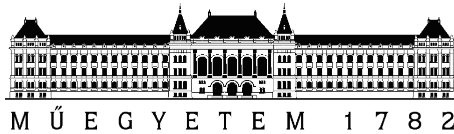


Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Towards Application of Modular Robotics In Future Factory: A Simulation Study

STUDENTS' SCIENTIFIC CONFERENCE PAPERS

Authors	*Department Supervisor*	*Industrial Supervisor*
Levente Vajda	Dr. Attila Vidács	Dr. Géza Szabó
József Pető	Associate Professor	Ericsson Ltd.

October 27, 2018

Contents

Összefoglaló 3

Abstract 4

1 Introduction 5

2 Related work 8

3 Background 11

3.1 Robot Operating System . . . 11
3.1.1 ROS Concepts . . . 12
3.1.2 ROS Filesystem Structure . . . 12
3.1.3 Other important ROS concepts . . . 14
3.1.4 ROS Computation Graph . . . 15
3.1.5 ROS Community . . . 19
3.1.6 ROS Names . . . 20
3.1.7 Package Resource Names . . . 20
3.2 Gazebo . . . 20
3.3 UR10 Robot arm . . . 21
3.4 Used ROS packages . . . 21
3.4.1 universal_robot package . . . 21
3.4.2 ros_control package . . . 22
3.4.3 gazebo_ros_pkgs package . . . 23
3.4.4 xacro package . . . 24
3.4.5 roslaunch package . . . 24
3.4.6 ariac package . . . 24
3.4.7 hebiros package . . . 24

4 Implementation 25

4.1 Implementation of a modular robotic architecture . . . 25
4.1.1 Actuator modules . . . 25

4.1.2 Connection mechanism . . . 26
4.1.3 Energy . . . 26
4.1.4 Sensors . . . 26
4.1.5 Communications . . . 26
4.1.6 Control . . . 26
4.2 Controller for our modular approach . . . 27
4.3 Hints to identify our contribution . . . 28
4.4 Attaching the parts with vacuum grippers . . . 30
4.5 Issues with the servos . . . 30
4.6 Real time factor optimization . . . 31

5 Results and evaluation **33**
5.1 Building a modular robotic arm . . . 33
5.2 Demonstrating the capabilities of the robot arm . . . 36
5.2.1 Lift the box . . . 36
5.2.2 Step . . . 37
5.2.3 Alternative use case of the robot parts . . . 38

6 Conclusion and further work **40**

Bibliography **42**

Kivonat

Moduláris robotok alatt olyan robotokat értünk, amelyek képesek újrakonfigurálni felépítésüket – önállóan vagy külső segítséggel – az elvégzendő feladathoz igazodva. A moduláris robotika története az 1972-es évig nyúlik vissza, de ipari környezetben való alkalmazásuk nem jellemző. Ez nem is meglepő, hiszen a legtöbb ilyen robot nem arra lett tervezve, hogy az iparban használt robotok feladatait ellássák. Sok moduláris robotot arra terveznek, hogy extrém körülmények között – például a világűrben, a mélytengereken vagy nukleáris erőművekben – használják. Felhasználva azt, hogy a robot képes a meghibásodott alkatrészét kicserélni, vagy átkonfigurálni magát.

A kiinduló kérdés az volt, hogy hogyan tudna egy robot, amely az elvégzendő feladathoz igazodva épül fel, helytállni egy ipari környezetben. Egy robotszimulációs szoftverrel gyorsan lehet tesztelni olyan ötleteket, amelyeket a valóságban eszközök megvétele nélkül nem lehet megvalósítani.

A munkám célja egy olyan robot bemutatása, amely képes az adott feladathoz igazodva manuálisan átkonfigurálni magát. Ehhez a Gazebo szimulációs szoftvert használtam, amellyel hatékonyan lehet szimulálni komplex környezeteket. Valamint Robot Operating Systemet (ROS) használtam, amely egy robot alkalmazásfejlesztő platform.

Bemutatom egy ilyen moduláris robotnak a megépítését, amelyhez egy UR10 robotkart használok fel. Példákat mutatok a robotkar lehetséges használatára, valamint a robotkar építőelemeinek egy másik lehetséges felhasználására.

Abstract

The history of modular robotics dates back to 1972, but their application in industrial environment is not common. This is not surprising since most of these robots are not designed to handle the roles of the industrial robots. Many modular robots are designed to be used in extreme conditions such as space, deep sea or nuclear power plants utilizing their ability that the robot can replace its defective component or reconfigure itself.

Our goal is to investigate if a modular robot could be applied in an industrial environment. We present a robot that is capable of self-reconfiguration in a task specific way. Our analysis is performed on a robot simulation software, Gazebo interacting with ROS. Gazebo is an open source simulation software that can be used to simulate complex environments. Robot Operating System (ROS) is a robotic application development platform.

In this paper, we discuss the assembly process of a modular robotic arm. The parts are gathered with an UR10 robot arm and the remote-controlled connectors are used to fix them together. Finally, we present some alternative assembly setups of the modular robotic kits.

Chapter 1

Introduction

In the past few years, there has been an increasing demand from customers towards the manufacturing industry to provide more and more customized products [1]. Personalized production is one of the key motivations for manufacturers to start leveraging new technologies that enable to increase, for instance, the flexibility of production lines. High flexibility in general is needed to realize cost effective and customized production by supporting fast reconfiguration of production lines, as well as easy application development.

Applying standard, expensive industrial robots in a highly varying production line is not practical in the long run as it will only be used for a particular task and will be doing nothing the rest of the time. A modular self-reconfigurable robot cell optimized for task specific deployment is the desired state in the future [2]. Manifold requirements are needed to be fulfilled technologically:

- low energy consuming actuators;
- some embedded intelligence in the local controller;
- capable of any kind of IoT communication.

If we aim for a completely self-contained robot module, it should be also self-propelled by internal battery and remotely controlled via wireless access. The advances in battery technologies can provide the first, while the upcoming 5G supports the latter. Application of such a wireless technology in manufacturing enables, for instance, to reduce cabling in a factory. Cableless communication is a real enabler of many applications that is difficult to achieve with production systems depending on wired connections e.g., jet engine manufacturing during the milling of the blades [3].

One can argue that it is difficult to fulfil the same requirements by modular robotics that is provided by industrial grade arms. The industrial robot has many metrics and measurable characteristics, which will have a direct impact on the robustness of the robot during the execution of its tasks. The main measurable characteristics are repeatability

and accuracy. In a nutshell, the repeatability of a robot might be defined as its ability to achieve repetition of the same task. While, accuracy is the difference (i.e., the error) between the requested task and the realized task (i.e., the task actually achieved by the robot). Practically, repeatability is doing the same task over and over again, while accuracy is hitting your target each time. For more details about the calculation of accuracy and repeatability, see [4]. The objective is to have a robot that can repeat its actions while hitting the target every time. When the current mass production assembly lines are designed, robots are deployed to repeat a limited set of tasks as accurately and the fastest possible to maximize the productivity and minimize the number of faulty parts. The reprogramming of the robots rarely occurs e.g., per week, per month basis and it takes a long time e.g., days and it is a difficult task requiring lot of expertise.

We argue that there can be tasks identified in a robotic cell that needs relaxed requirements and by speeding up the whole process it is beneficial for the overall Manufacturing Cycle Time [5] of the robotic cell. In this work we go for the identification of these tasks. To achieve this we test a some use cases in a simulator.

Recent advances in simulator technology go beyond process level simulation e.g., [6] and with the application of rigid body simulation, a detailed, close-to-real world implementation study can be performed. We chose Gazebo as our target robot simulation environment. Gazebo [7] offers the ability to efficiently and accurately simulate a great number of robots in complex indoor and outdoor environments. It has a robust physics engine, convenient programmatic and graphical interfaces and high-quality graphics. Gazebo is free and widely used among robotic experts. The physics engine is used to model the behaviors of objects in space. These engines allow the simulation of different types of bodies to be affected by various physical stimuli. There are two types of physics engines: real-time and the high precision. Most real-time engines are inaccurate and only provide reduced approximation of the real world, while most high-precision engines are too slow for everyday applications. Physics engines are based on the laws of classical mechanics. The used models determine how accurate these simulations are in dynamical simulations. Gazebo uses the later, sacrificing performance over accuracy, which can be fine-tuned by several parameters.

A rigid body simulator would be difficult to apply for the evaluation of complex robotic cell task, but due to a recent robotic competition, Gazebo starts to gain new features to support this. A recent competition Agile Robotics for Industrial Automation Competition (ARIAC)[8] targets industrial related applications. ARIAC is a simulation-based competition designed to promote agility in industrial robotics using the latest developments in artificial intelligence and robot planning. The general goal of the first and second editions (2017, 2018) of the ARIAC competition were to motivate further development and adoption of agile industrial robotics by providing an environment where teams could work

on solutions towards more productive and autonomous robots that would also require less time from shop floor workers. The competition involved a simulation of the infrastructure where teams would have to complete a set of tasks. The simulation infrastructure was built on top of Gazebo [7] and ROS [9]. The tasks were made to comprehend four specific areas: failure identification and recovery, automated planning, fixtureless environment, and plug and play robots. The tasks or challenges were explored with different simulation trials, which represent the configuration of the simulated environment as well as its goals. ARIAC tasks revolve around collecting a set of part pieces and placing them on a tray to be sent for assembling.

We started our journey to evaluate the feasibility of a pick and place use case performed by modular robots in a robotic cell. In this work we discuss the very first steps, preparing modular functional robot blocks that can be interconnected, assembled and perform any non-trivial action integrated in a robotic cell. This requires the implementation of new features into the existing framework and a lot of difficult integration steps. The paper introduces these steps and discusses them.

In the long run, we would like to perform quantitative measurements to check if any improvement of the ARIAC KPIs is feasible with the modular robotic arms.

Chapter 2

Related work

Authors of [10] collected and discussed many papers in their survey. The design trends of modular reconfigurable robots in terms of docking, powering, reconfiguration, communications, locomotion, DoF, size, and control have been analyzed, reaching some common conclusions that are summarized in the following paragraphs.

Regarding reconfiguration, it has been shown that although the trends have usually been to build self-reconfigurable robots, in the last years many manually configurable robots have also appeared. Although self-reconfigurable robots seem to have general purposes, manually configurable robots are usually task specific. Communication has two clear trends. The first one is to use intramodule communication protocols because the module complexity is increasing and there is a growing need to interconnect several devices (mainly micro-controllers and dedicated controllers) inside the module. The second one is to use wireless protocols (WiFi, Bluetooth, and even ZigBee) for communication between modules and external controllers. The use of wireless controllers simplifies the docking design and allows free movements (no cable dependency). The drawback is that it is not possible to power the robot through a cable. Regarding the size feature, it highly depends on the design of the power-source and the actuators used, overlooking the tasks for which the robot is being designed. While external power source can help in reducing the size of modules, the use of LiPo batteries can also derive in autonomous modules of tenths of centimetres large. Brushless motors are being intensively used to provide modules with actuated joints, without compromising the weight and size but with the drawback of the power consumption. The designs presented in the last 15 years tend to be cubic or spherical. However, this feature is utterly established based on the tasks meant for the robotic system which may not follow the previous statement. There seems to be no clear trends in terms of docking, powering and DoF use. Docking presents no clear trend in the design, and nowadays there are several technologies used: permanent magnets, magnetic, electromagnetic, mechanical, electromechanical, and so on. It is the same case in powering, where most robots seem to use a tether to power the system, but it is not usually pointed out in the published papers.

Modular robots that use batteries use different types: lead acid, NiCd, Lithium, and so on. About the use of DoF, trend in chain-type modules remains stable, using one or at most two DoF. But recent lattice-type modules show higher flexibility with higher degree of connectivity and autonomy. In terms of locomotion, micro-locomotion for module autonomy a trend can be seen in recent chain-type and hybrid-type modular robots. Wheel-based locomotion for both micro- and macro-locomotion is also a new trend in chain- and hybrid-type modular robots. Locomotion through self-reconfiguration in lattice-type systems has remained unchanged, although some recent hybrid-type systems have demonstrated locomotion capability of individual modules on 2D and 3D surfaces, embedded with docking units. In the early days of modular robotics, controllers tend to be more central and less scalable. In the recent years, the focus has been to develop highly distributed and scalable controllers. Controllers have always been highly dependent on intermodule communication for synchronization and coordination among modules. A recent trend in controllers also reflects on biologically inspired control models.

Authors of [11] reviewed the history and state of the art of Reconfigurable Modular Robots (RMR). After analyzing various designs, they found that the concepts of modularity and reconfigurability have been penetrating the design of all the RMRs. Te article made a classification of the existing RMRs: modular mobile robots and modular restructured robots. The two major categories can be further divided into several sub-categories: joint-motion robots and joint-reconfiguration robots; macro-sized reconfigurable manipulators and mini-sized reconfigurable or self-reconfigurable robots (chain, lattice, and hybrid types). Two comparative analyses was demonstrated to clarify the typical characteristics of RMRs. These characteristics contain module shape, module DOF, module attribute, connection mechanism, interface autonomy, locomotion mode, and workspace. Advantages and weaknesses of different sub-categories have been discussed. Furthermore, an evolutionary cobweb evaluation model has been proposed to assess the autonomy level of selected robots. The key technologies, the design methodology of modules and robot configurations, the challenge, and the self-reconfiguration algorithms were also summarized in this survey.

Authors of [12] discuss the application of modular robots in industry to reach hazardous environments. In large facilities there is a with wide range of tasks and modular robots are a flexible robot solution. Some of the tasks to be performed can vary from achieving locomotion with different modular robot (M-Robot) configurations or the execution of cooperative tasks such as moving objects or manipulating objects with multiple modular robot configurations (M-Robot colony) and existing robot deployments. The coordination mechanisms enable the M-Robots to perform cooperative tasks as efficiently as specialised or standard robots. The approach is based on the combination of two communication types i.e., Inter Robot and Intra Robot communications. Through this communication architecture, tight and loose cooperation strategies are implemented to synchronise modules

within an M-Robot configuration and to coordinate M-Robots belonging to the colony. These cooperation strategies are based on a closed-loop discrete time method, a remote clock reading method and a negotiation protocol. The coordination mechanisms and cooperation strategies are implemented into a real modular robotic system, SMART. The need for using such a mechanism in hazardous section of large scientific facilities is presented along with constraints and tasks. Locomotion execution of the mobile M-Robots colony in a bar-pushing task is used as an example for cooperative task execution of the coordination mechanisms and results are presented.

Authors of [13] consider the issue of increasing the number of robots working in sectors characterized by dynamic and unstructured environments. Specifically, the paper deals with a modular robotics based approach to allow the fast deployment of robots to solve specific tasks. They claim that some other authors have proposed modular architectures, mostly in laboratory settings, but their design was usually based on what could be built instead of what was necessary for industrial operations. In this paper they consider the problem by defining the industrial settings the architecture is aimed at and extract the main features that would be required from a modular robotic architecture to operate successfully in these kinds of environments. These requirements are then taken into account to design a particular heterogeneous modular robotic architecture and a laboratory implementation of it. A prototype is also built in order to test its capabilities and show its versatility using a set of different configurations including manipulators, climbers and walkers.

Authors of [14] propose a modular robot platform which relied on a designed module library based on the screw theory and module theory. Then, the configuration design method of the modular robot was proposed and the different configurations of modular robot system were built, including industrial mechanical arms, the mobile platform, six-legged robot and 3D exoskeleton manipulator. Finally, the simulation and verification of one system among them have been made, using the analyses of screw kinematics and polynomial planning. The overlook of a high level Matlab simulation is discussed shallowly.

In all the above papers the common point is that modular robotics is considered only if no industrial grade solution is available on the market e.g., surveying robots in catastrophe areas. They are not recognized as real options if there is an existing solution but a one-fit-for all type robot which can perform any kind of task in a tolerable success rate. In this paper we argue the about the opposite: there are tasks in which modular robots can outperform even the industrial grade ones or at least greatly improve the robotic cell total performance.

Chapter 3

Background

This chapter describes the basics of Robot Operating System, the packages used in our work, the UR10 robot arm, and the Gazebo simulation software.

3.1 Robot Operating System

Robot Operating System (ROS)[15] is used to control the movement of the robot arm. ROS is an open-source, meta-operating system for robot software development. It provides standard services that would be expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

ROS is not a real-time framework, though it is possible to integrate ROS with real-time code.

ROS was designed to be as distributed and modular as possible, so the users can use as much or as little of ROS as they desire. The distributed nature of ROS also fosters a large community of user-contributed packages that add a lot of value on top of the core ROS system. At last count there were over 3,000 packages in the ROS ecosystem This is only the ROS packages that people have taken the time to announce to the public. These packages range in fidelity, covering everything from proof-of-concept implementations of new algorithms to industrial-quality drivers and capabilities. The ROS user community builds on top of a common infrastructure to provide an integration point that offers access to hardware drivers, generic robot capabilities, development tools, useful external libraries, and more.

The ROS framework is easy to implement in any modern programming language. It is already implemented it in Python, C++, and Lisp, and there are experimental libraries in Java and Lua.

ROS currently only runs on Unix-based platforms. Software for ROS is primarily tested on Ubuntu and Mac OS X systems, though the ROS community has contributed support for Fedora, Gentoo, Arch Linux and other Linux platforms. [9, 16]

3.1.1 ROS Concepts

ROS has three levels of concepts: the Filesystem level, the Computation Graph level, and the Community level. [17]

3.1.2 ROS Filesystem Structure

3.1.2.1 Packages

Packages[18] are the main unit for organizing software in ROS. In the file system they are represented by folders which contain a package manifest.

A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together. Packages are the most atomic build item and release item in ROS. Meaning that the most granular thing you can build and release is a package.

ROS packages tend to follow a common structure.

- include/package_name: C++ include headers
- msg/: Folder containing Message (msg) types
- src/package_name/: Source files, especially Python source that are exported to other packages.
- srv/: Folder containing Service (srv) types
- scripts/: executable scripts
- CMakeLists.txt: CMake build file
- package.xml: Package manifest
- CHANGELOG.rst: Many packages will define a changelog which can be automatically injected into binary packaging and into the wiki page for the package

3.1.2.2 Catkin

ROS utilizes a custom build system, catkin[19], that extends CMake to manage dependencies between packages.

The build system is needed, because ROS is a very large collection of loosely federated packages. That means lots of independent packages which depend on each other, utilize various programming languages, tools, and code organization conventions.

Because of this, the build process for a target in some package may be completely different from the way another target is built. catkin specifically tries to improve development on large sets of related packages in a consistent and conventional way. In other words, both rosbuild and now catkin aim to make building and running ROS code easier by using tools and conventions to simplify the process. Efficiently sharing ROS-based code would be more difficult without it.

3.1.2.3 Message types

Message descriptions, stored in .msg files, define the data structures for messages sent in ROS.[20]

This description makes it easy for ROS tools to automatically generate source code for the message type in several target languages. Message descriptions are stored in .msg files in the msg/ subdirectory of a ROS package.

There are two parts to a .msg file: fields and constants. Fields are the data that is sent inside of the message. Constants define useful values that can be used to interpret those fields (e.g. enum-like constants for an integer value). Each field consists of a type and a name, separated by a space:

For example:

```
fieldtype1 fieldname1
fieldtype2 fieldname2
fieldtype3 fieldname3
```

The field name determines how a data value is referenced in the target language. For

```
int32 x
int32 y
```

example, a field called 'pan' would be referenced as 'obj.pan' in Python, assuming that 'obj' is the variable storing the message.

Field types can be:

- a built-in type, such as "float32 pan" or "string name"
- names of Message descriptions defined on their own, such as "geometry_msgs/PoseStamped"
- fixed- or variable-length arrays (lists) of the above, such as "float32[] ranges" or "Point32[10] points"

- the special Header type, which maps to std_msgs/Header

ROS provides the special Header type to provide a general mechanism for setting frame IDs for libraries like tf. While Header is not a built-in type (it's defined in std_msgs/msg/Header.msg), it is commonly used and has special semantics. If the first field of your .msg is "Header header", it will be resolved as std_msgs/Header.

3.1.2.4 Service types

Service descriptions, stored in a.srv files[21], define the request and response data structures for services in ROS. A service description file consists of a request and a response msg type, separated by '—'. Any two .msg files concatenated together with a '—' are a legal service description.

```
string str
---
string str
```

3.1.3 Other important ROS concepts

3.1.3.1 Client Library

A ROS client library [22] is a collection of code that eases the job of the ROS programmer. It takes many of the ROS concepts and makes them accessible via code. In general, these libraries let you write ROS nodes, publish and subscribe to topics, write and call services, and use the Parameter Server. Such a library can be implemented in any programming language, though the current focus is on providing robust C++ and Python support. Main client libraries are roscpp, rospy and roslisp.

3.1.3.2 Coordinate frames

In a robotic system there are multiple coordinate frames that change in time. Converting vectors between them correctly is not simple.

tf[23] (and its successor tf2[23]) is a package that lets the user keep track of multiple coordinate frames over time. tf maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time.

3.1.3.3 Unified Robot Description Format

The Unified Robot Description Format (URDF)[24] is an XML specification to describe a robot. It is designed to be as general as possible, but obviously the specification cannot describe all robot. Only tree structures can be represented, ruling out all parallel robots.

The specification assumes the robot consists of rigid links connected by joints; flexible elements are not supported. The format can be used to specify the kinematic and dynamic description of the robot, the visual representation of the robot and the collision model of the robot.

3.1.3.4 Plugins

The pluginlib[25] package provides tools for writing and dynamically loading plugins using the ROS build infrastructure. To work, these tools require plugin providers to register their plugins in the package.xml of their package.

It is a C++ library for loading and unloading plugins from within a ROS package. Plugins are dynamically loadable classes that are loaded from a runtime library (i.e. shared object, dynamically linked library).

With pluginlib, one does not have to explicitly link their application against the library containing the classes – instead pluginlib can open a library containing exported classes at any point without the application having any prior awareness of the library or the header file containing the class definition. Plugins are useful for extending/modifying application behavior without needing the application source code.

3.1.4 ROS Computation Graph

The ROS Computation Graph is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure. The basic Computation Graph concepts of ROS are Master, nodes, Parameter Server, messages, services, topics, and bags, all of them provide data to the Graph in different ways.

3.1.4.1 Master

The ROS Master[26] acts as a nameservice in the ROS Computation Graph. It stores topics and services registration information for ROS nodes. Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections appropriate. The Master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are run.

The Master is implemented via XMLRPC[27], which is a stateless, HTTP-based protocol. XMLRPC was chosen primarily because it is relatively lightweight, does not require a stateful connection, and has wide availability in a variety of programming languages.

3.1.4.2 Parameter Server

The parameter server[28] is a shared, multi-variate dictionary that is accessible via network APIs. It runs inside of the ROS Master. Nodes use this server to store and retrieve parameters at runtime. As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters. It is meant to be globally viewable so that tools can easily inspect the configuration state of the system and modify if necessary.

The Parameter Server API is also implemented via XMLRPC[27]. The use of XML-RPC enables easy integration with the ROS client libraries and also provides greater type flexibility when storing and retrieving data. The Parameter Server can store basic XML-RPC scalars (32-bit integers, booleans, strings, doubles, iso8601 dates), lists, and base64-encoded binary data. The Parameter Server can also store dictionaries (i.e. structs).

3.1.4.3 Nodes

Nodes[29] are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server. The use of nodes in ROS provides several benefits to the overall system. There is additional fault tolerance as crashes are isolated to individual nodes. Code complexity is reduced in comparison to monolithic systems. Implementation details are also well hidden as the nodes expose a minimal API to the rest of the graph and alternate implementations, even in other programming languages, can easily be substituted.

Every node has a URI, which corresponds to the host:port of the XMLRPC server it is running[27]. The XMLRPC server is not used to transport topic or service data: instead, it is used to negotiate connections with other nodes and also communicate with the Master. This server is created and managed within the ROS client library, but is generally not visible to the client library user. The XMLRPC server may be bound to any port on the host where the node is running.

A ROS node is written with the use of a ROS client library, such as roscpp or rospy.

3.1.4.4 Messages

Nodes communicate with each other by publishing messages to topics[30]. A message is a simple data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as arrays of primitive types. Messages can include arbitrarily nested structures and arrays.

They are defined by .msg files that are simple text files specifying the data structure of

a message. The ROS Client Libraries implement message generators that translate .msg files into source code, so the messages are programming language independent.

3.1.4.5 Topics

Messages are routed via a transport system with publish / subscribe semantics[31]. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.

ROS currently supports TCP/IP-based and UDP-based message transport. The TCP/IP-based transport is known as TCPROS and streams message data over persistent TCP/IP connections. TCPROS is the default transport used in ROS and is the only transport that client libraries are required to support. The UDP-based transport, which is known as UDPROS and is currently only supported in roscpp, separates messages into UDP packets. UDPROS is a low-latency, lossy transport, so is best suited for tasks like teleoperation.

For example, the sequence by which two nodes begin exchanging messages is:[27]

1. Publisher node registers with the Master by sending its name, XMLRPC host:port, topic to publish to and topic type. [XMLRPC]
2. Subscriber node registers with the Master by sending its name, XMLRPC host:port, topic to subscribe to and topic type. [XMLRPC]
3. Master notices that there is a node that is interested in a topic that has a publisher, so it sends the XMLRPC address of the publisher to the subscriber. [XMLRPC]
4. The Subscriber sends a connection request to the XMLRPC address of the Publisher, sending its name, the topic name and a list of supported protocols. [XMLRPC]
5. The Publisher responds with a selected protocol and the address which uses the negotiated protocol. [XMLRPC]
6. The Subscriber connects to the address using the negotiated protocol.
7. The connection is established, data is sent from the publisher to the subscriber.

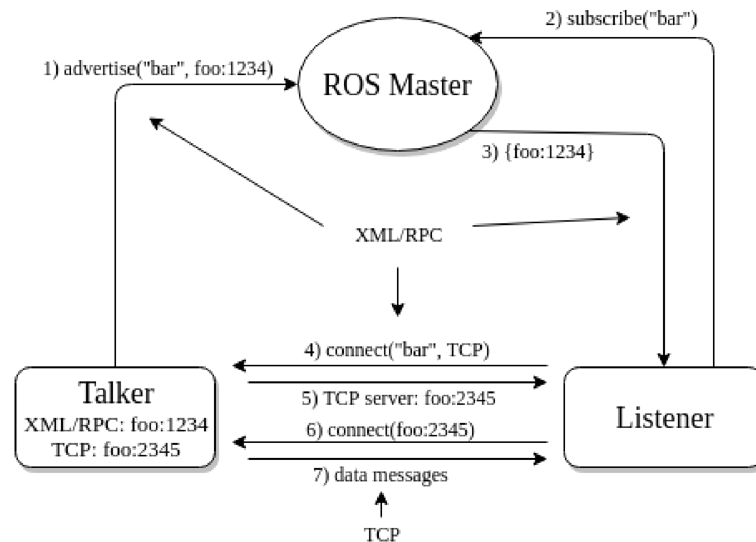


Figure 3.1: *The sequence of connection*

The Master keeps track of the publishers and subscribers of all topics, so when there is a new publisher to a topic, it can notify the subscribers of that topic to connect to that publisher. Also, when there is a new subscriber, it will send all publishers address to it so it can connect to them all.

Consequently, the order in which the nodes are registered does not matter, simplifying the startup processes of complicated computation graphs.

3.1.4.6 Services

The publish / subscribe model of topics is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request/reply interactions, which are often required in a distributed system. Request/reply is done via services[32] , which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply. ROS client libraries generally present this interaction to the programmer as if it were a remote procedure call. Services are defined using .srv files, which like .msg files are compiled into source code by a ROS client library.

3.1.4.7 Actions

In any large ROS based system, there are cases when someone would like to send a request to a node to perform some task, and also receive a reply to the request. This can currently be achieved via ROS services. In some cases, however, if the service takes a long time to execute, the user might want the ability to cancel the request during execution or

get periodic feedback about how the request is progressing. The actionlib package[33] provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server.

3.1.4.8 Bags

Bags [34] are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms. Bags are usually created using the rosbag command-line tool.

3.1.5 ROS Community

There are ROS resources that enable separate communities to exchange software and knowledge.

3.1.5.1 Distributions

ROS Distributions[35] are collections of versioned stacks that you can install. Distributions play a similar role to Linux distributions: they make it easier to install a collection of software, and they also maintain consistent versions across a set of software. There are 4 distributions that are maintained at the time of writing.

Table 3.1: *Recent distributions*

Distribution	**Release Date**	**End of Life Date**
ROS Melodic Morenia	May 23rd, 2018	May, 2023
ROS Lunar Loggerhead	May 23rd, 2017	May, 2019
ROS Kinetic Kame	May 23rd, 2016	May, 2021
ROS Indigo Igloo	July 22nd, 2014	April, 2019

Melodic Morenia and Kinetic Kame are LTS (Long Term Support) distributions, meaning they receive updates for 5 years. Lunar Loggerhead, not being LTS release, is only updated for 2 years.

3.1.5.2 ROS Wiki

The ROS community Wiki[36] is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute their own documentation, provide corrections or updates, write tutorials, and more.

3.1.6 ROS Names

3.1.6.1 Graph Resource Names

Graph Resource Names[37] provide a hierarchical naming structure that is used for all resources in a ROS Computation Graph, such as Nodes, Parameters, Topics, and Services.

They are an important mechanism in ROS for providing encapsulation. Each resource is defined within a namespace, which it may share with many other resources. In general, resources can create resources within their namespace and they can access resources within or above their own namespace. Connections can be made between resources in distinct namespaces, but this is generally done by integration code above both namespaces. This encapsulation isolates different portions of the system from accidentally grabbing the wrong named resource or globally hijacking names.

Names are resolved relatively, so resources do not need to be aware of which namespace they are in. This simplifies programming as nodes that work together can be written as if they are all in the top-level namespace.

Any name within a ROS Node can be remapped when the node is launched at the command-line.

3.1.7 Package Resource Names

Package Resource Names[37] are used in ROS with Filesystem level concepts to simplify the process of referring to files and data types on disk. Package Resource Names are very simple: they are just the name of the Package that the resource is in plus the name of the resource. For example, the name "std_msgs/String" refers to the "String" message type in the "std_msgs" Package.

3.2 Gazebo

Gazebo[7] is a 3D dynamic simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a rich library of robot models and environments, a suite of sensors, and interfaces for both users and programs. Gazebo is free and widely used among robotic experts.

Typical uses of Gazebo include: testing robotics algorithms, designing robots, performing regression testing with realistic scenarios.

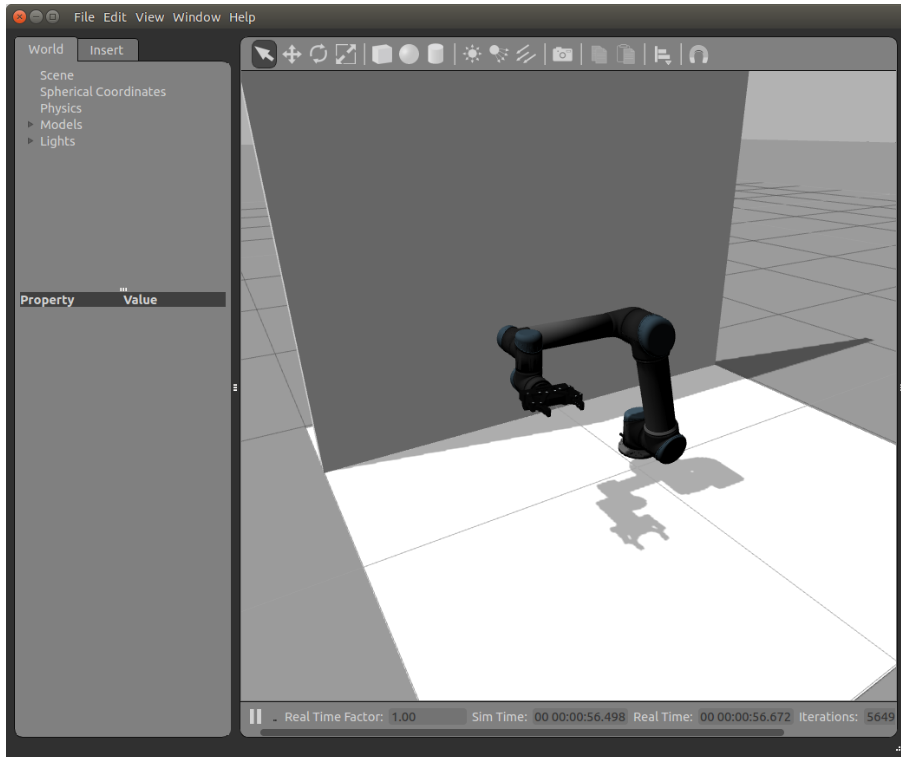


Figure 3.2: *Gazebo simulation of an UR5 robot arm*

3.3 UR10 Robot arm

The UR10[38] robot arm designed by Universal Robots has 6 degrees of freedom with its 6 rotating joints. Its payload can be up to 10 kg. It has a reach of 1300 mm. It is controlled by sending text commands to it using a TCP/IP connection. The commands are in a special script language called URScript[39]. By sending commands you can control the robot's Cartesian position, velocity, joint angle and velocity.

3.4 Used ROS packages

To avoid reinventing the wheel, multiple ready made packages were used to create our setup.

3.4.1 universal_robot package

The `universal_robot metapackage`[40] contains packages that provide nodes written in Python for communication with Universal's industrial robot controllers and URDF models for various robot arms (UR3, UR5, UR10).

- ur_description: This package contains the model of the robot. The urdf and the

mesh files are describing the robot links;

- ur_gazebo:This package contains files that aid in starting a robot simulation.

3.4.2 ros_control package

Ros_control [41] is a set of packages defining a set of interfaces, which are designed to abstract away differences between robot hardware.

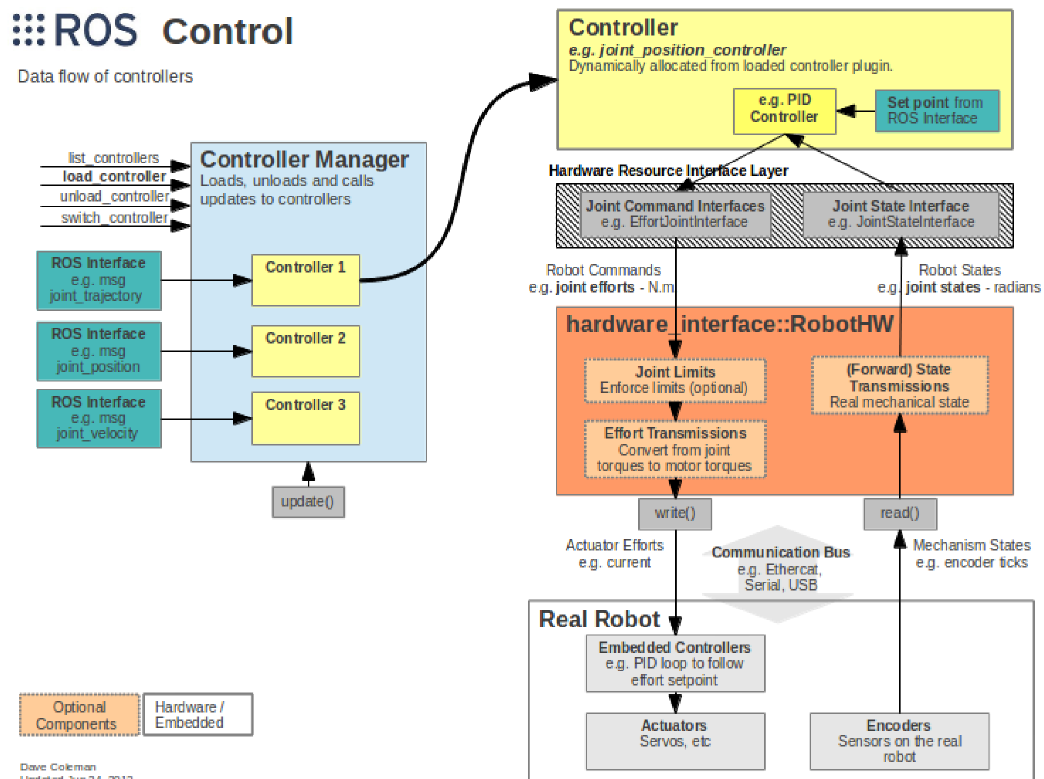


Figure 3.3: *Overview of ros_control [41]*

There are controllers, which provide standard ROS interfaces (topic or service) in order to allow communication between the robot and other ROS nodes using not robot-specific topics, and messages. The controllers are not robot specific, but they are using interfaces that are C++ classes to read and write. These interfaces represent hardware elements (e.g., VelocityJointInterface that can represent a joint that can be controlled by using velocity commands). The interfaces are basically shared memory where command can be written to and state can be read from.

The controllers for example can use PID controllers to control the interfaces, that way they can for example receive position commands from a topic, and through a PID

controller it can control a VelocityJointInterface.

The controller can also read from the interface, so it can publish information about the hardware represented by the interface (e.g., joint state, torque information).

The interfaces are implemented in a hardware specific driver extending hardware_interface::RobotHW, that takes care of communicating with the robot using its hardware specific communication method (serial, Modbus, Ethernet, USB).

The controllers and drivers are implemented using the pluginlib package to make them dynamically loaded.

3.4.2.1 joint_state_controller/JointStateController

This is a controller that reads state data (joint angles, velocities, efforts) from JointStateInterfaces, and publishes them in sensor_msgs/JointState messages to the /joint_state topic.

3.4.2.2 velocity_controllers/JointTrajectoryController

It is a controller for executing joint-space trajectories on a group of joints. Trajectories are specified as a set of waypoints to be reached at specific time instants, which the controller attempts to execute as well as the mechanism allows. Waypoints consist of positions, and optionally velocities and accelerations.

3.4.3 gazebo_ros_pkgs package

gazebo_ros_pkgs[42] is a set of ROS packages that provide the necessary interfaces to simulate a robot in the Gazebo 3D rigid body simulator for robots. The package integrates with ROS using ROS messages, services and dynamic reconfigure.

It contains a converter that converts URDF into SDF which is the world description language that Gazebo uses. This way there is no need to maintain two sets of models.

gazebo_ros_pkgs package

gazebo_ros_pkgs also contains the gazebo_ros_control_package which is a ROS package for integrating the ros_control controller architecture with the Gazebo simulator.

It provides a Gazebo plugin which instantiates a ros_control controller manager and connects it to a Gazebo model. The Gazebo plugin also loads in the DefaultRobotHWSim plugin through pluginlib which creates the hardware_interfaces (position, velocity or effort) for each joint as defined in the loaded URDF.

3.4.4 xacro package

The xacro package[43] is most useful when working with large XML documents such as URDFs. Xacro is an XML macro language. With xacro, you can construct shorter and more readable XML files by using macros that expand to larger XML expressions.

3.4.5 roslaunch package

roslaunch[44] is a tool for easily launching multiple ROS nodes locally and remotely via SSH, as well as setting parameters on the Parameter Server. It includes options to automatically respawn processes that have already died. roslaunch takes in one or more XML configuration files (with the .launch extension) that specify the parameters to set and nodes to launch, it is also possible to upload configurations to the Parameter Server from YAML files.

3.4.6 ariac package

The ariac package contains the simulation environment and the GEAR interface [45]. GEAR provides a ROS interface to control all available actuators, read sensor information and send/receive notifications.

3.4.7 hebiros package

The hebiros package [46] is an API provided by Hebi Robotics as source code that can be compiled into a node in a catkin workspace. The package contains the hebiros node and several example nodes to demonstrate the use of the API. The hebiros node is responsible for controlling modules and for creating and maintaining communication. The examples show how to:

- identify robot modules on the network,
- receive feedback from the modules,
- send commands to the robot,
- execute trajectories,
- simulate the robot arm with Gazebo[7],
- integrate and visualize trajectory planning with MoveIt[47].

Chapter 4

Implementation

4.1 Implementation of a modular robotic architecture

In this section we provide a brief description of the solutions we have adopted in order to implement a modular architecture. We follow the division as [13] uses to divide the architecture of a modular system.

4.1.1 Actuator modules

We used the X-Series Actuators [48] (see Figure 4.1) of Hebi Robotics, which are 'smart' series-elastic actuators that integrate a brushless motor, geartrain, spring, encoders, and control electronics into a compact package that run on standard DC voltages.

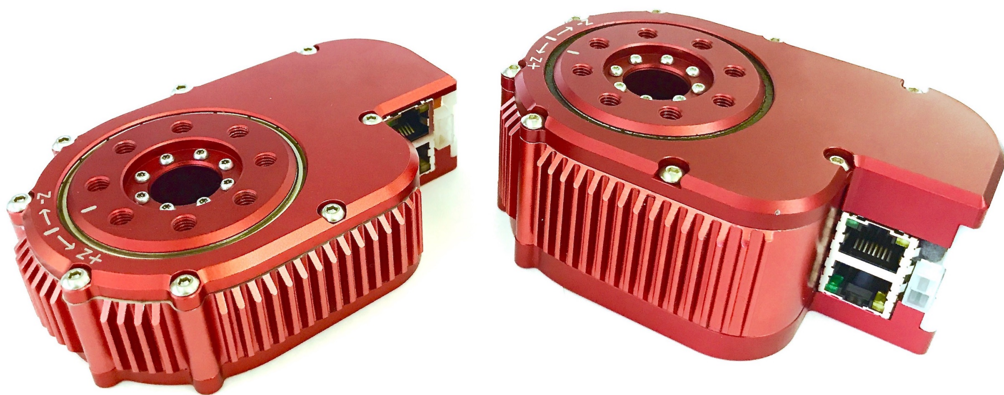


Figure 4.1: *Hebi Robotics' X-Series Actuators*

4.1.2 Connection mechanism

Hebi Robotics connects the actuators to other building blocks with bolts. Our connection mechanism is able to join two modules mechanically in a few seconds. This solution is explained in more details in the 4.4 chapter.

4.1.3 Energy

In this work, we aim to design a fully autonomous and flexible modular architecture, without the need for any wire or tether, which would limit the resulting robots' motions and their independence.

However the real world modules run on DC voltages between 24V-48V. The actuators can operate from external power supply, battery, or Power over Ethernet (PoE). The internal electronics of the modules are designed to control and automatically scale motor control parameters as the bus voltage changes.

4.1.4 Sensors

All of the actuators contain specific sensors to measure the position as well as an accelerometer to provide their spatial orientation. The modules send feedback on their status, e.g.:

- the amount of current draw of the motor of an actuator,
- sensed position,
- sensed velocity,
- sensed force or torque.

4.1.5 Communications

The X-Series Actuators run firmware that allow them to communicate over a standard 10/100Mbps Ethernet connection. In our solution we communicate using ROS topics over TCP.

4.1.6 Control

Hebi actuator modules can be controlled by using position, velocity, and effort. Typically, the best performance can be achieved by coordinating and controlling a combination of commanded positions, velocities, and effort. To combine these control inputs, PID control

loops are used on position, velocity, and effort. These controllers can be cascaded and combined in different preset configurations.

A controller was made by Hebi Robotics that can be used for simulation purposes, but it was not suitable for our modular approach, therefore we have revised it according to our needs. For more information, see chapter 4.2.

4.2 Controller for our modular approach

In order to control the modular robot, we created a controller that is capable of controlling the servos of the robot arm without pre-assigning the number of elements to control. Hebi Robotics' solution was not suitable for our modular approach because the degree of freedom of the modular robot arm is not constant, therefore the number of servos to be controlled is not constant either.

We created a control interface that processes the received commands and sends position command to the right servo. At he beginning of the simulation when the models spawn a hebiros node is created for each servo. The original code contained one hebiros node that controls all servos. Our hebiros node is responsible for controlling only one servo. Each servo is controlled by a proportional-integral-derivative (PID) controller.

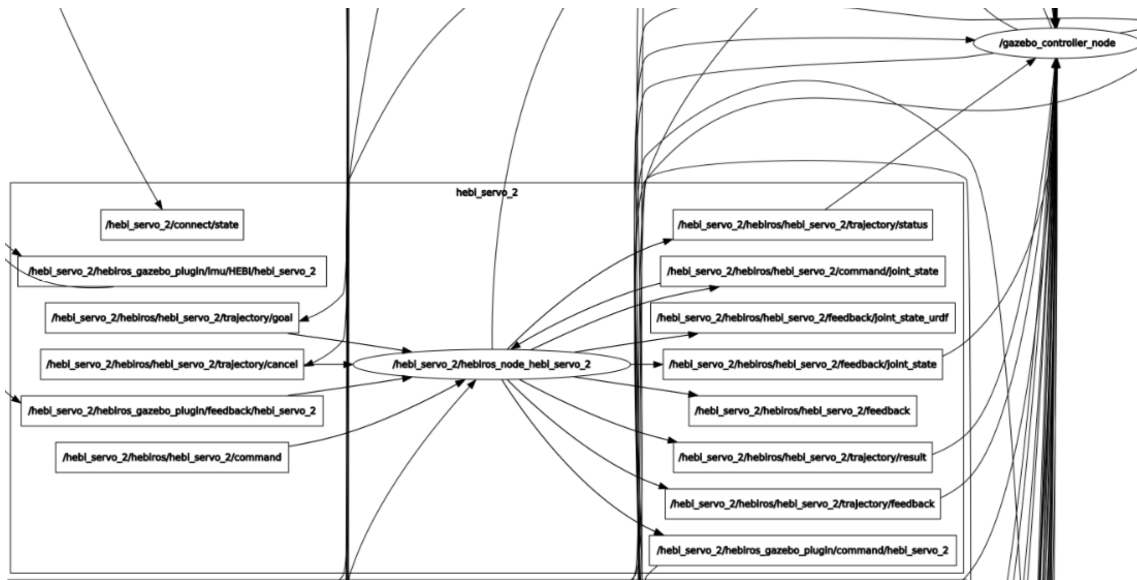


Figure 4.2: *A hebiros node from the system, the oval circle denotes the nodes, and the rectangles are the topics*

The controlling interface called "gazebo_controller_node" receives the number of servos from the parameter server. With this solution, the control system is independent of the number of servos in the simulation. It is not self-evident what element in the control vector represents which module. The developer is responsible to provide the correct com-

mand to the right servo through the control interface. The 4.2 figure shows which topics are connected to the hebiros node and the controller interface.

It is possible to send position commands by command-line interfaces or programmatically. Command-line example:

```
rostopic pub /hebiros/commands/mine hebiros_6dof_ariac/ServoCommandMsg '[.NAN, .NAN
    ↪ , .NAN, .NAN ,NAN, .NAN, .NAN, .NAN, .NAN ]' '3'
```

If NaN value is sent then the servo status remains unchanged. It is important that the input vector has the same size as the number of servos in the simulation. The unit of values of the vector is radian and the last value is second. The last value determines how much time the controller gets to set to the specified position. The possible values are not limited because the servos can turn around more than once.

4.3 Hints to identify our contribution

In order to use the Hebi Robotics components in the ARIAC environment, the contents of the hebiros package had to be modified to serve only one module. The name of the module is given at the beginning of the simulation. Thus, each module has a unique identifier so every module can communicate on separate topics. We had to create one unified control interface to send commands to all modules.

Figure 4.2 shows the topics for the modules. It can be seen that the beginning of each topic is the same as the module name. Figure 4.3 shows only the ROS nodes and the connection between them. The modules and the control interface were highlighted with green. Table 4.1 presents which files had to be created or modified in the ARIAC and hebiros packages. ur_action and hebi_action files are mainly motion control functions, but include service calls as well as callback functions. The scheduler organizes coordination between the two robots.

Figure 4.3: *System overview*

Table 4.1: *A brief summary of the files that are needed to be written or modified*

File name	Number of lines / added lines	Package	Programming language
hebiros_node	54/1	hebiros_package	cpp
hebiros_actions	117/10	hebiros_package	cpp
hebiros_clients	60/4	hebiros_package	cpp
hebiros_parameters	92/12	hebiros_package	cpp
hebiros_publishers	57/4	hebiros_package	cpp
hebiros_publishers_gazebo	20/1	hebiros_package	cpp
hebiros_services	154/3	hebiros_package	cpp
hebiros_services_gazebo	165/10	hebiros_package	cpp
hebiros_subscriber_gazebo	61/3	hebiros_package	cpp
hebiros_gazebo_controller	441/14	hebiros_package	cpp
hebiros_gazebo_group	93/6	hebiros_package	cpp
hebiros_gazebo_joint	24/3	hebiros_package	cpp
hebiros_gazebo_plugin	198/15	hebiros package	cpp
hebiros_gazebo_controller_node	254/254	hebiros package	cpp
ur_actions	1613/1613	ariac_package	python
scheduler	164/164	ariac_package	python
hebi_actions	141/141	ariac_package	python

4.4 Attaching the parts with vacuum grippers

A simulated pneumatic gripper was attached to each robotic part. These can be remotely controlled by ROS services. An object will be attached to the gripper if they are making contact. The gripper regularly publishes its status. The published message contains whether the suction is enabled or disabled or whether there is an object attached to the gripper. For example:

```
$ rostopic echo /hebi_servo_1/gripper/state
enabled: False
attached: False
---
```

The idea of using vacuum grippers for binding object together originates from the 2018 Agile Robotics for Industrial Automation Competition (ARIAC) where the same gripper was attached to the end effector of the UR10[38] robot arm.

4.5 Issues with the servos

In the simulation, we have noticed that the servos get infinite acceleration from even little forces, for example from falling from a low altitude. This issue is related to the inertia of the models. We assumed that the given values are correct and by increasing the mass,

the inertia will change in a direct proportion. However we have increased the mass of the servos the issue was still persisted.

The inertia tensor encodes both the mass and the distribution of mass, so it does depend on the mass of the object and also where it is located. Each tensor is defined relative to a coordinate frame or set of axes. Determining the correct values can be performed by using CAD software that includes this feature or manually for simple objects. It is required to have a good approximation for these values to get an accurate simulation in Gazebo.

We assumed that the servos are homogeneous bodies and the shape is proximate to a cuboid. This is not a drastic simplification because, except for the part where Ethernet cables can be connected, the servos are shaped like a cuboid. Calculating the inertia tensor for a 4.4 cuboid is quite simple by the 4.1 formula.

$$\mathbf{I} = \begin{bmatrix} \frac{1}{12}m(h^2+d^2) & 0 & 0 \\ 0 & \frac{1}{12}m(w^2+d^2) & 0 \\ 0 & 0 & \frac{1}{12}m(w^2+h^2) \end{bmatrix} \tag{4.1}$$

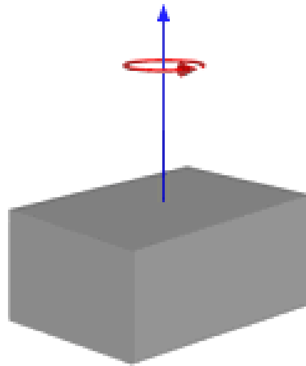


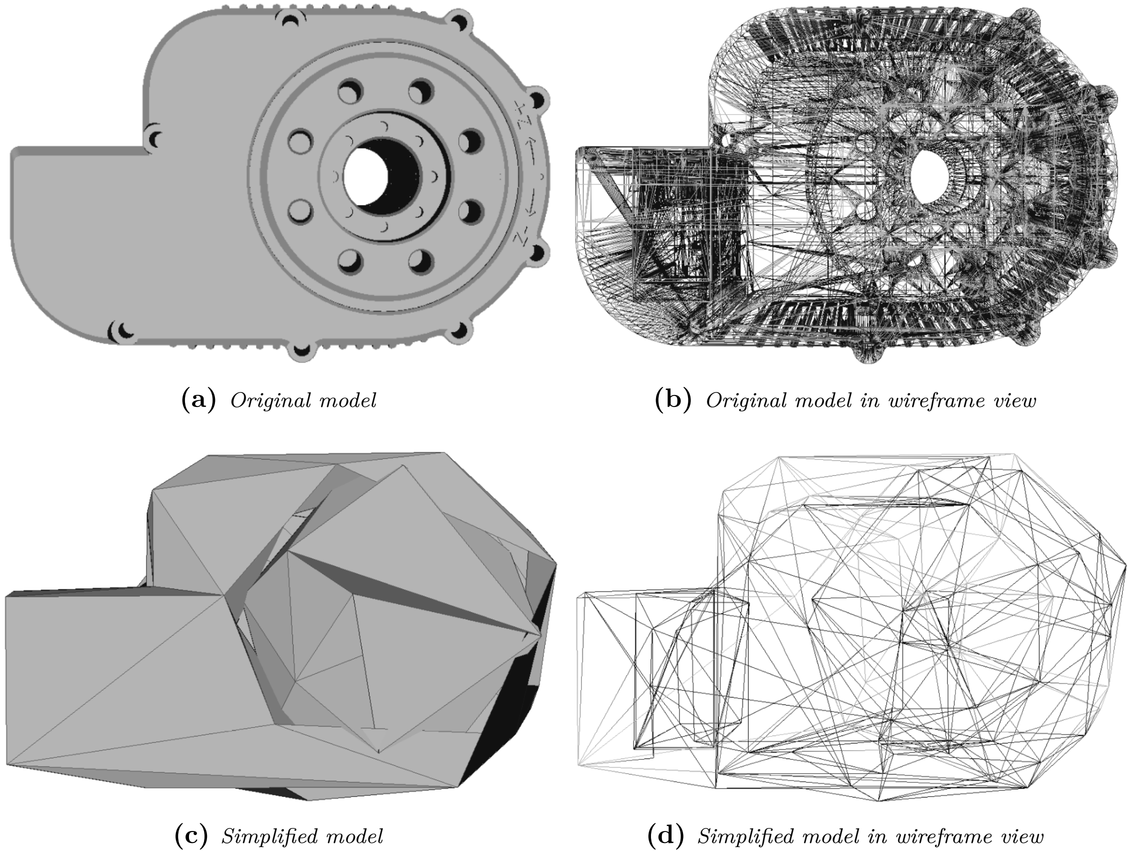
Figure 4.4: *Solid cuboid of width w, height h, depth d, and mass m*

4.6 Real time factor optimization

Real time factor (RTF) is the ratio of simulation time to real time. It is important to keep this number as high as possible otherwise the simulations will last much longer. This factor can be increased by simplifying the environment or using a better computer. Our problem was that high number of robot parts significantly reduced the RTF when there was no change in the environment. The reason is that the collision and visual model are the same in the URDF[24] file.

We need to reduce its geometric complexity, creating a geometry with the same shape but with less triangles (or points). For this purpose we used MeshLab [49] which is an open source software for 3D triangular mesh editing. In Meshlab one can choose from

many simplification algorithms. We used Quadric Edge Collapse Decimation which is able to preserve the boundary and normal of a 3D mesh. We significantly reduced the face number of our models. We simplified a 2.5 MB file 4.5a to a 45.5 kB (see 4.5c) which is one-sixtieth of the original file. This method was able to improve the real time factor by 0.18.



(a) *Original model* **(b)** *Original model in wireframe view*

(c) *Simplified model* **(d)** *Simplified model in wireframe view*

Figure 4.5: *The images show that the detail-rich model was simplified to the unrecognizability*

An example of models before and after simplification is shown in the figure 4.5. Another way to improve the RTF is to simplify the complexity of our system. Instead of creating a separate interface for each servo controller, one can combine the interfaces to achieve a better performance.

Chapter 5

Results and evaluation

5.1 Building a modular robotic arm

Our first goal is to build a robot arm which has at least 4 degrees of freedom. We placed a base with vacuum gripper on which the assembly will take place. The parts spawn in the yellow storage bins at the beginning of the simulation. To prevent the components from slipping, the parts are placed on the lower edge of the containers. By this method we can guarantee that the parts will always be in the same place in the simulation.

To ensure that the modular robots perform closest to the industrial grade counterparts we rely the self-reconfiguration task on a regular industrial robotic arm. Its existence is considered as a baseline in the ARIAC 2018 cell. It can ensure that the assembly of the robotic modules with the links are accurate. Still there can be accuracy issues which cannot be addressed solely by the application of the industrial robotic arm, for example eccentric assembly of the links on the vacuum gripper due to the one-fit-for-all parameterized PID controllers, but we intend to deal with those in the further work.

The robot arm is made up of three types of components: servo, bracket and tube. The parts are moved by the UR10 robot arm.

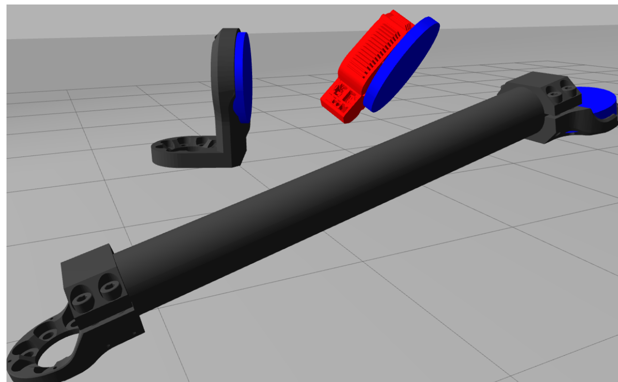


Figure 5.1: *Robotic parts are all equipped with a vacuum gripper*

The assembly can be divided into three main steps:

1. the robot arm moves to the right storage bin,
2. it picks up the robotic part,
3. it attaches the component to the robot arm.

Each operation requires a unique movement, but these movements can be split into smaller elementary operations, for example: move to a yellow storage container, go above the parts or go out of the container. The control code[45] for the UR10 robot arm is provided by Open

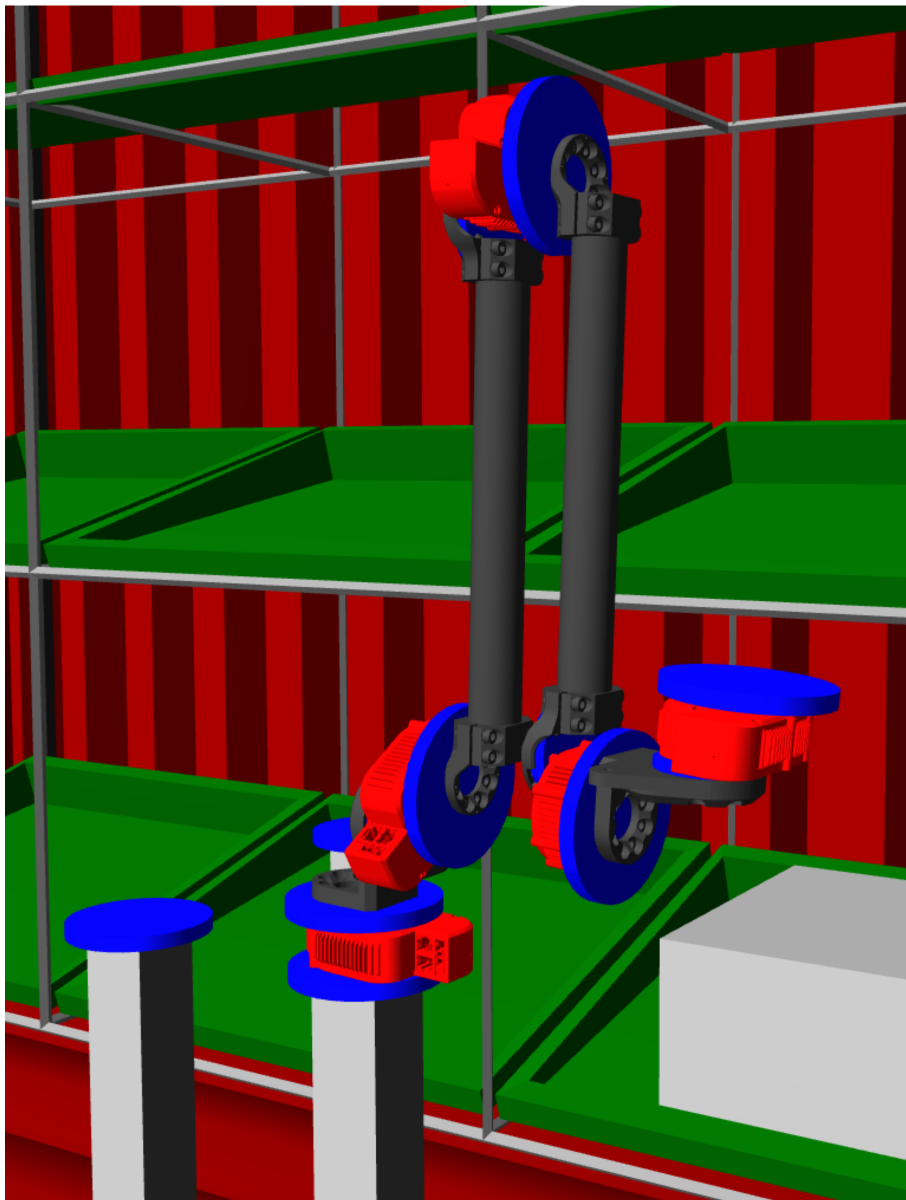


Figure 5.2: *The finished 5 DOF robot arm*

Source Robotics Foundation (OSRF) in the 2018 Agile Robotics for Industrial Automation Competition (ARIAC). The controller of the UR10 is subscribed to a topic that can be used to control all of the joints of the robot arm. Desired position, velocity acceleration and/or effort for each joint can be sent to this topic using the trajectory_msgs/JointTrajectory message type. The controller sets the specified values.

```
std_msgs/Header header
string[] joint_names
trajectory_msgs/JointTrajectoryPoint[] points
```

Rqt's Joint Trajectory controller is a graphical user interface for controlling the UR10 robot arm. We used it to design and test trajectories. This interface is shown in Figure 5.3.

Figure 5.3: *This is the graphical user interface for executing joint-space trajectories on a group of joints*

One can manually set the value for each joint and the speed of the action execution. This is a very useful tool for testing and developing. If we know the desired joint states from the graphical interface, it is also possible to set these value programmatically. We divided a function to more atomic movements, for example picking up a part that we can

reuse for other parts as well.

It is also necessary to control the servos while assembling the modular robot arm, otherwise the two elements will be unevenly connected. In some cases the UR10 robot arm would not have access to the element to which it should be attached. Thanks to the vacuum grippers topic one can check when the two parts are in contact. This could be useful when the robot arm is fixed in order to keep the attached part in the proper position.

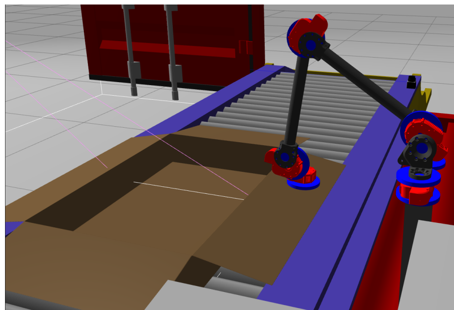
The completed robot arm has 5 degrees of freedom. Every joint of the modular robot arm can rotate without restrictions. The assembled robot arm is depicted in the figure 5.2.

5.2 Demonstrating the capabilities of the robot arm

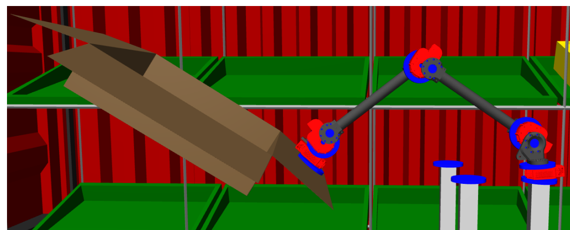
Although the completed robot arm demonstrates modularity in itself, it does not present the possibilities that it can achieve. We came up with a few tasks that can be performed with the modular robot arm.

5.2.1 Lift the box

The first task is that a robot arm should be able to pick an object. Next to the modular robotic arm, there is a conveyor belt that can be controlled by service calls. Activating the conveyor belt, the shipping box on it can be moved to the robot, which will be lifted and thrown away. It is worth to observe that once the vacuum gripper releases the box, the robot arm swings in the opposite direction. This can be explained by that it takes time for the controller of the modules to adjust the correct torque to the current load. This operation is shown in the figure 5.4.



(a) *The beginning of the action*



(b) *The end of the action before release*

Figure 5.4: *Demonstrating that the modular robot arm can lift and throw objects*

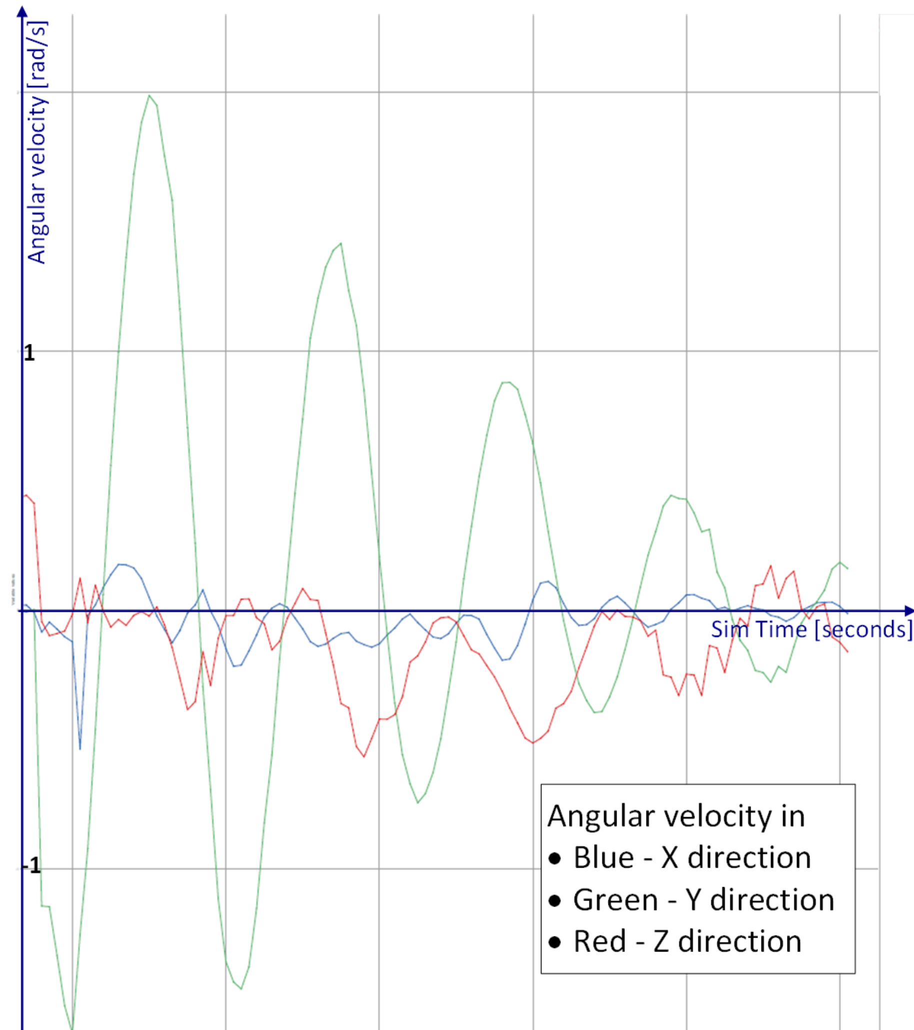


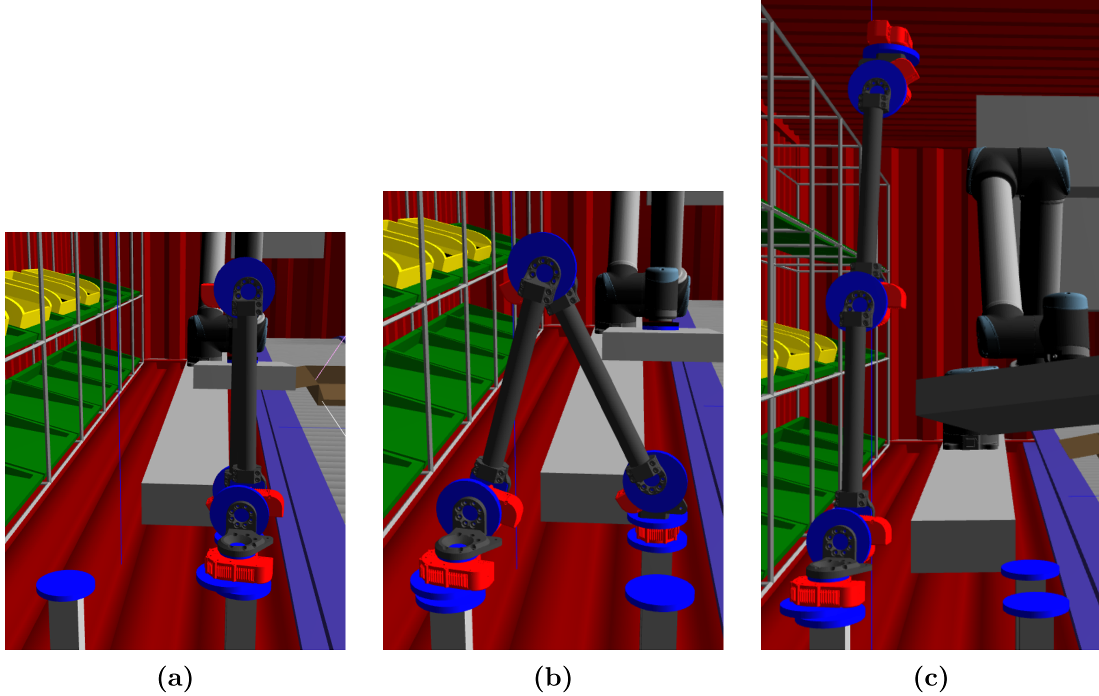
Figure 5.5: *Angular velocity in X-,Y-,Z-direction of the wrist module of the modular robot arm after releasing the shipping box*

The Figure 5.5 shows the speed of the module at the end of the robot arm at its x-, y-, z-direction. It is shown in the figure what happens when the robot arm releases the shipping box. The last module of the robot arm swings in the Y-direction. The robot arm slowly controls itself back into the initial state, which requires a considerable amount of time because all modules have their own controller that does not take into account the effects of the other controllers.

5.2.2 Step

The modular robot arm is not permanently attached to a static base. It is possible that another robot arm or even itself will change its position. We have created a stepping feature. From the point where the assembly took place, the robot arm steps to another platform. This simple example demonstrates that the movement of the robot arm is not

limited and there are many opportunities to take advantage of by moving the arm. This action is shown in the figure 5.4.



(a) (b) (c)

Figure 5.6: *Demonstrating that the modular robot arm can change its position without external intervention*

5.2.3 Alternative use case of the robot parts

One of the most important features of the servos is that it can rotate without restriction. This can be exploited to rotate the wheel of a car. Naturally, this is a very schematic model, not a real car, but it is good for example to demonstrate the modularity and reusability of the modular robotic parts.

To create the car we had to modify the URDF files of the servos. The vacuum gripper is moved to the other side of the servo and the cylinder is rotated not the vacuum gripper. In order to smoothly rotate the wheel its inertia matrix 5.1 had to be determined. For a cylindrical body (see Figure 5.7) this can be calculated by hand.

$$\mathbf{I} = \begin{bmatrix} \frac{1}{12}m(3r^2 + h^2) & 0 & 0 \\ 0 & \frac{1}{12}m(3r^2 + h^2) & 0 \\ 0 & 0 & \frac{1}{2}mr^2 \end{bmatrix} \tag{5.1}$$

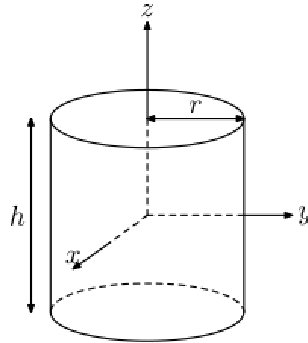
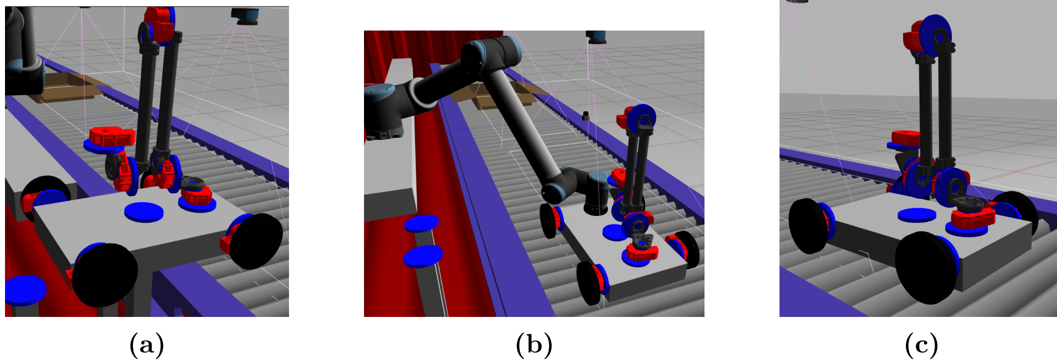


Figure 5.7: *Solid cylinder of radius r, height h and mass m*

The base of the car is a cuboid with a vacuum gripper on top. The purpose of the gripper is to allow the car to carry equipment or another robot. The car's assembly consists of two steps:

1. the UR10 robot arm takes the body of the car to the assembly site
2. it attaches the wheels to the base of the car

After assembly, the UR10 robot arm moves the modular robot arm to the car (see Figure 5.8a). The robot arm exerts a greater torque on the UR10 that it could compensate. This will result that we have to wait for each movement of the robot arm to move to the correct position. After the robot arm is attached to the car, the UR10 moves them onto the conveyor belt (see Figure 5.8b) where the car can move freely (see Figure 5.8c).



(a) **(b)** **(c)**

Figure 5.8: *The demonstration of what else can be assembled from the robot components.*

Chapter 6

Conclusion and further work

Conclusion

This work presents a possible approach to modular robotics that can be used in industrial environments. We used robotic parts provided by Hebi Robotics to implement a modular robotic architecture. The parts are gathered with an UR10 robot arm and a simulated pneumatic gripper was attached to each robotic part to join two modules mechanically. Hebi Robotics' solution is not suitable for our modular approach as the degree of freedom of the modular robot arm is not constant, therefore the number of servos to be controlled is not constant either. In order to control the modular robot, we created a controller that is capable of controlling the servos of the robot arm without pre-assigning the number of elements to control.

Our first goal was to build a robot arm which could be applied in an industrial environment. The completed robot arm is capable of reconfiguration in a task specific way. Every joint of the modular robot arm can rotate without restrictions, thus the robot arm is able to move in a much larger state space than a conventional robot arm. Although the robot arm demonstrates modularity in itself, it does not present the possibilities that it can achieve. We came up with a few tasks that can be performed with the modular robot arm. It was tested for simple operations such as manipulation, walking or transportation. The results demonstrate the success of our modular architecture to execute similar task like the conventional robots, combined with the flexibility to reconfigure itself to another task.

Further work

As a next step we would like to perform quantitative measurements to check if any improvement of the ARIAC KPIs is feasible with the modular robotic arms. To achieve this we have to achieve that the same ARIAC tasks are solved with the modular robotics approach. That needs a lot of study and work including robotic planning, actuation in Cartesian space and interfacing with various parts of ROS.

Another direction is the issue with accuracy and repeatability. While the servos provide industrial grade accuracy, the assembly inaccuracies of the robot arm can lead to inaccurate pick and place actions. This issue can be addressed during the fine tuning of the assembly task and the fine tuning of the arm movement of the misassembled arm as well. Techniques with learning the inverse kinematics of robotic arms are widely applied nowadays e.g., [50, 51].

Bibliography

[1] Yoram koren, the global manufacturing revolution: Product-process-business integration and reconfigurable systems, john wiley & sons, inc. "http://onlinelibrary.wiley.com/book/10.1002/9780470618813, 2010.

[2] Table of disruptive technologies. https://www.imperial.ac.uk/media/imperial-college/administration-and-support-services/enterprise-office/public/Table-of-Disruptive-Technologies.pdf, 2018.

[3] 5G ultra-low latency propels jet engine manufacturing. https://www.ericsson.com/en/networks/cases/5g-ultra-low-latency-propels-jet-engine-manufacturing, 2018.

[4] ISO: International Organization for Standardization. 1998. Manipulating industrial robots – Performance criteria and related test methods, NF EN ISO9283. https://www.iso.org/standard/22244.html, 1998.

[5] The Top 5 Cobot KPIs, How To Measure And Iimprove The Performance of Collaborative Robots. https://blog.robotiq.com/top-five-kpis-for-cobots, 2018.

[6] Siemens Plant Simulation. https://www.plm.automation.siemens.com/store/en-us/plant-simulation/, 2018.

[7] Gazebo Robot Simulator. http://gazebosim.org.

[8] Agile Robotics for Industrial Automation Competition (ARIAC). http://gazebosim.org/ariac, 2017.

[9] Introduction to ROS, ROS Wiki. http://wiki.ros.org/ROS/Introduction.

[10] A. Brunete, A. Ranganath, S. Segovia, J. Perez de Frutos, M. Hernando, and E. Gambao. Current trends in reconfigurable modular robots design. *International Journal of Advanced Robotic Systems*, 14(3):1729881417710457, 2017.

[11] J. Liu, X. Zhang, and G. Hao. Survey on research and development of reconfigurable modular robots. *Advances in Mechanical Engineering*, 8(8):1687814016659597, 2016.

[12] J. Baca, P. Pagala, C. R., and M. Ferre. Modular robot systems towards the execution of cooperative tasks in large facilities. *Robotics and Autonomous Systems*, 66:159 – 174, 2015.

[13] A. Faíña, F. Orjales, D. Souto, F. Bellas, and R. J. Duro. Designing a modular robotic architecture for industrial applications. In *2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS)*, volume 02, pages 874–879, Sept 2013.

[14] C. Jiang, H. Wei, and Y. Zhang. Automatic modeling and simulation of modular robots. *IOP Conference Series: Materials Science and Engineering*, 320(1):012003, 2018.

[15] Main website of Robot Operating System. `http://www.ros.org/`.

[16] Is ROS for me? `http://www.ros.org/is-ros-for-me/`.

[17] ROS Concepts, ROS Wiki. `http://wiki.ros.org/ROS/Concepts`.

[18] ROS Packages, ROS Wiki. `http://wiki.ros.org/Packages`.

[19] ROS catkin Build system, ROS Wiki. `http://wiki.ros.org/catkin`.

[20] ROS Msg Files, ROS Wiki. `http://wiki.ros.org/msg`.

[21] ROS Msg Files, ROS Wiki. `http://wiki.ros.org/srv`.

[22] ROS Client Libraries, ROS Wiki. `http://wiki.ros.org/ClientLibraries`.

[23] ROS Coordinate Frames, ROS Wiki. `http://wiki.ros.org/tf`.

[24] ROS URDF, ROS Wiki. `http://wiki.ros.org/urdf`.

[25] ROS Plugins, ROS Wiki. `http://wiki.ros.org/pluginlib`.

[26] ROS Master, ROS Wiki. `http://wiki.ros.org/Master`.

[27] ROS Technical Overview, ROS Wiki. `http://wiki.ros.org/ROS/TechnicalOverview`.

[28] ROS Parameter Server, ROS Wiki. `http://wiki.ros.org/Master`.

[29] ROS Nodes, ROS Wiki. `http://wiki.ros.org/Nodes`.

[30] ROS Messages, ROS Wiki. `http://wiki.ros.org/Messages`.

[31] ROS Common Messages, ROS Wiki. `http://wiki.ros.org/common_msgs`.

[32] ROS Services, ROS Wiki. http://wiki.ros.org/Services.

[33] ROS actionlib package, ROS Wiki. http://wiki.ros.org/actionlib.

[34] ROS Bags, ROS Wiki. http://wiki.ros.org/Bags.

[35] ROS Distributions, ROS Wiki. http://wiki.ros.org/Distributions.

[36] ROS Wiki, ROS Wiki. http://wiki.ros.org/.

[37] ROS Names, ROS Wiki. http://wiki.ros.org/Names.

[38] UR10 Robot Arm. https://www.universal-robots.com/products/ur10-robot/.

[39] URScript. https://www.universal-robots.com/how-tos-and-faqs/how-to/ur-how-tos/ethernet-socket-communication-via-urscript-15678/â??, 2017.

[40] ROS universal_robot package, ROS Wiki. http://wiki.ros.org/universal_robot.

[41] ROS ros_control packages, ROS Wiki. http://wiki.ros.org/ros_control.

[42] ROS gazebo_ros_pkgs package, ROS Wiki. http://wiki.ros.org/gazebo_ros_pkgs.

[43] ROS xacro package, ROS Wiki. http://wiki.ros.org/xacro.

[44] ROS roslaunch package, ROS Wiki. http://wiki.ros.org/roslaunch.

[45] Gear interface used in the ariac 2018. https://bitbucket.org/osrf/ariac/wiki/2018/competition_interface_documentation, 2018.

[46] HebiROS package, ROS Wiki. http://wiki.ros.org/hebiros_description, 2018.

[47] Ioan A. Sucan and Sachin Chitta. Moveit! http://moveit.ros.org/.

[48] Hardware information about the x-series actuators. http://docs.hebi.us/hardware.html#assembly-instructions, 2018.

[49] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. In Vittorio Scarano, Rosario De Chiara, and Ugo Erra, editors, *Eurographics Italian Chapter Conference*. The Eurographics Association, 2008.

[50] A. Csiszar, J. Eilers, and A. Verl. On solving the inverse kinematics problem using neural networks. In *2017 24th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, pages 1–6, Nov 2017.

[51] Ahmed R. J. Almusawi, Lale Dülger, and Sadettin Kapucu. A new artificial neural network approach in solving inverse kinematics of robotic arm (denso vp6242). 2016:10, 08 2016.