliance with safety standards. The robot's movements and operations are constantly monitored to ensure they are safe for human interaction. In case of an error, Polyscope provides clear error messages that help users quickly identify and fix problems. The software also offers options for customizing safety protocols, allowing users to define acceptable force limits, speeds, and operational boundaries.

Due to all its features, Polyscope is an essential component of UR collaborative robots, offering a straightforward, intuitive, and flexible solution for programming and controlling automation tasks. Its easy-to-use graphical interface, powerful tools, and real-time feedback make it ideal for users with little to no programming experience, empowering businesses to implement and

adapt automation quickly and efficiently. By simplifying the complexity of robot programming, Polyscope plays a crucial role in expanding the accessibility of collaborative robotics to businesses of all sizes, so that they can program and reprogram their cobots in-house, saving on costs and time.

2.3 URCaps

Universal Robots Caps (URCaps) are essential components in the ecosystem of Universal Robots' collaborative robotic arms.

URCaps are Java-based software packages designed to extend the functionality and versatility of robotic arms, allowing users to integrate additional features, tools, and capabilities into their automation processes. These software packages are developed by Universal Robots or third-party developers and can be easily installed and managed through the intuitive interface of the UR robot controller [3].

URCaps serve as a bridge between the Universal Robots' robotic arms and a range of accessories, peripherals, and software tools, enabling seamless integration and interoperability. These software packages encapsulate a wide array of functionalities, including advanced motion control algorithms, vision systems, force/torque sensing capabilities, and communication protocols. Moreover, URCaps facilitate the implementation of complex tasks such as pick-and-place operations, assembly tasks, machine tending, quality inspection, and collaborative workflows. They empower users to achieve the full potential of their robotic systems, enhancing precision, flexibility, and efficiency in various industrial and research settings.

Integrating URCaps on Polyscope, the graphical user interface (GUI) used to program and control Universal Robots' robotic arms, is a straightforward process designed to streamline customization and deployment. Within the Polyscope environment, users can easily access and manage URCaps through the dedicated interface.

Polyscope provides intuitive features for installing, configuring, and utilizing URCaps directly from the robot controller: users can navigate through the interface to browse available URCaps, select the desired packages, and install them with just a few clicks. Once installed, URCaps become integrated into the programming environment, allowing users to access their functionalities and incorporate them into their robot programs. Within the programming interface of URCaps, program nodes represent specific functionalities or operations that users can incorporate into their robot programs. Each program node corresponds to a particular task or action that the robot arm is going to

perform, such as movement commands, sensor readings, or logic decisions. Users drag and drop these program nodes onto a visual programming canvas to define the robot sequence of actions.

Furthermore, Polyscope offers comprehensive support for URCaps development, including access to development tools, documentation, and resources to create custom URCaps in order to fulfil specific application requirements. Developers can leverage Polyscope's software development kit (SDK) to build and deploy URCaps that extend the capabilities of UR robotic arms, enabling advanced automation solutions tailored to diverse industrial needs.

Overall, the integration of URCaps in Polyscope enhances the flexibility, versatility, and functionality of Universal Robots' robotic arms.

URCaps are composed of:

- **Activator**

An activator is a Java class responsible for initializing and managing the lifecycle of the URCaps bundle within the Universal Robots environment. When the URCaps bundle is loaded onto the robot controller, the activator's *start()* method is invoked, allowing it to perform tasks such as registering custom Services, setting up event listeners, or initializing resources required by the URCaps. Similarly, the *stop()* method is used to perform cleanup tasks when the URCaps bundle is unloaded.

- **Service**

Services of a URCap are components that enable interaction between the external software (URCap) and the robot's controller, providing additional functionalities and customizations. It essentially acts as the controller that initializes the configuration. Services are registered in the Activator, each with a specific purpose.

- **View and Contribution on Java Swing**

When developing URCaps using Java Swing for the user interface components, Contribution refers to the integration of custom Swing components and widgets into the URCaps interface. Developers can integrate custom View with User Interface (UI) components such as buttons, sliders, or panels to enhance the user experience and provide intuitive controls for configuring and interacting with the URCaps functionality. These Swing components can be integrated into the URCaps development environment, allowing users to access and interact with them as part of their programming workflow.

In Swing, the developer is responsible for linking the View-class (the user interface) with the Contribution-class (the controller and logic code), through public methods and callbacks associated with the various UI elements. The View calls the provider whenever it needs to fetch methods and

datas from the Contribution. Generally, the Contribution knows the valid data, and what is currently the state of the node. Whenever the openView-call happens, the Contribution calls the View, to update it to reflect the correct settings of the node. The View captures events by the user, and channels the new values into the Contribution, so these can be stored in the DataModel [4].

Whenever the user needs to insert an input, the View must be programmed to present a keyboard to the user when he clicks the input field. The interaction between the user and the input field is being notified to the system by adding a mouse listener on the input field. The keyboard is shown to the user when the View calls the *getKeyboardForInput* method through the provider from the Contribution class where it is defined. The View then displays a text input keyboard and returns an instance of itself. After the user inserts its input on the keyboard, the provider calls the *getKeyBoardCallBack* method, which is also defined in the contribution class, to save that input inside the DataModel.

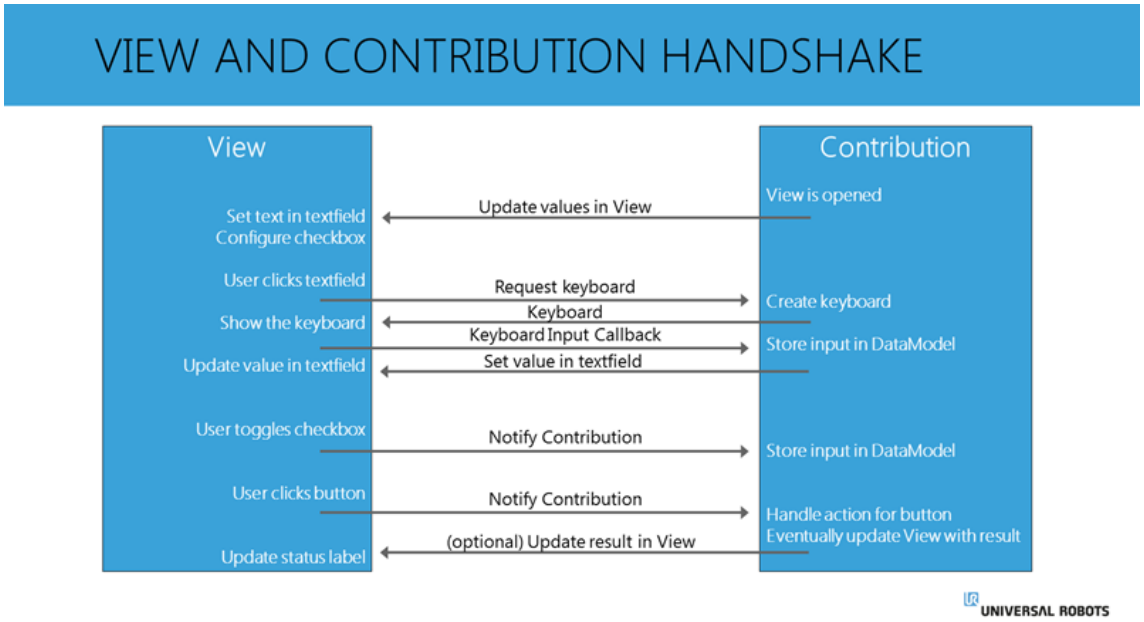

Figure 2-3 View and Contribution handshake [4]

2.3.1 MotionPlus

MotionPlus is an advanced technology developed by Universal Robots to enhance the movement control and interaction of their collaborative robots, such as the UR series [5].

The MotionPlus URCap aims to provide a simple and accurate way for customers to integrate external axes with their UR robots. An External Axis is a mechanical component that can produce motion, that has a motor controller that can accept position and/or velocity commands. An axis is “external” if it is not one of the robot’s native joints. The robot and axis group move simultaneously and complete their desired motions at the same time which is critical for part positioners used in welding.

MotionPlus incorporates force and torque sensors into the robot's end effector, allowing it to "feel" and adapt to the forces exerted on it during operation. This capability is crucial for tasks that require a high degree of sensitivity and precision, such as assembling delicate components or handling fragile objects. The sensors enable the robot to measure the forces applied during activities like pressing, lifting, or assembling, ensuring precise control to avoid damage.

MotionPlus integrates with UR Polyscope software, known for its intuitive and easy-to-use interface. The product is divided into two components:

The **Controller URScript API** and the **EtherCAT URCap** (The EtherCAT URCap also contains an EtherCAT-specific URScript API).

The Controller URScript API allows customers to build the kinematics of a group of external axes. It enables jogging of external axes with desired target positions or velocities, performing frame tracking with a moving axis, synchronizing the timing of the robot and external axes, performing coordinated motion by combining frame tracking and timing synchronization, and calibrating an axis relative to the robot. On its own, the controller generates and publishes target setpoints for the external axes, which are also published over RTDE (Real Time Data Exchange) on an internal non-public message bus. The controller relies on other components to perform the lower-level communication with hardware using the published target setpoints.

The EtherCAT URCap implements the lower-level EtherCAT communication for the controller’s target setpoints. The EtherCAT component utilizes the non-public message bus.

The MotionPlus URCap includes three sub-components: a Polyscope installation page GUI for setting up and starting EtherCAT communication, a daemon that runs alongside the UR controller

and handles low-level EtherCAT communication, and a URScript API that enables programmers to manage EtherCAT communication with external axes in their own applications.

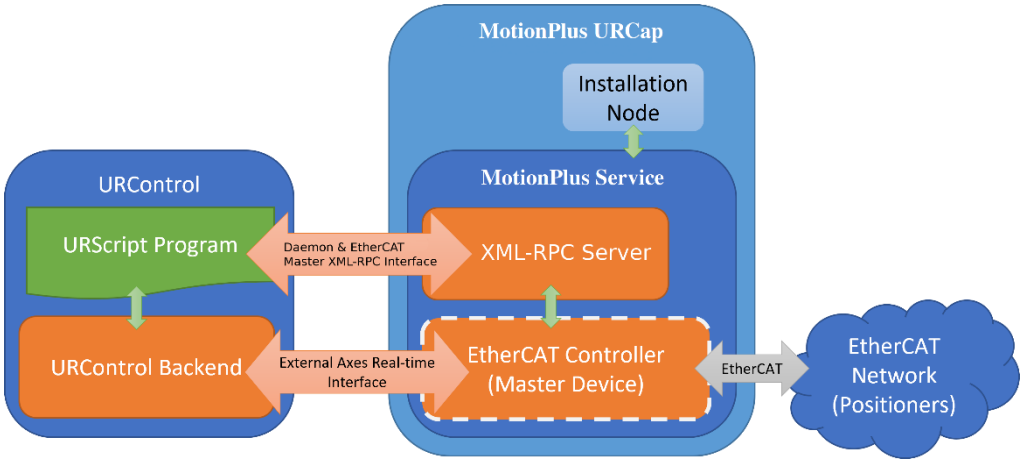

Figure 2-4: MotionPlus URcap architecture [6]

The MotionPlus URcap architecture shown in Figure 2-4 consits of:

- **MotionPlus Service Process**

It hosts an EtherCAT Master Device Implementation and an XML-RPC (Extensible Markup Language – Remote Procedure Call) server to help interface with the EtherCAT Master Device Implementation from the URScript program. The URCap provides an URScript API that interacts with the XML-RPC interface.

- **Installation Node**

It provides a GUI (Graphical User Interface) for configuring, starting, and stopping the MotionPlus Service Process.

- **EtherCAT URScript API**

It provides a URScript interface for lower-level EtherCAT functionality that is loaded into the Polyscope program preamble by the Installation Tab. The API abstracts much of the XML-RPC communication with the MotionPlus Service Process.

2.4 EtherCAT

EtherCAT (Ethernet for Control Automation Technology) is a high-performance, real-time industrial Ethernet protocol developed by the Beckhoff Automation company. It was introduced in 2003 as a solution for the limitations faced by traditional industrial communication systems, including slow communication speeds, high latency, and limited scalability.

EtherCAT is a powerful, high-speed industrial Ethernet protocol that offers numerous benefits for real-time control and automation applications. Its unique "processing on the fly" principle enables fast communication with minimal latency, making it ideal for use in industries such as manufacturing, automotive, energy, and robotics.

It operates based on a standard Ethernet physical layer but is designed to offer significantly faster communication speeds and reduced transmission delays compared to conventional Ethernet protocols. Its widespread adoption has been driven by its suitability for highly dynamic, time-critical applications in industrial automation, robotics, and motion control [8].

EtherCAT was conceived to overcome the deficiencies of earlier industrial Ethernet solutions, such as PROFINET and Ethernet/IP: while these protocols are based on standard Ethernet, they suffer from high transmission latency due to the need for each device to process the entire Ethernet frame. EtherCAT's unique approach is based on a "processing on the fly" principle, where data is processed as it passes through network devices, significantly enhancing speed and efficiency.

It utilizes a master-slave topology, with one central EtherCAT master controlling the network, and multiple EtherCAT slaves connected in a daisy-chain fashion. The key to its speed is that each EtherCAT slave device does not fully process the Ethernet frame; instead, it processes data directly while the frame is being passed through it in a "pass-through" mode referred as "processing on the fly." When an Ethernet frame is sent by the master, it sequentially travels through each slave, and relevant data is extracted or inserted into the frame before it moves to the next device. This results in lower latency and faster transmission speeds compared to traditional Ethernet-based communication systems.

EtherCAT frames are structured to minimize overhead: unlike traditional Ethernet frames, which typically include a significant amount of control information, EtherCAT frames are optimized for high-speed communication with only the necessary information. EtherCAT uses a "distributed clock" mechanism to synchronize all connected devices with microsecond precision. This ensures consistent timing across devices and allows for real-time control of industrial processes.

EtherCAT's scalability is another defining feature, allowing networks with hundreds of devices to be easily set up and managed without a significant drop in performance. The communication cycle time does not deteriorate as additional devices are added, unlike in traditional systems, where network congestion can lead to delays.

To summarize some key Features of EtherCAT are:

- **High-Speed Communication**: with a cycle time in the range of microseconds, EtherCAT allows high-frequency data exchange, crucial for real-time control applications.

- **Low Latency**: EtherCAT achieves minimal latency, allowing for the precise control of systems, especially in motion control and automation tasks.
- **Cost-Effective**: using standard Ethernet infrastructure and components makes EtherCAT a cost-effective solution for industrial automation.
- **Flexibility and Scalability**: EtherCAT's can handle networks with many devices without compromising performance.
- **Deterministic Timing**: the use of the distributed clock system ensures that all devices are synchronized.
- **Compatibility**: EtherCAT supports integration with other communication protocols, such as CANopen and PROFINET.

3 MotionInit URCap Code Structure

The MotionInit URCap is intended to configure the welding environment, presenting the operator with an easy-to-use and intuitive interface that can be easily understood and modified. At the same time, it provides the programmer with a robust infrastructure for gathering and processing all necessary data, and for generating a script at the program's initiation. The data collected from the URCap is then used by the MotionPlus technology to mimic a welding process.

The code implements a URCap system, which allows users to configure motion parameters like Gear Ratio, Counts per Revolution, and Feed Constants, select the name of the group of axis in use and the name of the singular axes which are enabled in the setup. The MotionInit URCap has been coded on the Eclipse IDE for the installation node and then deployed inside Polyscope.

The module consists of four main components: the **Activator**, the **MotionInitInstallationNodeService**, the **MotionInitInstallationNodeView**, and the **MotionInitInstallationNodeContribution**, which will be described in the next paragraphs and can be seen in Figure 3-1.

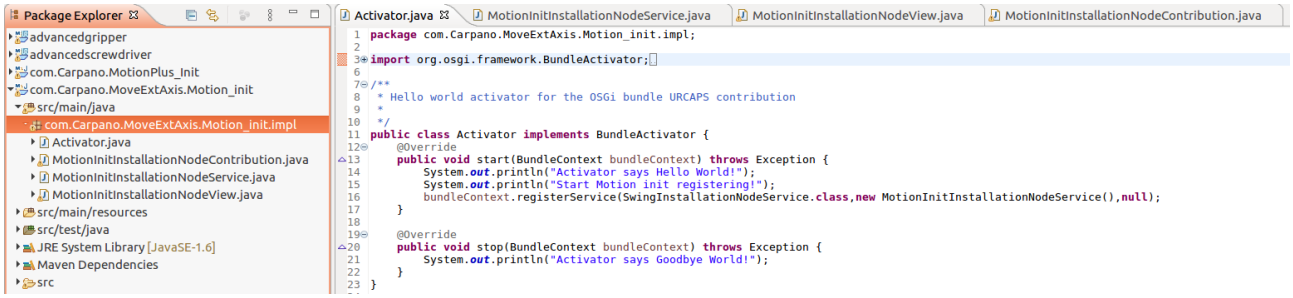

Figure 3-1 MotionInit URCap structure on Eclipse IDE

3.1 Activator

The Activator class is responsible for managing the lifecycle of the URCap within the framework. The scope of the Activator is to register the Service whenever a URCap is launched.

When the URCap is launched, the *start()* method is invoked. This method registers the *MotionInitInstallationNodeService*, which provides a custom user interface. The Service registration code line *bundleContext.registerService(...)* binds the *MotionInitInstallationNodeService* to the *SwingInstallationNodeService* interface, making it available to the controller.

Whenever the URCap ceases its activity, the *stop()* method is triggered and a message is logged on the debug screen indicating that the URCap is being stopped.

```
package com.Carpano.MoveExtAxis.Motion_init.impl;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import com.ur.urcap.api.contribution.installation.swing.SwingInstallationNodeService;

/**
 * Hello world activator for the OSGi bundle URCAPS contribution
 *
 */
public class Activator implements BundleActivator {
	@Override
	public void start(BundleContext bundleContext) throws Exception {
		System.out.println("Activator says Hello World!");
		System.out.println("Start Motion init registering!");
		bundleContext.registerService(SwingInstallationNodeService.class,new
MotionInitInstallationNodeService(),null);
	}

	@Override
	public void stop(BundleContext bundleContext) throws Exception {
		System.out.println("Activator says Goodbye World!");
	}
}
```

3.2 MotionInitInstallationNodeService

The MotionInitInstallationNodeService, which is registered in the Activator, handles the interaction between the system and the installation node. It essentially acts as the controller that initializes the configuration.
The Service code is composed of 4 methods:

1. ***configureContribution()*** defines the configuration of the Contribution for the installation node

```
package com.Carpano.MoveExtAxis.Motion_init.impl;

import java.util.Locale;

import com.ur.urcap.api.contribution.ViewAPIProvider;
import com.ur.urcap.api.contribution.installation.ContributionConfiguration;
import com.ur.urcap.api.contribution.installation.CreationContext;
import com.ur.urcap.api.contribution.installation.InstallationAPIProvider;
import com.ur.urcap.api.contribution.installation.swing.SwingInstallationNodeService;
import com.ur.urcap.api.domain.data.DataModel;

public class MotionInitInstallationNodeService implements
SwingInstallationNodeService<MotionInitInstallationNodeContribution,
MotionInitInstallationNodeView>{

    @Override
    public void configureContribution(ContributionConfiguration configuration) {
        // TODO Auto-generated method stub

    }
```

2. ***getTitle()*** defines the name of the URCap that will be displayed on Polyscope

```
@Override
public String getTitle(Locale locale) {
	// TODO Auto-generated method stub
	return "Motion Init MNG";
}
```

3. ***createView()*** establishes the MotionInitInstallationNodeView, which is responsible for managing the user interface of the installation node.

```
@Override
public String getTitle(Locale locale) {
	// TODO Auto-generated method stub
	return "Motion Init MNG";
}

@Override
public MotionInitInstallationNodeView createView(ViewAPIProvider apiProvider) {
	// TODO Auto-generated method stub
	return new MotionInitInstallationNodeView(apiProvider);
}
```

4. ***createInstallationNode()*** initializes a MotionInitInstallationNodeContribution object, which represents the logic of the contribution and binds the view and data during the creation of the process.

```
        @Override
        public MotionInitInstallationNodeContribution
createInstallationNode(InstallationAPIProvider apiProvider,
                MotionInitInstallationNodeView view, DataModel model, CreationContext
context) {
            // TODO Auto-generated method stub
            return new MotionInitInstallationNodeContribution(apiProvider,view,model);
        }

    }
```

3.3 MotionInitInstallationNodeView

The View is the user interface that allows interaction with the configuration parameters. It is responsible for building the graphical components such as text fields, labels, and checkboxes for Gear Ratio, Counts per Revolution, Feed Constant assigned to every available axis.

Following the entire code of the MotionInitInstallationNodeView is reported, but only the method that builds the User Interface is described .

```
package com.Carpano.MoveExtAxis.Motion_init.impl;


import java.awt.Component;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JCheckBox;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;
import com.ur.urcap.api.contribution.ContributionProvider;
import com.ur.urcap.api.contribution.ViewAPIProvider;
import com.ur.urcap.api.contribution.installation.swing.SwingInstallationNodeView;
import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardNumberInput;
import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardTextInput;
```

```
public class MotionInitInstallationNodeView implements
SwingInstallationNodeView<MotionInitInstallationNodeContribution> {

    private final ViewAPIProvider apiProvider;
    private int MAXIMUM_FRACTION_DIGITS = 5;
    private MotionInitInstallationNodeContribution locprovider = null;
    private boolean isInitialized = false;

    public MotionInitInstallationNodeView(ViewAPIProvider apiProvider) {
       this.apiProvider = apiProvider;
    }
```

The ***buildUI()*** method builds the interface for the user to interact with. The interface is composed of:

- checkboxes for every available axis
- text fields where the user can insert the name of the axis Group
- text fields for the name of the single available axis (which will be part of the same axis group)
- numerical fields (Gear Ratio, Counts Per Revolutes, Feed Constants) for every available axis.

Figure 3-2 shows the User Interface created by the view in the case of two available axis. The layout is constructed with vertical boxes using Swing components like JTextField and JCheckBox.

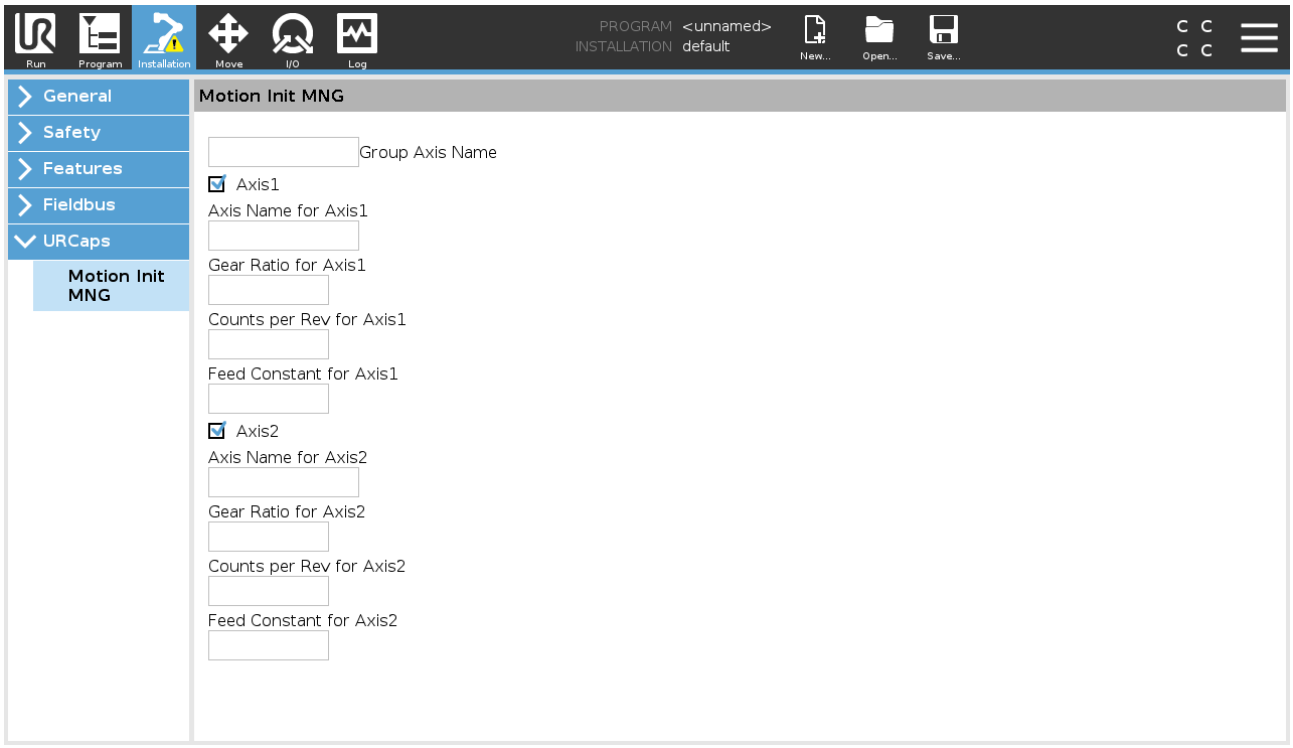

Figure 3-2 MotionInit URCap User Interface

When the user interacts with any of the fields (for instance, clicking the Gear Ratio field), a numeric keyboard will be displayed to the user to collect inputs. All these inputs will be called, used and manipulated by the Contribution.

```
        @Override
        public void buildUI(JPanel panel, final MotionInitInstallationNodeContribution contribution) {
            panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
            this.locprovider = contribution;

            // Axis Selection
            panel.add(createSpacer(10));



            boolean[] availableAxes = locprovider.getAvailableAxes();
            final JTextField axisGroupNameField = new JTextField();
            axisGroupNameField.setPreferredSize(new Dimension(150, 30));
            axisGroupNameField.setMaximumSize(axisGroupNameField.getPreferredSize());
            panel.add(createGroupAxisName(axisGroupNameField, "Group Axis Name", new
MouseAdapter() {
            public void mousePressed(MouseEvent e) {
```

```
                    KeyboardTextInput keyboardInput =
contribution.getKeyboardStringGroupAxis("GroupAxisName");
                                    keyboardInput.show(axisGroupNameField,
contribution.getCallbackTextGroupAxis("GroupAxisName", axisGroupNameField));

                    }
                    }));

                for (int i = 0; i < availableAxes.length; i++) {
                   if (availableAxes[i]) {
                      String axisKey = "Axis" + (i + 1);
                      // Creation of a checkbox for each axis
                      panel.add(createSpacer(5));


                      final JCheckBox axisCheckBox = new JCheckBox(axisKey);
                      axisCheckBox.setSelected(true);
                      final int axisIndex = i;
                      axisCheckBox.addActionListener(new ActionListener() {
                         @Override
                         public void actionPerformed(ActionEvent e) {
                            boolean isSelected = axisCheckBox.isSelected();
                            locprovider.updateAxisSelection(axisIndex, isSelected);
                         }
                      });

                      panel.add(axisCheckBox);

                      panel.add(createSpacer(5));
                      panel.add(createDescription("Axis Name for " + axisKey));


                      final JTextField axisNameField = new JTextField();
                      axisNameField.setPreferredSize(new Dimension(150, 30));
                      axisNameField.setMaximumSize(axisNameField.getPreferredSize());
                      panel.add(createAxisName(axisNameField,"", i, new MouseAdapter() {
```

```
                    public void mousePressed(MouseEvent e) {
                        KeyboardTextInput keyboardInput =
contribution.getKeyboardString("AxisName");
                            keyboardInput.show(axisNameField,
contribution.getCallbackText("AxisName", axisNameField, axisIndex));
                        }
                            }));

                // Creation of Gear Ratio, Counts per Revolute and Feed constants of each available axis
                panel.add(createSpacer(5));
                panel.add(createDescription("Gear Ratio for " + axisKey));



                final JTextField gearRatioField = new JTextField();
                gearRatioField.setPreferredSize(new Dimension(120, 30));
                gearRatioField.setMaximumSize(gearRatioField.getPreferredSize());
                panel.add(createGearRatio(gearRatioField,"", i, new MouseAdapter() {
                  public void mousePressed(MouseEvent e) {
                            KeyboardNumberInput<Double> keyboardInput =
contribution.getKeyboardNumber1("GearRatio");
                            keyboardInput.show(gearRatioField,
contribution.getCallbackNumber1("GearRatio", gearRatioField, axisIndex));
                        }
                            }));

                panel.add(createSpacer(5));
                panel.add(createDescription("Counts per Rev for " + axisKey));



                final JTextField countsPerRevField = new JTextField();
                countsPerRevField.setPreferredSize(new Dimension(120, 30));
                countsPerRevField.setMaximumSize(countsPerRevField.getPreferredSize());
                panel.add(createCountsPerRev(countsPerRevField,"", i, new MouseAdapter() {
                  public void mousePressed(MouseEvent e) {
                            KeyboardNumberInput<Integer> keyboardInput =
contribution.getKeyboardNumber("CountsPerRev");
```

```
                                keyboardInput.show(countsPerRevField,
contribution.getCallbackNumber("CountsPerRev", countsPerRevField, axisIndex));

                            }
                                }));


                    panel.add(createSpacer(5));
                    panel.add(createDescription("Feed Constant for " + axisKey));

                    final JTextField feedConstantField = new JTextField();
                    feedConstantField.setPreferredSize(new Dimension(120, 30));
                    feedConstantField.setMaximumSize(feedConstantField.getPreferredSize());
                    panel.add(createFeedConstant(feedConstantField, "", i, new MouseAdapter() {
                      public void mousePressed(MouseEvent e) {
                                KeyboardNumberInput<Double> keyboardInput =
contribution.getKeyboardNumber2("FeedConstant");
                                keyboardInput.show(feedConstantField,
contribution.getCallbackNumber2("FeedConstant", feedConstantField, axisIndex));

                            }
                                }));
                    }
                }

                // Call InitView() to setup initial values
                InitView();
            }

            public void InitView() {
                final JTextField axisGroupNameField = new JTextField();
                setValueString(locprovider.getGroupAxisName(), axisGroupNameField);
                boolean[] availableAxes = locprovider.getAvailableAxes();
                for (int i = 0; i < availableAxes.length; i++) {
                    if (availableAxes[i]) {
                        final JTextField gearRatioField = new JTextField();
```

```
            final JTextField countsPerRevField = new JTextField();
            final JTextField feedConstantField = new JTextField();
            final JTextField axisNameField = new JTextField();
            //Setup Initial Values
            setValueDouble(locprovider.getGearRatio(i), gearRatioField);
            setValueInteger(locprovider.getCountsPerRev(i), countsPerRevField);
            setValueDouble(locprovider.getFeedConstant(i), feedConstantField);
            setValueString(locprovider.getAxisName(i), axisNameField);

            Double gearRatio = locprovider.getGearRatio(i);
            if (gearRatio != 0.0) {
               setValueDouble(gearRatio, gearRatioField);
            }
            Integer countsPerRev = locprovider.getCountsPerRev(i);
            if (countsPerRev != 0) {
               setValueInteger(countsPerRev, countsPerRevField);
            }
            Double feedConstant = locprovider.getFeedConstant(i);
            if (feedConstant != 0.0) {
               setValueDouble(feedConstant, feedConstantField);
            }
         }
      }
   }


   public void setValueInteger(Integer value, JTextField Text) {
      DecimalFormat df = new DecimalFormat("#");
      String stringValue = df.format(value);
      Text.setText(stringValue);
   }

      public void setValueDouble(Double value, JTextField Text) {
            DecimalFormat df = new DecimalFormat("0");
            df.setMaximumFractionDigits(MAXIMUM_FRACTION_DIGITS);
            String stringValue = df.format((double) value);
```

```
                    Text.setText(stringValue);
                }

                public void setValueString(String value, JTextField text) {

                    text.setText(value);
                }

            private Box createDescription(String desc) {
                Box box = Box.createHorizontalBox();
                box.setAlignmentX(Component.LEFT_ALIGNMENT);
                JLabel label = new JLabel(desc);
                box.add(label);
                return box;
            }

            private Box createGearRatio(final JTextField inputField, String label, final int axisIndex,
MouseAdapter mouseAdapter) {
                Box box = Box.createHorizontalBox();
                box.setAlignmentX(Component.LEFT_ALIGNMENT);
                JLabel jlabel = new JLabel(label);

                inputField.setFocusable(true);
                inputField.setPreferredSize(new Dimension(120, 30));
                inputField.setMaximumSize(inputField.getPreferredSize());
                inputField.addMouseListener(mouseAdapter);

             // Add an ActionListener that updates the value when the user changes it
                inputField.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                        try {
                            double gearRatio = Double.parseDouble(inputField.getText());
                            locprovider.setGearRatioForAxis(axisIndex, gearRatio);  //Saves the value in the
contribution
                        } catch (NumberFormatException ex) {
                            System.out.println("Error in the value of Gear Ratio for " + "Axis" + (axisIndex + 1));
```

```
                }
            }
        });

        box.add(createSpacer(5));
        box.add(inputField);
        box.add(jlabel);

        return box;
    }

    private Box createCountsPerRev(final JTextField inputField, String label, final int axisIndex,
MouseAdapter mouseAdapter) {
        Box box = Box.createHorizontalBox();
        box.setAlignmentX(Component.LEFT_ALIGNMENT);
        JLabel jlabel = new JLabel(label);

        inputField.setFocusable(true);
        inputField.setPreferredSize(new Dimension(120, 30));
        inputField.setMaximumSize(inputField.getPreferredSize());
        inputField.addMouseListener(mouseAdapter);


        inputField.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    int countsPerRev = Integer.parseInt(inputField.getText());
                    locprovider.setCountsPerRevForAxis(axisIndex, countsPerRev);
                } catch (NumberFormatException ex) {
                    System.out.println("Error in the value of Counts per Rev for " + "Axis" + (axisIndex +
1));
                }
            }
        });

        box.add(createSpacer(5));
```

```
            box.add(inputField);
            box.add(jlabel);

            return box;
        }

        private Box createFeedConstant(final JTextField inputField, String label, final int
axisIndex,MouseAdapter mouseAdapter) {
            Box box = Box.createHorizontalBox();
            box.setAlignmentX(Component.LEFT_ALIGNMENT);
            JLabel jlabel = new JLabel(label);

            inputField.setFocusable(true);
            inputField.setPreferredSize(new Dimension(120, 30));
            inputField.setMaximumSize(inputField.getPreferredSize());
            inputField.addMouseListener(mouseAdapter);

            inputField.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    try {
                        Double feedConstant = Double.parseDouble(inputField.getText());
                        locprovider.setFeedConstantForAxis(axisIndex, feedConstant);  // Salviamo il valore
nella contribution
                    } catch (NumberFormatException ex) {
                        System.out.println("Error in the value of Feed Constant for " + "Axis" + (axisIndex +
1));
                    }
                }
            });

            box.add(createSpacer(5));
            box.add(inputField);
            box.add(jlabel);

            return box;
        }
```

```
        private Box createAxisName(final JTextField inputField, String label, final int
axisIndex,MouseAdapter mouseAdapter) {
            Box box = Box.createHorizontalBox();
            box.setAlignmentX(Component.LEFT_ALIGNMENT);
            JLabel jlabel = new JLabel(label);

            inputField.setFocusable(true);
            inputField.setPreferredSize(new Dimension(150, 30));
            inputField.setMaximumSize(inputField.getPreferredSize());
            inputField.addMouseListener(mouseAdapter);



            inputField.addActionListener(new ActionListener() {
               public void actionPerformed(ActionEvent e) {
                  try {
                     String axisName = inputField.getText();
                     locprovider.setAxisNameForAxis(axisIndex, axisName);
                  } catch (NumberFormatException ex) {
                     System.out.println("Error in the value of Axis Name for " + "Axis" + (axisIndex + 1));
                  }
               }
            });

            box.add(createSpacer(5));
            box.add(inputField);
            box.add(jlabel);

            return box;
         }

        private Box createGroupAxisName(final JTextField inputField, String label, MouseAdapter
mouseAdapter) {
            Box box = Box.createHorizontalBox();
            box.setAlignmentX(Component.LEFT_ALIGNMENT);
            JLabel jlabel = new JLabel(label);
```

```
        inputField.setFocusable(true);
        inputField.setPreferredSize(new Dimension(150, 30));
        inputField.setMaximumSize(inputField.getPreferredSize());
        inputField.addMouseListener(mouseAdapter);



        inputField.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    String axisName = inputField.getText();
                    locprovider.setGroupAxisNameForAxis(axisName);
                } catch (NumberFormatException ex) {
                    System.out.println("Error in the value of Axis Group Name ");
                }
            }
        });

        box.add(createSpacer(5));
        box.add(inputField);
        box.add(jlabel);

        return box;
    }

    private Component createSpacer(int height) {
        return Box.createRigidArea(new Dimension(0, height));
    }
}
```

3.4 MotionInitInstallationNodeContribution

The logic of the configuration resides in the Contribution: it links the Service and the View, managing the data flow and all the interactions.
It is used to define and manage the motion parameters up to six axes (2 in this specific case) in a robotic system. These parameters are essential for the robot's motion control, affecting how the robot's motors and encoders work together.
The MotionInitInstallationNodeContribution class is a critical part of the framework because it integrates user inputs through the UI, stores configuration values, and generates a corresponding script that will be executed on the robot to initialize motion settings.

The MotionInitInstallationNodeContribution code is composed by the following methods:

- ***getConfigFile()*** reads the configuration data from a CSV file located at a designated directory inside the robot. The file contains key-value pairs, where the key is the axis name (e.g., Axis1, Axis2) and the value is a boolean (True/False) indicating whether the axis is enabled or not. These Boolean values are stored inside an array of strings, which is used in the view to show only the enabled axis.
- ***getAvailableAxes()*** returns an array indicating which axes are available (enabled), based on the values stored in the array of booleans.
- ***updateAxisSelection()*** allows to update the axis selection by modifying the values in the array.
- ***"Set and Get"***

The class provides several setter and getter methods for managing motion parameters, including gear ratios, counts per revolution, feed constants, axis names, and the axis group name. These methods are essential for both retrieving and updating configuration values dynamically, jumping between View and Contribution.

- ***Keyboard Input***

The class provides methods to handle user input via the keyboard, which will be used on the View side.

- ***openView()***
- ***generateScript()*** generates a script based on the current configuration which will be placed on top of the Program, with the parameters set in the installation process. It generates lines of code that will be executed by the robot, defining things like axis names, gear ratios, encoder resolutions, and feed constants, creating a global variable for each of these parameters. It also configures the robot's axes, velocity limits, and other parameters necessary for proper motion control.

The reason why the variables are defined as global is because they can be accessed by the Program even if the script of the Installation node is generated before the start of the Program.

```
package com.Carpano.MoveExtAxis.Motion_init.impl;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;

import javax.swing.JTextField;

import com.Carpano.MoveExtAxis.Motion_init.impl.MotionInitInstallationNodeView;
import com.ur.urcap.api.contribution.InstallationNodeContribution;
import com.ur.urcap.api.contribution.installation.InstallationAPIProvider;
import com.ur.urcap.api.domain.data.DataModel;
import com.ur.urcap.api.domain.script.ScriptWriter;
import com.ur.urcap.api.domain.undoredo.UndoRedoManager;
import com.ur.urcap.api.domain.userinteraction.inputvalidation.InputValidationFactory;
import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardInputCallback;
import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardInputFactory;
import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardNumberInput;
import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardTextInput;
import com.ur.urcap.api.domain.variable.VariableException;
import com.ur.urcap.api.domain.variable.VariableFactory;

public class MotionInitInstallationNodeContribution implements InstallationNodeContribution {

    private MotionInitInstallationNodeView view;
    private DataModel model;
    private InstallationAPIProvider apiProvider;
```

```
        private KeyboardInputFactory keyboardInputFactory;
        private InputValidationFactory validatorFactory;

        // Initialization of Variables
        private Double[] gearRatios = new Double[6]; // 6 axes
        private Integer[] countsPerRevs = new Integer[6];
        private Double[] feedConstants = new Double[6];
        private boolean[] axisEnabled = new boolean[6];
        private String[] axisNames = new String[6];
        private String axisGroup = new String();


        public MotionInitInstallationNodeContribution(InstallationAPIProvider apiProvider,
MotionInitInstallationNodeView view, DataModel model) {
            this.view = view;
            this.model = model;
            this.apiProvider = apiProvider;
            this.keyboardInputFactory =
apiProvider.getUserInterfaceAPI().getUserInteraction().getKeyboardInputFactory();
            this.validatorFactory =
apiProvider.getUserInterfaceAPI().getUserInteraction().getInputValidationFactory();

            for (int i = 0; i < 6; i++) {
                this.gearRatios[i] = 0.0; // Default Gear Ratio
                this.countsPerRevs[i] = 0; // Default Counts per Rev
                this.feedConstants[i] = 0.0; // Default Feed Constant
        }

            getConfigFile();
        }

        // Read the configuration of CSV file
        public void getConfigFile() {

System.out.println("*****************************cfg*****************************");
```

```
try {
    FileReader filename = new FileReader("/home/ur/Desktop/Axis_Config.csv");
    BufferedReader csvReader = new BufferedReader(filename);
    String row;

    while ((row = csvReader.readLine()) != null) {
        String[] data = row.split("=");
        if (data.length == 2) {
            String key = data[0].trim();
            String valueStr = data[1].trim();

            if ("true".equalsIgnoreCase(valueStr) || "false".equalsIgnoreCase(valueStr)) {
                boolean value = Boolean.parseBoolean(valueStr);
                if (key.equalsIgnoreCase("Axis1")) {
                    axisEnabled[0] = value;
                } else if (key.equalsIgnoreCase("Axis2")) {
                    axisEnabled[1] = value;
                } else if (key.equalsIgnoreCase("Axis3")) {
                    axisEnabled[2] = value;
                } else if (key.equalsIgnoreCase("Axis4")) {
                    axisEnabled[3] = value;
                } else if (key.equalsIgnoreCase("Axis5")) {
                    axisEnabled[4] = value;
                } else if (key.equalsIgnoreCase("Axis6")) {
                    axisEnabled[5] = value;
                }
            }
        }
    }
    csvReader.close();
} catch (IOException e) {
    e.printStackTrace();
}

// Debug
for (int i = 0; i < 6; i++) {
```

```
                System.out.println("Axis" + (i + 1) + " state: " + axisEnabled[i]);
            }

System.out.println("*******************************cfg*****************************");
        }


        public void setGearRatioForAxis(int axisIndex, double gearRatio) {
            if (axisIndex >= 0 && axisIndex < 6) {
                gearRatios[axisIndex] = gearRatio;
                System.out.println("Set GearRatio for Axis " + (axisIndex + 1) + ": " + gearRatio);
            }
        }


        public void setCountsPerRevForAxis(int axisIndex, int countsPerRev) {
            if (axisIndex >= 0 && axisIndex < 6) {
                countsPerRevs[axisIndex] = countsPerRev;
                System.out.println("Set CountsPerRev for Axis " + (axisIndex + 1) + ": " + countsPerRev);
            }
        }

        public void setFeedConstantForAxis(int axisIndex, double feedConstant) {
            if (axisIndex >= 0 && axisIndex < 6) {
                feedConstants[axisIndex] = feedConstant;
                System.out.println("Set Feed Constant for Axis " + (axisIndex + 1) + ": " + feedConstant);
            }
        }

        public void setAxisNameForAxis(int axisIndex, String axisName) {
            if (axisIndex >= 0 && axisIndex < 6) {
                axisNames[axisIndex] = axisName;
                System.out.println("Set Axis Name for Axis " + (axisIndex + 1) + ": " + axisName);
            }
        }
```

```
    public void setGroupAxisNameForAxis(String axisName) {

        axisGroup = axisName;
        System.out.println("Set Axis Group Name for Axis " + ": " + axisName);

    }

    public Double getGearRatio(int axisIndex) {

                return gearRatios[axisIndex];
    }


    public Integer getCountsPerRev(int axisIndex) {

       return countsPerRevs[axisIndex];

    }

       public Double getFeedConstant(int axisIndex) {

       return feedConstants[axisIndex];

    }

    public String getAxisName(int axisIndex) {

       return axisNames[axisIndex];

    }

    public String getGroupAxisName() {

       return axisGroup;

}
```

```
    public boolean[] getAvailableAxes() {
        return axisEnabled;
    }


    public void updateAxisSelection(int axisIndex, boolean isSelected) {
        if (axisIndex >= 0 && axisIndex < 6) {
            axisEnabled[axisIndex] = isSelected;
            System.out.println("Axis " + (axisIndex + 1) + " selection updated to: " + isSelected);
        }
    }
    public KeyboardNumberInput<Integer> getKeyboardNumber(String Key) {
        KeyboardNumberInput<Integer> keyboard = keyboardInputFactory.createIntegerKeypadInput();
        keyboard.setInitialValue(model.get(Key, 0));
        return keyboard;
    }

    public KeyboardInputCallback<Integer> getCallbackNumber(final String Key, final JTextField Text, final int axisIndex) {
        return new KeyboardInputCallback<Integer>() {
            @Override
            public void onOk(Integer value) {
                System.out.println("Saving countsPerRev value: " + value);
                countsPerRevs[axisIndex] = value;
                model.set(Key, value);
              view.setValueInteger(value, Text);
              Text.revalidate();
              Text.repaint();
            }
        };
    }
```

```
        public KeyboardNumberInput<Double> getKeyboardNumber1(String Key) {
            KeyboardNumberInput<Double> keyboard = keyboardInputFactory.createPositiveDoubleKeypadInput();
            keyboard.setInitialValue(model.get(Key, 0D));
            return keyboard;
        }

        public KeyboardInputCallback<Double> getCallbackNumber1(final String Key, final JTextField Text, final int axisIndex) {
            return new KeyboardInputCallback<Double>() {
                @Override
                public void onOk(Double value) {
                    System.out.println("Saving GearRatio value: " + value);
                    gearRatios[axisIndex] = value;
                    model.set(Key, value);
                  view.setValueDouble(value, Text);
                  Text.revalidate();
                  Text.repaint();
                }
            };
        }

        public KeyboardNumberInput<Double> getKeyboardNumber2(String Key) {
            KeyboardNumberInput<Double> keyboard = keyboardInputFactory.createPositiveDoubleKeypadInput();
            keyboard.setInitialValue(model.get(Key, 0D));
            return keyboard;
        }

        public KeyboardInputCallback<Double> getCallbackNumber2(final String Key, final JTextField Text, final int axisIndex) {
            return new KeyboardInputCallback<Double>() {
                @Override
                public void onOk(Double value) {
```

```
                    System.out.println("Saving FeedConstant value: " + value);
                    feedConstants[axisIndex] = value;
                    model.set(Key, value);
                view.setValueDouble(value, Text);
                Text.revalidate();
                Text.repaint();
            }
        };
    }

    public KeyboardTextInput getKeyboardString(String Key) {
        KeyboardTextInput keyboard = keyboardInputFactory.createStringKeyboardInput();
        keyboard.setInitialValue(model.get(Key, ""));
        return keyboard;
    }

    public KeyboardInputCallback<String> getCallbackText(final String Key, final JTextField Text,
final int axisIndex) {
        return new KeyboardInputCallback<String>() {
            @Override
            public void onOk(String value) {
                System.out.println("Saving Axis Name: " + value);
                axisNames[axisIndex] = value;
                model.set(Key, value);
                view.setValueString(value, Text);
                Text.revalidate();
                Text.repaint();
            }
        };
    }

    public KeyboardTextInput getKeyboardStringGroupAxis(String Key) {
        KeyboardTextInput keyboard = keyboardInputFactory.createStringKeyboardInput();
        keyboard.setInitialValue(model.get(Key, ""));
        return keyboard;
    }
```

```
        public KeyboardInputCallback<String> getCallbackTextGroupAxis(final String Key, final
JTextField Text) {
            return new KeyboardInputCallback<String>() {
                @Override
                public void onOk(String value) {
                    System.out.println("Saving Axis Name: " + value);
                    axisGroup = value;
                    model.set(Key, value);
                    view.setValueString(value, Text);
                    Text.revalidate();
                    Text.repaint();
                }
            };
        }

        @Override
        public void openView() {
            getConfigFile();
            view.InitView();
        }

        @Override
        public void closeView() {
            // TODO Auto-generated method stub
        }

        @Override
        public void generateScript(ScriptWriter writer) {
            String axisGroup = getGroupAxisName();
            writer.appendLine("   global GroupAxisname = \"" + axisGroup + "\"");
            for (int i = 0; i < 6; i++) {
                if (axisEnabled[i]) {
                    Double gearRatio = getGearRatio(i);
                    Integer countsPerRev = getCountsPerRev(i);
                    Double feedConstant = getFeedConstant(i);
                    String axisName = getAxisName(i);

                    System.out.println("GearRatio for Axis " + (i + 1) + ": " + gearRatio);
```

```
                    System.out.println("CountsPerRev for Axis " + (i + 1) + ": " + countsPerRev);
                    System.out.println("Feed Constant for Axis " + (i + 1) + ": " + feedConstant);
                    System.out.println("Axis Name for Axis " + (i + 1) + ": " + axisName);

                    writer.appendLine("    global axis" + (i + 1) + " = \"" + axisName + "\"");
                    writer.appendLine("    global GEAR_RATIO" + (i + 1) + " = " + gearRatio );
                    writer.appendLine("        global  ENCODER_RESOLUTION" + (i + 1) + " = " +
countsPerRev );
                    writer.appendLine("    global FEED_CONSTANT" + (i + 1) + " = " + feedConstant);

                }
            }
            writer.appendLine("    reset_world_model()");
            writer.appendLine("    axis_group_add(\"" + axisGroup +"\", p[0,0,0,0,0,0], \"base\")");


        writer.appendLine("    axis_group_add_axis(\""+ axisGroup +"\", \""+ getAxisName(0) +"\",\"\" ,
Point_1,0,d2r(3),d2r(4.7))");
            writer.appendLine("    axis_group_add_axis(\""+ axisGroup +"\", \""+ getAxisName(1) +"\",
\"" + getAxisName(0) +"\", p[0.06844, 0.09904, 0.29675, 1.64,0.87439,-1.63615],0,0.5,d2r(47))");
            writer.appendLine("end");
          }
        }
```

4 Application of MotionInit URCap for Coordinated Movement

To show how the MotionInit URCap works, a Program has been developed in the Program section of Polyscope. This Program exploits all the components that have been described so far (URCap, EtherCAT, and MotionPlus) to achieve coordinated movement between the robot and the external axes. The program configures the robot's axes, moves the robot along defined waypoints, and manages the tracking of an object, called "pezzo" (part) in the script, throughout the process.

The MotionInitURCap is placed inside the Installation node of Polysope and it generates all the global variables before the start of the Program. Since the variables created by the URCap are global, they can be used by the Program even if they are generated before its start. Whenever the Program is started, the compiler produces a script that contains all the instructions that have been "drag and dropped" in the Program tree.

Here follows the script generated by the compiler to control the UR Cobot that uses a dual-axis movement system and EtherCAT communication for precise control of its actions.

```
Program
  Variables Setup
  BeforeStart
    var_2 = False
    var_1 = pose_trans(pose_inv(Point_1),Rotazione)
    Wait: 1.0
    MotionPlus_Init
  Robot Program
     Script: homing.script
     MoveJ
       MoveCApp
    add_frame("pezzo",p[0.768,0.619,0.467,3.73,3.27,0],"base")
    add_frame("wp1",pose_trans( pose_inv(PezzoFeature_const), MoveCApp),"pezzo")
    add_frame("wp2", pose_trans(pose_inv(PezzoFeature), MoveCStart),"pezzo")
    add_frame("wp3", pose_trans(pose_inv(PezzoFeature), Waypoint_8),"pezzo")
    add_frame("wp4", pose_trans(pose_inv(PezzoFeature), Waypoint_9),"pezzo")
    add_frame("wp5", pose_trans(pose_inv(PezzoFeature), Waypoint_5),"pezzo")
    add_frame("wp6", pose_trans(pose_inv(PezzoFeature), Waypoint_7),"pezzo")
    attach_frame("pezzo",axis1)
```

```
    attach_frame("wp1", "pezzo")
    attach_frame("wp2", "pezzo")
    attach_frame("wp3", "pezzo")
    attach_frame("wp4", "pezzo")
    attach_frame("wp5", "pezzo")
    attach_frame("wp6", "pezzo")
    frame_tracking_enable("pezzo")
    Loop
      movel_with_axis_group("wp1", a=0.1, v=0.005, name=GroupAxisname, axis_target=[d2r(0),0])
      movel_with_axis_group("wp2", a=0.1, v=0.005, name=GroupAxisname, axis_target=[d2r(0),0])
      movec_with_axis_group("wp3", "wp4", a=0.1, v=0.005, r=0.002, mode=0, name=GroupAxisname,
axis_target=[d2r(20),d2r(15)])
      movep_with_axis_group("wp5", a=0.1, v=0.009, r=0.02, name=GroupAxisname,
axis_target=[d2r(20),d2r(15)])
      movel_with_axis_group("wp6", a=0.1, v=0.008, name=GroupAxisname, axis_target=[0,0])
      movej_with_axis_group("wp1", a=0.1, v=0.08, name=GroupAxisname, axis_target=[d2r(0),0])
    frame_tracking_disable()
    ethercat_stop( True )
    If  False
      MoveJ
        MoveCApp
        MoveCStart
        Waypoint_8
        Waypoint_9
        Waypoint_5
        Waypoint_7
```

First, the script starts by setting up some variables. Specifically, var_2 is set to False, and var_1 is initialized as a pose transformation, which is based on the inverse of a point called Point_1 and a rotation matrix called “Rotazione”. These variables are used to guide the robot's movements and define its starting position.

Then, the Program controls the robot to perform the following two specific tasks:

- It issues the command *ethercat_stop(True)* to halt EtherCAT communication, which could be still running from a previous Program, to prevent any conflict.

- It triggers the *MotionPlus_Init*, that is another URCap that has been created to make the program flow lighter. It handles the EtherCAT configuration of the external axes with all the data that the user has entered in the Installation node of the Motion_Init URCap. URCaps are meant to simplify Program preparation and the MotionPlus_Init is a clear example: the operator only needs to add the URCap to the Program tree and enter the desired data, without having to worry about the complex mechanics behind the EtherCAT configuration because the programmer has already prepared a pre-packaged solution with everything the operator needs.

  Here follows the *generateScript()* method inside the Contribution of the *MotionPlus_Init*, which contains the configuration and enabling of external axes with global variables “axis”, “ENCODER_RESOLUTION”, “GEAR_RATIO” and “FEED_CONSTANT” .

```
@Override
	public void generateScript(ScriptWriter writer) {
		// TODO Auto-generated method stub
		writer.appendLine("    ethercat_clear_error()");
		writer.appendLine("    ethercat_stop(  True  )");
		    writer.appendLine("    ethercat_clear_error()");
		writer.appendLine("        ethercat_config_axis(    axis1    ,  2,  ENCODER_RESOLUTION1, GEAR_RATIO1 , FEED_CONSTANT1 , 0)");
		writer.appendLine("        ethercat_config_axis(    axis2    ,  1,  ENCODER_RESOLUTION2, GEAR_RATIO2 , FEED_CONSTANT2 , 0)");

		writer.appendLine("    ethercat_set_parameter(\"dc_enable\", True)");
		writer.appendLine("    ethercat_start(10)");

		writer.appendLine("    ethercat_enable_axis(axis1)");
		writer.appendLine("    ethercat_enable_axis(axis2)");

	}
```

Next, the Program runs the Homing script, which allows the robot to perform a homing sequence for both of the axes. Homing is a crucial step to ensure that the robot's axes are correctly

calibrated. The script takes into account the 2 global variables “axis1”, “axis2”, that contain the name of the two external axes inserted by the user.

```
Homing
    ethercat_home_axis(axis1,1,0,[1000000,1000000],[1000000],timeout=300)
    Wait: 1.0
    ethercat_home_axis(axis2,3,0,[1000000,100000],[1000000],timeout=300)
```

The *ethercat_home_axis* command moves the axes to a predefined home position and ensures that the robot knows its starting position accurately thanks to the homing sensors mounted on external axes.

After the homing phase, the Program defines and manipulates the Robot’s work area through the usage of Frames. The script creates Frames for different waypoints (e.g., wp1, wp2, wp3, etc.) as well as for the part ("pezzo"). These frames are critical because they allow the robot to understand and navigate the workspace more precisely. The *add_frame* commands relate every waypoint Frame to the *pezzo* Frame, and the *attach_frame* commands link them together, passing from the robot’s reference Frame to the *pezzo*’s reference Frame. Attaching the *pezzo*’s Frame to the "axis1" ensures that the part will move along with the rotation axis during the robot's operations.

Next, the program performs thee frame tracking, which is one of the key features of this script. The *frame_tracking_enable("pezzo")* command ensures that the robot continuously tracks the part ("pezzo") during the movements. This allows the robot to adjust its positioning if the part moves or shifts.

Afterward, the Program executes a loop, during which the robot performs various movements following different waypoints. Several motion types are used, such as:

- *movel_with_axis_group*: Linear movement between two points.
- *movec_with_axis_group*: Circular movement between points.
- *movep_with_axis_group*: Point-to-point movement with a specified radius.

These commands move the robot with a controlled acceleration (a) and velocity (v), and they are executed within the context of the defined axis group "GroupAxisname," a global variable decided by the user, which ensures that the movement is coordinated between both axes belonging to the same axis group.

Finally, the Program performs a final coordinated movement to reposition the robot using *movej_with_axis_group*. Once the robot reaches its final position, EtherCAT communication is stopped using *ethercat_stop(True)* to safely disable the control. Frame tracking is disabled thanks to *frame_tracking_disable()*.

5 Conclusions

In this thesis, the development of a URCap for initializing the welding environment has been presented, focusing on the integration of external axes with the welding robot, demonstrating the capabilities of Universal Robots' MotionPlus technology. This work aimed to provide an intuitive and user-friendly interface for operators while offering a robust infrastructure for programmers to manage the welding process effectively. The development of the MotonInit URCap has demonstrated significant improvements in the precision and efficiency of welding applications, with reduced setup times and optimized control of the welding parameters.

The integration of external axes with the main welding robot has proven to be a valuable tool for increasing the flexibility of welding movements, allowing for more complex and precise trajectories. The developed software not only simplifies the interaction between the operator and the system but also enhances the safety of the work environment, minimizing the risk of accidents or injuries in hazardous conditions.

The experience gained during the internship at Carpano Equipment Srl, particularly with the collaboration with Universal Robots, highlighted the advantages of collaborative robots (Cobots) in the welding industry. The ease of programming, coupled with the integration of external motorized axes, represents a significant step forward in optimizing industrial welding processes, providing better throughput and reducing overall operational costs.

Future work could focus on further optimizing the flexibility of the Programs, creating custom URCaps able to simplify even further the operator work. Additionally, the functionality of the Program could be extended to allow for more advanced real-time adjustments based on the welding process leading to even more efficient production cycles.

References

[1]. Swapnil Patil, V.Vasu and K.V.S. Srinadh (2023). Advances and perspectives in collaborative robotics: a review of key technologies and emerging trends https://link.springer.com/article/10.1007/s44245-023-00021-8

[2]. Universal Robots - https://www.universal-robots.com/it/

[3]. Universal Robots - https://www.universal-robots.com/articles/ur/urplus-resources/urcap-basics/

[4]. Universal Robots - https://www.universal-robots.com/articles/ur/urplus-resources/urcap-choosing-swing-or-html/

[5]. Universal Robots - https://docs.universal-robots.com/motionplus/mp1.1/index.html

[6]. Universal Robots - https://docs.universal-robots.com/motionplus/mp1.2/urcap.html

[7]. Universal Robots - https://www.universal-robots.com/products/polyscope-5/

[8]. EtherCAT Technology Groug - https://www.ethercat.org/en/technology.html

[9]. Carpano Equipment Srl - https://www.carpano.it/it/#aboutus.

[10]. Cobot 2023 - https://www.carpano.it/wp-content/uploads/2023/09/COBOT-2023-ITA.pdf.

Acronyms

API	Application Program Interface
COBOTS	Collaborative Robots
GUI	Graphical User Interface
IDE	Integrated Development Environment
SDK	Software Development Kit
UI	User Interface
UR	Universal Robots
URCaps	Universal Robots Capabilities

List of Figures