LUT UNIVERSITY
LUT School of Energy Systems
LUT Mechanical Engineering
BK10A0402 Bachelor's thesis

Development of a Control Framework for a Serial Robot Manipulator and Gripper Based on Robot Operation System Platform (ROS)

Seriaalisen Robottimanipulaattorin ja Tarttujan Ohjauskehyksen Kehittäminen Robot Operating System Ohjelmistoalustalla (ROS)

In Lappeenranta 22.06.2020
Tuomas Syyrilä
Examiners Professor Heikki Handroos
D.Sc. Tech. Eng. Hamid Roozbahani

TIIVISTELMÄ

LUT-Yliopisto
LUT School of Energy Systems
LUT Kone

Tuomas Syyrilä

Seriaalisen Robottimanipulaattorin ja Tarttujan Ohjauskehyksen Kehittäminen Robot Operating System Ohjelmistoalustalla (ROS)

Kandidaatintyö
2020

63 sivua, 32 kuvaa, 6 taulukko ja 7 liitettä

Tarkastajat: Prof. Heikki Handroos
TkT Hamid Roozbahani

Hakusanat: ROS, UR10, Teollisuusrobotit, Poimi ja aseta, Liikkeen suunnittelu

Tämä projekti toteutettiin osana LUT-yliopiston älykkäiden koneiden laboratorion (Lappeenranta-Lahti University of Technology) tutkimusta kehitettäessä ROS-pohjaista yhteistyörobotti valmistusasemaa. Tämän tutkimus keskittyy vakaan ROS-pohjaisen kehyksen kehittämiseen Universal Robotin ja Robotiq-tarttujan hallitsemiseksi. Kehys on suunniteltu modulaariseksi tulevien antureiden, kuten kameran, integroimiseksi tulevaisuudessa. Tutkimuksessa kuvataan robotin ja tarttujan integrointi ROS-alustaan sekä menetelmät laitteiston manipuloimiseksi. Tutkimuksessa esitetään myös laitteiston ja ohjelmistokomponenttien välisen viestinnän ylemmän tason rakenne. Tutkimuksen tuloksena kehitettiin toimintakehys, joka pystyy tehokkaasti hallitsemaan robottia ja tarttujaa. Tutkimuksessa myös ohjelmoitiin poimi ja aseta sovellus onnistuneesti.

ABSTRACT

LUT University
LUT School of Energy Systems
LUT Mechanical Engineering

Tuomas Syyrilä

Development of a Control Framework for a Serial Robot Manipulator and Gripper Based on Robot Operating System Platform (ROS)

Bachelor's thesis
2020

63 pages, 32 figures, 6 tables and 7 appendices

Examiners: Professor Heikki Handroos
D. Sc. Tech. Eng. Hamid Roozbahani

Keywords: ROS, UR10, Industrial robots, Pick and Place, Motion planning

This project was conducted as a part of research for the Laboratory of Intelligent Machines at LUT University (Lappeenranta-Lahti University of Technology) in developing a ROS-based collaborative robot manufacturing station. The research in this thesis focuses on the development of a stable ROS based framework for controlling a Universal Robot and Robotiq gripper. The framework is designed to be modular for future integration of further hardware such as vision sensors. The research describes the requirements and actual integration of the robot and gripper as well as the methods for manipulating the hardware. The higher/level concepts of communication between hardware and software components are also presented. The research resulted in the development a working framework that can control both the robot and gripper efficiently. Furthermore, software for a perception free pick and place application was successfully developed.

ACKNOWLEDGEMENTS

This project could not have been successful without the help of certain individuals at LUT whom I must extend my gratitude to. Firstly, I would like to thank Prof. Heikki Handroos and Dr. Hamid Roozbahani for giving me the opportunity to be a part of this research. Especially I must thank Dr. Roozbahani for the endless support, patience, and wisdom he provided during the project. Additionally, I would like to thank Juha Koivisto for the technical support and multitude of hours he spent on ensuring the functionality of the project hardware. I would also like extend my gratitude for the Ultrasimmi internal investment project at LUT for the funding that was provided for the research.

Finally, I want to thank my girlfriend for the continued support throughout the project.

Tuomas Syyrilä

Tuomas Syyrilä

Lappeenranta 22.06.2020

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS (IF NEEDED)

GUI	Graphical User Interface
IFR	International Federation of Robotics
KPIECE	Kinematic Planning by Interior-Exterior Cell Exploration
LBKPIECE1	Lazy Bi-directional KPIECE with one level of discretization
LUT	Lappeenranta-Lahti University of Technology
OMPL	Open Motion Planning Library¨
PRM	Probabilistic Roadmap Method
ROS	Robot Operating System
RRT	Rapidly expanding random trees
RTU	Remote Terminal Unit Protocol
TCP	Tool Center Point
TCP	Modbus Transmission Control Protocol
URDF	Unified Robot Description Format
UR	Universal Robots
XML	Extensible Markup Language

1 INTRODUCTION

This thesis is based upon laboratory research on designing and developing a framework for a manufacturing working station that could enable a human and robot to work together. The research was conducted for the Laboratory of Intelligent Machines at LUT University (Lappeenranta-Lahti University of Technology).

1.1 Background and motivation

Automation of manufacturing processes in the industrial field is a very current global topic. Businesses are constantly evolving their methods to further optimize production while improving the ergonomics of their employees' work environment. The advancements in the robotics field allow repetitive and strenuous tasks to be carried out by machines as opposed to human workers. As a result, manufacturers world-wide have accepted automation as a standard in industry. However, the next evolutionary step for the industrial field is the collaboration of humans and robots. This nature of collaboration has not been widely implemented yet and is addressed in this research.

A collaborative robot is a robot that is by design able to perform alongside a human safely and efficiently in a physical capacity. Robots have been widely used in automation previously but in a primarily simple and limited manner. Collaborative robots differ from common robots by not only being fully autonomous but by acting as an additional supportive set of hands in the workspace that can safely interact with humans. The implementation of collaborative robots in industrial environments is growing exponentially. Already in 2017, the International Federation of Robotics (IFR) predicted that the number of collaborative robots in factory environments would increase more than tenfold within a few years (Ferraresi & Quaglia 2018, p.242).

The project test bed is based upon a UR10 (Universal Robot) with an integrated Robotiq 2F-85 adaptive gripper controlled through Robot Operating System (ROS). To understand the nature of the research it must be stated that the project began with a relatively wide scope that include the model of the robot and the objective to design a human-robot collaborative workstation. The physical concept idea of the workstation is displayed in figure 1.

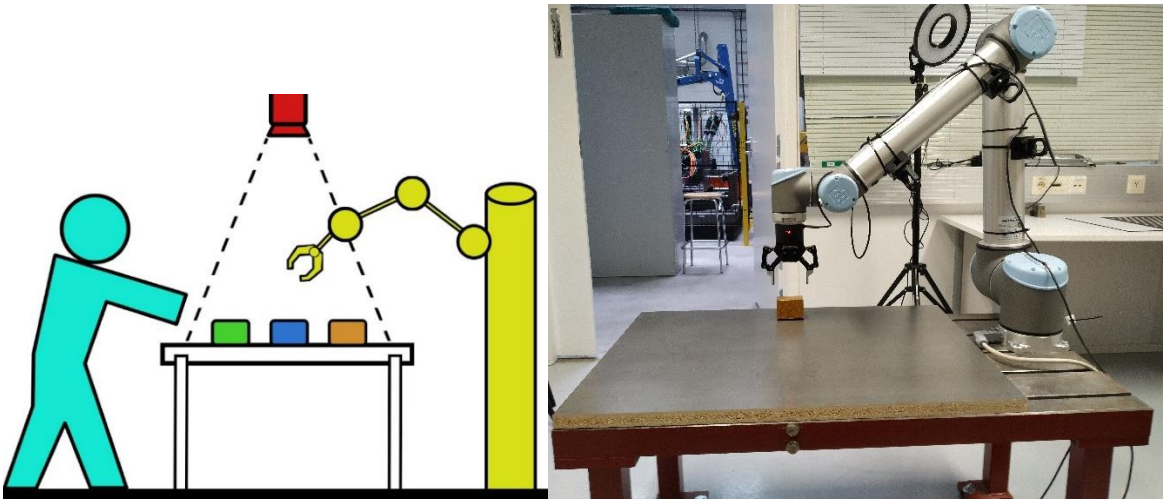

Figure 1. Collaborative robot manufacturing station

1.2 Research problem

As previously defined, a collaborative robot must be able to interact with the operator to be considered as such. The robot is of no use if it requires manual control to function. In addition, a sole UR10 without an end effector has of little value. Prior to this research, no connection or interface had been made with the UR10 or gripper. Only manual control through the teach pendant existed with no communication between the robot and gripper. The Laboratory of Intelligent Machines at LUT had a vision to utilize the UR10 to its full potential as an intelligent collaborative robot. The lack of a framework to build an intelligent system on had to be addressed.

1.3 Objectives

The primary objective of the project was to develop and entirely autonomous system that could intelligently collaborate with a human worker. The robot would receive real-time information from a smart camera to interact according to the operators' actions. Additionally, augmented reality glasses were to be implemented to provide the operator with additional inputs from the workstation. During the project, the scope of the project was scaled back to first develop the basic framework required for the robot to work remotely. The integration of augmented reality glasses and smart vision were delayed until further research could be completed. The main objectives of the project were set as:

- Research of compatible software for the chosen hardware
- Integration of the UR10 and 2F-85 gripper in ROS

- Development of code so that the UR10 can perform simple tasks such as moving objects

1.4 Research methods

The research in this project was conducted to actualize the concept idea of a collaborative robot workstation. The efficient integration of 3rd party hardware with the UR10 required a meta-operating system, such as ROS, where all required hardware could simultaneously be controlled. This project is entirely found upon the freedom and modularity that ROS provides. Therefore, much of the research entailed learning the very basics of ROS from the ground up, to the plethora of community dependent open source software that the ROS community provides. Different distributions of ROS and compatible software had to be tested to find a viable solution to develop a real-world application of the concept idea.

1.5 Novelty and Contribution

The outcome of the research has impact in multiple ways. Primarily the project provides a tested and working framework for controlling a UR10 with an integrated gripper entirely through ROS. Previously neither the co-bot nor the gripper had been controlled in ROS and required manual control through the teach pendant. The software associated with ROS is extremely volatile to change and therefore compatibility amongst all the required hardware and software can cause problems. Furthermore, much of the community developed software has only been tested on relatively old versions of UR firmware and cannot guarantee compatibility with newer versions. During this project, compatibility issues became apparent very early on. Training material and tutorials were often outdated, and their execution resulted in errors. This research provides a proven to work solution that utilizes the latest versions of all respective software to ensure a maximized period of third-party support.

Furthermore, the groundwork completed in the research provides a modular framework that can easily be further developed. Collaborative robots require various sensors that provide force, speed, and distance feedback to ensure safety in the work environment. The implementation of these sensors can easily be done to the framework. The end goal for the workstation is for it to be a fully autonomous, safe, and intelligent. The solution provided by this project enables further research to achieve that end goal.

2 THEORITICAL STUDY AND LITERATURE REVIEW

In this section, a brief history and the origin of the industrial robot is examined to understand the challenges predecessors of the modern robot had to overcome.

The very definition of a robot, as defined by the Merriam-Webster (2020) online dictionary, is “a machine that resembles a living creature in being capable of moving independently (as by walking or rolling on wheels) and performing complex actions (such as grasping and moving objects) “. One of the first examples of robots resembling this definition was a remotely controlled boat engineered by none other than Nikola Tesla himself. Tesla claimed his invention to be the “first of a race of robots, mechanical men which will do the laborious work of the human race”. (O’Neill 1944, p. 170)

The design of early industrial robots began in the United States with a patent, presented in figure 2, for a position-controlling apparatus by Willard V. Pollard in 1938. The apparatus was to be programmed by storing positions within drum memory to control the movement and positioning of a spray gun for coating automobile parts. An example of a similar spray-paint robot, the Trallfa spray-paint robot, developed between 1965 and 1967 is presented in figure 3. Three primary arms and three secondary arms contributed to a total of six axes of motion that were to be controlled by three electric motors. (Pat. US 2286571A 1938, p. 1)

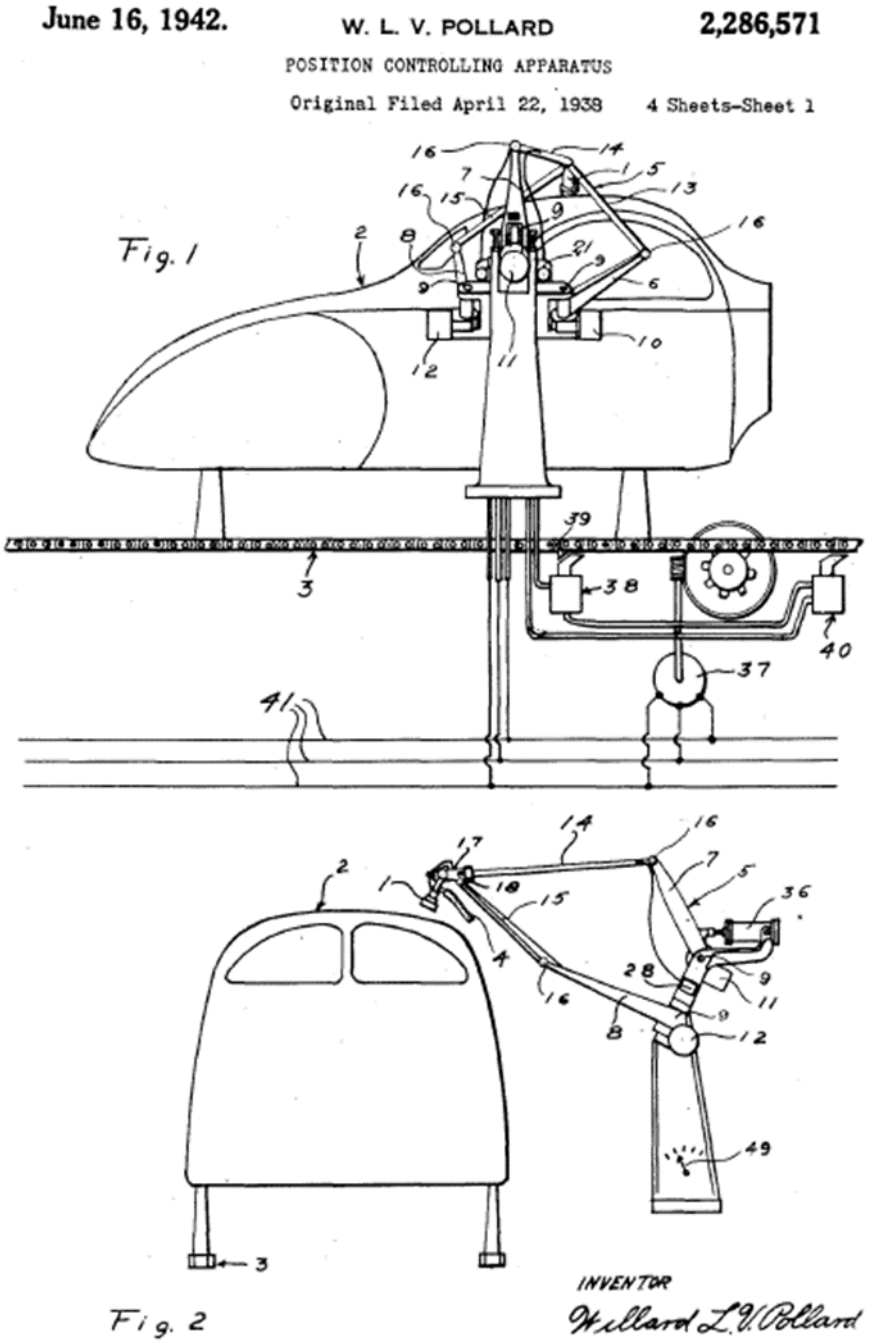

Figure 2. Patent for a spray gun position-controlling apparatus (Pat. US 2286571A 1938, p. 1)

Figure 3. Trallfa spray-paint robot (Cyberneticzoo 2013)

Pollard's vision was a step in the right direction despite his patent never being fully utilized to its' complete potential. The real foundation for industrial robots was built by George Devol and Joe Engleberger who developed the first industrial robot, the Unimate, in 1961 (Robotics Industries Association 2020). The Unimate was based on the patent for programmed article transfer Devol had previously applied for 1954 and used the same drum memory technology as Pollard. The patent describes a general-purpose industrial robot with six degrees of freedom that can be programmed for a variety of real-world applications. In his patent Devol defines that the end effector is manipulated "by a mechanical power source through a sequence of strokes" and that the parameters for the motion could be set by the selected program controller. One of the key components in the patent was the feedback loop Devol describes for achieving precise position and orientation for the end effector. Essentially movement of the end effector would displace a position detector and the status of the detector would be compared to set parameters in the program controller. (Pat. US 2988237A 1954, p. 1) The Unimate was designed as a universal robot that could be easily reprogrammed to cater to a specific task which rendered it a pioneer in the industrial field.
The Unimate, presented in figure 4, by the company Unimation was the first ever industrial robot that made it to production. The Unimate 1900 model was implemented at the New Jersey General motors plant in 1961 with a primary application of manipulating hot die castings. According to IEEE (2020) the Unimate had a position repeatability of 1 mm and could be programmed with hundreds of specific steps, which made it remarkable for its' time.

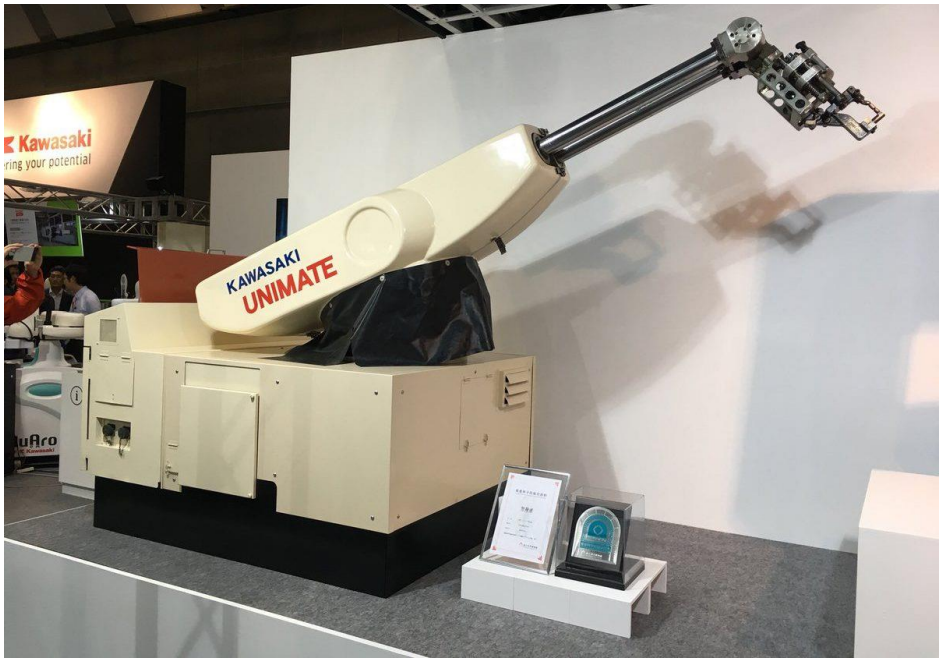

Figure 4. The Unimate 1900 (Kutcher 2020)

The introduction of the Unimate 1900 to industrial factories allowed General Motors to increase efficiency and quality significantly. The real breakthrough came when the General Motors Ohio plant integrated the new Unimate spot welding robots in 1969. Cart production per hour was more than doubled to 110 per hour from the previous maximum threshold that could be achieved without automation. (Robotics Industries Association, 2020 The Unimate 1900 and spot-welding robot also had some serious flaws. The magnetic drum memory made programming slow and difficult. The robots had no sensors, which rendered them oblivious of their work environment creating a serious safety issue that had to be answered. Furthermore, control of the robots was extremely simple point-to-point manipulation.

Up to this point, robotic manipulators had relied heavily on hydraulics to function. In 1969, Victor Scheinman (Irati et al. 2017) developed the first sensor integrated mobile robot at the Stanford Research Institute. The robot, dubbed as Shakey, was the first fully electronically driven robot designed specifically for computer control. The integration of tactile and vision sensors as well as the QA3 question/answer system enabled Shakey to be the first of a generation of intelligent robots that could perform tasks requiring actual planning, as opposed to the previous method of step-to-step programming. (SRI International 2020).

Shortly after in 1973, the Famulus robot by KUKA was developed with design of 6 electric motor-driven axes (Shepherd & Buchstab 2014, p.373) The electromechanical design of the Famulus defined the way modern robots function to this day. The replacement of hydraulic and pneumatic systems allowed industrial robots to be better optimized for assembly line work where the manipulated loads were light, and agility was paramount.

The Sigma (Generic Integrated System for Automatic Handling) robot developed in 1974 by an Italian company called Olivetti was the first robot to have a Cartesian co-ordinate system (Parent & Laurgeau 1984). A Cartesian coordinate robot is a definition for a robot design configuration which has three (X,Y,Z) linear axes of control. (Zhang & Wei B & Rosen, 2017). Fundamentally, it means that the arms of the robot are at right angles to each other and move in a linear fashion. Cartesian coordinate robots are best suited to specific applications such as welding where a linear motion is preferred over more complex motions. An example of a modern cartesian coordinate robot is presented in figure 5. The Sigma respectively was used in assembly, drilling and welding applications where the Cartesian coordinate design was suitable.

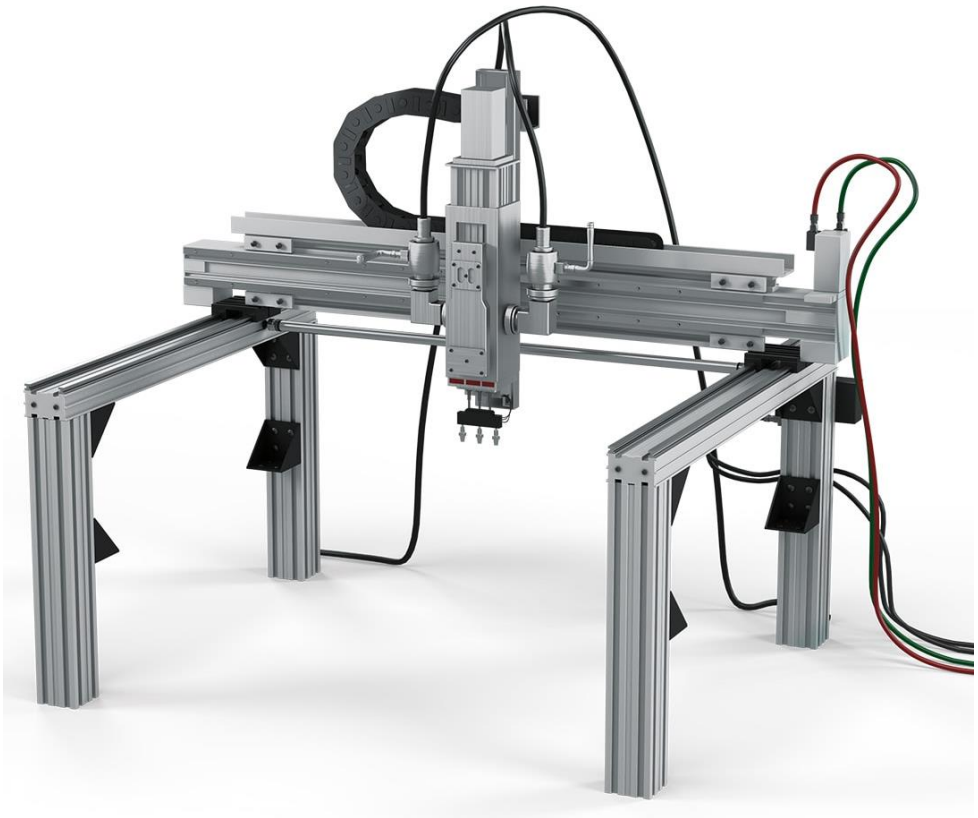

Figure 5. Modern cartesian coordinate robot (Trelleborg 2020)

The Sigma had two arms with each having three degrees of freedom, totaling to a full six degrees of freedom. The modularity of the Sigma enabled the degree of freedom configuration to be altered by adding or removing arms up to eight in total. The Sigma Robot specifications are presented in Table 1. The use of force sensors in the Sigma's grippers granted the ability to detect and report failures. In turn assembly reliability was improved as the Sigma could handle outlier scenarios where a defective part would appear, instead of causing a production stop. (Salmon & d'Auria 1979)

Table 1. Olivetti Sigma robot specifications (Salmon & d'Auria 1979)

Overall dimensions	1950 x 1200 x 2600 mm
Working are	1010 x 400 x 400 mm
Total Weight	1500kg
Maximum Payload	10 kg
Number of arms	1 to 4
Controlled degrees of freedom	Up to 8 step motors
Arm speed	30 m/s
Force feedback: Maximum detectable force Lowest detectable force Force discrimination	 2 kg 50 g 16 g
Data input:	Joystick for position data and/or TeleTYpewriter for Sigla language instructions

The Sigma also had some faults regarding the variety of objects that could be manipulated. Only equally shaped and sized objects could be manipulated without changing the end effector. Retooling the Sigma amounted to 30-50% of the overall cost which made it economically unfeasible. Olivetti Sigma robot specifications (Salmon & d'Auria 1979)

The Programmable Universal Manipulation Arm (PUMA) developed by Victor Scheinman in 1978 and produced by Unimation resembles many of the modern industrial robots present today in industrial assembly lines. The PUMA 560, presented in figure 6, follows the same standard 6 degree of freedom design as the Unimate but all 6 joints are actuated. DC brushed permanent magnet servo motors power each joint and provide position and speed feedback to the control unit (Jaber 2017, p.75).

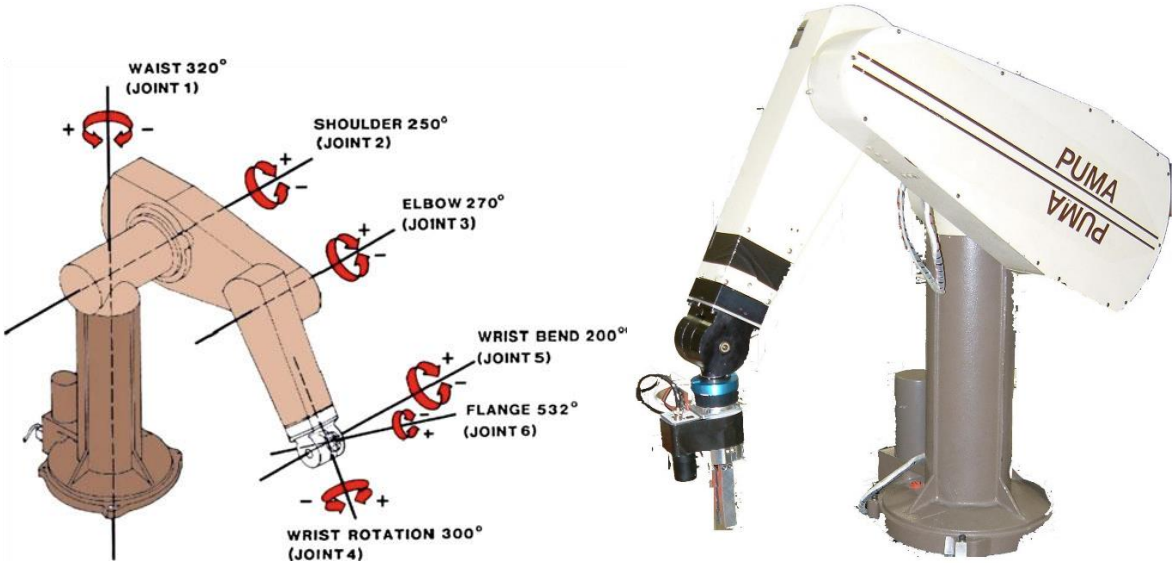

Figure 6. PUMA 560 robot joints with their respective axes and rotation limits (Rutherford 2012)

By the 1980s the investment in robots by companies began to rise dramatically. Sales of industrial robots rose by up to 80% compared to previous years as companies sought to automate assembly lines for arbitrary tasks such as painting, soldering, and moving. Robots began to have certain attributes, such as having computers as dedicated controllers and their own robot programming languages. (Irati et al. 2017) From this point on to the 2000s technological advancements in industrial robots were primarily upgrades to pre-existing systems.

One of these technological advancements was the first use of machine vision in an industrial application by General Motors in 1981 (IFR 2020). The vision system, Consight, used a

combination of light and a stationary line-scan camera to compute the position and orientation of unsorted parts on a conveyor belt (Appleton & Williams 1987, p.83). The GM Consight vision system was capable of sorting up to 6 specific castings at a rate of 1400 per hour. This laid the foundation for future bin picking systems which are a core component of implementing robots in production lines.

In 1984 the first direct drive robot AdeptOne, presented in figure 7, was developed by the American company Adept. Their direct drive SCARA (Selective Compliant Articulated Robot for Assembly) featured electric drive motors that were directly connected to the arms. The direct drive feature made the use of intermediate gears and chains redundant improving accuracy and execution speed significantly. (Gasparetto & Scalera 2019, p31)

Figure 7. AdeptOne the first direct drive robot

Ever increasing efficiency demands in the 1990s pushed researchers to explore innovative solutions regarding kinematic chain designs. One of the developed solutions was the Delta robot that featured a parallel kinematic chain with three translational DOFs and a rotational DOF. (Gasparetto & Scalera 2019, p.33) The Delta robot was based on a patent by Reymond Clavel that defined a robot configuration based on parallellograms.

Essentially the use of parallelograms allowed an output link to remain fixed relative to an input link. The use of three parallelograms rendered the orientation of the mobile platform to be constrained to three translational degrees of freedom. The design of the delta enabled the robot to achieve accelerations rates of up to 12 Gs proving it to be an excellent concept for fast pick and place applications. (Bonev 2001) The functionality of the delta robot design was proven when ABB developed the world's fastest picking robot, the FlexPicker, in 1998. The ABB FlexPicker is presented in figure 8.

Figure 8. World's fastest picking robot the FlexPicker by ABB

From the 2000s on to present day the evolution of industrial robots has focused on the incorporation of Artificial Intelligence (AI), advanced sensors that perform analysis on the gathered data as well as collaboration with humans (Irati et al. 2017). Esben Østergaard, Kasper Støy, and Kristian Kassow founded universal Robots, a Danish company, in 2005

with an investment by Syddansk Innovation. Their vision was to design and produce affordable robot technology that could be applied by small and medium sized manufacturers. Their first collaborative robot, the UR5 presented in figure 9, was released in 2008. The UR5 was a pioneer in the robotics field with it being the first lightweight and flexible collaborative robot. (Universal Robots 2020a) After the initial success of their first cobot Universal Robots decided to develop and manufacture a model with a greater payload and reach called the UR10.

Figure 9. Universal Robots UR5 model with teach pendant

The complex nature of robotics has also sparked an ideology that researchers all over the world should have open access to relative information. As a result, much of the robotics related software, information is open-source, and collaboration is a pillar of the worldwide robotics community. A collaborative project between Willow Garage, a Silicon Valley startup, and the STanford AI Robot (STAIR) and Personal Robots (PR) programs in 2007 resulted in the birth of ROS. (ROS 2020a) ROS has since then revolutionized the robotics industry by providing, as Irati et al. (2017) stated, a “de-facto standard for robot software development”.

3 METHODS AND EQUIPMENT

3.1 Hardware

In this section the used hardware, hardware specifications and physical work environment setup is presented.

3.1.1 UR10

The UR10, presented in figure 10, is a 6-axis collaborative robot manufactured by Universal Robots (UR). Due to its lightweight and flexible design, the applications and redeployment of the robot are practically infinite. A collaborative robot or as they are nicknamed co-bots are designed to ensure safe interaction with humans by including features such as rounded contours, padding and sensors (IFR 2018, s.1). The design of UR-robots embodies these standards and the extensive safety protocols allow the robots to immediately stop upon collision with itself or a human. Technical specifications of the UR10 are provided in Appendix I.

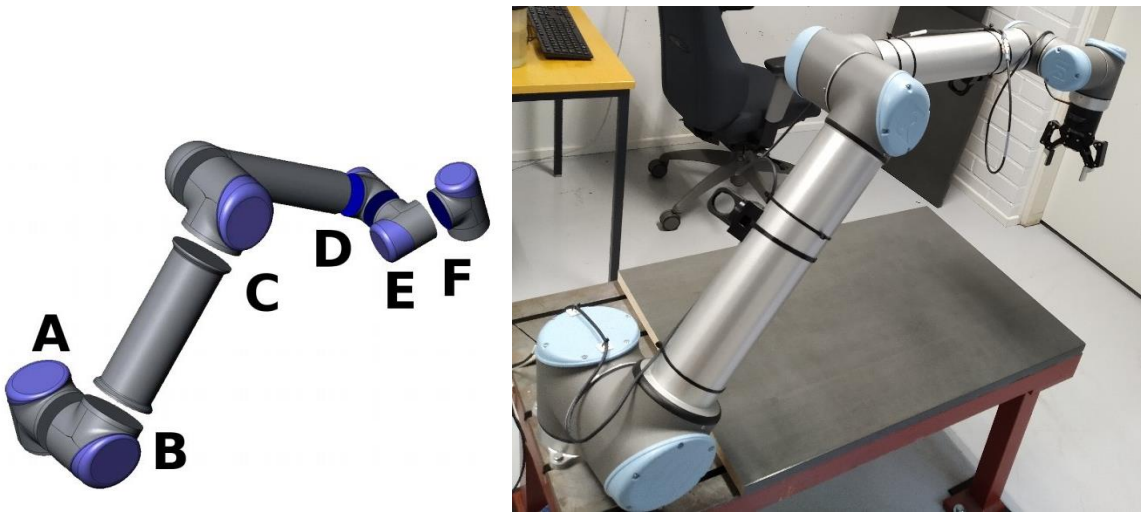

Figure 10. UR10 collaborative robot and respective joints; A: Base, B: Shoulder, C: Elbow, D: Wrist 1, E: Wrist2, F: Wrist 3 (Universal Robots 2020)

The UR10 also includes a teach pendant that can be used to control and program the robot. Effectively the teach pendant is a touchscreen tablet that has pre-installed software, referred to as PolyScope software, which works as a programming graphical user interface (GUI). The intuitive PolyScope software allows a new user to quickly and effortlessly either

individually move each of the six joints or utilize the forward kinematics of the robot to automatically adjust the current pose. The UR10 specifications are presented in Table 2.

Table 2. UR10 Specifications (Universal Robots 2020)

Payload	10 kg
Reach	1300 mm
Degrees of freedom	6 rotating joints
Average Power Consumption	350 W
Pose Repeatability	+- 0.05 mm
Working Range of Joints	+-360 º
Maximum joint speed	Base and shoulder joints 120 º/s Elbow and wrist joints 180 º/s
Typical TCP speed	1 m/s
Robot mounting	Any orientation
Footprint	190 mm
Weight	33.6 kg
I/O Power supply	24V 2A
Communication	500 Hz Control frequency, Modbus TCP, Profinet, Ethernet/IP, USB 2.0, USB 3.0

3.1.2 Robotiq 2F-85 two-finger adaptive gripper

The UR10, like most other collaborative robots, can be equipped with a wide selection of end effectors depending on the required application. The gripper used in the project was a two-finger gripper as shown in figure 11. Robotiq offers both two and three finger options for their grippers. A two-finger model was chosen since it is more efficient at grasping more complex objects evenly. Whilst a three-finger gripper might grasp a complex object unevenly and cause unwanted rotation, or even fail to grasp it at all.

The Robotiq 2F-85 has a stroke of 85mm, grip force of 20-235N and maximum closing speed of 150 mm/s (Robotiq 2020). The fingertips of the 2F-85 are also customizable depending

on the intended application. The Robotiq 2F85 gripper specifications are presented in Table 3 and Appendix II.

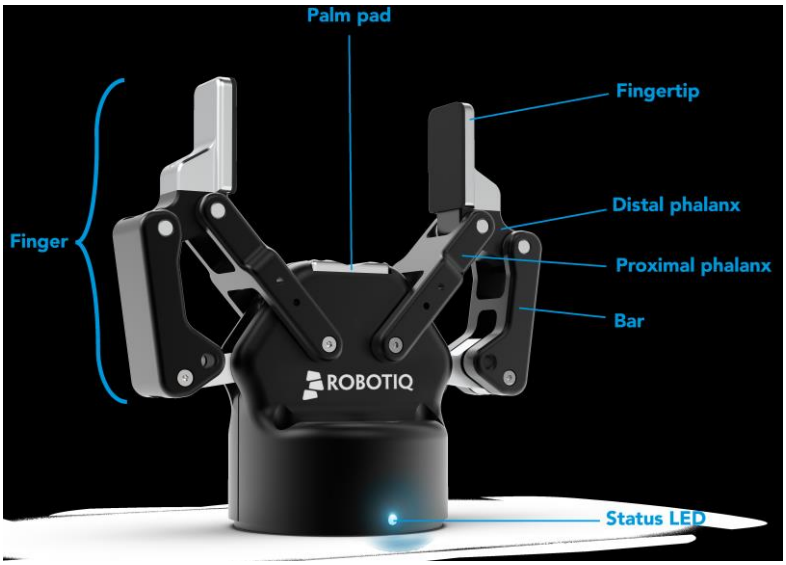

Figure 11. Robotiq 2F-85 two-finger adaptive gripper (Robotiq 2020)

Table 3. Robotiq 2F85 Adaptive Gripper specifications (Robotiq 2020)

Stroke	85 mm
Griper Force	20 – 235 N
Form-fit Grip Payload	5 kg
Friction Grip Payload	5 kg
Gripper weight	0.9 kg
Closing speed	20 – 150 mm/s
Position resolution	0.2 mm
Communication protocol	Modbus RTU RS-48, RS-232

3.1.3 Physical setup and communications network

The framework developed in this research is based on running a virtual machine on top of a windows 10 host system. The virtual machine was installed with an Ubuntu 18.04.4 LTS distribution. Virtualization was conducted via the Oracle VirtualBox (VB) version 6.0.12 r 133076 software. The physical connections of the framework are presented in figure 12. The host machine specifications are presented in table 4.

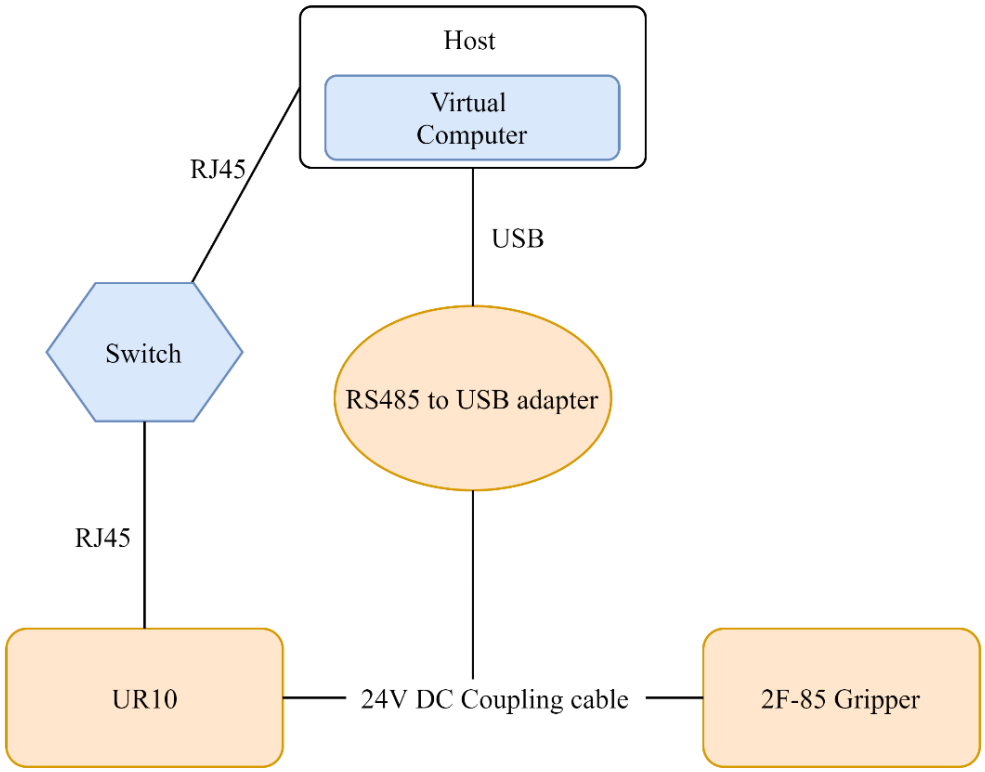

Figure 12. Physical connections of the framework

Table 4. Host machine specifications

Model	Dell Inc. OptiPlex 7050
OS	Windows 10 Enterprise 64-bit
Processor	Intel® Core™ i7-7700 CPU @ 3.6GHz (8 cores)
Ram	32 GB
Graphics card	NVIDIA Quadro P600

The full system required a modular communications network to be established between all hardware. The network had to be designed in a way so that new hardware could easily be integrated in further research. The solution was to use a switch as a hub for connecting the host computer, UR10 and gripper. In this way, other sensors such a vision system can be added to the framework without altering the existing communications network. The switch used was HP Procurve switch 408. Switch specifications are presented in Table 5.

Table 5. HP 408 Switch series specifications (HPE 2020)

Weight	0.68 kg
Ports	8 auto-sensing 10/100 ports (IEEE 802.3 Type 10Base-T, IEEE 802.3u Type 100Base-TX)
Memory and processor	32 KB per port
Performance	Switching capacity: 1.6 Gb/s Routing table size:1000 entries
Standards and protocols	IEEE 802.3x Flow Control
Power consumption	15 W

3.2 Software

In this section, the used software is outlined.

3.2.1 Robot Operating System (ROS)

Quigley et al. (2009) state that ROS provides a structured communications layer above the host operating system of a heterogenous compute cluster and therefore it could be described more accurately as a meta-operating system. The traditional operating system services it provides include hardware abstraction, implementation of commonly used functionality, low-level device control, message-passing between processes and package management. Further features include tools and libraries for obtaining, building, writing, and running code across multiple machines. (ROS 2018a) Designing a framework that caters to all hardware with multiple components and tasks is extremely difficult. Due to this ROS was designed to be flexible and to allow wide-spread collaboration. ROS also supports many programming languages such as Python, C++, and Lisp.

ROS is based on a three-level concept design consisting of the Filesystem level, the Computation level, and the Community level. The Filesystem level consists of stored resources. The Computation level (Computation graph) contains the network of ROS processes that interact with each other. Finally, the Community level refers to ROS resources that allow the exchange of software and knowledge. (ROS 2014) The computation level components are presented in figure 13.

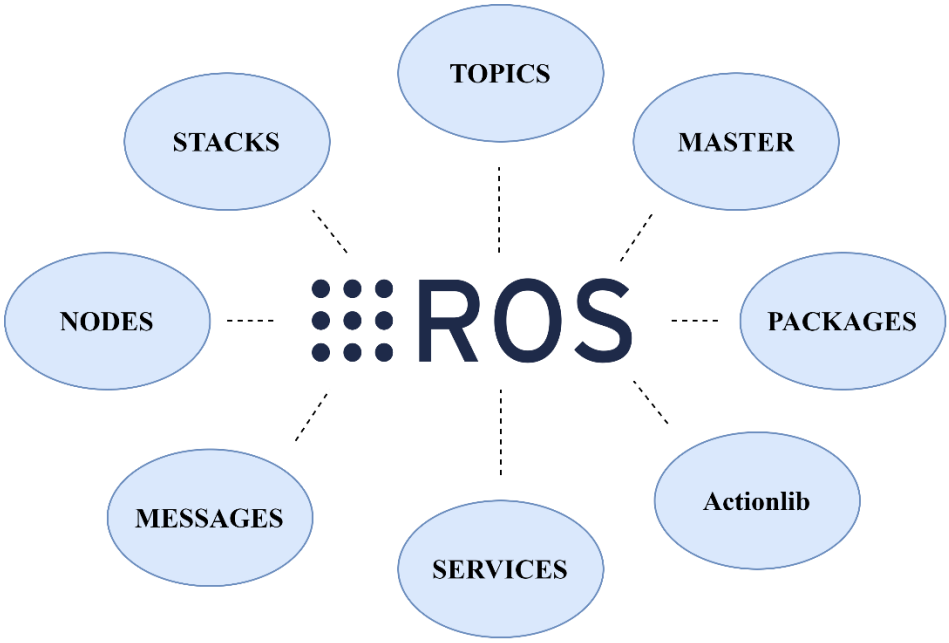

Figure 13. ROS computation level components

The Computation level consists of a Master, Nodes, Topics, Messages and Services that together form a network to process data as presented in figure 14. Actual computation and for example physical actions of a robotic arm, are executed by nodes. A single node will typically control a single arm. These processes contain C++ or Python code to take the designated actions. Nodes can communicate with each other with Messages.

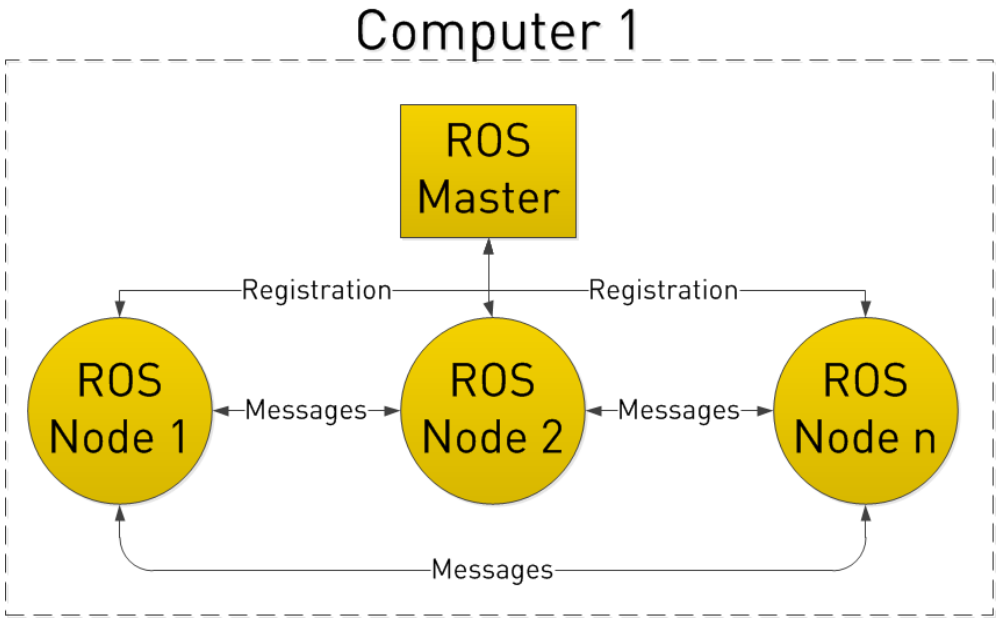

Figure 14. Basic concept of ROS architecture and communication between nodes (CLEARPATH ROBOTICS, 2015)

Messages contain simple data and only support primitive data types. Topics acts as a channel between nodes to transport messages. Topics can have two types of participants: subscribers and publishers. A node with relevant data required in another node can publish to a Topic whilst the other node subscribes to the same Topic. In this manner relevant data, such as a message containing co-ordinates, can be sent from a vision node to an action node through a Topic. This manner of communication between nodes is visualized in figure 15. Nodes can also communicate via services or parameters. Services provide more complex communication such as request / reply interaction that are not appropriate for Topics. The ROS master acts as a host in the computation graph by enabling the processes to register, find each other and communicate. (ROS 2014)

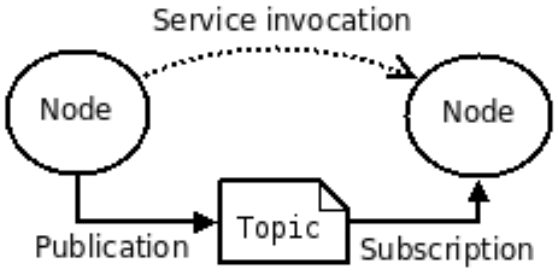

Figure 15. Node communication concept (ROS 2014)

Organization of software in ROS is executed via packages in the File system level. Packages are the most basic form of individual software that can be built in ROS and contains nodes, datasets, third-party software, and other modules. (ROS 2014) At its core a package is a unit that has everything required to build a specific program and run it. A library of packages is a unit referred to as a stack.

The ROS distribution used in this project was ROS Melodic Morenia for Ubuntu Bionic 18.04 LTS. Furthermore 3 external libraries were used which included *MoveIt, Universal Robots ROS Driver* and *Robotiq*.

3.2.2 Imported ROS libraries

Universal Robots ROS Driver- Integration of the UR10 in ROS is achieved with the Universal Robots ROS driver library. The driver has been specifically developed, by

Universal Robots and the FZI Research Center for Information Technology, to provide an efficient interface for UR-robots in ROS. Its predecessors, *ur driver* and *ur modern driver*, were community based and had no affiliation with Universal Robots.

The new driver comes with multiple essential new features such as the ability to extract factory calibration settings from the robot for precise cartesian target acquisition. Operation without factory calibration can cause centimeters of deviation in the simulated position of the tool center point (TCP) and its' real-world position. (GitHub 2020a) In practical applications the orientation of the TCP is of utmost concern as most operations function by manipulating the robot to cater to a desired TCP end pose. The Universal Robots ROS Driver enables control of the UR10 in ROS by interacting with the respective action server or for example using the rqt joint trajectory controller. (GitHub 2020a) Control of individual joints is however not very efficient for any practical applications. A motion planning framework was required for more complex operation.

MoveIt- The *MoveIt* library is an open-source motion planning work that incorporates motion planning, manipulation, inverse kinematics, control, 3D perception and collision checking. Essentially *MoveIT* allows a user to create their own custom *MoveIt package* based on Unified Robot Description Format (URDF) file that contains the specifications of the robot. Then the *MoveIt package* can be configured by calculating self-collisions, defining virtual joints and end effectors as well as setting planning groups for specific kinematic chains. Once the base *MoveIt package* is configured, the robot can be visualized in *RViz*, a ROS 3D visualization tool, and manipulated. The visualization of the UR10 in *RViz* is presented in figure 16. The user can set desired start and end poses for the robot and then execute path planning using a specific planner. Planners in *MoveIt* are path planning algorithms.

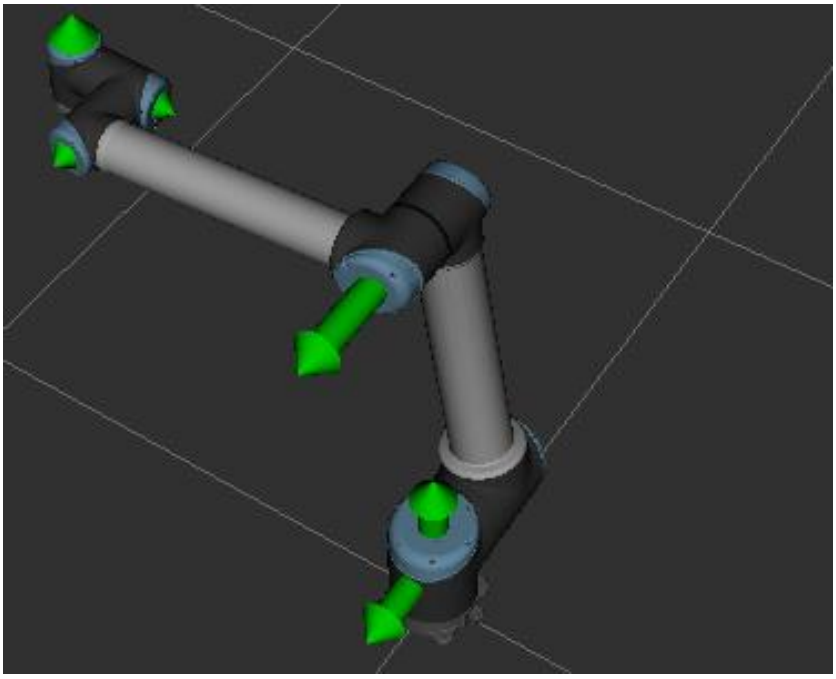

Figure 16. Visualization of the UR10 in *RViz* with respective axes of the six joints

Robotiq- The last external library, *Robotiq*, was required to integrate the gripper in ROS. The *Robotiq* library allows control in ROS of both the 2- and 3-finger grippers as well as the Force Torque Sensor that the company provides. The stack is compatible with either Modbus Transmission Control Protocol (TCP) or Remote Terminal Unit (RTU) protocol.

3.3 Method for defining orientation in three-dimensional space

Defining the orientation of a robot in three-dimensional space is a major component of robot manipulation. In practical applications, the orientation of the end effector must be precisely controlled. A common method to define orientation in robotics is with the use Euler angles. According to Euler's rotation theorem "an arbitrary rotation may be described by only three parameters". (Weisstein 2020). The Euler angle χ-convention is visualized in figure 17.

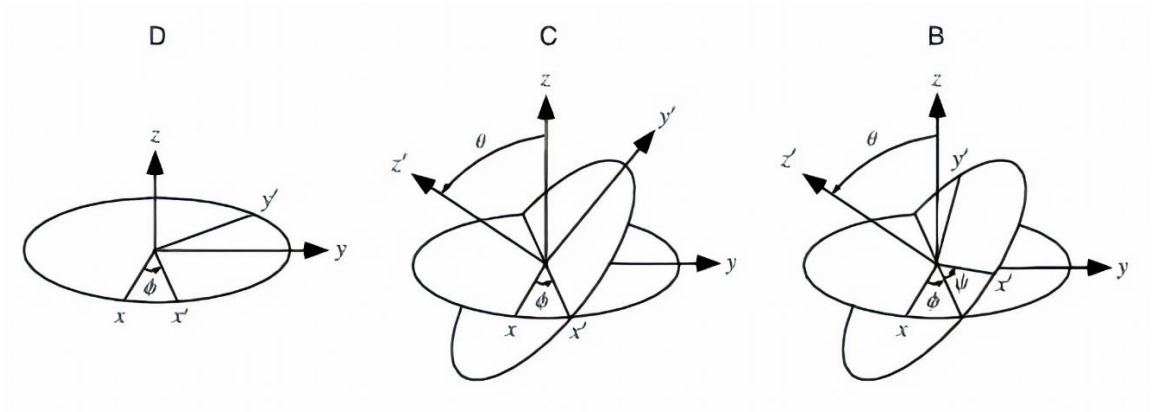

Figure 17. Euler angle χ-convention

If the parameters are written as rotation matrices the resulting rotation matrix A can be written as:

$$\mathbf{A = B\,C\,D} \tag{1}$$

For the χ-convention of the Euler angles the rotations must be applied as follows in the stated order (Weisstein, 2020):

1. A rotation by an angle ϕ about the z-axis (D)
2. A rotation by an angle θ about the x'-axis (C)
3. A rotation by an angle ψ about the z'-axis (B)

Therefore, the rotation matrices for each rotation are (Weisstein 2020):

$$D \equiv \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & cos\ \phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

$$C \equiv \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\ \theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \tag{3}$$

$$B \equiv \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -sin\ \psi & cos\ \psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(4)

With the equations 1. – 4. the resulting Eulerian rotation matrix can be presented as:

$$A \equiv \begin{bmatrix} \cos\phi\ \cos\psi - \sin\phi \sin\psi \cos\theta & -cos\phi \sin\psi - sin\phi \cos\psi \cos\theta & \sin\psi \sin\theta \\ sin\phi \cos\psi + cos\phi \sin\psi \cos\theta & -sin\phi \sin\psi + cos\phi \cos\psi \cos\theta & -cos\phi \sin\theta \\ \sin\psi \sin\theta & \cos\psi \sin\theta & \cos\theta \end{bmatrix}$$

(5)

The use of Euler angles does however have an issue called gimbal lock. Gimbal lock refers to a scenario where two axes coincide after a rotation with a permutation of 90 degrees about a third axis. Gimbal lock causes a temporary loss of a degree of freedom (Rotenberg 2016). This is where quaternions, the allocated way to define orientation in *MoveIt*, come in. Quaternions are a method for describing three-dimensional rigid body orientations as four-dimensional vectors. A quaternion is written as (Mordecha 2018):

$$q_0 + q_1 i + q_2 j + q_3 k$$

(6)

Where q_0 is a real number and the rest form a vector in imaginary space. The complex parts (i, j, k) are satisfied by these identities (Mordechai 2018):

$$i^2 = j^2 = k^2 = -1$$

(7)

$$ij = k, ji = -k$$

(8)

$$jk = i, kj = -i$$

(9)

$$ki = j, ik = -j$$

(10)

For example, a vector in three-dimensional space in quaternion form would look as follows (Mordechai, 2018):

$$q = 0 + xi + yj + zk$$

(11)

For defining orientation, a unit quaternion is used. A unit quaternion is defined in equation 12. A unit quaternion consists of vectors that collaborate to from the surface of a four-dimensional hypersphere with a radius of 1 (Rotenberg 2016).

$$|\boldsymbol{q}| = \left(\sqrt{q_0 + q_1 + q_2 + q_3}\right) = 1$$

(12)

As an example, a quaternion format for representing a rotation around an axis a by an angle θ is presented in equation 13 (Rotenberg 2016).

$$\boldsymbol{q} = \left[\cos\frac{\theta}{2}, \quad a_x \sin\frac{\theta}{2}, \quad a_y \sin\frac{\theta}{2}, a_z \sin\frac{\theta}{2}\right]$$

(13)

3.4 Integration of the hardware in ROS

In this section, the methods for establishing a connection in ROS for the UR10 and Robotiq 2F85 gripper are described.

3.4.1 Control of the UR10

Integration of the UR10 in ROS begins with the installation of the externalcontrol-1.0.urcap on the UR10. URCaps could be summarized as apps but for UR-robots. They can perform a multitude of things such as provide GUIs for connected 3rd party hardware such a grippers or other sensors. In this case the external control URcap acts as a consent program on the PolyScope software that grants external control of the UR10 in ROS. (Github 2020a) Secondly forward and inverse kinematics calibration parameters must be extracted from the

hardware to ensure accuracy of the end effector positions. The extracted kinematics calibration file is presented in appendix III. Kinematics calibration is extracted and save with the following command:

```
roslaunch ur_calibration calibration:correction.launch
\robot_ip=192.168.100.103
target_filename:"${~catkin_ws/src/support/config}/calibr
ation.yaml
```

Once these prerequisites are met, the actual driver can be started with the following command that simultaneously passes the previously attained calibration file to the driver:

```
roslaunch ur_robot_driver ur10_bringup.launch
robot_ip=192.168.100.103 \ kinematics_config:=$(rospack
find support/config}/calibration.yaml
```

To finalize the connection of the UR10 in ROS the external control URcap must be executed on the teach pendants' PolyScope software. When the connection is successful and control is achieved, the driver node terminal will display the output "*Robot ready to receive control commands"*. At this point, the robot can be controlled by using the */scaled_pos_traj_controller/follow_joint_trajectory* action server (Github 2020a).

3.4.2 Control of the 2F-85 Gripper

To integrate the gripper in ROS either Modbus RTU or TCP can be used. In this project RTU protocol with a serial to USB converter was used due to its' practicality. It is worth noting that the TCP protocol could be more practical in real world applications if the host computer is located further from the collaborative robot. The gripper requires two packages, *Robotiq 2f gripper control* and *robotiq modbus rtu*, from the *Robotiq* library to be integrated in ROS. First, the respective serial port the USB is attached to must be configured. The right serial port can be located and then consecutively configured to give the proper privileges to the user with the following commands:

```
dmesg | grep tty
```

```
usermod -a -G dialout USERNAME
```

After the serial configuration, the gripper driver can be launched with the following command:

```
Rosrun robotiq_2f_gripper_control
 Robotiq2FGripperRtuNode.py /dev/ttyUSB0
```

The driver is subscribed to the Robotiq2FGripperOutput topic and received messages are converted and sent as commands to the gripper (ROS 2018b). A Terminal-based GUI simple controller is also included in the package and can be used to test the functionality of the gripper within a terminal as shown in figure 18. The GUI controller node can be run with the following script:

```
rosrun Robotiq_2f_gripper_cotrol
 Robotiq2FGripperSimpleController.py
```

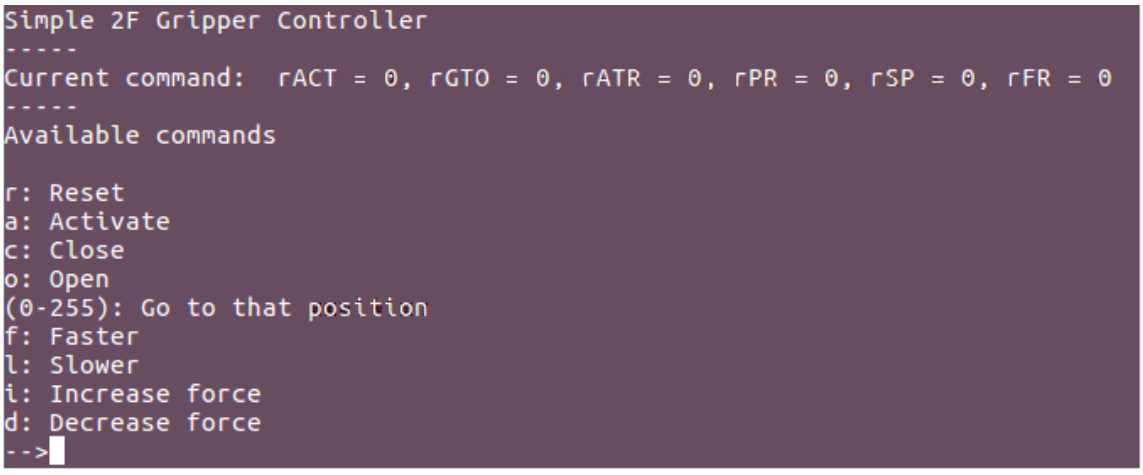

Figure 18. Terminal based GUI for interacting with the gripper

4 CASE STUDY: PICK AND PLACE APPLICATION

4.1 Methodology and code development

In this chapter, the steps of the code development for a simple pick and place application are presented. The code development consisted of generating a custom *MoveIt* package, manipulation of the UR10, Manipulation of the gripper and motion planning. All the aspects of code development presented in this chapter are combined to form a pick and place node that can be setup to perform the picking and placing of objects with preset positions and orientations. It must be stated that this part of the research would not have been possible without the *MoveIT* (GitHub 2020b) and ROS Industrial (GitHub 2019) tutorials. A visualization of the framework required for the pick and place application is presented in figure 19.

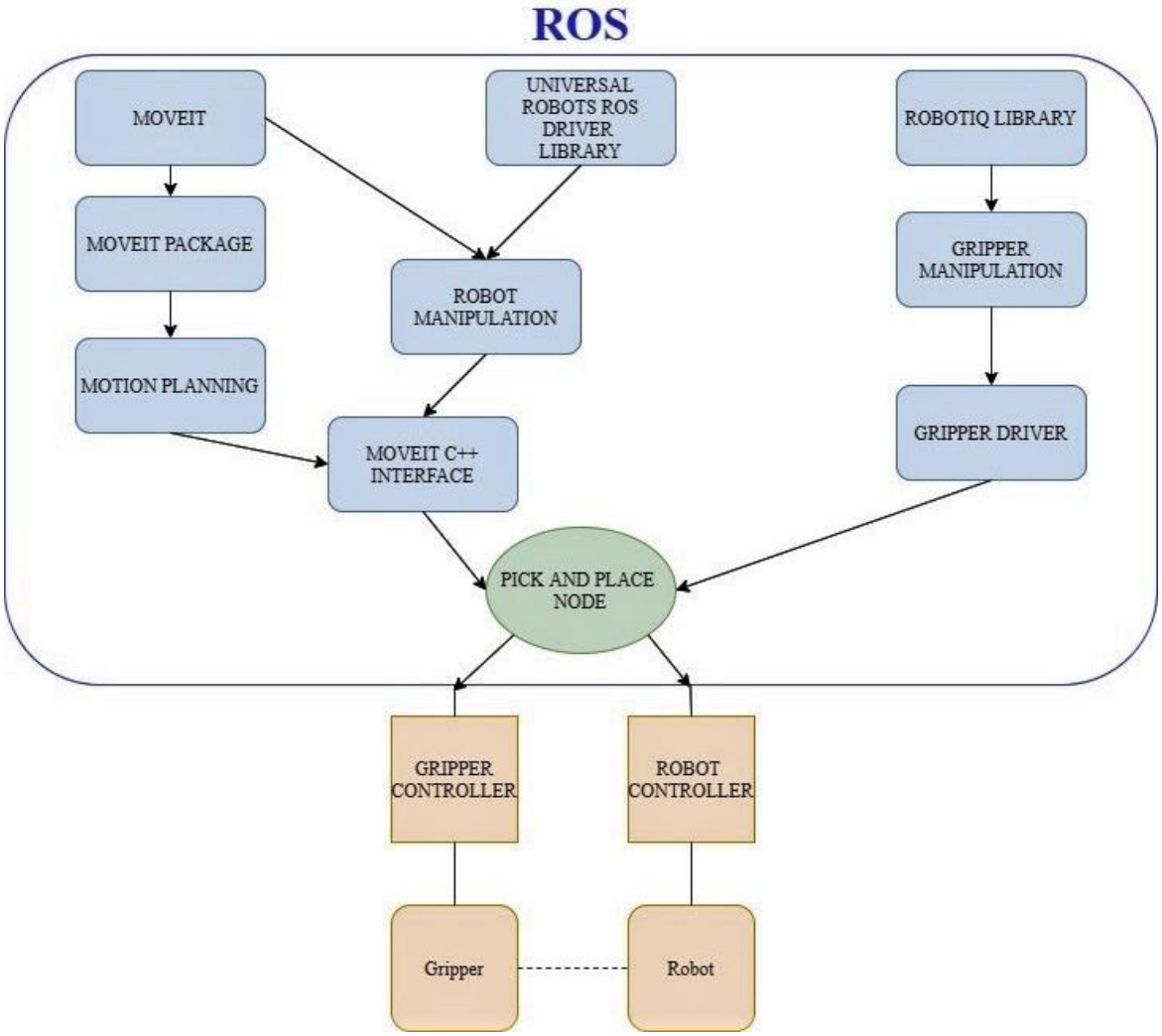

Figure 19. Pick and place application framework

4.2 *MoveIt* package

A *MoveIt* package consists of configuration and launch files required to utilize the *MoveIT* motion planning library with a specific robot configuration. To build a new *MoveIt* package a URDF file must be defined. URDF is an Extensible Markup Language (XML) based way to describe a robot in a tree structure format that consists of link and joint elements. The URDF defines the kinematic and dynamic description, visualization, and collision model of the robot (ROS 2012). An example of the URDF tree structure is presented in figure 20.

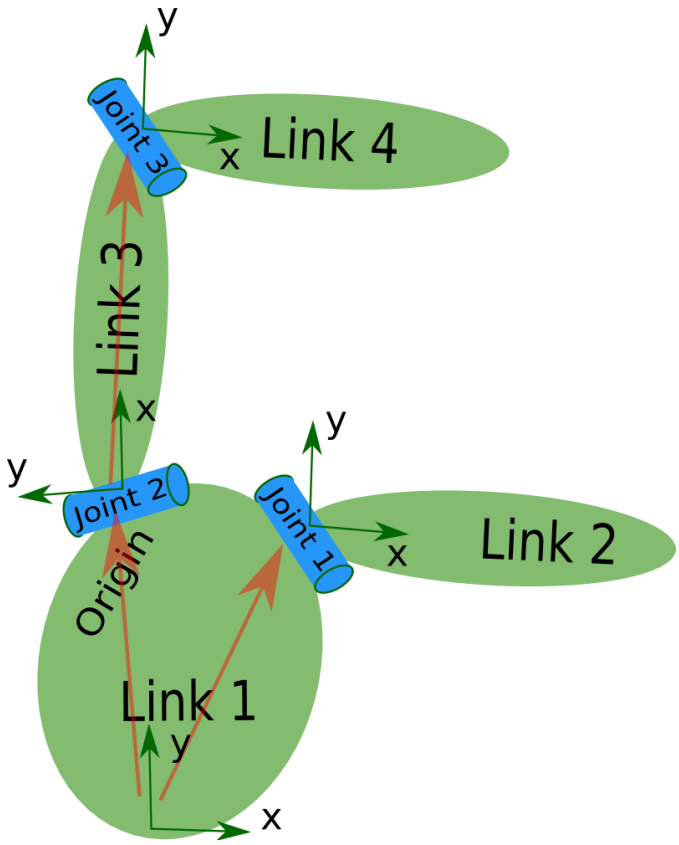

Figure 20. Visual representation of a URDF file tree structure (ROS 2012).

Link elements describe a physical rigid body between joints such as a part of the ur10 arm that has inertial, visual and collision elements. The inertial element is constructed by defining an origin for the inertial reference frame relative to the link reference frame, setting the mass of the link and by including a 3×3 symmetrical rotational inertia matrix. Collision and visual elements are both defined by specifying their geometry. Often the collision and visual properties can be very similar; however, collision elements can be simplified to optimize computation time (ROS 2012).

Joints elements complement link elements by providing the required kinematics and dynamics descriptions. Joint elements bind link elements together into a chain by defining a parent and child link that are attached by the joint. A joint element is comprised of a parent and child element that defines which links the joint is attached to in the tree structure. Optional elements such as calibration, limit, mimic, safety controller and dynamics can also be included. Particularly the dynamics element, which defines physical damping and friction values for the joint, is important for simulation.

The packages used in this project provided URDF files for both the UR10 and the 2F85 gripper constructed by their respective manufacturers. The URDF files had to combined in a single *Xacro* file to include both in simulation and collision checking. *Xacro* is an XML macro language used to simplify URDF files by utilizing macros to call on other XML files to expand to larger XML expressions (ROS 2020b).

The master *Xacro* file calls macros for the UR10 and 2F85 gripper assemblies. Both files define and instantiate the URDFs required. Therefore, the master *Xacro* must only define the final links and joints required to build the tree structure. The master *Xacro* defines a virtual world link and arm *tcp* link with no geometry. The world link is used to attach the UR10 to the virtual world. The arm *tcp* link is used to simulate the change in the actual TCP of the end effector caused by the addition of the gripper. According to the Robotiq (2020) user manual, the TCP of the 2F85 is at 171 mm on the z-axis. The three joints included in the xacro (world to robot, gripper to robot and arm joint tcp) define how the links are attached to the tree structure. The complete master Xacro is included in Appendix IV.

The completion of the master *Xacro* allowed a new *MoveIT* configuration package to be build using the *MoveIt* setup assistant. The configuration begins by generating a self-collision matrix for all links in the *Xacro*. Then a virtual base joint is added that matches the root link that was defined in the Xacro. Motion planning in *MoveIT* also requires a planning group to be set that consists of the entire kinematic chain to be manipulated. In the case of the UR10, the kinematic chain is set from base link to tool 0. Due to the addition of the gripper, the kinematic chain was in this case set to end at the new tool center point. Motion planning for the gripper was not conducted via *MoveIT* and therefore a planning group was

not set for the gripper. All joints of the gripper were set as passive therefore rendering it under actuated. The final *MoveIT* package was built with the previously stated parameters.

Once the *MoveIt* package was built, some modifications were required before actual application in motion planning. The package requires the creation of a controller (Appendix V), joint names (Appendix V), controller manager (Appendix V) and planning execution file (Appendix VI). In the controller file the UR10 controller to be used is with *MoveIT* motion planning is set. In this research, the scaled position trajectory controller was used. The joint names file defines the joints to be controlled by the selected controller. The controller manager launches the set controller during execution. The planning execution launch file executes the actions presented in figure 21.

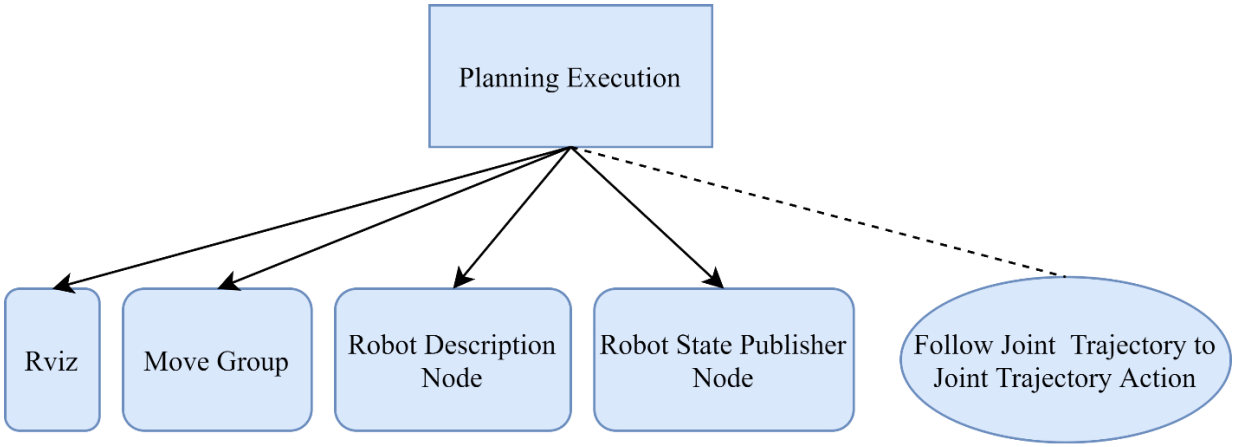

Figure 21. Actions of the planning execution file with arrows indicating a launch action and dashed line indicating a remap action.

4.3 UR10 manipulation

MoveIt provides a simple user interface for the manipulation of robots using scripts with its C++ and python wrappers. In this project, the *MoveIt* C++ wrapper was used for manipulation of the UR10. The MoveGroup class included in MoveIt provides the C++ user interface. The *MoveGroup* class allows the user to script very practical actions such as setting joint specific or pose specific goals as well as create motions plans for these goals. The class also allows the user to add collision objects in the working environment or set path constraints to guide path planning. All off the *MoveGroup* class functionalities are conducted by interacting with the *MoveGroup* node through topics, services, and actions.

Manipulation through the *MoveIt* C++ interface is based on planning groups. The planning group defined in the constructed *MoveIt* configuration package, “manipulator”, is used to define certain goals. The manipulator-planning group contains the full set of joints defined in the kinematic chain from the UR10 base link to the arm tcp link. The planning group is called upon in the interface and stored in the JointModelGroup (also referred to as PLANNING GROUP) object at the beginning of the script and the MoveGroup class is initialized:

```
Static const std::string PLANNING_GROUP = “manipulator”;
moveit::planning_interface::MoveGroupInterface
move_group(PLANNING_GROUP);
```

According to the *MoveIt* MoveGroup C++ interface tutorial (GitHub, 2018) raw pointers are used for the planning group due to performance.

```
Const robot_state::JointModelGroup*
joint_model_group = move_group.getCurrentState()
->getJointModelGroup(PLANNING_GROUP);
```

Once the planning group is set, a move goal can be defined in multiple ways. The C++ interface allows the user to set a pose, joint-space or Cartesian path goal for the end effector. In this research, the pose goal method was used as the method to define manipulation goals. The *move_group.SetPoseTarget* method requires a target pose to be set according to the *geometry_msgs:Pose* message format. The pose message is formed of a point position and quaternion orientation.

Point position fields:

```
float64 x
float64 y
float64 z
```

Quaternion orientation fields:

```
float64 x
float64 y
float64 z
float64 w
```

After the desired goal pose for the end effector is defined, the path from the start state to the goal state can easily be computed using the Plan function and stored:

```
moveit::planning_interface:MoveGroupInterface::Plan
my_plan;
bool success = (move_group.plan(my_plan) ==
moveit::planning_interface::MoveItErrorCode::SUCCESS);
```

Once a motion plan has been successfully computed it can be executed by calling the *move.group* to move:

```
move_group.move();
```

Manipulation of the UR10 using this method is simple but allows to user to script a wide range of actions for the end effector. Path constraints and collision objects can be included to cater to applications that are more complex where the workspace is not completely free. The actions of the UR10 can also be fined tuned to an application by setting certain move group parameters, such as the max velocity scaling factor and planning time, accordingly.

4.4 Gripper manipulation

The method for controlling a gripper in ROS requires three components as stated by the ROS (2010) wiki and is presented in figure 22. The required top-level component is a program that sends higher-level commands to an action node. The second component is an action node that accepts the higher-level commands and converts them to low-level commands. After the conversion, the action node sends the low-level commands to a controller. The controller acts as the third component and commands the grippers' joint motors directly.

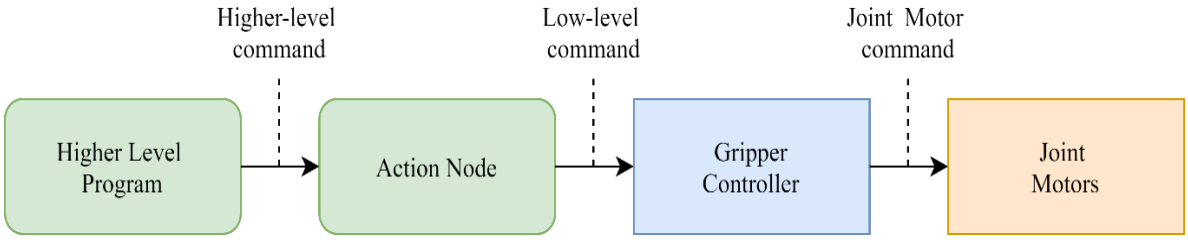

Figure 22. Method for controlling a gripper in ROS

In this project, the respective controller is the embedded controller in the gripper, the action node is the gripper driver provided by Robotiq and the higher-level program is the pick and place node. The process of control is visualized in figure 23.

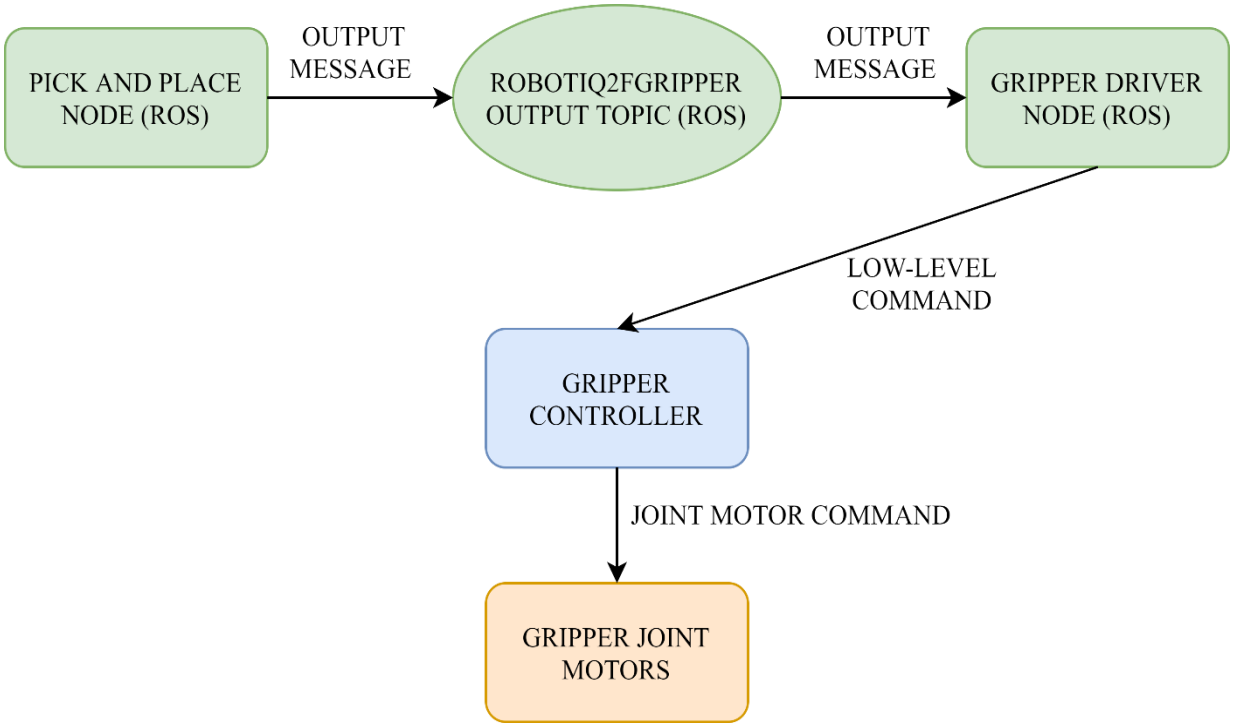

Figure 23. Gripper control process

Due to Robotiq's user friendly design only the higher-level node had to be constructed to control the grippers' actions. The gripper driver node is subscribed to the *Robotiq2FGripperRobotOutput* topic and can receive messages of the *Robotiq2FGripper robot output* type. The respective message type contains six fields of unsigned 8-bit int primitives.

```
Uint8 rACT
```

```
Uint8 rGTO
Uint8 rATR
Uint8 rPR
Uint8 rSP
Uint8 rFR
```

The names of the fields are defined in the Robotiq user manual. Definitions of field names are presented in Table 6.

Table 6. Robotiq2FGripper robot output message field name definitions (Robotiq 2020b)

rACT	Activates gripper
rGTO	Moves the Gripper fingers to a defined position.
rATR	Slowly opens the Gripper fingers until all motion axes reach their mechanical limits.
rPR	Desired position (0-255). At 0 the gripper is fully opened and 255 fully closed.
rSP	Desired speed (0-255).
rFR	Desired force (0-255). 0 for very fragile objects. 127 for solid and fragile objects. 255 for solid and strong objects.

To control the gripper with a desired action the correct output message must be defined and sent from the pick and place node. It must also be stated that the gripper will not accept any other commands before it has been activated by sending an activation command. First, the right message type must be included in the pick and place node and the node must be configured to publish the respective message to the right topic. In this case, the topic is *Robotiq2FGripperRobotOutput* and the message type is *Robotiq2FGripper_robot_output.*

```
ros::Publisher
pub=node_handle.advertise<robotiq_2f_gripper_control::Ro
botiq2FGripper_robot_output>("Robotiq2FGripperRobotOutpu
t",100);
```

A loop rate is also set, and a commands class is defined for storing the parameters for the gripper message.

```
ros::Rate loop_rate(1); //sets a rate of 1 per second
robotiq_2f_gripper_control::Robotiq2FGripper_robot_outpu
t commands;
```

Then the desired action is configured by setting the fields appropriately, in this case the fields are set to activate the gripper, and the message is published to the topic:

```
commands.rACT = 1;
commands.rGTO = 1;
commands.rATR = 0;
commands.rPR = 0;
commands.rSP = 255;
commands.rFR = 150;
pub.publish(commands); // publishes the message
loop_rate.sleep(); //tells ROS to sleep until the rate
is fulfilled
```

This method can be replicated to perform any desired action for the gripper. To open the gripper, the parameters are configured as:

```
commands.rACT = 1;
commands.rGTO = 1;
commands.rATR = 0;
commands.rPR = 0;
commands.rSP = 255;
commands.rFR = 150;
```

To close the gripper, the parameters are configured as:

```
commands.rACT = 1;
commands.rGTO = 1;
commands.rATR = 0;
commands.rPR = 255;
commands.rSP = 255;
```

```
commands.rFR = 150;
```

4.5 Path planning and collision environment

Motion planning in *MoveIt* is conducted via the motion planner plugin interface. The plugin allows the user to choose a planner from multiple external libraries (*MoveIt*, 2020). The *MoveIt* motion planning pipeline concept is presented in figure 24. The default library called the Open Motion Planning Library (OMPL) was used in this research. OMPL is a sampling-based motion planning library containing numerous planning algorithms including the Probabilistic Roadmap Method (PRM), Rapidly-expanding Random Trees (RRT) and Kinodynamic Planning by Interior-Exterior Cell Exploration (KPIECE). (Șucan, Moll & Kavraki 2012)

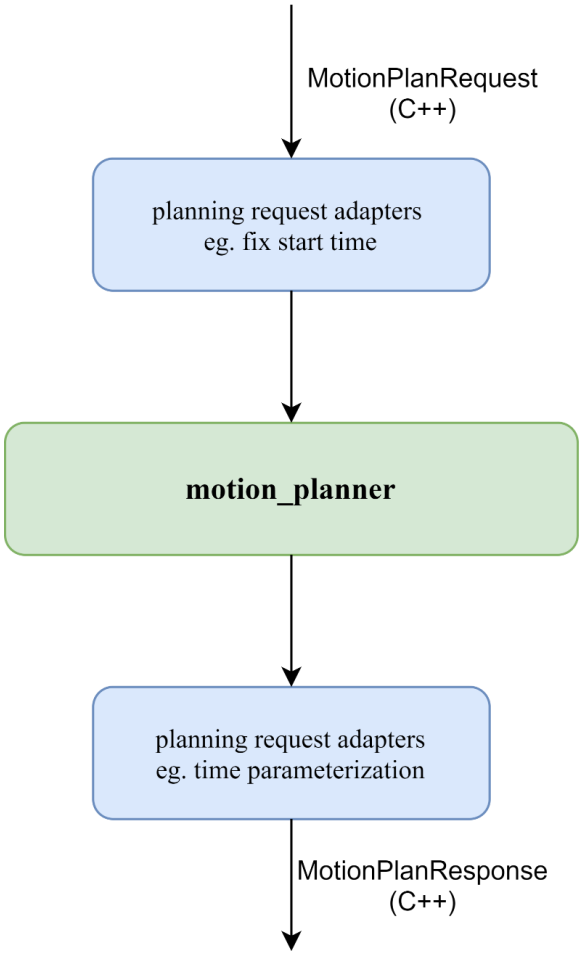

Figure 24. *MoveIt* motion planning pipeline (MoveIt 2020)

Sampling-based motion planning refers to a method that attempts to solve queries without constructing a map of the entire workspace. The method determines whether the robot configuration is in collision at any point in the given solution. (Khaksar, Sahari & Hong 2016) According to OMPL (2019) the Kinematic Planning by Interior-Exterior Cell Exploration (KPIECE) and Lazy Bi-directional KPIECE with one level of discretization (LBKPIECE1) planners have been proven to work with real-world problems and therefore the LBKPIECE and KPIECE planners were used.

Utilizing the path planning algorithms to their full potential requires describing the real-world environment surrounding the robot through the *MoveIt* C++ interface. The planning scene topic is used to define and uphold the environment state as well as robot state. The planning scene monitor from the move group node maintains the actual planning scene. (*MoveIt* 2020) A visual representation of the planning scene workflow is presented in figure 25.

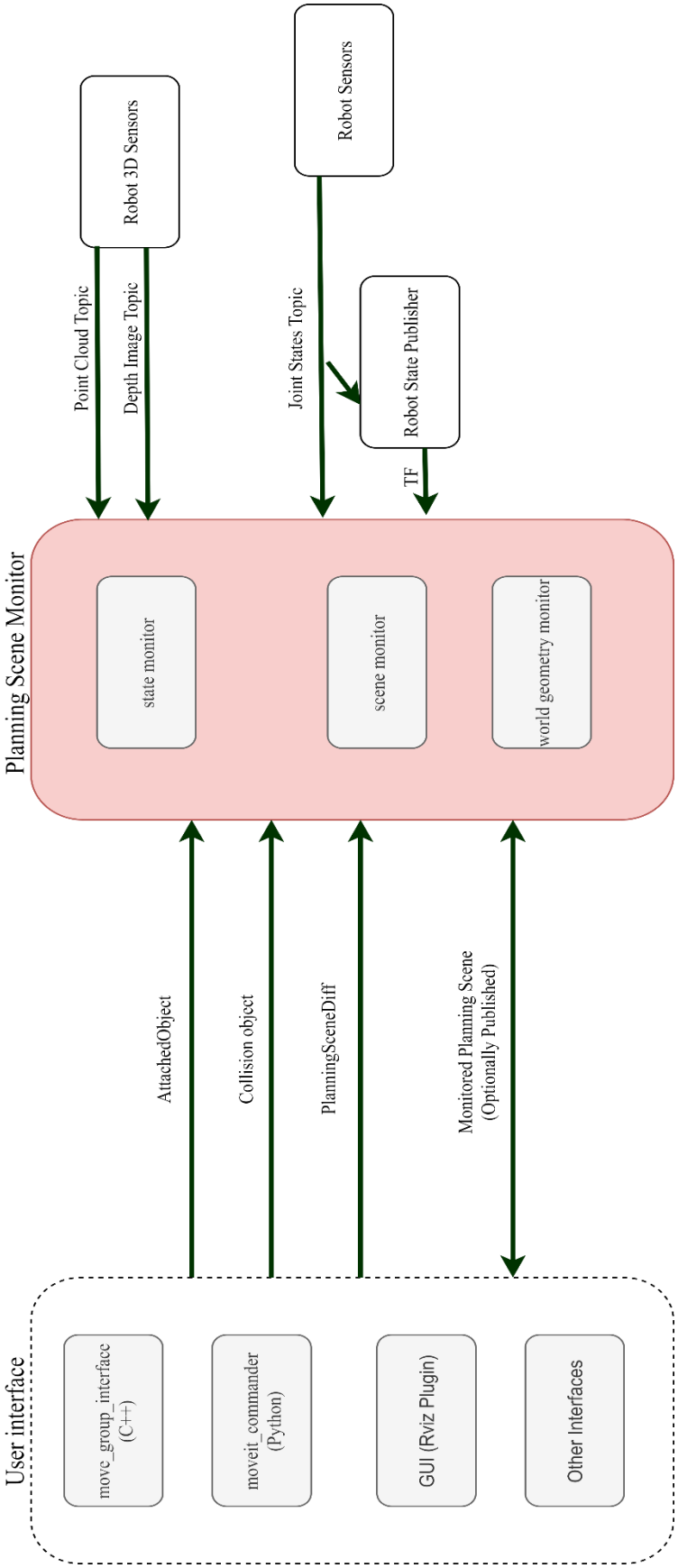

Figure 25. Representation of the planning scene workflow (*MoveIt* 2020)

Adding collision objects to the planning scene in *MoveIt* is done by initializing the planning scene interface class.

```
moveit::planning_interface::PlanningSceneInterface
planning_scene_interface;
```

Then the desired collision object from the real world must be simulated and defined as a collision object message. Essentially the message defines the object's action, type, pose and geometry. Collision checking in *MoveIt* is supported for meshes as well as primitive shapes such as boxes, cylinders and planes (*MoveIt* 2020). In this project, the box primitive type was used as the work environment was relatively open and the only real limitation for movement was the workstation itself. The workstation is defined as a collision object by first initializing a collision object and adding an id.

```
moveit_msgs::CollisionObject collision_object;

collision_object.header.frame_id =
move_group.getPlanningFrame();

collision_object.id = "table";
```

Then the object type and geometry are defined.

```
shape_msgs::SolidPrimitive primitive;
primitive.type = primitive.BOX;
primitive.dimensions.resize(3);
primitive.dimensions[0] = 0.6;
primitive.dimensions[1] = 1;
primitive.dimensions[2] = 0.025;
```

Next, the pose of the object is defined.

```
geometry_msgs::Pose box_pose;
```

```
box_pose.orientation.w = 1.0;
box_pose.position.x = 0;
box_pose.position.y = 0.6;
box_pose.position.z = 0;
```

Finally, the geometry, pose and action are set.

```
collision_object.primitives.push_back(primitive);
collision_object.primitive_poses.push_back(box_pose);
collision_object.operation = collision_object.ADD;

std::vector<moveit_msgs::CollisionObject>
collision_objects;
collision_objects.push_back(collision_object);
```

Once the collision object is fully defined, it can be added to the planning scene by sending the collision object message.

```
planning_scene_interface.addCollisionObjects(collision_o
bjects);
```

The final representation of the workstation defined as a box primitive collision object and visualized in *RViz* is presented in figure 26.

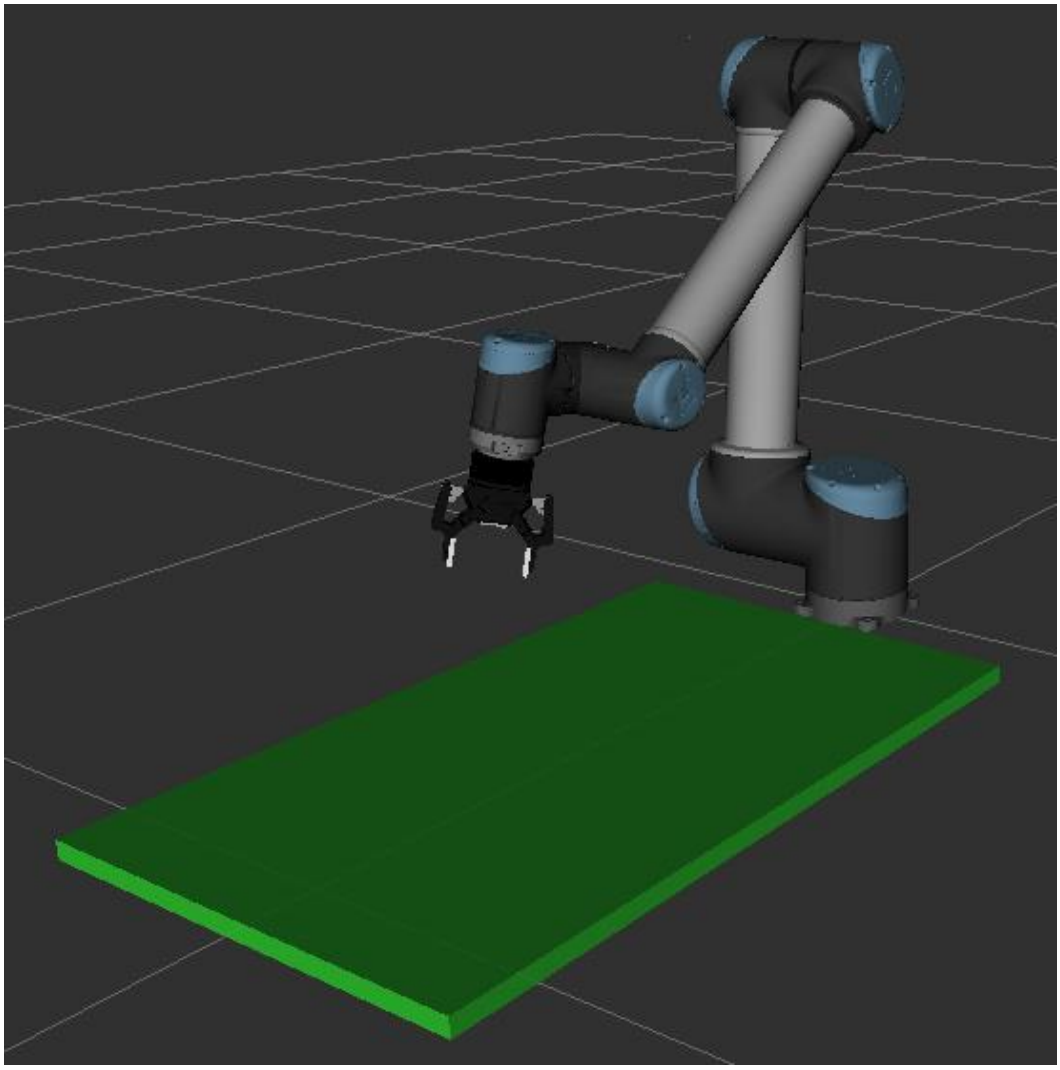

Figure 26. Simulation of the workstation as a collision object in *RViz*

4.6 Experiment

The functionality of the configured motion-planning framework was tested with a simple task of planning and executing between two set poses. The UR10 was manipulated from an initial *start* pose to a *wait* pose and back to the *start* pose to simulate a simple pick and place task. The task was repeated with the RTT connect, KPIECE and LBKPIECE1 planners. The *start* pose is presented in figure 27 and the wait pose in figure 28, respectively.

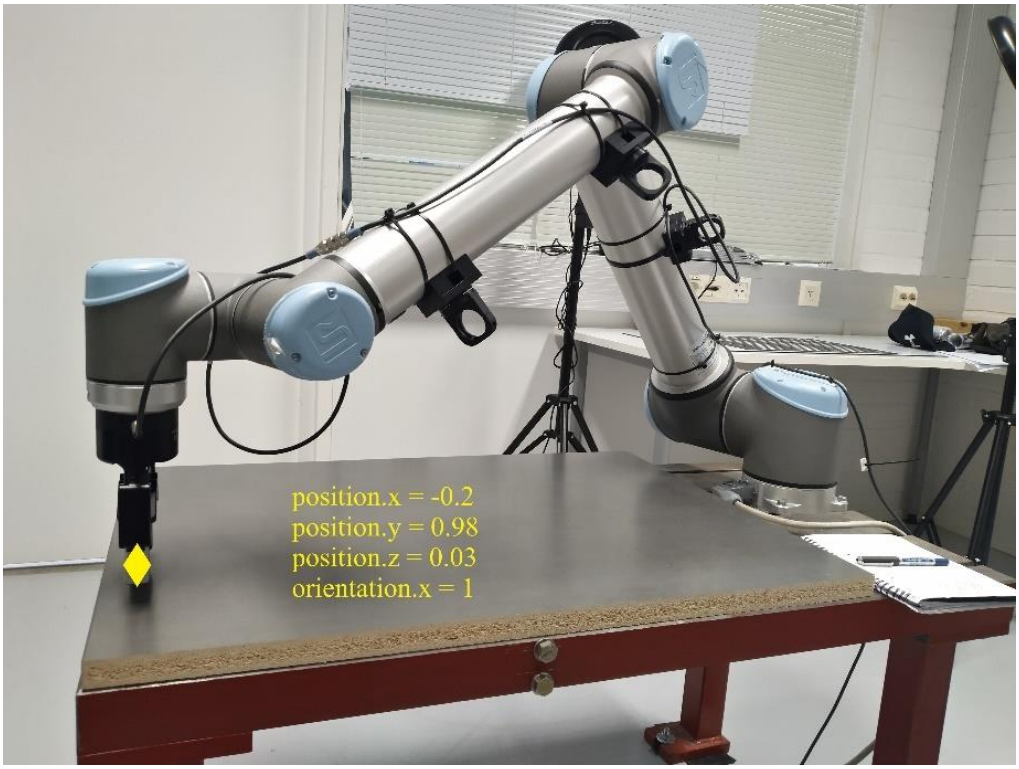

Figure 27. Start pose

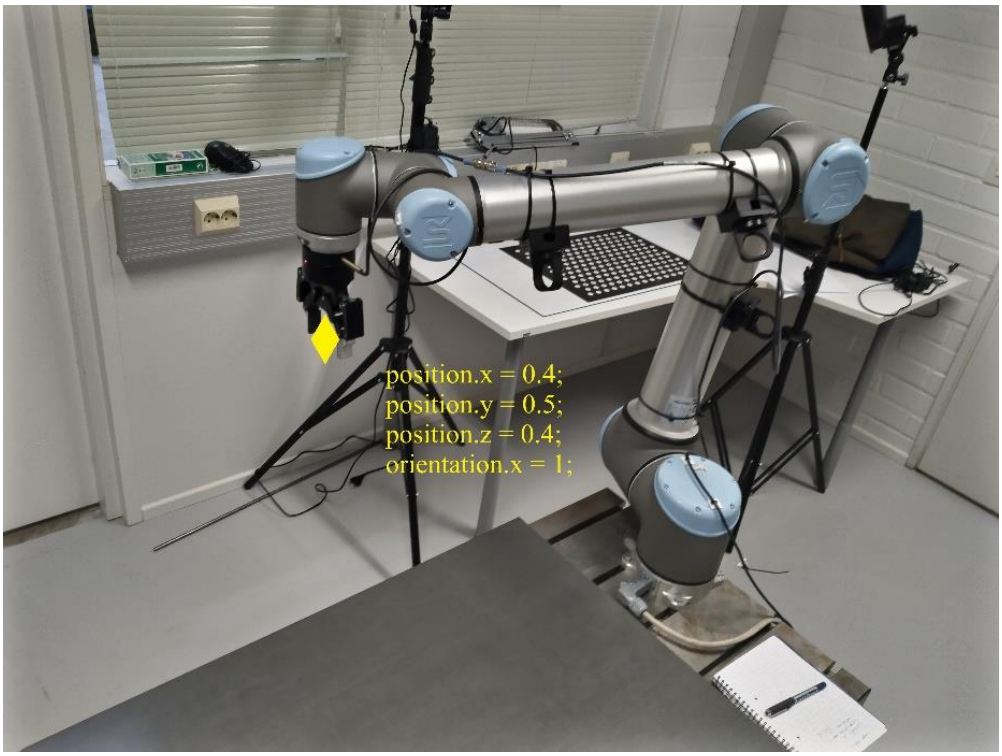

Figure 28. Wait pose

5 RESULTS AND DISCUSSION

In this section, the results of the research are presented including an overview of the final framework and of all the components as well as the results for the efficiency test. During the research, a functional framework for control of the hardware was established. The computation level *rqt* graph of the communication framework in ROS is presented figure 29. The complete pick and place application code is presented in appendix VII.

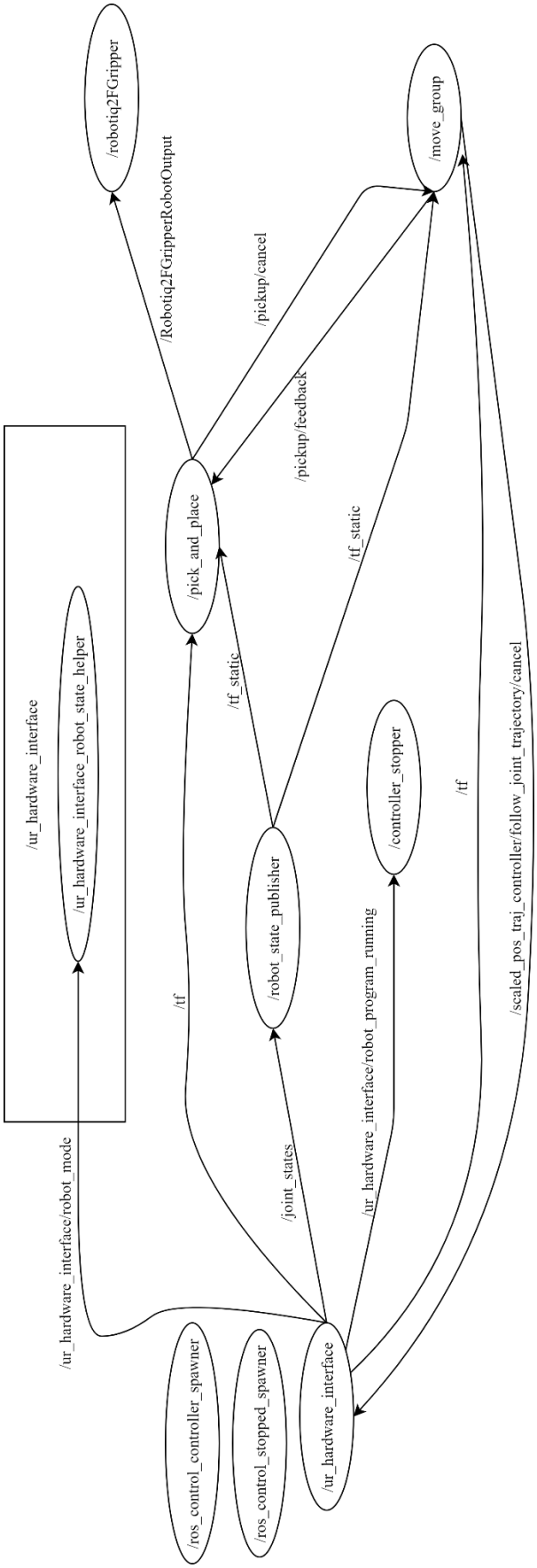

Figure 29. Computation level of the framework in ROS

5.1 UR10 and Robotiq 2F-85 Gripper

As presented in the previous chapters the UR10 was integrated in ROS with the Universal Robots ROS driver. The driver in combination with the external control *URcap* enables full control of the UR10 in ROS. Furthermore, factory calibration data for forwards and inverse kinematics were extracted for precise control of the end effector. A specifically configured *MoveIt package* was built based on a master *Xacro* to include the gripper in collision checking. The robot and gripper configuration and planned paths were visualized in *RViz* in real time. The actions of the gripper were not visualized. Path planning for the UR10 was conducted with the OMPL included in *MoveIt.* The *MoveIt* C++ interface was used to manipulate the robot based on X-Y-Z Cartesian coordinates and a quaternion orientation. The collision environment of the robot was defined through the *MoveIt* planning scene interface.

The Gripper was integrated in ROS with the *Robotiq2FGripperRtuNode*, which acts as a higher-level driver. The driver enables direct control of the gripper by interacting with the */Robotiq2FGripperRobotOutput* topic. The desired actions for the gripper were set using the pick and place node and sent as a message to the */Robotiq2FGripperRobotOutput* topic. . The grippers' position, speed and force can all be configured to suit specific applications. Communication for the status of the gripper was established but not utilized in this project. A rqt graph of the gripper communication network is presented in figure 30.

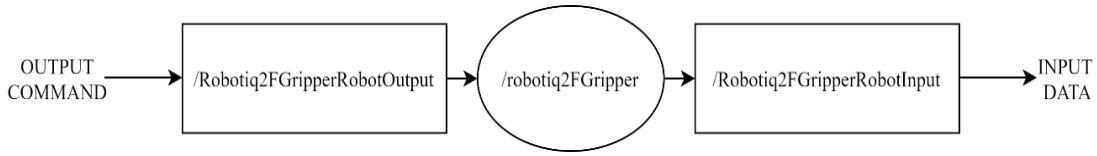

Figure 30. *Rqt* graph of the gripper driver node and related topics in ROS

5.2 Experiment results

The efficiency of the path planning algorithms KPIECE, LBKPIECE1 and RTT connect were tested to see how they affected latency during manipulation for a simulated pick and place task. The results for the planning time with the three planners are presented figure 31. From the testing, it became apparent that the LBKPIECE1 planner was not suited for tasks of this nature. The KPIECE and RTT connect planners were significantly faster for planning simple manipulation tasks. Especially the RTT connect path planning algorithm had an

insignificant amount of delay during planning and was best suited for the specific application. The simplification time of each solution contributes to the total planning time of the task. The simplification times for both solutions with each planner are presented in figure 32. The simplification times followed the same trend with the LBKPIECE1 performing significantly slower than the KPIECE and RTT connect. However, the absolute magnitude of the simplification times for the LBKPIECE were insignificant with the planning times requiring a multitude of seconds.

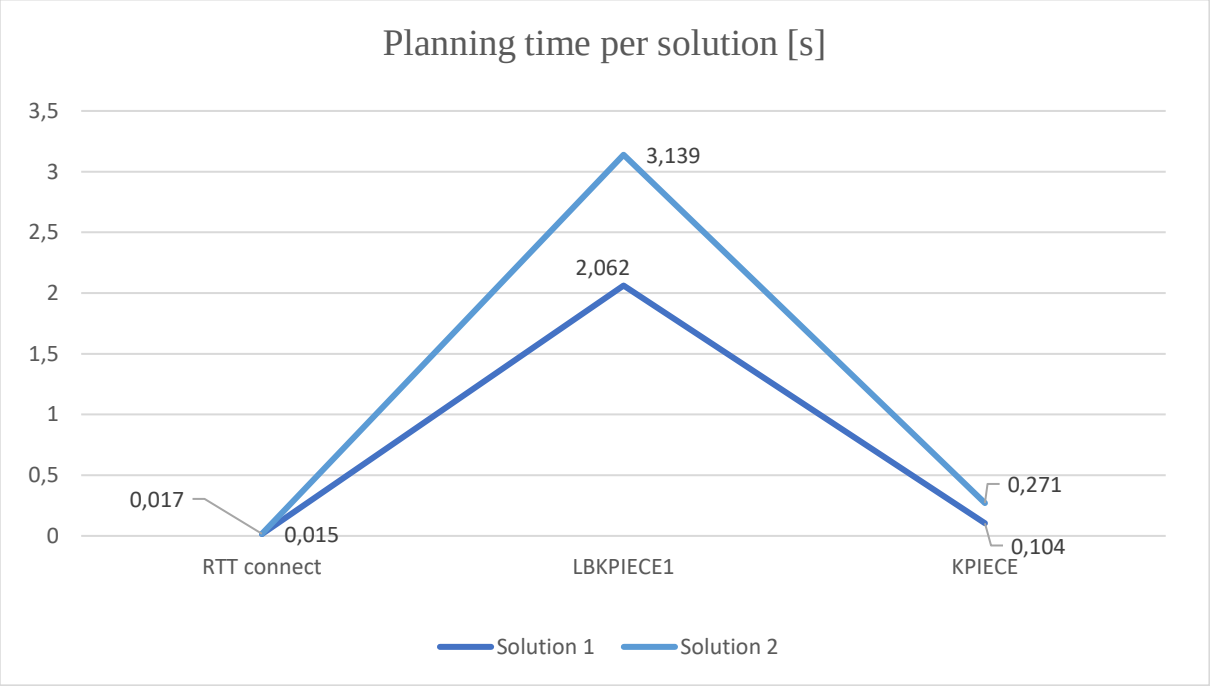

Figure 31. Results for planning time with the designated planners

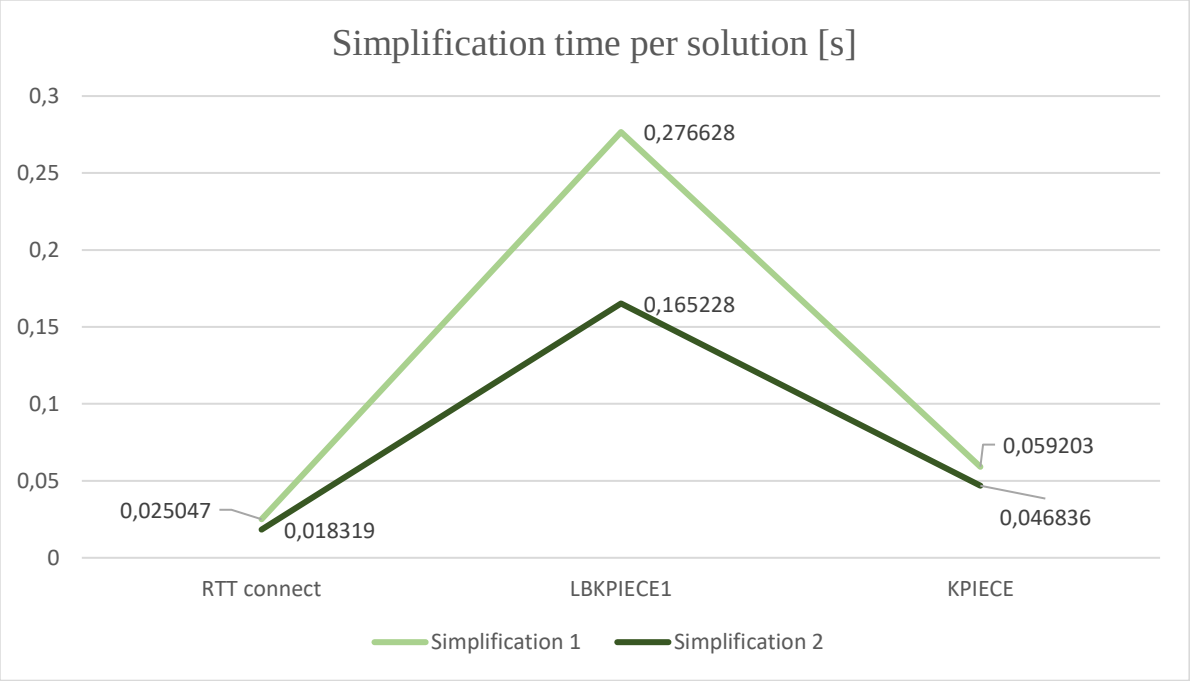

Figure 32. Results for the simplification time with designated planners

5.3 Discussion of the results

In summary the framework works. The UR10 and Robotiq 2F85-gripper can both be controlled in ROS efficiently. The developed framework accomplishes the main objectives of the project that were defined as:

- Research of compatible software for the chosen hardware
- Integration of the UR10 and 2F-85 gripper in ROS
- Development of code so that the UR10 can perform simple tasks such as moving objects

The code that was developed can be configured to suit a variety of applications where simple picking and placing is required. The framework has not been implemented with a vision system and therefore requires a preset position and orientation of an object to be known to function. This constricts the efficiency of the framework drastically and must be corrected in further research. Further sensors can however be relatively easily integrated to the pre-existing framework to further develop the capacity at which the UR10 can operate.

Furthermore, the foundation for two-way communication with the gripper was established. However, the gripper was only utilized with one-way communication where commands were sent to the gripper. The grippers' status was not verified due to time constraints with the research. In practical applications, the framework must be aware of grippers' status in case of a malfunction.

In addition, the functionality test revealed the chosen planning algorithms to not be optimal for the pick and place application. The LBKPIECE1 and KPIECE planners were significantly slower than the RTT Connect planners. This could be a result of an error in the understanding of the planners on the researcher's part or a fault in the frameworks design. In either case, the chosen path planner must be further researched to verify which is most optimal for the specific framework and designated real-world application.

6 CONCLUSION

This project was a part of research for the development of a collaborative human-robot manufacturing working station for the Laboratory of Intelligent Machines at LUT University. The scope of the project was to develop a control framework for a UR10 and Robotiq 2F-85 gripper based on the ROS platform. The main objectives of the research were achieved, and a fully functional framework was developed. The code for a simple pick and place application was also developed and the efficiency of path planning was tested.

The framework is not perfect and requires further research and optimization to be deployed in a real-world application. The research does however provide a stable foundation developed with the latest distributions of respective software to provide a maximal period of compatibility with future hardware.

Future research regarding this project will aim to incorporate a vision system. The implementation of a vision system will enable the framework to cater to a vastly greater variety of applications and work environments. The operator will not be required to enter the position and orientation of an object so that it can be manipulated. Furthermore, the two-way communication of the gripper will be utilized to ensure that the framework can operate in cases where a malfunction occurs.

The framework in this research uses a single node to control both the manipulation of the UR10 and gripper, which does not comply with the standard ideology in ROS where a single node controls a single action. In further research the architecture of the pick and place application should be further developed. A separate node should be built to control the gripper with higher-level commands.

LIST OF REFERENCES

Appleton, E. & Williams, D.J. 1987. Industrial Robot Applications. New York: Halsted press and John Wiley & Sons. 240p.

Bonev, I. 2001. Delta Parallel Robot — the Story of Success [web document]. Updated 6.5.2001. [Referred 16.06.2020]. Available: http://www.parallemic.org/Reviews/Review002.html

CLEARPATH ROBOTICS. 2015. INTRO TO ROS. [web document]. [Referred 17.06.2020]. Available: http://www.clearpathrobotics.com/assets/guides/kinetic/ros/Intro%20to%20the%20Robot%20Operating%20System.html

Cyberneticzoo. 2013. 1965-7 – Trallfa spray-paint robot – Ole Molaug and Sverre Bergene (Norweigan) [web document]. Updated 12.3.2013. [Referred 15.06.2020]. Available: http://cyberneticzoo.com/early-industrial-robots/1965-7-trallfa-spray-paint-robot-ole-molaug-and-sverre-bergene-norweigan/

Ferraresi, C. & Quaglia, G. 2018, Advances in Service and Industrial Robotics. Proceedings of the 26th International Conference on Robotics. Alpe-Adria-Danube Region, RAAD 2017. Technical University Politecnico di Torino, Turin, Italy. 21-23.6.2017. P. 242-252.

Gasparetto, A. & Scalera, L. 2019. A Brief History of Industrial Robotics in the 20th Century. In: Advances in Historical Studies. Volume 8. P 24-35.

GitHub. 2018. Move Group C++ Interface [web document]. Updated 10.7.2018. [Referred 04.06.2020]. Available: https://github.com/ros-planning/moveit_tutorials/blob/kinetic-devel/doc/move_group_interface/move_group_interface_tutorial.rst

GitHub. 2019. ROS Industrial (Melodic) Training Exercises [web document]. Updated 8.10.2019. [Referred 04.06.2020]. Available: https://github.com/rosindustrial/industrial_training/blob/melodic/gh_pages/index.rst

GitHub. 2020a. Universal Robots ROS Driver [web document]. 7.5.2020. [Referred 08.05.2020]. Available: https://github.com/UniversalRobots/Universal_Robots_ROS_Driver

GitHub. 2020b. MoveIt Tutorials [web document]. Updated 27.02.2020. [Referred 04.06.2020]. Available: https://github.com/ros-planning/moveit_tutorials/blob/master/index.rst

HPE. 2020. HPE 408 Switch Series – Overview [web document]. [Referred 06.06.2020] Available: https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-c01846427

IEEE. 2020. Unimate [web document]. [Referred 01.06.2020]. Available: https://robots.ieee.org/robots/unimate/

International Federation of Robotics. 2018. Demystifying Collaborative Robots [web document]. Published 12.2018. [Referred 10.4.2020]. Available: https://ifr.org/downloads/papers/IFR_Demystifying_Collaborative_Robots.pdf

International Federation of Robotics. 2020. Robot history [web document]. [Referred 02.06.2020]. Available: https://ifr.org/robot-history

Irati, Z., Kojcev, R., Hernandez, A., Muguruza, I., Usategui, L., Bilbao, A., Mayoral, V. 2017. Dissecting Robotics - historical overview and future perspectives. 8 p. Available: https://www.researchgate.net/publication/316538824_Dissecting_Robotics_-_historical_overview_and_future_perspectives

Jaber A.A. 2017. PUMA 560 Robot and Its Dynamic Characteristics. In: Design of an Intelligent Embedded System for Condition Monitoring of an Industrial Robot. Springer Theses (Recognizing Outstanding Ph.D. Research). Springer, Cham

O'Neill, J.J. 1944. Prodigal Genius The Life of Nikola Tesla. New York: Ives Washburn. 368 p.

Khaksar W., Sahari K.S.M., Hong T.S. 2016. Application of Sampling-Based Motion Planning Algorithms in Autonomous Vehicle Navigation. Autonomous Vehicle. Andrzej Zak. IntechOpen. September 7th, 2016. Available: https://www.intechopen.com/books/autonomous-vehicle/application-of-sampling-based-motion-planning-algorithms-in-autonomous-vehicle-navigation

Kutcher, D. 2020. Film-like Story of the First Real Robot Unimate in History [web document]. Updated 20.02.2020. [Referred 15.06.2020]. Available: https://www.somagnews.com/film-like-story-first-real-robot-unimate-history/

Merriam-Webster.com. 2020. Robot [web document]. [Referred 01.06.2020]. Available: https://www.merriam-webster.com/dictionary/robot

Mordechai B. 2018. A Tutorial on Euler Angles and Quaternions [web document]. Department of Science Teaching Weizmann Institute of Science. Updated 19.07.2018. [Referred 06.06.2020]. Available: https://www.weizmann.ac.il/sci-tea/benari/sites/sci-tea.benari/files/uploads/softwareAndLearningMaterials/quaternion-tutorial-2-0-1.pdf

MoveIt. 2020. Concepts. [web document]. [Referred 03.06.2020]. Available: https://moveit.ros.org/documentation/concepts/

OMPL. 2019. Available Planners [web document]. [Referred 03.06.2020]. Available: https://ompl.kavrakilab.org/planners.html#geometric_planners

Parent M., Laurgeau C. 1984. Programming languages. In: Logic and Programming. Robot Technology, vol 5. Springer, Boston, MA

Pat. US 2286571A. 1938. Position-controlling apparatus. Willard L V Pollard. Appl. US203634A, 1938-04-22. Publ. 1942-06-16. 4 p.

Pat. US 2988237A. 1954. Programmed article transfer. Jr George C Devol. Appl. US 474574A, 1954-12-10. Publ. 1961-06-13. 13 p.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote T., Leibs J., Berger, E., Wheeler. R., Ng A. 2009. ROS: an open-source Robot Operating System. ICRA Workshop on Open Source Software. 3.

ROS. 2010. Moving the gripper [web document]. Updated 24.8.2010. [Referred 03.06.2020]. Available: http://wiki.ros.org/pr2_controllers/Tutorials/Moving%20the%20gripper

Robotics Industries Association. 2020. Unimate // The First Industrial Robot [web document]. [Referred 01.06.2020]. Available: https://www.robotics.org/joseph-engelberger/unimate.cfm

Robotiq. 2020a. 2F85-140-Adaptive-Robot-Gripper [web document]. [Referred 08.05.2020]. Available: https://robotiq.com/products/2f85-140-adaptive-robot-gripper

Robotiq. 2020b. Robotiq 2F-85 & 2F-140 for Universal Robots [web document]. [Referred 03.06.2020]. Pp. 41-47.Available https://assets.robotiq.com/website-assets/support_documents/document/2F-85_2F-140_UR_PDF_20200211.pdf?_ga=2.62132652.1039012859.1591191670-564883962.1574854987

RobotWorx. 2020. Universal Robots UR5 [web document]. [Referred 16.06.2020]. Available: https://www.robots.com/robots/universal-robot-ur5

ROS. 2012. XML Robot Description Format (URDF) [web document]. Updated 19.06.2012. [Referred 26.05.2020]. Available: http://wiki.ros.org/urdf/XML/model

ROS. 2014. Concepts [web document]. Updated 21.6.2014. [Referred 06.05.2020]. Available: http://wiki.ros.org/ROS/Concepts

ROS. 2018a. Introduction [web document]. Updated 18.12.2018. [Referred 06.05.2020]. Available: http://wiki.ros.org/ROS/Introduction

ROS. 2018b. Control of a 2-Finger Gripper using the Modbus RTU protocol (ros kinetic and newer releases [web document]. Updated 4.11.2018. [Referred 08.05.2020]. Available: http://wiki.ros.org/robotiq/Tutorials/Control%20of%20a%202-Finger%20Gripper%20using%20the%20Modbus%20RTU%20protocol%20%28ros%20kinetic%20and%20newer%20releases%29

ROS. 2020a. History [web document]. [Referred 02.06.2020]. Available: https://www.ros.org/history/

ROS. 2020b. Xacro [web document]. Updated 17.2.2020. [Referred 28.05.2020]. Available: http://wiki.ros.org/xacro

Rotenberg S. 2016. Orientation & Quaternions [web document]. University of California San Diego, CSE169: Computer Animation. [Referred 06.06.2020]. Available: https://cseweb.ucsd.edu/classes/sp16/cse169-a/slides/CSE169_03.pdf

Rutherford, J. 2012. Using the PUMA 560 robot.

Salmon M., d'Auria A. 1979. Programmable Assembly System. In: Dodd G.G., Rossol L. (eds) Computer Vision and Sensor-Based Robots. Springer, Boston, MA

Shepherd S., Buchstab A. 2014. KUKA Robots On-Site. In: McGee W., Ponce de Leon M. (eds) Robotic Fabrication in Architecture, Art and Design 2014. Springer, Cham

SRI International. 2020. Shakey [web document].. [Referred 02.06.2020] Available: http://www.ai.sri.com/shakey/

Șucan I.A., Moll M., Kavraki L.E. 2012. The Open Motion Planning Library. IEEE Robotics & Automation Magazine. 19(4):72–82. December 2012

Universal Robots. 2020a. Our history. [Universal Robots webpage]. [Referred 03.06.2020]. Available: https://www.universal-robots.com/about-universal-robots/our-history/

Trelleborg. 2020. Application Examples for Cartesian Robots [web document]. [Referred 15.06.2020]. Available: https://www.tss.trelleborg.com/en/upcoming/robotics/application-examples/cartesian-robot

Universal Robots. 2020b. Universal Robots UR10/CB3 Original instructions (en) US Version [web document]. Available: https://s3-eu-west-1.amazonaws.com/ur-support-site/69445/99237_UR10_User_Manual_en_US.pdf

Weisstein E.W. 2020. Euler Angles. [web document]. Updated 27.05.2020. [Referred 04.06.2020]. Available: https://mathworld.wolfram.com/EulerAngles.html

Zhang D., Wei B., Rosen M. 2017. Overview of an Engineering Teaching Module on Robotics Safety. In: Zhang D., Wei B. (eds) Mechatronics and Robotics Engineering for Advanced and Intelligent Manufacturing. Lecture Notes in Mechanical Engineering. Springer, Cham

UR10

Performance

Repeatability	±0.1 mm / ±0.0039 in (4 mils)
Ambient temperature range	0-50°
Power consumption	Min 90W, Typical 250W, Max 500W
Collaboration operation	15 advanced adjustable safety functions. TüV NORD Approved Safety Function Tested in accordance with: EN ISO 13849:2008 PL d

Specification

Payload	10 kg /22 lbs
Reach	1300 mm / 51.2 in
Degrees of freedom	6 rotating joints
Programming	Polyscope graphical user interface on 12 inch touchscreen with mounting

Movement

Axis movement robot arm	Working range	Maximum speed
Base	± 360°	± 120°/Sec.
Shoulder	± 360°	± 120°/Sec.
Elbow	± 360°	± 180°/Sec.
Wrist 1	± 360°	± 180°/Sec.
Wrist 2	± 360°	± 180°/Sec.
Wrist 3	± 360°	± 180°/Sec.
Typical tool		1 m/Sec. / 39.4 in/Sec.

Features

IP classification	IP54	
ISO Class Cleanroom	5	
Noise	72dB	
Robot mounting	Any	
I/O ports	Digital in	2
	Digital out	2
	Analog in	2
	Analog out	0
I/O power supply in tool	12 V/24 V 600 mA in tool	

Physical

Footprint	Ø 190mm
Materials	Aluminium, PP plastics
Tool connector type	M8
Cable length robot arm	6 m / 236 in
Weight with cable	28,9 kg / 63.7 lbs

CONTROL BOX

Features

IP classification	IP20	
ISO Class Cleanroom	6	
Noise	<65dB(A)	
I/O ports	Digital in	16
	Digital out	16
	Analog in	2
	Analog out	2
I/O power supply	24V 2A	
Communication	TCP/IP 100Mbit, Modbus TCP, Profinet, EthernetIP	
Power source	100-240 VAC, 50-60 Hz	
Ambient temperature range	0-50°	

Physical

Control box size (WxHxD)	475mm x 423mm x 268mm / 18.7 x 16.7 x 10.6 in
Weight	17 kg / 37.5 lbs
Materials	Steel

TEACH PENDANT

Features

IP classification	IP20

Physical

Materials	Aluminium, PP
Weight	1,5 kg / 3.3 lbs
Cable length	4,5 m / 177 in

APPENDIX II

SPECIFICATIONS	2F-85	2F-140
Stroke (adjustable)	85 mm	140 mm
Grip force (adjustable)	20 to 235 N	10 to 125 N
Form-fit grip payload	5 kg	2.5 kg
Friction grip payload	5 kg	2.5 kg
Gripper mass	0.9 kg	1 kg
Position resolution (fingertip)	0.4 mm	0.6 mm
Closing speed (adjustable)	20 to 150 mm/s	30 to 250 mm/s
Communication protocol	Modbus RTU (RS-485)	
Ingress protection (IP) rating	IP40	IP40
* All specifications provided for reference only. See user manual at support.robotiq.com for official specifications.		

APPENDIX III

```
kinematics:
  shoulder:
    x: 0
    y: 0
    z: 0.1273
    roll: -0
    pitch: 0
    yaw: -0
  upper_arm:
    x: 0
    y: 0
    z: 0
    roll: 1.570796327
    pitch: 0
    yaw: -0
  forearm:
    x: -0.612
    y: 0
    z: 0
    roll: -0
    pitch: 0
    yaw: -0
  wrist_1:
    x: -0.5723
    y: 0
    z: 0.163941
    roll: -0
    pitch: 0
    yaw: -0
  wrist_2:
    x: 0
    y: -0.1157
    z: -2.373046667922381e-11
    roll: 1.570796327
    pitch: 0
    yaw: -0
  wrist_3:
    x: 0
    y: 0.0922
    z: -1.891053610911353e-11
    roll: 1.570796326589793
    pitch: 3.141592653589793
    yaw: 3.141592653589793
  hash: calib_17227329492635474227
```

```
<?xml version="1.0" ?>

<robot name="myworkcell"
xmlns:xacro="http://ros.org/wiki/xacro">

<xacro:include filename="$(find
ur_description)/urdf/ur10_robot.urdf.xacro" />

<xacro:include filename="$(find
robotiq_2f_85_gripper_visualization)/urdf/robotiq_arg2f_85_mo
del.xacro" />

<link name="world"/>
<link name="arm_tcp_link"/>

<joint name="world_to_robot" type="fixed">
    <parent link="world" />
    <child link = "base_link" />
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 3.1416" />
  </joint>

<joint name="gripper_to_robot" type="fixed">
  <parent link="tool0"/>
  <child link="robotiq_arg2f_base_link"/>
  <origin xyz="0 0 0" rpy="0 0 1.5708"/>
</joint>

<joint name="arm_joint_tcp" type="fixed">
    <parent link="tool0" />
    <child link = "arm_tcp_link" />
    <origin xyz="0.0 0.0 0.171" rpy="0.0 0.0 0.0" />
  </joint>

</robot>
```

APPENDIX V

controllers.yaml:

```
controller_list:
  - name: /scaled_pos_traj_controller
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    joints:
      - shoulder_pan_joint
      - shoulder_lift_joint
      - elbow_joint
      - wrist_1_joint
      - wrist_2_joint
      - wrist_3_joint
```

joint_names.yaml:

```
controller_joint_names: [shoulder_pan_joint,
shoulder_lift_joint, elbow_joint, wrist_1_joint,
wrist_2_joint, wrist_3_joint]
```

myworkcell_moveit_controller_manager.launch:

```
<launch>

  <!-- loads moveit_controller_manager on the parameter server
which is taken as argument
    if no argument is passed, moveit_simple_controller_manager
will be set -->
  <arg                        name="moveit_controller_manager"
default="moveit_simple_controller_manager/MoveItSimpleControl
lerManager" />
  <param    name="moveit_controller_manager"    value="$(arg
moveit_controller_manager)"/>

  <!-- loads ros_controllers to the param server -->
  <rosparam                                       file="$(find
ur10_2f85_moveit_config)/config/controllers.yaml"/>
</launch>
```

APPENDIX VI

planning_execution.launch:

```
<launch>
<!-- The planning and execution components of MoveIt! configured to run -->
<!-- using the ROS-Industrial interface. -->

<rosparam command="load" file="$(find ur10_2f85_moveit_config)/config/joint_names.yaml"/>

<!-- load the robot_description parameter before launching ROS-I nodes -->
<include file="$(find ur10_2f85_moveit_config)/launch/planning_context.launch" >
<arg name="load_robot_description" value="true" />
</include>

<!-- publish the robot state (tf transforms) -->
<node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />


<include file="$(find ur10_2f85_moveit_config)/launch/move_group.launch">
<arg name="publish_monitored_planning_scene" value="true" />
</include>

<include file="$(find ur10_2f85_moveit_config)/launch/moveit_rviz.launch">
<!-- <arg name="config" value="true"/> -->
</include>

</launch>
```

pick_and_place.cpp:

```
#include <moveit/move_group_interface/move_group_interface.h>
#include
<moveit/planning_scene_interface/planning_scene_interface.h>
#include <moveit_msgs/DisplayRobotState.h>
#include <moveit_msgs/DisplayTrajectory.h>
#include <moveit_msgs/AttachedCollisionObject.h>
#include <moveit_msgs/CollisionObject.h>
#include
<robotiq_2f_gripper_control/Robotiq2FGripper_robot_output.h>



int main(int argc, char** argv)
{
  ros::init(argc, argv, "move_group_interface_tutorial");
  ros::NodeHandle node_handle;
  ros::AsyncSpinner spinner(1);
  spinner.start();
  ros::Publisher
pub=node_handle.advertise<robotiq_2f_gripper_control::Robotiq
2FGripper_robot_output>("Robotiq2FGripperRobotOutput",100);
  ros::Rate loop_rate(1);
  //PLanning Group
  static const std::string PLANNING_GROUP = "manipulator";

  //:move_group_interface:`MoveGroupInterface` class

  moveit::planning_interface::MoveGroupInterface
move_group(PLANNING_GROUP);

  //Planning scene interface class for collision objects
  moveit::planning_interface::PlanningSceneInterface
planning_scene_interface;

  // Raw pointers for improved performance.
  const robot_state::JointModelGroup* joint_model_group =
      move_group.getCurrentState()-
>getJointModelGroup(PLANNING_GROUP);


  // Gripper

  // Activation
  robotiq_2f_gripper_control::Robotiq2FGripper_robot_output
commands;
```

```
  commands.rACT = 1;
  commands.rGTO = 1;
  commands.rATR = 0;
  commands.rPR = 0;
  commands.rSP = 255;
  commands.rFR = 150;
  pub.publish(commands);
  loop_rate.sleep();

  //table collision object
  moveit_msgs::CollisionObject collision_object;

  collision_object.header.frame_id =
move_group.getPlanningFrame();
  collision_object.id = "box";

  shape_msgs::SolidPrimitive primitive;
  primitive.type = primitive.BOX;
  primitive.dimensions.resize(3);
  primitive.dimensions[0] = 0.6;
  primitive.dimensions[1] = 1;
  primitive.dimensions[2] = 0.025;

  geometry_msgs::Pose box_pose;
  box_pose.orientation.w = 1.0;
  box_pose.position.x = 0;
  box_pose.position.y = 0.6;
  box_pose.position.z = 0;

  collision_object.primitives.push_back(primitive);
  collision_object.primitive_poses.push_back(box_pose);
  collision_object.operation = collision_object.ADD;

  std::vector<moveit_msgs::CollisionObject>
collision_objects;
  collision_objects.push_back(collision_object);

planning_scene_interface.addCollisionObjects(collision_object
s);

  // planning time setting
  move_group.setPlanningTime (10.0);


  // Planning to approach pose
  // ^^^^^^^^^^^^^^^^^^^^^^^^
```

```
  geometry_msgs::Pose target_pose1;
  target_pose1.position.x = 0.2;
  target_pose1.position.y = 0.98;
  target_pose1.position.z = 0.3;
  target_pose1.orientation.x = 1;
  move_group.setPoseTarget(target_pose1);

  moveit::planning_interface::MoveGroupInterface::Plan
my_plan;

  bool success = (move_group.plan(my_plan) ==
moveit::planning_interface::MoveItErrorCode::SUCCESS);

  move_group.move();

  // Planning to grasp pose
  // ^^^^^^^^^^^^^^^^^^^^^^

  geometry_msgs::Pose target_pose2;
  target_pose2.position.x = 0.2;
  target_pose2.position.y = 0.98;
  target_pose2.position.z = 0.05;
  target_pose2.orientation.x = 1;
  move_group.setPoseTarget(target_pose2);

  success = (move_group.plan(my_plan) ==
moveit::planning_interface::MoveItErrorCode::SUCCESS);

  move_group.move();

  //gripper close
  commands.rACT = 1;
  commands.rGTO = 1;
  commands.rATR = 0;
  commands.rPR = 255;
  commands.rSP = 255;
  commands.rFR = 150;
  pub.publish(commands);
  loop_rate.sleep();

  // Plan and execute to retreat
  move_group.setPoseTarget(target_pose1);
  success = (move_group.plan(my_plan) ==
moveit::planning_interface::MoveItErrorCode::SUCCESS);

  move_group.move();
```

```
  // Planning to approach pose
  // ^^^^^^^^^^^^^^^^^^^^^^^^

  geometry_msgs::Pose target_pose3;
  target_pose3.position.x = -0.2;
  target_pose3.position.y = 0.98;
  target_pose3.position.z = 0.3;
  target_pose3.orientation.x = 1;
  move_group.setPoseTarget(target_pose3);

  success = (move_group.plan(my_plan) ==
moveit::planning_interface::MoveItErrorCode::SUCCESS);

  move_group.move();

  // Planning to payload release pose
  // ^^^^^^^^^^^^^^^^^^^^^^^^

  geometry_msgs::Pose target_pose4;
  target_pose4.position.x = -0.2;
  target_pose4.position.y = 0.98;
  target_pose4.position.z = 0.05;
  target_pose4.orientation.x = 1;
  move_group.setPoseTarget(target_pose4);

  success = (move_group.plan(my_plan) ==
moveit::planning_interface::MoveItErrorCode::SUCCESS);

  move_group.move();
  commands.rACT = 1;
  commands.rGTO = 1;
  commands.rATR = 0;
  commands.rPR = 0;
  commands.rSP = 255;
  commands.rFR = 150;
  pub.publish(commands);
  loop_rate.sleep();
// End of sequence
  ros::shutdown();
  return 0;
}
```