**Technische Universität Dresden**

**Chair for Compiler Construction**

# Bachelor's Thesis

# Lingua Franca in Robotics

**Submitted by**
**Benedict Mehnert**

**First Corrector:**
Prof. Jeronimo Castrillon

**Second Corrector:**
Prof. Roberto Calandra

[July, 2024]

# Contents

# 1 Introduction

The field of robotics is advancing rapidly. Robots require increasingly adaptive and safe software frameworks to handle complex tasks in real-time environments. One such framework is Lingua Franca (LF), a coordination language designed for implementing real-time and cyber-physical systems. Lingua Franca combines deterministic execution, robust concurrency support and scalability. In this thesis we will explore whether Lingua Franca is suitable in robotic applications where precision, reliability and efficiency are of importance. To highlight Lingua Franca's advantages, we will integrate a camera with a robotic arm. The demo will feature the arm tracking an object seen by the camera, demonstrating LF's real-time performance. The Lingua Franca network will dynamically process image, robot, and state data to generate control output for the robot. In the end, we will compare Lingua Franca to ROS 2 (Robot Operating System 2) in the context of robotic applications.

# 2 Background

## 2.1 Lingua Franca

While nondeterminism can be useful, the majority of computational tasks widely profits from repeatable behavior, especially when it comes to debugging, testing and understanding the code. Lingua Franca preserves determinism by default and only allows non-determinism if explicitly introduced by the developer. A logical timeline is used to order events and ensure deterministic execution. Lingua Franca is a polyglot coordination language. It is designed to coordinate the execution of mainstream target programming languages like C++, C or Rust. It equips those target languages with a deterministic concurrency model. The LF compiler synthesizes plain target program code. This code then is compiled using standard tool chains. Since Lingua Franca coordinates and declares explicitly data dependencies between reactors, independent reactions can be executed in parallel[13].

Lingua franca specifies the interactions between components called reactors. A reactor can be described as a deterministic actor with a discrete-event execution semantics. It has explicitly declared ports and connections. Reactors are comparable to classical object oriented classes, encapsulating state and methods. They offer a form of inheritance and are parametrized at instantiation. But contrary to the classical object oriented paradigms, methods cannot be invoked by other reactors, they only serve to manage code inside a reactor. Triggering functionality of different reactors is only possible by emitting events. A reactor is comparable to a software component that sends messages, with these messages typically carrying values that are then passed to the reaction. Lingua Franca programs usually consist of a reactor network. The LF diagram generation tool visualize the network which helps developers.

In Lingua Franca, all computation is performed in reactive code segments called reactions. Reactions are implemented in the target language. They must explicitly declare their triggers, dependencies, and potential effects. The reaction may access the reactors state or schedule events via logical actions or output ports addressing subsequent reactions in the same reactor or in different reactors. A reaction is seen as logically instantaneous to its triggering event. If two reactions of the same reactor triggered at the same logical time are accessing the reactor state, the reactions are invoked after each other. Interaction between reactors in Lingua Franca occur through event emissions via ports. Each event that is invoked bears the same timestamp as its triggering event and be associated a specific trigger object. In addition, events can carry objects that can be passed to reactions. As reactions invoke downstream reactions, each reaction can be assigned a level. Dependencies are determined by the LF dependency analysis. All reactions within the same level can be executed safely in parallel. LF threads, called workers, are managed by the runtime environment, which maps reactions to workers. The number of workers

is predefined by the developer. The scheduler implementation schedules the next level of reactions when the current level of execution is finished. This introduces a key feature and strength of Lingua Franca, it promotes deterministic behavior while still allowing concurrency.

The reactor model introduced so far is purely instantaneous. But a key feature of the model is its semantic notion of time, differentiating between two variants: the *logical* and the *physical* time. Physical time should be regarded as the 'real' time, while the logical time is a system-internal concept introduced to reason deterministically about the execution order of events. The logical time 'chases' the physical time. The runtime only processes the events associated with a certain tag once the current physical time exceeds the time value of the tag. This allows for setting deadlines. The interaction with the physical time is however also possible under explicitly defined semantics[13].

The event-driven architecture enables responsive behavior to sensor inputs, commands or environmental changes. Hardware interface reactors can encapsulate interfaces to sensors and actuators, transmitting joint data and sensor data into an LF network. Control reactors react to input and generate control output. In the context of robot control, Lingua Franca offers advantages for managing complex robotic systems.

## 2.2 Robot Control

Effective robot control typically includes several key components. Data gathering, detecting objects and targets is achieved by sensors. Based on the sensed information, robots make decisions, this includes path planning and responding to changes. Once path planning is finished, robots execute the movement or task using actuators such as motors. With continuous feedback, robots adjust their actions based on real-time data.

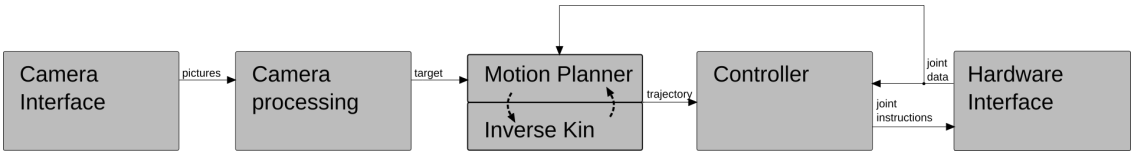


Figure 1: Classical ROS2 robot control flow[7]

**Motion planning** is a computational problem that involves determining a sequence of valid configurations to guide the robot to its target location[24]. Robot kinematics examines the relationship between the position, velocity, and acceleration of each link in a robotic system[26]. The process of computing joint parameters for a specified position of the end-effector is known as **inverse kinematics**.

The robotic demo built within this thesis integrates a camera, the XArm 7 robotic arm and Lingua Franca into a real-time robot application. The robotic arm will

attempt to trace an object seen by the camera. A typical robot control flow for this setup is shown in Figure 1. The Camera processes image data to identify and localize a target, this target is sent to the 'Motion Planning' component. Given the position of the target object and the current joint configuration, the target joint position of the end-effector of the robotic arm can be calculated via inverse kinematics. With this target joint position and the current position of the robotic arm, a complete joint trajectory mapping joint position output to time is transmitted to the controller. The 'Controller', shown in the Figure 1, directly regulates the joint positions usually with PID controllers as it follows the given trajectory.

## 2.3 XArm 7

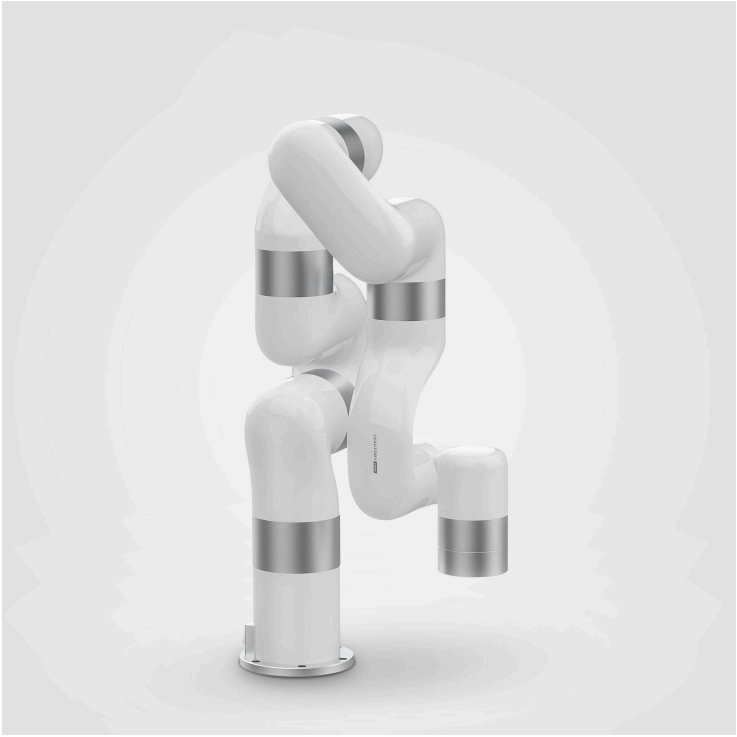

Figure 2: The XArm7, 7 degrees of freedom, partially made of carbon, load capacity of 3.5 kg and repeatability of $\pm 0.1$ mm

The robot used within this thesis is a robotic arm. The XArm 7 is produced by UFACTORY[21]. It is capable of performing a variety of tasks, from holding a camera to precisely packaging an object. The arm has seven joints promoting seven degrees of freedom. The system consists of the robotic arm mounted on a table and a control box. The control box serves as the interface between the robot and a computer and can be connected to the computer via a LAN cable. An emergency stop button is installed on the control, instantly stopping all activities of the robot in case of an emergency. The last joint is known as the tool side. It can be used to

connect the end-effector. UFACTORY provides a vacuum gripper and a classical gripper, but custom end-effectors can also be mounted. The robot features collision detection. If the torque deviation of a joint exceeds the normal range during arm movement, the arm will automatically stop to prevent damage. Besides sending commands to the robot via an SDK, the XArm can also be controlled with Ufactory Studio, a graphical user application for user without programming background. It incorporates various modes such as 'Live Control', which allows users to adjust the posture of the robot arm. Track recording is also featured, as well as 'Blockly', enabling users to program the robot by simply dragging and dropping code blocks.

The robot implements several sorts of motion modes. The **servo motion** mode will be the mode through which the robot is controlled using Lingua Franca. This thesis focuses exclusively on positional control. In this mode, the robot reaches the specified Cartesian position with maximum velocity and acceleration. Since velocity and acceleration are at their peak, precise robot control is mandatory to prevent overloading the robot joints with too much force. Commands are not buffered and only the last received target point is executed. In this mode, the robot ought to be controlled using motion planning. The maximum receiving frequency of the control box is 250 Hz. Rather than transmitting only the target position, the robot is provided with intermediate positions along the path to the target at a high frequency. With each cycle, the robot makes a small step towards the target. Further modes are **joint velocity control**, where angular velocities are provided to the robot. Or **cartesian velocity control** mode, instead of specifying end-effector positions, the end-effector velocity vector is provided to the robot[21].

## 2.4 Lingua Franca Control

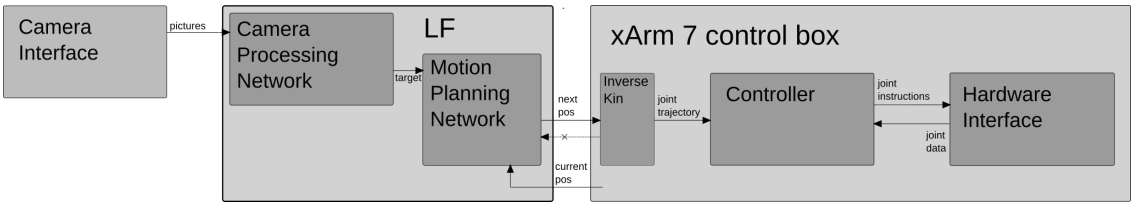


Figure 3: The Lingua Franca control approach

The control approach promoted within this thesis will differ from the control flow introduced in Section 2.2. Instead of transmitting timestamped trajectories, the Lingua Franca networks shown in Figure 3 will produce robot end-effector configurations, consisting of the position in cartesian space and the end-effector's roll, pitch, and yaw, periodically at a high frequency. Every cycle, the network will fetch state data from the robot and the camera to produce the control output. To translate the end-effector position to joint motions, the network relies on the XArm 7 control box to perform the inverse kinematics and joint position control.

In order to track the object, the color and depth frame provided by the camera are fed into the Lingua Franca network. The 'Camera Processing Network' processes and interpretes the depth and color images and transmits the target to the 'Motion Planning Network'.

The 'Motion Planning Network' is described in detail in Section 3.2 and is the main control component designed within this thesis. It consists of many Lingua Franca reactors. As highlighted in Figure 3, this network takes camera data and positional joint data as input and determines the next cartesian position of the end-effector and its roll, pitch and yaw in real-time.

## 2.5 ROS

A very established framework in robotics is ROS2 (Robot Operating System 2). It is an open-source meta-operating system designed for robots. It offers a framework for developing, managing, and deploying robotic applications. ROS 2 provides support for a wide range of hardware platforms and software architectures. At its core, it simplifies robotics development through abstracting low-level device interfaces providing essential libraries and tools for implementing commonly-used functionalities such as motion control, perception and navigation. Message passing facilitates the communication between modular components (nodes). This allows the development and maintaining of modular and reusable software components. Tools allow runtime parameter configuration or visualization of network and robot.

Nodes are a fundamental unit of computation in ROS2. Each node serves a specific purpose, ranging from data publication and subscription to providing services and managing system parameters and lifecycles. Publisher Nodes can publish messages to a specific topic. This could include publishing camera sensor data or joint sensor data to a topic for other nodes to consume. A Subscriber node could receive and process camera images from to perform object detection. Service server nodes provide services that can be called by other nodes to execute specific tasks like calculating the inverse kinematics. Service client nodes can call services provided by other nodes to request a specific task. Parameter nodes manage and share configuration parameters across the ROS 2 system such as sensor calibration values. Controller nodes implement control algorithms to manage actuators based on sensor data. For example, a node computing motor commands based on sensor feedback to control a robotic arm's joint position[15] is a Controller node. Often nodes are a complex combination of some or all different node types.

Nodes mainly communicate through a publisher-subscriber model. Communication means are topics, services, actions and parameters. Services allow nodes to send requests and receive responses. Actions are similar to services, but are designed for long-running tasks that may provide periodic feedback and have a goal completion status. Parameters allow storing and retrieving configurations settings dynamically during runtime. This forms a distributed system for controlling robotic system.

### 2.5.1 Resource Manager

The Resource Manager plays a critical role in managing the allocation and utilization of hardware resources within a robotic system. It abstracts physical hardware and its drivers for the ROS 2 control framework. It loads the components, manages their lifecycles, states and command interfaces. This allows reuse of implemented hardware components. It oversees the allocation and utilization of hardware resources such as CPU cores, memory, but also sensors and actuators. This also includes that critical processes receive adequate resources, e.g. ensuring that control loops and sensor data processing receives sufficient CPU time, in real-time and safety critical applications.

### 2.5.2 Controller Manager

The Controller Manager (CM) connects the controllers and hardware abstraction sides of the ROS 2 control framework. It translates between high-level control commands and low-level hardware-specific instructions. The CM manages controllers (e.g. loading, activating, deactivating, unloading) and their required interfaces. It gives controllers access to hardware via the Resource Manager when activated. Or it reports an error if there is an access conflict and manages the loading and unloading of controller plugins. When the system parameters change over time, the controller manager allows passing parameters and configuration settings to controllers allowing dynamic adjustment.

### 2.5.3 Controller

Controllers are software components responsible for managing and executing control algorithms that regulate the behavior of robotic actuators. They play a crucial role in translating high-level commands or desired states into low-level signals that drive actuators such as motors or joints. Position controllers ensure actuators achieve and maintain specific positions, often using PID algorithms. Other types include velocity controllers and effort/torque controllers. Trajectory controllers execute complex motion trajectories by interpolating between key points. Controllers are typically implemented as ROS 2 nodes or components, subscribing to sensor data topics and computing control commands based on desired goals or trajectories.

### 2.5.4 Hardware Components

The hardware components realize communication to physical hardware and represent its abstraction in the framework. There are three basic types of components: The 'Actuator', a simple robotic hardware component like motors or valves with 1 DOF. The 'Sensor', related to a joint (such as an encoder) or a link (like a force-torque sensor) and only capable of reading data. And the 'System' component, which stands out from the Actuator component by using complex transmissions through a single logical communication channel[4].

### 2.5.5 Hardware Description in URDF

The ROS2 framework uses URDF(Unified Robot Description Format, XML) files to describe its components, i.e. the physical configuration of the hardware setup. The URDF files are essential for simulation and visualization, defining the dynamic and kinematic properties of the robot. Controllers use this information to compute necessary actions to achieve desired states of trajectories. The URDF files also include collision geometry. Important components of URDF are **links**, representing rigid bodies with specific shapes, sizes and physical properties. There are also **joints** (revolute, prismatic, fixed, etc.), specifying the type of motion allowed between links describing the axis of rotation of translation and any limits on the motion. **Transmissions** connect actuators to joints and specify how motor commands translate into joint movements. This allows for integrating camera data into the system[9][4].

We have established the groundwork of ROS2. Before, we introduced the essential aspects Lingua Franca. We can proceed to the practical implementation. The next part of this thesis illustrates the development of a robotic application in Lingua Franca, showcasing the integration of the XArm 7 robotic arm and real-time camera data to achieve precise control and object interaction.

# 3 Design and Implementation

## 3.1 Interface to the Robot

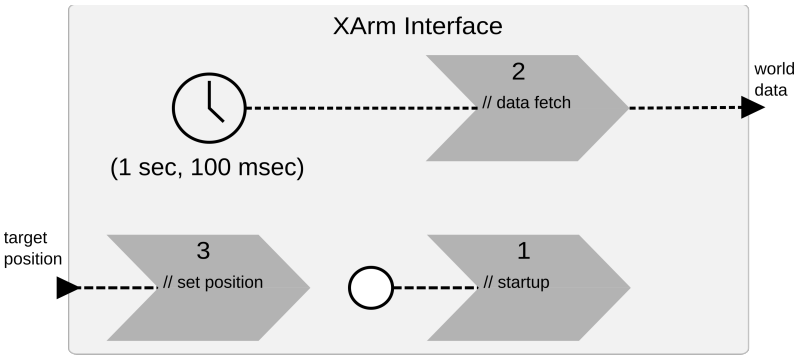


Figure 4: XArm Interface reactor, the reaction triggered by the timer fetches data from the robot, whereas reaction 3 sets the cartesian position

To interact with the robot, UFACTORY provides an SDK for C++. The SDK can be accessed within a Lingua Franca reactor. The reactor that will communicate with the robot is the 'XArm Interface' reactor shown in Figure 4 and further detailed in Figure 5. Its purpose is to send commands to the robot and to receive positional data from it. In addition to that, it is connected to the Motion Planning Network introduced in 2.4. It sends positional data to the network and receives control output which the reactor forwards to the robot. It contains three reactions, represented by the gray chevrons in Figure 4. Also shown in the reactor diagram are input and output diagrams and the timer specifying the period at which the Lingua Franca network fetches data.

The data transfer within the Motion Planning Network highlighted in Figure 3 consists in parts of objects of the class `WorlData` encapsulating positional and joint data from the robot. As of now, the network only utilizes positional and rotational data from the end-effector, without examining the remaining factors such as joint angles, torques, and velocities stored in `WorlData` objects.

The startup reaction, which is invoked once at the beginning of the program, initializes the state variable `XArmAPI` which will serve as the interface and allows to communicate with the control box. The startup function also prepares the robot for movement, as illustrated in lines 12 and 13, by configuring the motion mode to servo motion control and setting the state to `motion`.

The second reaction to the timer fetches positional and joint data. It initializes the procedure of the determination of the next position given the current data by sending a `WorldData` object to the Motion Planning network via the output port as shown in line 24 of Figure 5.

```
reactor RoboXArm7 (timer_start_moving: time = 1s, // .. further parameters ...// {
  input robo_next_position_checked: Position;
  output interface_world_data: WorldData;
  timer t(timer_start_moving, timer_initiate_movin_period);
  state arm: {=XArmAPI*=};

  reaction(startup) {=
    // startup reaction code

    this->arm = new XArmAPI(host);
    // .. //
    arm->set_mode(1); // setting servo-joint control mode
    arm->set_state(0); // setting moving state
    =}

  reaction(t) -> interface_world_data{=
    // data fetch reaction code

    fp32 pose[6] = {0};  fp32 angles[7] = {0}; fp32 velocities[7] = {0}; fp32 effort[7] = {0};
    arm->get_position(pose);
    arm->get_joint_states(angles, velocities, effort);
    WorldData data = WorldData data{
      get_elapsed_physical_time(),
      std::vector<double>(std::begin(pose), std::end(pose)),
      // .. asngles, velocities and effort is passed accordingly //
    };
    interface_world_data.set(data);
    =}


  reaction(robo_next_position_checked) {=
    // set position reaction code

    Position position_next_position = *robo_next_position_checked.get();

    fp32 arr_next_position[6] = {
      position_next_position.X_,
      position_next_position.Y_,
      position_next_position.Z_,
      default_roll,
      default_pitch,
      default_yaw};

    int error_code = arm->set_servo_cartesian(arr_next_position);

    if (error_code != 0) {
      std::cout << "Error occured while calling set_servo_cartesian" << std::endl;
    }
    =}
```

Figure 5: The XArm Interface reactor code

The third reaction receives the control output. With the positional control commands provided, this reaction calls `set_position()` on the XArmAPI. This reaction is logically instantaneous to reaction 2, since that there are no delays in the reactor network. For every data fetch reaction, a corresponding set position reaction follows. Fetching data at a high frequency here allows precise control and smooth movement.

## 3.2 Motion Planning Network

The Motion Planning Network performs real-time motion planning. It produces the intermediate positions along the path to the target taking the current positional data of the robot into consideration. It also traces the taken path and applies sanity checks at runtime. Each time positional data is fetched and transmitted to the network, it responds through the interface reactor displayed in Figure 6. The

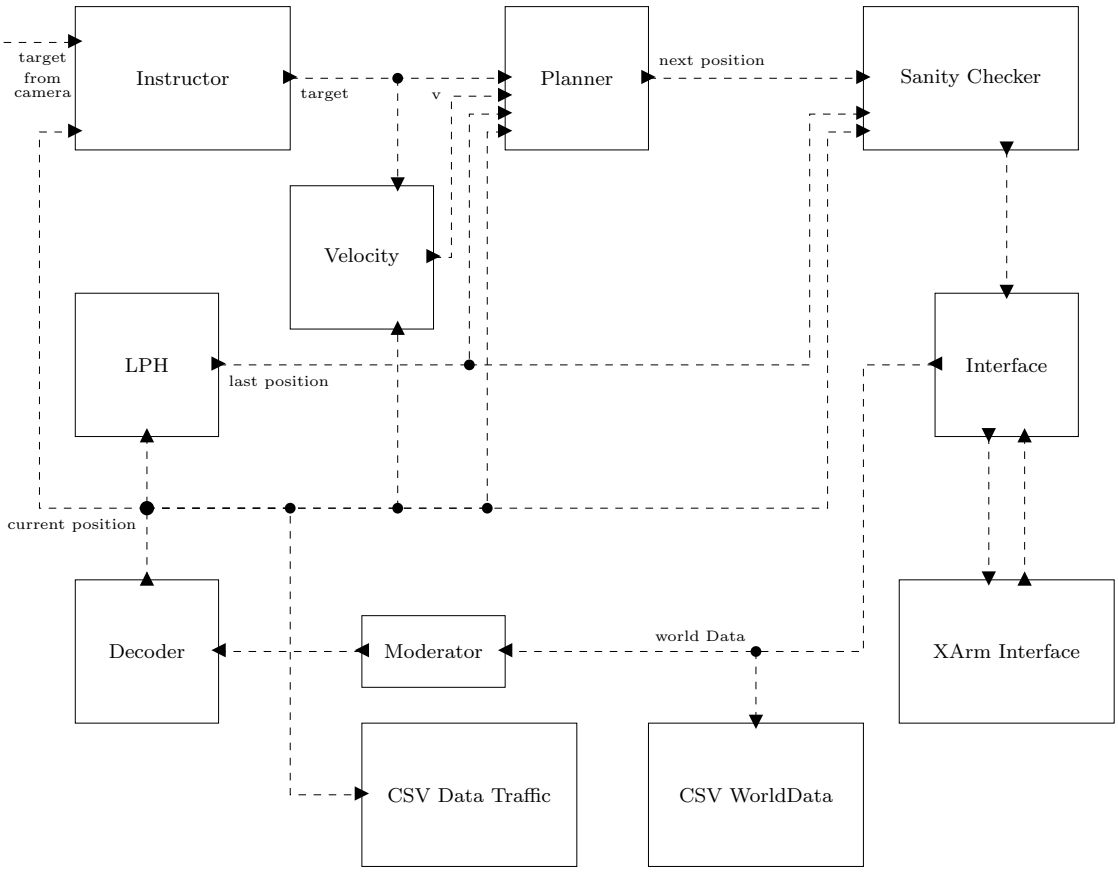


Figure 6: Lingua franca Motion Planning reactor network

response is the next intermediate position the robot is instructed to move to within this cycle and is transmitted via the XArm Interface reactor to the robot.

Data exchange within the Motion Planning Network mainly involves objects from the `Position` class, which serves as a basic container for Cartesian coordinates (x, y, and z) representing positions and the roll, pitch, and yaw of the end-effector. Define $\vec{p}_c[n]$ as the current position of the robot's end-effector at the point $n$ in logical time. Let $\vec{p}_{la}[n] = \vec{p}_c[n-1]$ denote the current position object from the previous iteration, represented as 'last position' in Figure 6. Furthermore, $\vec{p}_t[n]$ refers to the overall target target position, symbolized as 'target position' in the diagram. Let $\vec{p}_n[n]$ refer the next position in cartesian space on the path to the overall target position the robot will take on within the next cycle.

The Motion Planning Network consists of many reactors. The XArm Interface reactor just introduced is shown at the right hand side of Figure 6. Once the data is fetched and stored in an `WorlData` object, it is sent to the 'Interface' reactor. This interface does not implement any logic and as the name implies: It is the interface to the reactor network and allows connecting the network to a test reactor

or a simulation[1]. The 'CSV WorldData' reactor serves as data tracing network component. In the reactor the data is written to a `.csv` file for later analysis. The 'Moderator' reactor validates robot data output. As not all data within the `WorldData` object is relevant for the path planning cluster, the 'Decoder' specifically retrieves the `Position` object $\vec{p}_c[n]$ and sends $\vec{p}_c[n]$ to several different reactors. The 'Last Position Holder' (LPH) contains $\vec{p}_{la}[n]$ as a state variable. When its input port is set with $\vec{p}_c[n]$, its send $\vec{p}_{la}[n]$ to the 'Planner'. After that, the LPH overwrites its state variable $\vec{p}_{la}[n]$ with the recent $\vec{p}_c[n]$. The 'Instructor' reactor stores the current overall target position. In my application, the target is determined through the interpretation of camera image data, as described in Section 3.5. Once the input port of the instructor is set with $\vec{p}_c[n]$, it transmits the target position to the Planner. The 'Velocity' reactor coordinates the velocity of the robot, a proportional velocity controller here turned out to be working well in our use case, but one could consider applying more sophisticated controllers. The Planner takes the $\vec{p}_c[n]$, $\vec{p}_{la}[n]$, $\vec{p}_t[n]$ and the designated velocity $v[n]$ as input and calculates $\vec{p}_n[n]$. The 'Sanity Checker' receives the $\vec{p}_c[n]$ and $\vec{p}_n[n]$ and verifies $\vec{p}_n[n]$. It checks whether the planned movement adheres to the safety bounds. If it detects malfunctioning behavior like too fast deceleration, it will initiate a emergency stop. The robot will then execute a maneuver of deceleration just following its current direction and deceleration. In the default case of no emergency stop, the sanity checker just passes the next position to the interface which forwards it to the XArm control box.

The Planner plays a central role, as it performs the calculation of the control output. With the input of $\vec{p}_c$, $\vec{p}_{la}$, $\vec{p}_t$ and the current velocity $v[n]$, it determines the `Position` object $\vec{p}_n[n]$ referring to the position the robot will take on within the next control cycle.

## 3.3 Trajectory Planning

Our main goal is to determine a smooth path. Smooth means that only little acceleration is applied or from one iteration to another, the difference of the velocity vectors is only marginal. The force applying to the robot increases with the acceleration, the robot performs. Let $a_{max}[\mathrm{m/s^2}]$ denote the maximum acceleration.

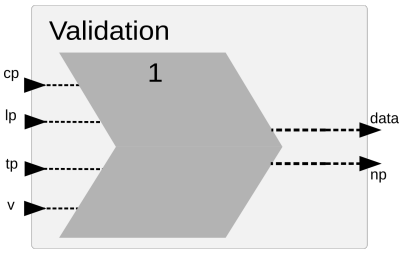

Figure 7: Planner

Let $\vec{r}(n)$ denote the position vector dependent on time which shows the robots head's location in space at a given point $n$ in logical time. The origin of the coordinate system is where the robot is securely mounted to the table. Let $T$ denote the constant period at which the XArm Interface reactor retrieves positional data from the robot. The velocity in the discrete-

[1]Alongside my work on the robot, a simulation of the robot was implemented, compare [20].

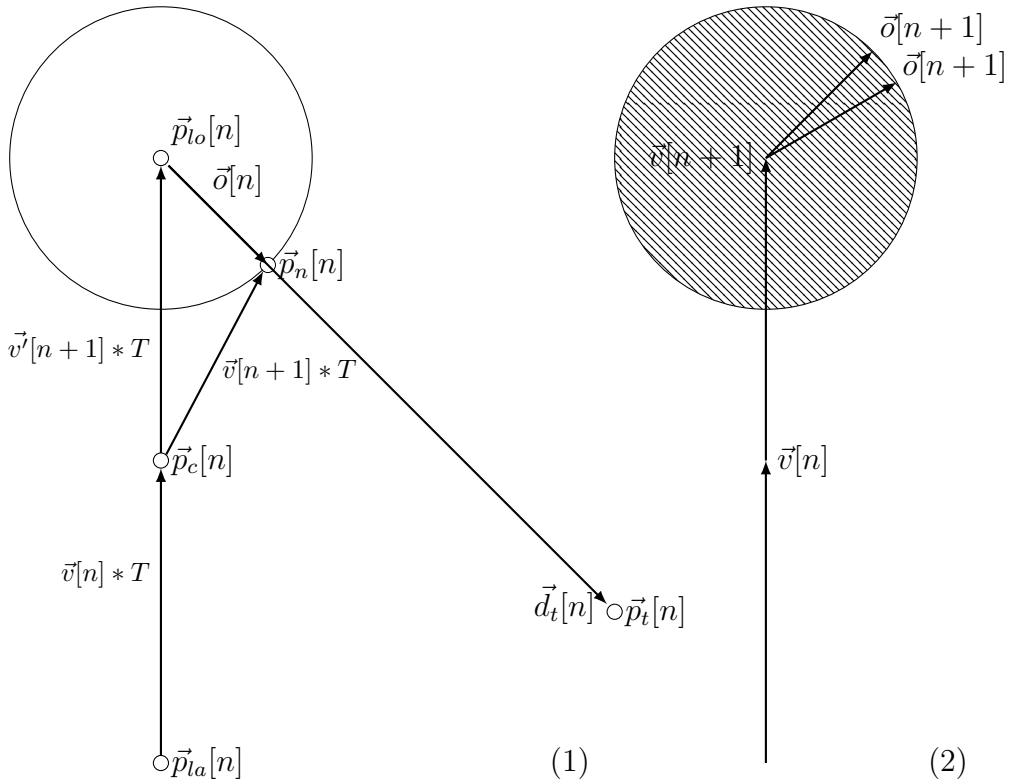


Figure 8: (1): Calculation of the next position on the path to the target, (2): Set of possible next positions

time domain is defined as the difference quotient: $\vec{v}[n] = \frac{\vec{r}[n]-\vec{r}[n-1]}{T}$ at a discrete time step $n$, whereas acceleration is expressed as $\vec{a}[n] = \frac{\vec{v}[n]-\vec{v}[n-1]}{T}$.

The robots state of motion can be represented by $T$ and the tuple of the position objects $\vec{p}_{la}[n], \vec{p}_c[n], \vec{p}_t[n]$ represented in Figure 8. With the state defined, the planner is able to determine $\vec{p}_n[n]$. The calculation of $\vec{p}_n[n]$ will be discussed in the following. It is assumed that while traveling between $\vec{p}_{la}$ and $\vec{p}_c$ the speed of the robots head remains constant. The arithmetic distance between those two point in the experiments with the robot was set to $<$ 1mm.

The planner has three input ports, referring to the three position objects just mentioned. Let $\vec{v}[n] = \frac{\vec{p}_c[n]-\vec{p}_{la}[n]}{T}$ be the current velocity vector. The robot might change speed and direction within the next cycle, therefore $\vec{v}[n+1] = \frac{\vec{p}_n[n]-\vec{p}_c[n]}{T}$ might be different to $\vec{v}[n]$.

When the robot changes speed or direction, the acceleration vector follows as: $\vec{a}[n+1] = \frac{\vec{v}[n+1]-\vec{v}[n]}{T}$. The force applying to the joints robot is proportional to the arithmetic length of this acceleration vector. Therefore we must ensure that the new instruction sent to the robot is according to our sanity bounds and is not overloading the robot with too much acceleration.

The objective is to determine $\vec{o}[n]$. If so, we can add $\vec{o}[n]$ to $\vec{v'}[n+1]*T$ to determine $\vec{v}[n+1]*T$ compare (1) of Figure 8. Subsequently, $\vec{v}[n+1]*T$ is added to $\vec{p_c}[n]$ to get $\vec{p_n}[n]$ which is the control output.

Let $\vec{p_{lo}}[n]$ be the position the robot would move to if we were to not apply any direction or velocity changes, $\vec{p_{lo}}[n] = \vec{p_c}[n] + \vec{v}[n] * T$. Let $\vec{d_t}[n]$ = $\vec{p_t}[n]$ - $\vec{p_{lo}}[n]$ be the difference vector between the overall target position $\vec{p_t}[n]$ and $\vec{p_{lo}}[n]$. By normalizing $\vec{d_t}[n]$, we get $\hat{d_t}[n]$ having the length 1[m]. Furthermore we define $a' = a_{max} * T^2[1/\text{m}]$. Note that $a'$ does not have a unit. We then choose $\vec{o}[n]$ as $\vec{o}[n] = \hat{d_t}[n] * a'$. Here $a'$ serves as a scaling constant for the offset vector $\vec{o}[n]$. The scaling constant $a'$ contains a larger value, when T is smaller and vice versa. Intuitively speaking, the higher the frequency, the shorter the offset vector should be to avoid sudden velocity changing. We choose the next position $\vec{p_n}[n]$ as follows:

$$
\begin{aligned}
(1) \quad \vec{p_n}[n] &= \vec{p_c}[n] + \vec{v'}[n+1] * T + \vec{o}[n] \\
(2) \quad &= \vec{p_c}[n] + (\vec{p_{lo}}[n] - \vec{p_c}[n]) + \vec{o}[n] \\
(3) \quad &= \vec{p_{lo}}[n] + \vec{o}[n]
\end{aligned}
$$

By choosing $\vec{p_n}[n]$ like this, we ensure, that $|\vec{a}[n+1]| <= al$.

$$
\begin{aligned}
(4) \quad \vec{v}[n+1] &= \frac{\vec{p_n}[n] - \vec{p_c}[n]}{T} = \frac{\vec{p_{lo}}[n] + \vec{o}[n] - \vec{p_c}[n]}{T} \\
(5) \quad \vec{v}[n] &= \frac{\vec{p_{lo}}[n] - \vec{p_c}[n]}{T}
\end{aligned}
$$

It should be noted that: $\vec{p_{lo}}[n] = \vec{p_c}[n] + \vec{v}[n] * T = 2\vec{p_c}[n] - \vec{p_{la}}[n]$ and therefore follows:

$$
\begin{aligned}
(6) \quad \vec{a}[n+1] &= \frac{\vec{v}[n+1] - \vec{v}[n]}{T} \\
(7) \quad &= \frac{(\vec{p_{lo}}[n] - \vec{p_c}[n]) + a' * \hat{d_t}[n] - \vec{p_c}[n] + \vec{p_{lo}}[n]}{T^2} \\
(8) \quad &= \frac{(\vec{p_c}[n] + \vec{p_c}[n] - \vec{p_{lo}}[n] - \vec{p_c}[n]) + a' * \hat{d_t}[n] - \vec{p_c}[n] + \vec{p_{lo}}[n]}{T^2} \\
(9) \quad &= \frac{a' * \hat{d_t}[n]}{T^2} \\
(10)
\end{aligned}
$$

If follows:

$$(11) \quad |\frac{a' * \hat{d_t}[n]}{T^2}| <= a_{max}$$

This vector has the length $a_{max}$ per Definition and adheres to the specified constraints. In fact all vectors that lie within the indicated sphere in Figure 8 satisfy

this condition and could Therefore be chosen as the next intermediate target. Most often, intermediate targets are selected on the sphere's edge to maximize range of motion and quickly approach the goal. However, for deceleration and velocity control, points inside the sphere might be chosen instead.

As shown in (1) of Figure 8 and discussed earlier, we construct the difference vector between $\vec{p}_{lo}[n]$ and $\vec{p}_t[n]$ to determine $\vec{d}_t[n]$, $\vec{o}[n]$ and subsequently $\vec{p}_n[n]$. Lets pretend the end-effector of the robot in Figure 8 missed the target and needs to turn around. In this scenario, the approach just introduced produces a sharp correction. Lets now pretend that the robot is near the target and moves directly in the targets direction. An alternative approach for choosing $\vec{p}_n[n]$ like in Equation 3 could involve constructing the difference vector between $\vec{p}_c[n]$ and $\vec{p}_t[n]$ instead of $\vec{p}_{lo}[n]$ and $\vec{p}_t[n]$ to determine $\vec{o}[n]$. In the latter scenario, this approach yields better results as the robot moves to the overall target position. Further improvements of the control could incorporate elements of both approaches, dynamically choosing between both.

## 3.4 Velocity Control

The velocity control sets the velocity of the robot. To set the speed within the next cycle of motion, we scale the offset vector that is added to $\vec{p}_c[n]$ while still adhering to the force constraints. This subsequent scaling operation also is performed by the Planner reactor. When $\vec{p}_n[n]$ is determined as in Equation 3, it always lies in the edge of the sphere, maximizing direction and speed change. In case of approaching the target, deceleration has to be applied. In Figure 9 we suppose that we are near to the target $\vec{p}_t[n]$. According to equation 3, the next intermediate target sent to the robot would be $\vec{p}_n[n]$. But in this scenario we want to reduce the speed. Let $\vec{s}_{max}[n] = \vec{p}_n[n] - \vec{p}_c[n]$. This is the offset vector to $\vec{p}_c[n]$ and can be scaled according to a minimum and maximum scaling limit as shown in Figure 9. Let $v_{min}[n]$ be the lower speed limit and $v_{max}[n]$ be the upper speed limit for the robots end-effector within the next cycle of motion. $|\vec{s}_{min}[n]| = |\vec{v}_{min}[n]| * T$ and $|\vec{s}_{max}[n]| = |\vec{v}_{max}[n]| * T$ are then the minimum and maximum distances traveled within the next cycle. Let $\vec{p}_{min}[n]$ denote the position to which the robot arm will move within the next cycle at minimum speed. Let $\vec{p}_{max}[n]$ denote the position to which it will move at maximum speed, both are displayed in Figure 9. In the same figure it is evident that $\vec{s}_{min}[n]$, $\vec{o}[n]$ and $\vec{v'}[n+1] * T$ form a triangle in the cartesian space. The triangle is well defined as long as $|\vec{o}[n]| < |\vec{v'}[n+1] * T|$. The objective is to determine $|\vec{s}_{min}[n]|$ and to scale $\vec{s}_{max}[n]$ accordingly.

Let $a = |\vec{o}[n]|$, $b = |\vec{s}_{min}[n]|$ and $c = |\vec{v'}[n+1] * T|$. We can calculate the angle $\alpha = \angle p_{min}[n] - \vec{p}_c[n] - \vec{p}_{lo}[n]$. With the law of sines we obtain two possible triangles: The triangle between the points $\vec{p}_c[n]$, $\vec{p}_{lo}[n]$ and $p_{min}[n]$, enclosing the angle $\gamma$ and on the other hand the triangle between the points $\vec{p}_c[n]$, $\vec{p}_{lo}[n]$ and $p_{max}[n]$, enclosing the angle $\gamma'$. The aim is to determine $|\vec{s}_{min}[n]|$, so between the two options, the

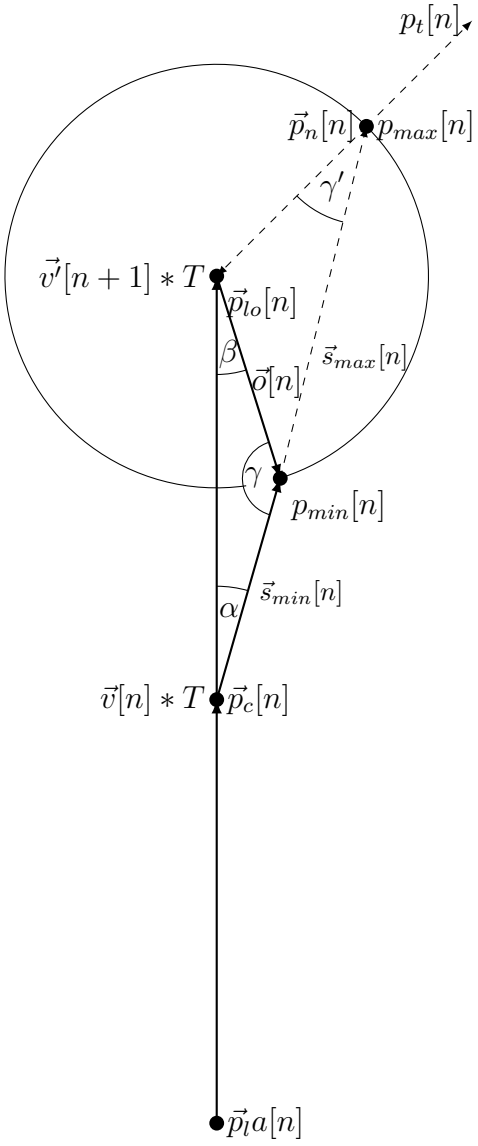


Figure 9: Determination of $|\vec{o}_{min}[n]|$

correct one is the former where the calculated $\gamma$ is an obtuse angle. To determine $\gamma$, $\gamma'$ is calculated beforehand. It follows:

$$\gamma' = \arcsin\left(\frac{\sin(\alpha)\cdot c}{a}\right) \tag{12}$$

$$\gamma = \pi - \gamma' \tag{13}$$

$$\beta = \pi - \gamma - \alpha' \tag{14}$$

$$b = |\vec{o}_{min}| = \frac{a\cdot\sin(\beta)}{\sin(\alpha)} \tag{15}$$

The entire procedure of calculating $|\vec{s}_{min}|$ is presented in Algorithm 1. If $|\vec{s}_{max}| >$

$m_v$, with $m_v$ being the maximum distance the robot head is able to move forward within each cycle at maximum speed, then $\vec{s}_{max}$ is trimmed to prevent the robot from achieving to much speed. As shown in 7, the Planner receives as input a target speed from the Velocity reactor. With the scaling interval now provided, the Planner sets the speed of the robot within the next cycle.

**Algorithm 1** Calculation of $|\vec{s}_{min}|$

**if** $|\vec{v'}[n+1] * T| < |\vec{o}_n|$ **then**
  **Return** 0
**else if** $linearDependent(\vec{v'}[n+1] * T, \vec{o}_n)$ **then**
  **Return** $|\vec{v'}[n+1]| - |\vec{o}_n|$
**else**
  **let** $a \leftarrow |\vec{o}[n]|$
  **let** $\alpha \leftarrow \angle p_{\min} - \vec{p}_c[n] - nl[n]$
  **let** $c \leftarrow |\vec{v'}[n+1] * T|$
  **let** $\gamma \leftarrow \pi - \arcsin\left(\frac{\sin(\alpha) \cdot c}{a}\right)$
  **let** $\beta \leftarrow \pi - \gamma - \alpha$
  **let** $b \leftarrow b = |\vec{s}_{min}| = \frac{a \cdot \sin(\beta)}{\sin(\alpha)}$
**end if**
**Return** $b$

Besides the positional end-effector data, the roll, pitch, and yaw has to be smoothly adjusted as the current and target position orientations might differ. For our use case proportional control was used, incrementally adjusting the current configuration towards the target orientation.

In summary, the described trajectory planning and velocity control methods enable the XArm 7 robotic arm to move smoothly and precisely with random position input and varying velocities. By calculating the next intermediate end-effector position and ensuring it adheres to acceleration constraints, the robot avoids sudden changes in velocity that could lead to instability or excessive forces on its components. Additionally, the velocity control mechanism allows for appropriate speed adjustments. It ensures that the robot can decelerate when approaching the target to avoid overshooting and can accelerate when more distance needs to be covered. This results in consistent and reliable movement for randomized input. It demonstrates the effectiveness of the proposed setup for achieving smooth and precise robotic motion. Incorporating additional data input and control logic in future work might enhance the systems capabilities. Now with the movement control working, we integrate a camera into the system.

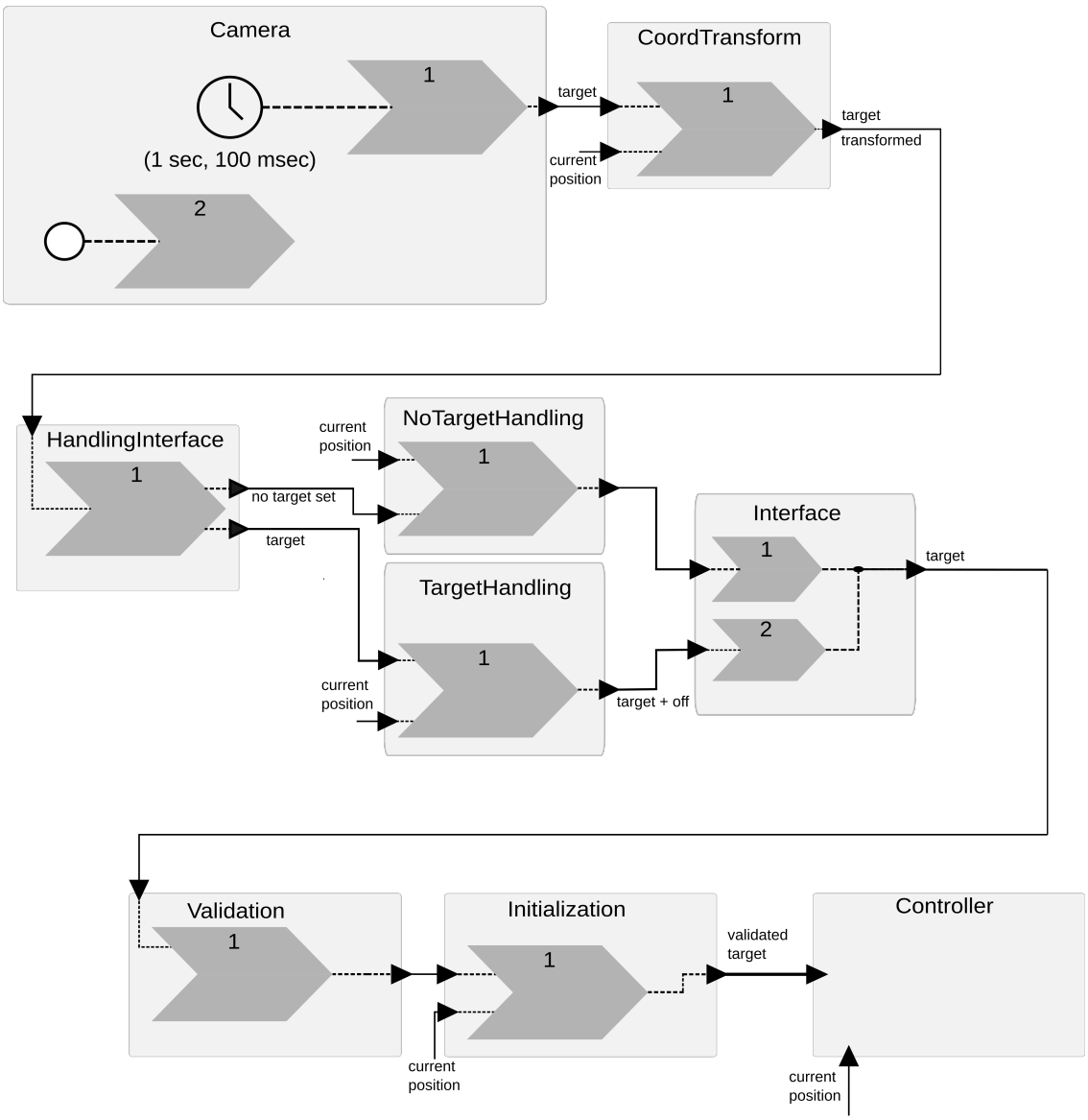


Figure 10: Camera, target processing

## 3.5 Perception

Robotic systems have become an integral part of modern technology. A key component of advanced robotic systems is their ability to perceive and interact with the environment. Image processing is an important aspect of robotic perception, enabling robots to understand and interpret visual information.

Within the demo built in this thesis, the robotic arm has to trace a target object. The purpose of the camera is to determine the current cartesian position of the detected object through image processing and image interpretation. An Intel REALSENSE depth camera D435i was used, featuring depth by combining image data

from two sensors to calculate depth information[8].

In stereo vision, binocular disparity is a key concept for reconstructing a scene's three-dimensional structure from two images. It refers to the difference in the location of similar features within two stereo images seen by two sensors. First, by using image rectification, both images are rotated. Once rectification is complete, disparities occur solely in the horizontal plane. This step is unnecessary if the cameras are accurately aligned horizontally. After rectification, the left and right images are scanned for corresponding features, which is referred to as the correspondence problem[23]. By triangulation, the distance of an object at a pixel to the camera is determined[5].

For localizing yellow objects, a simple image processing algorithm was used. The algorithm analyzes each pixel of a fetched frame, searches for the largest cluster of yellow pixels and calculates its centroid. The depth analysis provided by the camera enables determining the position of the object with respect to the camera.

The camera position and its rotation in space is constantly changing as the end-effector changes its position. The cameras rotation in space can be described by roll, pitch, and yaw. Roll is a counterclockwise rotation of $\alpha$ about the x-axis, while pitch and yaw are rotations about the y-axis and z-axis, respectively[22]. These rotations can be described by matrices:

(16)
$$\mathbf{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{R}_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \qquad \mathbf{R}_x(\gamma) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{pmatrix}$$

Each rotation is a simple extension of the 2D rotation matrix. A single rotation matrix can be formed by multiplying the yaw, pitch and roll rotation matrices. The combined rotation matrix $R$ yields:

$$R = R_{yaw}(\alpha) \cdot R_{pitch}(\beta) \cdot R_{roll}(\gamma)$$

(17)
$$R = \begin{bmatrix} \cos(\alpha)\cos(\beta) & \cos(\alpha)\sin(\beta)\sin(\gamma) - \sin(\alpha)\cos(\gamma) & \cos(\alpha)\sin(\beta)\cos(\gamma) + \sin(\alpha)\sin(\gamma) \\ \sin(\alpha)\cos(\beta) & \sin(\alpha)\sin(\beta)\sin(\gamma) + \cos(\alpha)\cos(\gamma) & \sin(\alpha)\sin(\beta)\cos(\gamma) - \cos(\alpha)\sin(\gamma) \\ -\sin(\beta) & \cos(\beta)\sin(\gamma) & \cos(\beta)\cos(\gamma) \end{bmatrix}.$$

It is important to note that $R$ performs the roll first, then the pitch and finally the yaw. Changing the order would yield a different rotation matrix as matrix multiplication generally is not commutative.

To add the new network component, the reactors shown in Figure 10 were connected with the reactor network of Figure 6. The output port of the 'Validation' reactor in Figure 10 is connected to the 'Controller' reactor in Figure 6 and overwrites periodically the controllers current target.

The 'Camera' reactor in Figure 10 serves as the interface to the camera and periodically fetches color and depth images. Its startup reaction initializes the camera and starts the image pipeline. Reaction 1 performs the image processing and transmits the position of the localized object to the 'CoordTransform' reactor. The transmitted cartesian position sent as an Object of type `Vector` is still in the reference system of the camera. The sent `Vector` is encapsulated within an `Optional` object to handle scenarios where the camera fails to localize an object. If the Camera reactor sends a valid `Vector` object, the CoordTransform reactor applies coordinate transformation and forwards the localized object's position in the robot's reference system to the 'HandlingInterface' reactor, an empty `Optional` object otherwise. This reactor, depending on the system's state, triggers a handling routine by communicating with the appropriate reactor. When an object is detected, it transmits to the 'TargetHandling' reactor. If no object is detected, it invokes the 'NoTargetHandling' reactor. Currently, if no object is assigned, the NoTargetHandling reactor commands the robot to stay in its current position. Alternatively, based on the application's objective, other commands, such as returning to a default waiting position, are imaginable. If on the other hand an object is determined by the camera, the TargetHandling reactor reacts. As for now the objective of the robotic system is to trace a yellow object (a lemon), the TargetHandling reactor ensures that the robot keeps a certain distance to the detected object by adding an offset vector. The interface passes the output of the handling routine to the Validation reactor. Given the limited range of motion for the robot, the Validation reactor ensures that the commands received by the camera adhere to specific motion constraints of the robot. The 'Initialization' reactor allows the robot to take on a default position before initiating movement. If we would want to change the applications, enhancing the robot's behavior would be simple thanks to the modular nature of Lingua Franca. This would likely only include adding network logic and linking it to the TargetHandling reactor and the NoTargetHandling reactor.

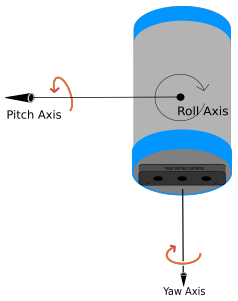

Figure 11: Roll, pitch, and yaw of the end-effector

With the camera system in place, the robotic demo is complete and traces objects. To explore its strengths and weaknesses, we will first discuss how the implementation of the demo could have been executed using ROS2.

## 3.6 Robot Control Design in ROS2

### 3.6.1 Creating the URDF file

Creating a virtual twin for the XArm7 is the first step in the process of classically constructing a robot control application. In the ROS2 control framework `.stl` and `.dae` files describe robots. The `.stl` files are coarse meshes used for fast collision checking, `.dae` files are used for visualization purposes. Blender is an open source 3D modeling software capable of processing both file types, the construction of a virtual twin can be performed using Blender. By convention, each `.stl` file expresses the position of its vertices in its own reference frame. Hence, a linear transformation (rotation and translation) has to be applied between each link to define the robot's full geometry when incorporating multiple `.stl` files to construct a robot[7].

```
1    hardware_interface.read();
2    //.. export of hardware data
3    controller.update();
4    //.. export of controller data
5    hardware_interface.update();
```

Figure 12: ROS2 main control loop

### 3.6.2 Construction

The hardware interface and joint controller play important roles in the main robot control loop, which coordinates both. Figure 12 provides an overview over the control flow: The `hardware_interface.read()` method fetches data from the hardware, the `controller.update()` method calculates the respective control output and the `hardware_interface.update()` method writes the control data to the joints. The three methods are executed on a real-time thread and therefor must obey real-time constraints. To develop an application for the XArm7, implementing both a hardware interface and a controller is mandatory[7]. Figure 13 illustrates the potential ROS2 control flow for the demo application developed in this thesis.

### 3.6.3 Hardware Interface

In ROS2, hardware system components are integrated via user defined driver plugins. Those plugin classes conform to the `HardWareInterface`. The interface demands the implementation 5 public methods: `on_init()`, `export_state_interfaces()`, `export_command_interfaces()`, `read()` and `write()`[18]. Every iteration of the ROS2 main control loop, `read()` is called on all hardware components implementing the `HardWareInterface`. The `read()` method receives the latest joint data and maps it to variables of class `command_interface` which hold a pointer to the joint data and

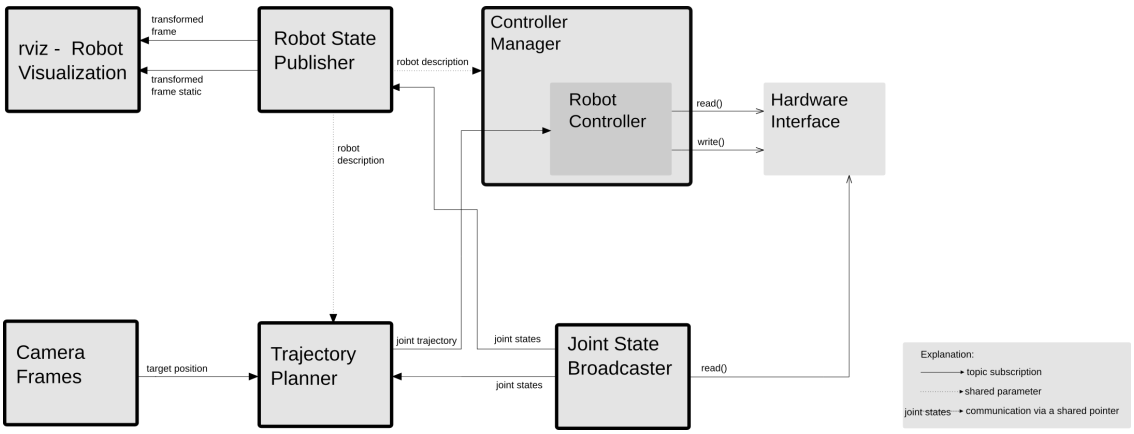


Figure 13: ROS2 node network, showcasing data exchange via topics, parameters and shared pointers

specify interface name and type, `command_interface` intuitively speaking are registers storing the current joint data. To stick to the application built within this thesis, `arm->get_joint_states()` introduced in line 21 in Figure 5 is called in `read()` to fetch joint data from the robot. Implementing the `on_init()` function requires the instantiation of an object of type `XArmAPI` as in line 10 in Figure 5 to initiate the communication to the robot. The methods `export_state_interfaces()` and `export_command_interfaces()` are used to share the pointer to the current joint data with the main control flow. Opposed to `read()`, the `write()` method operates on the hardware. It assumes that after the execution of `update()` in the main control flow, the `command_interface` data fields are overwritten by the controller. Dependent on the control mode used, the `write()` method hence could call the method `arm->vc_set_joint_velocity()` on the robot.

### 3.6.4 Writing a Controller

ROS2 provides a wide range of controllers. Controllers conform to the `ControllerInterface` in ROS2 and are similarly as the `HardwareInterface` loaded as plugins and modified using specific ROS2 parameters passed within a `.yaml` parameter file. Controllers exist in a finite set of states: `unconfigured`, `inactive`, `active` and `finalized`. During transition between these states, transition methods are called. When a controller is started, it is loaded into the memory, but is not yet initialized with the necessary parameters and configurations, the controller is in state `unconfigured`. It currently cannot perform any operations. After a configuration and a transition to `inactive`, the controller is fully operational, besides not controlling the hardware. When in `active` state and controlling the actuators, if the controller is shutdown, it transmits to state `finalized`. In this state, it is ready to be cleaned up or reinitialized by transition to unconfigured. These states allow

controller management at runtime.

### 3.6.5 Controller Implementation

The controller plugin is an object of a class that inherits from `controller_interface::ControllerInterface`. Nine methods are to be implemented. Six of them are transition callbacks like `on_configure()` or `on_activate()`. In the following, the most meaningful methods will be discussed. The methods `command_interface_configuration` and `state_interface_configuration` define which interfaces the controller need in order to operate. If a requested interface is not provided by the hardware, the controller fails. The method `on_configure()` is called when the controller transitions to `configured`. It sets up the subscription to the 'joint_trajectory' topic. The method `on_activate()`, called when the controller is activated, grants the controller access to the `command_interface` and `state_interface` data fields of the hardware. Claimed `command_interface` data fields are released in the method `on_deactivate()`. The `update()` method is regularly executed by the real-time control loop. The controller reads from its interfaces and calculates the control output. The output is written to the command interfaces, which will in turn control the hardware. The cooperation between the hardware interface and the controller enables accurate joint velocity control[17].

```
1  int main(int argc, char **argv)
2  {
3    rclcpp::init(argc, argv);
4    auto node = std::make_shared<TrajectoryPlanner>();
5    rclcpp::spin(node);
6    rclcpp::shutdown();
7    return 0;
8  }
```

Figure 14: TrajectoryPlanner main method

### 3.6.6 Further Network Nodes

We assume, that both the Robot Controller and the Hardware Interface are operational within the network shown in Figure 13. To expand the network, a node executing the planning operation has to be created. Server service nodes are executable cmake targets, `.cpp` files having a main method as shown in Figure 14. They are added as executables into the project via the projects CMakeLists.txt. The `spin()` method shown in the same Figure is a key method of the ROS2 node's lifecycle, it essentially keeps it running and allows processing incoming messages and the execution of callbacks. Server service nodes can be used for computational purposes. For our case, the 'Trajectory Planner' would be to subscribed to the 'target position' topic published by a camera node and to the 'joint states' topic referring

to the current state of the robot. The Trajectory Planner generates joint trajectories. Once the subscription callback from the 'target position' topic is executed, the inverse kinematics for the target end-effector position is determined. Computing the corresponding inverse kinematics using for example the KDL(Kinemtatics and Dynamics Library) is imaginable. The KDL library is able to generate the robot's kinematic tree from the URDF and creates an inverse kinematics velocity solver for the given end-effector position and its roll, pitch, and yaw. Given the current state published via the 'joint states' topic and the target state, the planner node would plan a trajectory by for example linear interpolation. If velocities for the start position and end position are given, cubic interpolation can be used. In our example the joint trajectory would consist of joint velocity configurations mapped to time stamps[19].

The 'Joint State Broadcaster' is a publisher node and receives joint data via shared pointer and publishes to the topic 'joint states'. Both the Trajectory Planner and the 'Robot State Publisher' are subscribed to this topic. The Robot State Publisher publishes required data for the visualization, generated from the URDF file and the joint states. It also manages the URDF representation of the robot and passes it to the 'Controller Manager' and to the Trajectory Planner.

To complete the networks, an additional node accessing camera data feeds processed camera data into the network. Similar to Figure 14, an object of the customly created class `RealSensePublisher` is instantiated and streams color and depths frames allowing image processing as described in 3.5 while spinning. Each time an image pair is processed, a target position is published to the 'target position' topic.

### 3.6.7 Summary

The implementation of the robot application with ROS2 started with the creation of a URDF file, constructing the robot's virtual twin. This enables control and simulation. The integration of the hardware is achieved through a user defined class inheriting from the ROS2 `HardwareInterface`. Controllers adhering to the `ControllerInterface` allow joint control and demonstrate, how the ROS2 control loop internally works. To integrate a visualization of the virtual twin, the robot state publisher preprocesses joint data. To allow interaction to the environment, a node that processes camera input and a trajectory planner are implemented. As shown, ROS2 allows for clear and fast application development. A developer using ROS2 can profit from a wide range of additional frameworks to control robots.

# 4 Evaluation

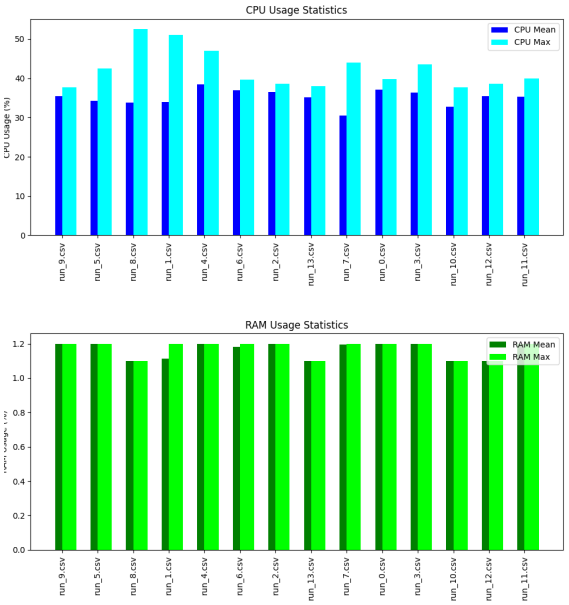


Figure 15: Planner

Several challenges while building the application were encountered, but many aspects also went very smoothly. Learning Lingua Franca is straightforward. The documentation is sufficient to fast get an overview and provided the minimalistic language design, the learning curve is steep. Once a large program is built, keeping the project structured is manageable. Modularization, factorization and the interpretation of the code is enhanced. Compared to C++, Lingua Franca benefits from the flexibility of cmake just as Cpp does. Implementing and design in Lingua Franca is faster than it is in C++. With the visualization tool, recent changes can be checked for consistency and correctness, aiding in development. When different developers concurrently work on the same Lingua Franca network, merge conflicts primarily occur in the reactor side of the program. In Lingua Franca, it's a good practice to let LF coordinate the code and implement the rest of the logic in external files. Therefore, reactor code remains short and manageable. Resolving merge conflicts in LF files was not labor-intensive.

Alongside the construction of the robot application demo, a simulation of the robot was built. We designed the reactor network to also connect to a simulation[20]. The simulation is integrated within the LF program and can be activated via connecting it with the Motion Planning Network through the Interface reactor shown in Figure 6. This allows integration testing.

The robot moved smoothly and consistently when run at speeds around 0.5[m/s]. The system reacts fast to input. By adjusting parameters like velocity and $a_{max}$ the robot performs fast movement, but a maximum velocity is reached quickly. The deceleration distance increases quadratically with speed. The higher the deceleration or acceleration, the higher the force. At higher speeds and increased $a_{max}$, the table and the robots head started to totter as the robot stopped and started abruptly. The control loops can be improved, but the physical properties of the XArm determine the maximum velocity and the precision at high speed. As the robot is mounted to the table, longer acceleration trajectories are not possible. When controlled at high speeds, either the mounting or the robot might break.

The current setup sometimes runs into problems as the self collision checking by the robot interrupts the program execution. Recall that the controller only determines the next intermediate position of the end-effector during a trajectory. At certain joint configurations, if it sends a control output that cannot be executed with the current joint configuration, the robot halts. A tighter integration with the robotic arm or shifting inverse kinematics externally while using direct joint control instead of position control can solve this problem.

CPU and RAM usage were measured over multiple test runs during multiple hours. In terms of RAM: even after longer test runs, the LF binary process did not significantly jitter in terms of RAM usage, the mean RAM usage always remained at around 1% of the total systems RAM capacity, see Figure 15. If a reaction finished, all data initialized is cleared, only the produced reactor output remains in the system. The system reacts to the current real time input and does not relay on large state data. This concretely indicates that the LF coordination network implemented within this thesis does not encounter memory leak problems, due to clear interface and lifecycle management. This implies stability over long periods of time. This is advantageous, as the target language Cpp is known for speed, but large programs can suffer from memory leaks.

In terms of CPU usage, jitter was encountered. During program execution, the image processing did in situations delay the system. When moving the target object close to the camera, recognition and clustering took to long as in proportion to the picture, the quantity of recognized pixels grow. This caused delays in the Lingua Franca execution flow and delayed control output and the robot performed sudden stops. Further improvements on the project might explore, whether this problem can be solved with the usage of the built-in feature 'enclaves': Reactions on the same 'level' determined by the LF dependency analysis can be executed concurrently. The current LF scheduler implementation only schedules the next level of reactions when the current level of execution is finished. This architecture can in practice cause independent reactions to block the execution of each other. This negatively impacts the performance. To handle this issue, LF provides the experimental feature enclaves. Each enclave has an own scheduler, reactions of different enclaves can advance their logical time independently[27]. Camera image processing is completely independent from the control network, partitioning both control flows into independent enclaves could solve this problem. Besides that, the remaining Lingua Franca network did not significantly produce jittering CPU utilization, as can be seen in Figure 15.

Race conditions can generally occur in robotic systems with multiple components that share data access. In LF, access to state variables is mutually exclusive and reactions bearing different timestamps are executed in order. Even tho introducing race conditions in Lingua Franca code is possible through passing pointers in messages, the documentation provides good guidelines to avoid them. During the

development and execution of the LF program, no problems with race conditions appeared.

Overall, Lingua Francas consistency in reaction time, CPU and RAM usage can enhance real-times applications' safety and predictability. The visualization tool also supports development and version control, providing a clear schema of the code. Lingua Franca allows developers to build robust, concurrent applications. This makes Lingua Franca an interesting choice in robotics.

The previous section focused on evaluating how well the Lingua Franca control logic operates the robot. The applied metrics, however, assume that the code is largely free of errors, a state that must first be achieved. While developing complex software systems, a developer can profit of good troubleshooting language features. Even if reproducable, finding bugs in bigger projects can become difficult. As Lingua Franca does not yet provide much debugging tooling, the following sections will discuss in detail the debugging process underwent to pin down errors during the development of the demo.

## 4.1 Debugging

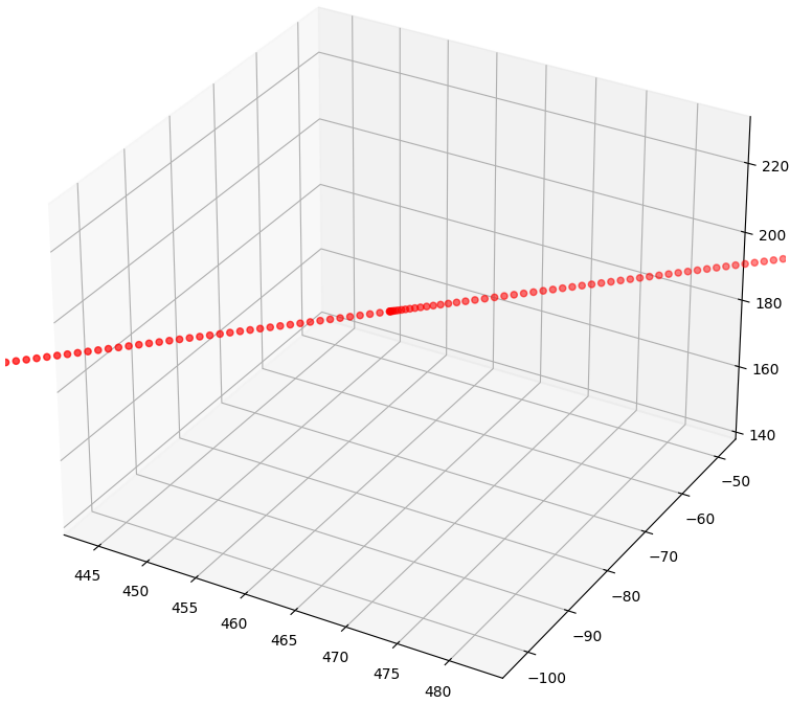


Figure 16: A normal trajectory, but the robot stops at a certain point and reaccelerates

Bugs are inevitable in software development. Finding them is often hard due to the growing complexity of codebases. Errors and bugs can arise from various sources like coding errors or design flaws. Often times, bugs are non-repeatable. Lingua Franca encourages the programmer to build modular and transparent code with clear defined interfaces. Therefore it might prevent errors. The bug that will be illustrated is robot specific, but the trouble shooting procedure can be generalized.

In Figure 16, the trajectory of the robot in the cartesian space, each point representing an intermediate position on the way to the overall target position. At a certain point however, it stops. This is non-desired behavior. The exact moment when the robot halts and then reaccelerates is detailed in Table 1. At entry 3491, the robot was moving at its maximum constant speed. However, entries CP(3512) and CP(3521) are identical, indicating that the robot indeed had stopped and then reaccelerated. When observing the robot's movement in real-time, it suddenly stopped before regaining control over its speed. This is the kind of behavior that is dangerous for the robots joints, especially when moving at max speed. Various reasons could cause the bug, the robot itself, malfunctioning logic in the Planner or Velocity Controller or Lingua Franca.

| **logical time** | $\vec{p}_{la}$ **x** | $\vec{p}_{la}$ **y** | $\vec{p}_{la}$ **z** | $\vec{p}_{c}$ **x** | $\vec{p}_{c}$ **y** | $\vec{p}_{c}$ **z** |
|---|---|---|---|---|---|---|
| 3491 | 464.429932 | -84.667854 | 199.990799 | 463.813354 | -85.530075 | 199.990829, |
| 3502 | 465.046509 | -83.805634 | 199.990768 | 464.429932 | -84.667854 | 199.990799, |
| 3512 | 465.046509 | -83.805634 | 199.990768 | 465.046509 | -83.805634 | 199.990768, |
| 3521 | 465.081421 | -83.756836 | 199.990768 | 465.046509 | -83.805634 | 199.990768, |

Table 1: Current position X,Y,Z — Last Position X,Y,Z

Collecting data is quite easy in Lingua Franca. By adding connections to a data handling reactor, data can be pushed into a log file. A similar reactor is shown in Figure 17. This reactor currently only logs $\vec{p}_c[n]$ and $\vec{p}_{la}[n]$. To extend data logging in the network, this reactor has to be adjusted as well as its connections to the rest of the network. Additional data from the inputs can be integrated by introducing new lines of code analogous to the existing retrieval(c.f. **??** line 7) and insertion(c.f. line 13-17) operations. This expansion is straightforward and does not necessitate extensive modifications.

It would be possible to automate this process. Let $I$ be a specialized reactor provided by LF, that only needs to be connected to the network, but is *not* implementable by the programmer. In our example, we define only one input port for $I$, but multiple are conceivable. Let $x$, $y$ be the objects, that are fed into the input ports of $I$. The purpose of $I$ is to store data, for example in a `.csv` file[2] automatically without having the programmer thinking about it. Future improvements of the Lingua Franca compiler could involve the generation of the code in the reaction

[2]different file types are possible and supposedly preferable

```
>

1
2     reaction(csv_last_position, csv_current_position, // .. extend)
3       {=
4
5         std::vector<std::string> data;
6
7         auto last_position_as_vector = (*csv_last_position.get()).get().to_vector();
8         auto current_position_as_vector  = (*csv_current_position.get()).get().to_vector();
9
10        // .. extend
11
12        vectors.insert(data.end(),
13        (*last_position_as_vector.begin(),
14        (*last_position_as_vector.end());
15
16        vectors.insert(data.end(),
17        (*current_position_as_vector.begin(),
18        (*current_position_as_vector.end());
19
20        // .. extend
21
22        Instruction_Collector ins_coll  =  Instruction_Collector{
23          get_elapsed_physical_time(),
24          data
25          };
26        ins_coll.write_instructions_to_csv(instructions_path);
27
28      =}
```

Figure 17: Reaction code of CSVDataTraffic

body in Figure 17 by itself and could automate the process of collecting data from the network. In this scenario, both $x$ and $y$ would need to implement an interface that includes the methods `to_vector()` and `to_header()`, respectively[11].

Another debugging approach can be Printf-Debugging which sometimes can become disorganized in classical programs. Printf-Debugging can be done in Lingua Franca elegantly by implementing a small printing reactor that just transfers data and prints it to the console. Once such a reactor is implemented it can be placed between any reactor. Consulting the visualization tool is sufficient to quickly get an overview of where printing is taking place and what is being printed. Such a print reactor could also be provided by a future Lingua Franca compiler enforcing only that the printed objects implement `to_string()`. Within the graphical visualization tool, manually placing the printing reactor within the graphic could be intuitive.

Debugging is not only about tracing data. It is also about testing the system. The unit that assumably could cause the robots halting in the context of the encountered bug is the 'Planner' reactor $P$ introduced in Figure 3.2 that encapsulates the logic that performs the positional control. At one iteration, the Planner receives via input ports a tuple $(\vec{p}_{la}[n], \vec{p}_c[n], \vec{p}_t[n])$ of three objects of type Position and $v[n]$, denoting the designated velocity determined by the Velocity reactor. Let $S_{3502} = (\vec{p}_{la}[3502], \vec{p}_c[3502], \vec{p}_t[3502], v[n3502])$ be input tuple of the Planner at logical timestamp 3502. The behavior of $P$ is deterministic and does not contain state variables. The second reactor within the Motion Planning Network impacting motion control is the 'Sanity Checker' reactor shown in 3.2. Its purpose is to execute an emergency stop if malicious behavior from the Planner is detected. In order to

properly test the Planner and the Sanity Checker, a second main reactor for testing purposes which is highlighted in Figure 18 was built.

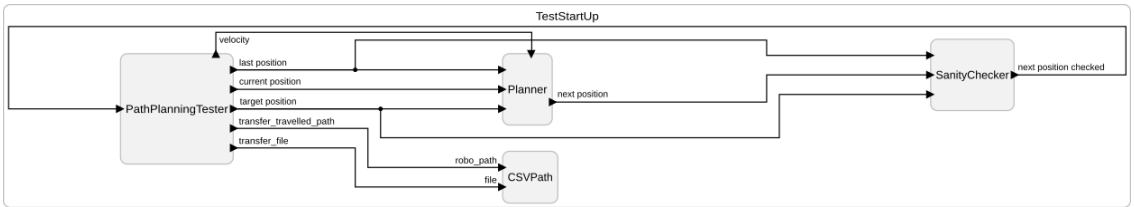


Figure 18: Testing reactor

The 'PathPlanningTester' $P_T$ mocks the robot. It holds two states, $S_i$ and a list $p = [\vec{p}_c[0], \vec{p}_c[1], \vec{p}_c[2], ..., \vec{p}_c[i-1]]$ referring to the trajectory calculated by the Planner. Each iteration, $S_i$ is passed to the Planner which determines $\vec{p}_n[i]$. The state of $P_T$ is updated, $S_{i+1} = (\vec{p}_{la}[i+1] \leftarrow \vec{p}_c[i], \vec{p}_c[+1] \leftarrow \vec{p}_n[i+1], \vec{p}_t[i+1] = \vec{p}_t[i], v[i+1])$. Once a trajectory is finished, $p$ is passed to the 'CSVPath' reactor and another trajectory is being initiated. Every path is stored in log files and can be analyzed afterwards. The purpose of this second main reactor is to see how the trajectory planning system works under different inputs on multiple different paths and starting conditions. Conceptually, the test inputs are designed to emulate a robot movement represented by $S_i$ at a random position and speed with a random target independent of the preceding robot movement. To assess the reliability of my motion planning units, it was tested with both the configuration that presumably caused the error, but also with generated random inputs to analyze the behavior of the two reactors.

This debugging process is hardly generalizable beyond robotic applications. It is comparable to classical unit testing classes in object oriented programming. It demonstrates, that unit testing in Lingua Franca is manageable, even though building the whole test setup was labor-intensive. Overall, testing the outsourced logic with classical test frameworks might be faster than constructing a second main reactor for testing purposes.

OMNeT++ is a simulation framework designed for modeling computer networks and network protocols. Generally, networks are tested. Just as in Lingua Franca the developer defines network components, message types and the network setup. In addition to that, in `.ini` files, the developer defines simulation setups with different parameter configurations. For the simulation, OMNeT++ features a graphical user interface that visualizes the network traffic, providing users with real-time insights into network operations. This interface allows users to control the simulation. One can either advance to the specific logical time points or pause to analyze the network's current state allowing deep insights in the network behavior[25]. A similar framework could be useful for Lingua Franca. As LF networks are designed to perform controlling in cyber-physical systems, if tested in a similar framework, mocking actuators and sensors with reactors will be necessary. For example, generating a reactor mocking a sensor through parsing a `.csv` file defining sensor output mapped

to timestamps is imaginable. Future work can explore whether such a framework can be built around Lingua Franca automatizing simulations combined with reactor generation to mock sensors and actuators.

In the end, the bug was caused by the robot itself. When fetching data with a frequency $f > 100Hz$, we surpass the updating rate of the control box's data registers and sometimes read the same value twice. This is a problem, as the Planner reactor assumes the robot stops since $\vec{p}_c[n] == \vec{p}_{la}[n]$. Therefor it falsely initiates an acceleration process. The network performs the calculation of control output on compromised input. To recover from this, a 'Moderator' Reactor placed between the interface reactor and the decoder reactor was implemented, as shown in Figure 6, that analyses the data and in the error case transmits polished data into the network. This solution is not ideal as it 'predicts' movement, but it turned out to work fairly well. Improvements here could consist of a adjustment of the planner reactor. Currently, only $\vec{p}_{la}[n]$ and $\vec{p}_c[n]$ are taken into consideration. An improved planner could incorporate state information from the traveled path. The current planner relies on the correctness of the fetched data from the robot. As a sudden stop could on the other mean a collision, the Moderator reactor also handles collision.

To further understand the strengths and weaknesses of Lingua Franca, it is beneficial to compare it with ROS 2 (Robot Operating System 2), which has established itself as a standard in the field of robotics. In the upcoming sections, the introduced ROS 2 will be compared with Lingua Franca in the context of robotic applications.

## 4.2 Comparison

Ease of development: ROS 2 provides a large set of tools, such as rviz for visualization, gazebo for simulation, and rqt for graphical interfaces. This allows prototyping and testing. The ROS ecosystem includes many libraries, such as the controller ros2_control package, but also KDL or MoveIt! for motion planning, inverse kinematics and collision checking[14]. ROS 2 is well documented. On youtube many guides can help diving into robot control. Troubleshooting, setup and usage is described on the official ROS wiki and ROS 2 documentation. A large community on various forums provide help. ROS 2 is an open-source framework. Maintenance and development is provided by developers and companies like Bosh or the Fraunhofer institute[3]. This large ecosystem plays a significant part in its success. Integrating sensors and actuators is easy as ROS 2 supports standardized messages and interfaces. ROS 2 is continuously evolving. Developers are actively contributing.

Lingua Franca is well documented, learning the language is easy if the developer already knows Cpp or different target languages. Learning LF from scratch is easier than learning ROS from scratch, as understanding the basic Lingua Franca structure is straightforward as it is modular and provides the network visualization. Setting up a basic environment in Lingua Franca is clear, the idea of designing

networks instead of class structures is convenient. Incorporating libraries like KDL or MoveIt! in Lingua Franca is as simple, as it is in ROS 2, as both use cmake. But including controllers like specified in the ros2_control package requires a more complex interface matching. ROS 2 hides much functionality from the user and provides management via structured configuration files. As much functionality is hidden from the developer, pinning down bugs sometimes is difficult. Here, the ROS' 2 documentation and guides provide help. Lingua Franca programs in contrast are more centralized.

Building on existing robot tooling is more difficult in LF. A LF developer will for example have to integrate URDF files and control mechanisms from scratch, if simulation is to be built. When it comes to debugging and troubleshooting, in terms of Lingua Franca and robotics, a developer can not profit from a large community, at least not yet. Finding help and sharing knowledge is not established in Lingua Franca in a robot application context.

Both Lingua Franca and ROS 2 are very modular. ROS is built around a modular architecture where individual functionalities are encapsulated in nodes. Each node performs a specific task and communicates with other nodes via topics, services, and actions. This enables reuse across different projects. Lingua Franca provides similar mechanisms, reactors performing tasks can be integrated into multiple systems if the interfaces match. Both are self-contained, their software architecture approaches focus on separation of functionality into many independent systems.

ROS 2 networks allow lifecycle management. This allows continuous integration and deployment and enhanced resource management. In complex robotic networks, different components for perception, navigation and manipulation can activate and deactivate to conserve energy. Future work might explore on how to generalize reactor state transition management and introduce semantics to the LF compiler.

### 4.2.1 Real-time Performance and Reliability

Developing a control system requires careful attention to real-time performance. Precise timing, reproducability and concurrency play important roles. Lingua Franca and ROS 2 are both frameworks designed to facilitate the development of complex, distributed, and real-time systems. However, they take different approaches to resource management and real-time capabilities. The ROS 2 non-real-time system architecture prevents it from guaranteeing fault tolerance, deadlines, or process synchronization. ROS 2 requires significant resources. CPU, memory, network bandwidth, threads and kernels are managed. Time constraints are hard to meet. ROS 2, the latest version of ROS, has improved real-time capabilities with the inclusion of the Data Distribution Service (DDS) middleware. However, most of of the ROS 2 ecosystem is currently built around Linux. The upper limit of real-time performance is determined by the operating system itself and improving the real-time performance lies in optimizing the performance of the operating system.

Robot control systems written in ROS 2 are not reproducable. Reproducing errors in testing environments is not possible. Ensuring reliability is more difficult and usually achieved by using fast, redundant hardware. ROS 2 introduced the Quality of Service (QoS) policies like deadlines and fault-tolerance. This allows fine-grained control over the communication behavior and can potentially improve real-time capacities and enhance reliability. As many companies use ROS 2, it continuosly undergoes testing and validation in real-world scenarios[16][2][12].

Lingua Franca supports directly real-time constraints by allowing developers to specify deadlines for invocations. LF is more focused on providing a concurrency oriented model suitable for real-time systems. Enclaves and workers enable complete concurrent deterministic behavior, controlled by the developer. LF excels in predictability and reliability, as Lingua Franca is independent from threads handled by the operating system. However, long reaction code execution times can delay the execution flow. Deadline handling enables recovering here.

### 4.2.2 Scalability and Node Management

Complex robotic systems involving multiple nodes and subsystems are imaginable. Contrary to ROS' 1 AMQP (Advanced Message Queuing Protocol), ROS 2 uses the network protocol Data Distribution Service (DDS). This communication middleware allows more efficient handling of large number of nodes and topics, as well as a better managing of the underlying resources. DDS also supports many-to-many communication patterns, this allows a more flexible and scalable organization of the robots in the system. Another aspect is the ability to handle failures and recover from them. DDS-based discovery and configuration improve the robustness of the system by enhancing its reliability. To measure the performance of a network, the framework irobot-ros provides tooling[10][2]. If multiple machines are configured to support ROS 2 and are in the same LAN-network, they form one logical robot application[6].

Lingua Franca promotes good scalability. Built-in deadlines allow micromanagement. When distributively executed, Lingua Franca does not loose its advantage of determinism over ROS 2. Even though the network jitter is unpredictable, given the same program network conditions, the programs output will not change and errors are repeatable. Recent work on distributed embedded system for autonomous driving has shown, that systems can profit from the extension with Lingua Franca, while still preserving determinism and exceeded throughput[1][11].

# 5 Conclusion and Outlook

In this thesis we showed that with Lingua Franca the control of a complex real-time robotic system is possible. Constructing a real-time control mechanism is direct. Expanding the network with new components to integrate camera data is transparent. When fully integrated, the system showed fast reaction times and precise movement. Similarly to ROS 2's node-based architecture, Lingua Franca's reactor-based approach allows for reusable, self-contained components. In addition, it offers precise timing and concurrency controls through features like deadlines, enclaves, and workers. They ensure deterministic behavior and robust performance.

While its modularity and understandability prevent bugs, debugging in Lingua Franca remains unintuitive. Future development of the Lingua Franca can include designing graphical interfaces and automating test reactor generation to facilitate debugging.

For projects where real-time constraints and deterministic behavior are important, Lingua Franca can excel, also showing promising performances for systems that have to run over very long periods of time. In contrast, projects benefiting from a vast ecosystem and a wide range of tools and libraries, ROS 2 remains a good choice. The decision between Lingua Franca and established frameworks will depend on the specific requirements and constraints of the robotic application.

# Acknowledgments

I would like to thank Dr. Christian Menard, whose feedback and guidance have been instrumental in the completion of this work. Your insights in Lingua Franca and support have greatly enhanced the quality of this project.

I am also thankful to Prof. Calandra for the expert advice concerning robotics, this accelerated the development of the application.

Special thanks to my father Steffen Mehnert, to Johannes Wünsche, to Leila Walter and to Enya Vogel for proofreading my thesis and providing feedback.

Thank you for your contributions and support.

# References


[1] Soroush Bateni et al. "Risk and Mitigation of Nondeterminism in Distributed Cyber-Physical Systems". In: *Proceedings of the 21st ACM-IEEE International Conference on Formal Methods and Models for System Design*. MEMOCODE '23. Hamburg, Germany: Association for Computing Machinery, 2023, pp. 1–11. ISBN: 9798400703188. DOI: 10.1145/3610579.3613219. URL: https://doi.org/10.1145/3610579.3613219.

[2] *Comparison of the scalability and performance of ROs1 and ROS2 in multi-robot systems — linkedin.com*. https://www.linkedin.com/pulse/comparison-scalability-performance-ros1-ros2-systems-fouad-anwar. [Accessed 03-07-2024].

[3] *Current Members — ROS-Industrial — rosindustrial.org*. https://rosindustrial.org/current-members. [Accessed 03-07-2024].

[4] ros2_control Development team. *Documentation/Getting started*. vhttps://control.ros.org/rolling/_images/components_architecture.png. [Accessed 30-06-2024].

[5] Digital Nuage. *Disparity and Depth Estimation from Stereo Camera*. [Online; accessed 27-June-2024]. 2024. URL: https://www.digitalnuage.com/disparity-and-depth-estimation-from-stereo-camera.

[6] ed. *Multiple machines tutorials — roboticsbackend.com*. https://roboticsbackend.com/ros2-multiple-machines-including-raspberry-pi/. [Accessed 03-07-2024].

[7] *Example: Full tutorial with 6DOF*. https://control.ros.org/master/doc/ros2_control_demos/example_7/doc/userdoc.html. [Accessed 02-07-2024].

[8] *Getting Started with RealSense D455*. https://www.mouser.de/applications/getting-started-with-realsense-d455/?_gl=1*1q3fn2j*_ga*dW5kZWZpbmVk*_ga_15W4STQT4T*dW5kZWZpbmVk*_ga_1KQLCYKRX3*dW5kZWZpbmVk. Accessed: 2024-06-26. 2024.

[9] *How do we describe a robot*. https://www.youtube.com/watch?v=CwdbsvcpOHM. [Accessed 30-06-2024].

[10] *Irobot/github.com*. https://github.com/irobot-ros/ros2-performance. [Accessed 03-07-2024].

[11] *Lingua Franca Tutorial, Part 6: Research Overview — youtube.com*. https://www.youtube.com/watch?v=afJowM35YHg&list=PL4zzL7roKtfXyKE3k8lOwPub9YEjulS4o&index=7&t=2877s. [Accessed 03-07-2024].

[12] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. "Exploring the performance of ROS2". In: *2016 International Conference on Embedded Software (EMSOFT)*. 2016, pp. 1–10. DOI: 10.1145/2968478.2968502.

[13] Christian Menard et al. "High-performance Deterministic Concurrency Using Lingua Franca". In: *ACM Trans. Archit. Code Optim.* 20.4 (Oct. 2023). ISSN: 1544-3566. DOI: 10.1145/3617687. URL: https://doi.org/10.1145/3617687.

[14] *moveit.ros.org*. https://moveit.ros.org/. [Accessed 03-07-2024].

[15] *Nodes2014; ROS 2 Documentation: Iron documentation — docs.ros.org*. https://docs.ros.org/en/iron/Concepts/Basic/About-Nodes.html. [Accessed 05-07-2024].

[16] *Real time performance and optimization*. https://cjme.springeropen.com/articles/10.1186/s10033-023-00976-5. [Accessed 03-07-2024].

[17] *ros2_control_demos/example_7/controller at master · ros-controls/ros2_control_demos — github.com*. https://github.com/ros-controls/ros2_control_demos/tree/master/example_7/controller. [Accessed 02-07-2024].

[18] *ros2_control_demos/example_7/hardware at master · ros-controls/ros2_control_demos — github.com*. https://github.com/ros-controls/ros2_control_demos/tree/master/example_7/hardware. [Accessed 02-07-2024].

[19] *ros2_control_demos/example_7/reference_generator/send_trajectory.cpp at master · ros-controls/ros2_control_demos — github.com*. https://github.com/ros-controls/ros2_control_demos/blob/master/example_7/reference_generator/send_trajectory.cpp. [Accessed 02-07-2024].

[20] Tassila Tanneberger and Benedict Mehnert. *Title of the Repository*. tud-ccc/lf-mujoco. 2024. URL: git@github.com:tud-ccc/lf-mujoco.git.

[21] uFactory. *XArm User Manual*. Version V2.0.0. Accessed: 2024-06-15. May 2023. URL: https://www.ufactory.cc/wp-content/uploads/2023/05/XArm-User-Manual-V2.0.0.pdf.

[22] Wikipedia. *Aircraft principal axes — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Aircraft%20principal%20axes&oldid=1213129487. [Online; accessed 17-June-2024]. 2024.

[23] Wikipedia. *Binocular disparity — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Binocular%20disparity&oldid=1226361244. [Online; accessed 27-June-2024]. 2024.

[24] Wikipedia. *Motion planning — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Motion%20planning&oldid=1212639048. [Online; accessed 11-June-2024]. 2024.

[25] Wikipedia. *Omnetpp — Wikipedia, The Free Encyclopedia*. https://de.wikipedia.org/wiki/OMNeT%2B%2B. Accessed: 2024-06-17.

[26] Wikipedia. *Robot kinematics — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Robot%20kinematics&oldid=1152657384. [Online; accessed 11-June-2024]. 2024.

[27] *www2.eecs.berkeley.edu*. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-131.pdf. [Accessed 03-07-2024].