HAIVE System

Camille, Ulrich
ulrich.camille@etu.u-bordeaux.fr

Summary

Chapter 1

Introduction

My work at Molcure.inc aimed to contribute to the development of the HAIVE system.
The HAIVE system is an innovative extension of modern technology aimed at discovering new drugs. It consists of interconnected modules that perform their own function but work together. It requires minimal human intervention and can achieve high accuracy and reliability.

My work at Molcure had several goals :

— Show to HAIVE customers, how HAIVE works and give them a try with a simulation

— Using this simulation for debugging and ensure in case of hypothetical problems if it came from hardware and not from software.

— develop some feature integrated to the HAIVE system such as MyCobot.

all this time I've been enhancing my skills in R.O.S. (Robotic Operating System), Unity, Qt and robot development.

Chapter 2

Molcure

In this chapter, let's present Molcure company and my place in the company

2.1 Presentation

Molcure is a company founded in 2013 and carry project around high-speed, high-quality, high-variation antibody screening design system combining biology, automation and bioinformatics.

They are finding pharmaceutical client asking for more automation tools. that's where Molcure act.

Molcure is composed of 3 teams

— Biology team : they are at the center of Molcure project since they ask other teams their need and expectation for the system.

— AI team : Work a lot with biologist for analysing tool or sequencing.

— Robotics team : Close to the hardware, we are designing robot for all automation that need to be designed.

All through my contract with Molcure, I've been working with the robotics team. Mainly Yutaro Kyono, manager of the Robotics team and also co-founder of Molcure. I've also working with Johannes Prizbilla in charge of robots software. I've been also interact with other member such as Suzuki in charge of robots hardware.

Main communications tools are Notion and Slack. Slack is a really interesting

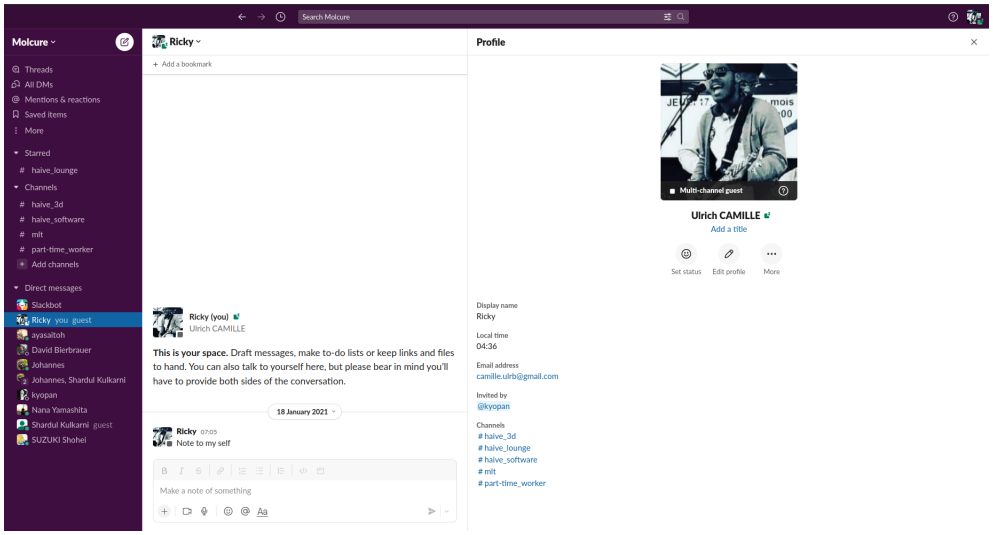

FIGURE 2.1: Slack platform

communication tool. It allows for multi-room communication, calls and meetings in a simple way and screen drawing tools. Slack also takes into account the time shift between me and other people. In fact Slack show the local time of any user and can turn off notifications when they are sleeping. I'm concerned by this case, but since Molcure works with different ethnicities (French, German, Indian, Japanese), sometimes people are working remotely from another country. This tool is great for scheduling meetings with people easily and without "surprise".

Beside, each week we have, a weekly meeting on Monday. Each team member presents their progress and may lead to discussions or other meetings scheduling. Notion is the support that each member use to report their progress then we are talk about on Slack.

Finally, we also had a monthly meeting called : lightning talk. The entire company organizes a meeting to learn about the progress of other teams, even if we don't work directly with them. For example, I've never heard about the IA team except at the lightning talks. At these meetings, the company organize some games and also always with the last word from the Molcure CEO.

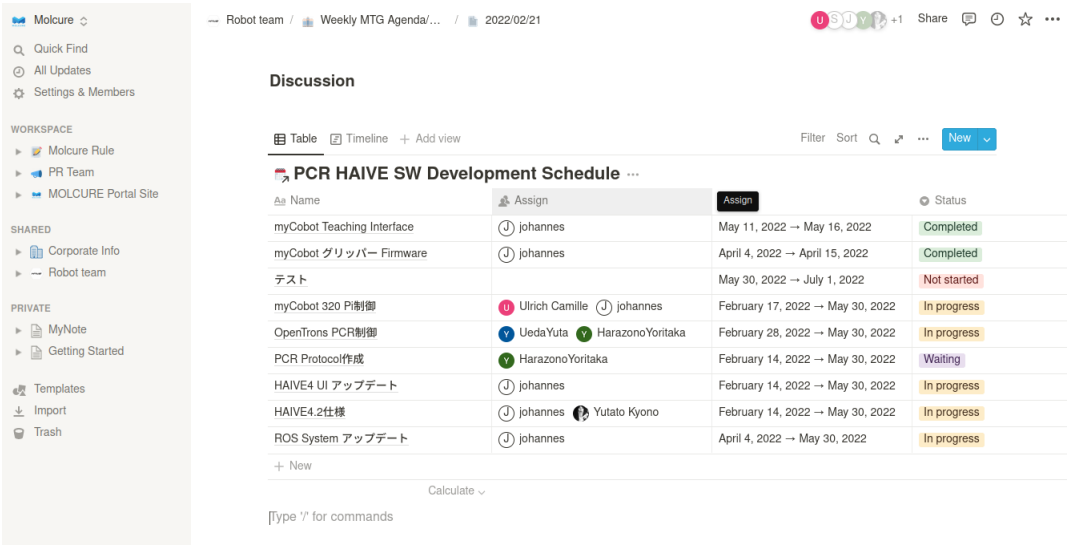

FIGURE 2.2: Notion platform

2.2 Remote working

I've been experiencing remote work at Molcure for more than a year. Of course, before starting, we'd to check my configuration and if it fits for Molcure project and software development. Since I use different software, many configurations are needed. As an example, Unity needed 2 different configurations :

— Linux 18, so that we can use both R.O.S. kinetic and Unity for Linux
— Window 10, for developping VR on unity. I've bought my self an Occulus Quest 2 and drivers are only for Windows 10 for now.

As a consequence, I configured my computer as I said. Later I also bought a new computer and installed Linux 20. Linux 20 supports R.O.S. Noetic, which is a great R.O.S. distribution because it uses Python3. R.O.S.2 can also be installed, but my last experience with R.O.S. showed me that we should rather wait for a Long Term Support distribution and more development. A virtual machine can also be used and can be useful if you just need simple operations from another OS version, but you won't be able to run Unity or large software in a virtual machine.

Chapter 3

HAIVE : Presentation

The HAIVE system is an autonomous robotic system and in the long term will be able to discover antibody alone using IA from protocols made itself and accelerate antibody discovery.

This system is composed of cells called HAIVE, and we can gather as many cells as needed for an experiment. Each of them has many tools that can reproduce scientific protocols based on the will of the biologists and the automation skills.

Test tubes are manipulated by the HAIVE system through containers that travel between HAIVEs, park slot to park slot. The heads of the HAIVEs contain test tubes with tips that can be changed, just as a machine would change tool according to the action he need. The other tips are held by containers.
Those containers can only move forward or backward and they can be aligned with a turntable held by each HAIVE in the center, providing access to all slots indexed with odd numbers (slot 1, 3, 5, 7, 9, 11).

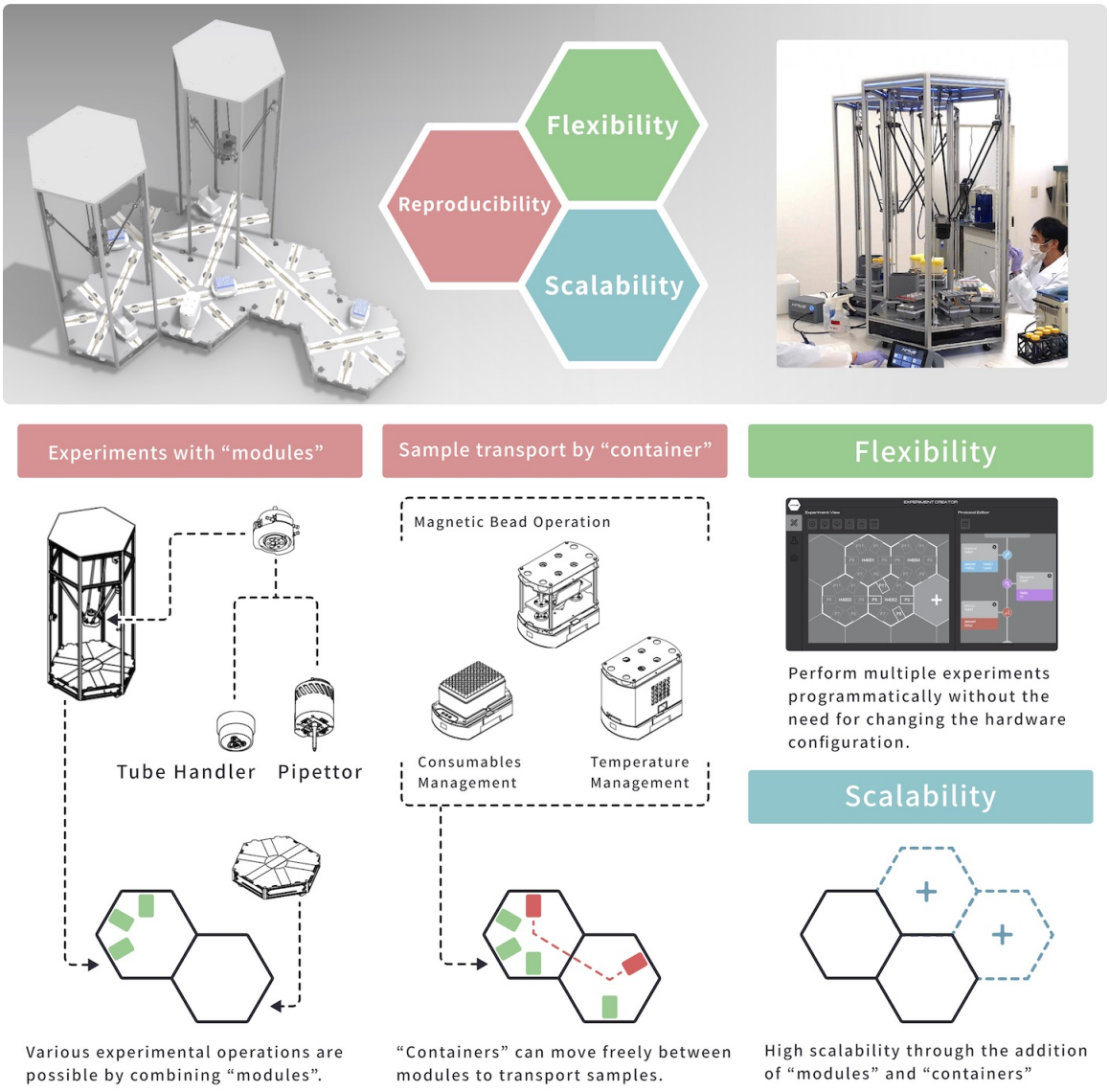

FIGURE 3.1: HAIVE overall presentation

3.1 Software structure

The HAIVE use 4 different softwares :

— Airtable, is a server that hold even protocols data but can also store their results. However airtable will be replace by another solution in the future.

— Qt, is software that allow to build on every platform an UI that gives order to the HAIVE
— R.O.S., is a middleware that receive high level command and give low level command directly to hardware.
— Unity, is the application used in order to build a simulation of the HAIVE hardware.

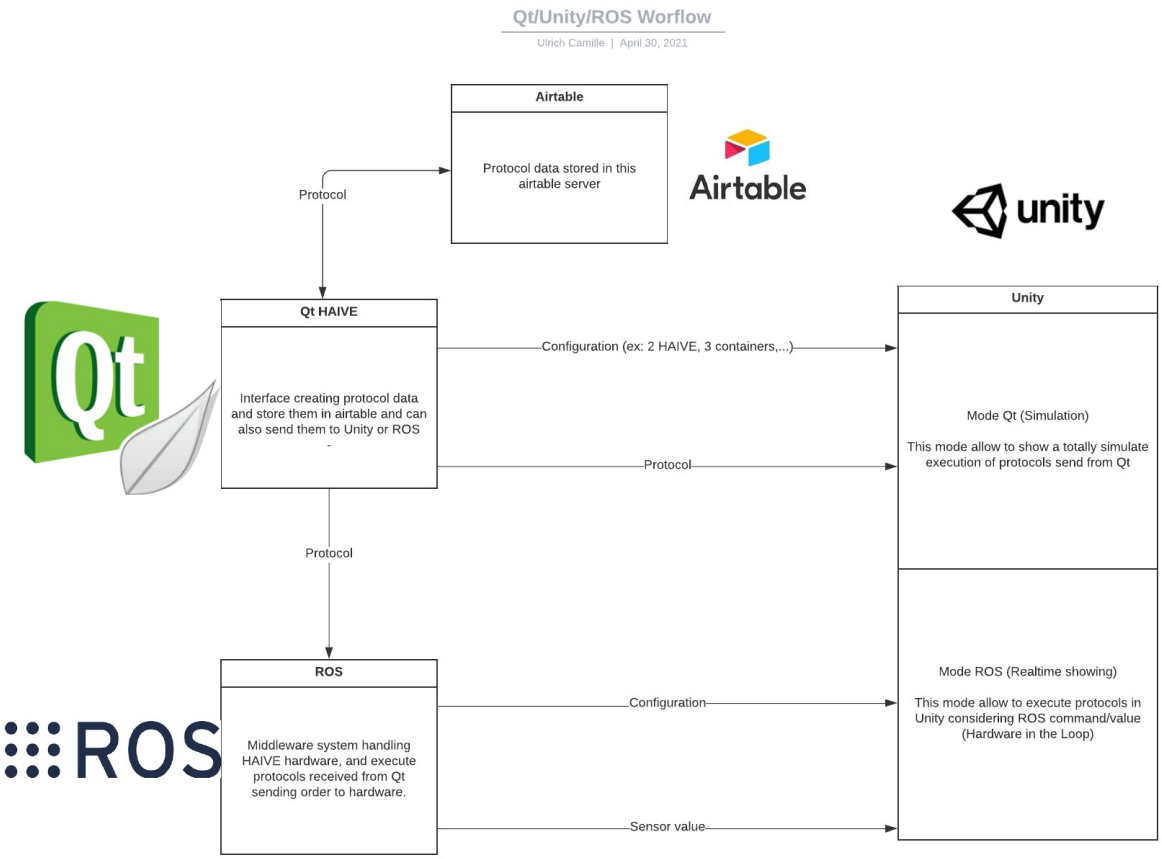

FIGURE 3.2: Software workflow

The Unity software can simulate either directly from the Qt user interface, receiving directly protocols, either unity is receiving HAIVE status values (actuator, configuration).

3.2 R.O.S.

R.O.S. (Robotics Operating System) rules the HAIVE software and is a middleware and thus a tool allowing to make the junction between hardware and software because this one presents the necessary tools to take care of the hardware part but also software (AI, interface, ...).

Robotic systems call for many skills such as mechanics, electrical engineering, control, computer vision, and many areas of computing such as real-time computing, parallelism, and networking. Advances in robotics often depend on technological locks solved in these scientific fields. A robotic system can therefore be complex and involve many people. In order to be efficient in the design of its robotic system, R.O.S. has practical aspects at several levels : simpler architecture design, debugging tools, multiple possible programming languages (c++, python, java, ...).

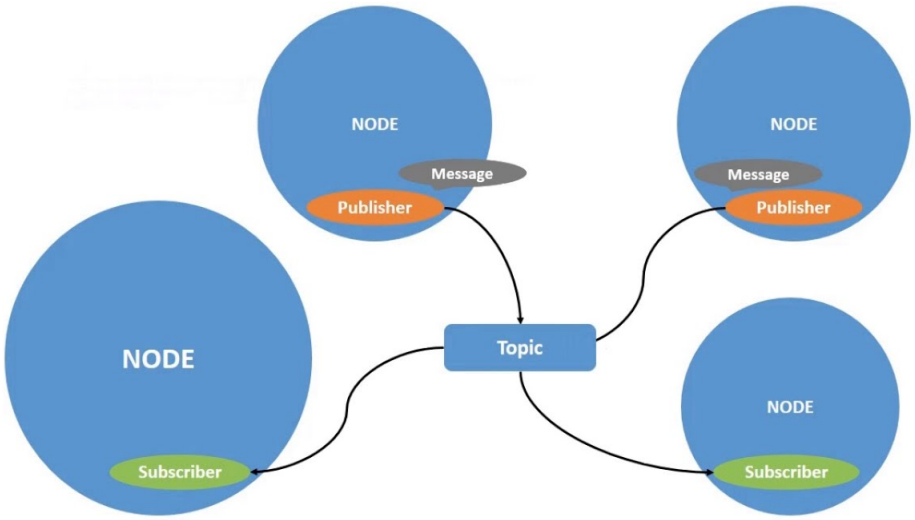

FIGURE 3.3: R.O.S. publishers and subscribers

Each node holds robot's functions. Usually there are separated in order to give each programmers a task. They can hold publisher that send data to a space call topic. Topic hold those serialized data, then any subscriber, whatever the code language chosen (Python, C, Java,...), can ask those data and use them.

For instance i was given the task with some supervision, to create a node that receive strings. By the way I've been initiated to the Layer 2 structure holding for example MyCobot commands (a HAIVE feature we will see later).

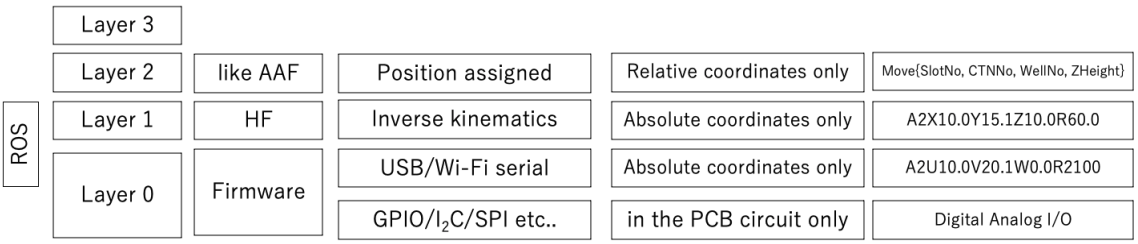

FIGURE 3.4: HAIVE layers command

3.3 Command

For now, Molcure is developing each layer of the HAIVE.
Each layer give order from low level to higher level.

Layers 0 and 1 (low layers) are managed by R.O.S. giving order that hardware can understand.

Layer 2 is managed by Qt allowing to create a entire protocol that a human can understand and easily design.

Finally, layer 3 or upper incoming layers are layers in development using IA to design the entire protocol.

These protocols have :

— P resources (HAIVE, containers, ...)
— N sequences
— $C_{P,N}$ command. the number of command for each resources and each sequences.

Every sequences, all resources execute their command. With m $\in \mathbb{N}$, once all commands on sequence m are executed, the sequence m+1 begin.

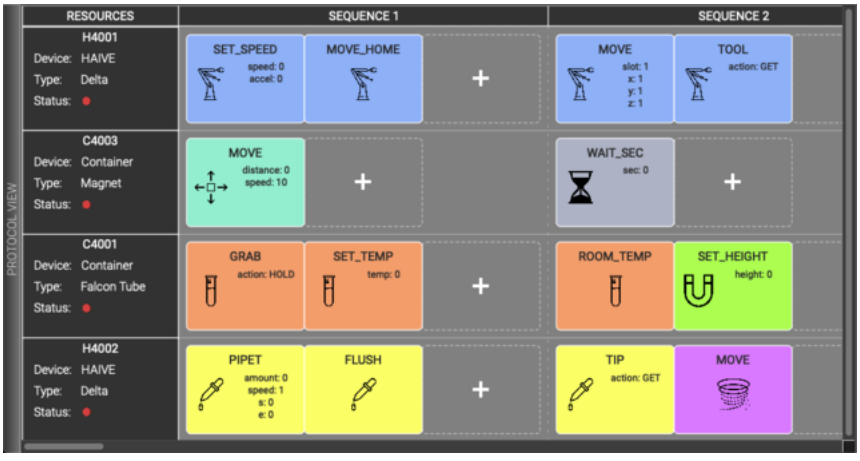

FIGURE 3.5: Protocol example for HAIVE

Chapter 4

HAIVE : Simulation

4.1 Unity

Unity is now a worldwide knowned software for game and simulation development.

Gazeboo or V-REP software could have been used because they have already many features for R.O.S. but Unity can be build on every platforms (Android,IOS ,Mac , Window, Linux) and can give higher rendering especially for VR purpose so that potential client can have a good and cool overview of the HAIVE system.
In order to realise this project, I've been working on Qt and R.O.S. integration in Unity.
Qt hold the HAIVE User Interface (UI) in order to give protocols or any command.
Robotic Operating System (R.O.S.) is a middleware that give an opportunity to give a functional structure of data and give order to the HAIVE hardware.

4.1.1 C.A.D.

The HAIVE simulation on Unity was developed directly by (Computer Aid Design) CAD of HAIVE. Two different HAIVE models were created, a high quality and a low quality. These two models are used depending on the frame rate requirement. The computer aims for 60 frames per second (FPS), but in the case of VR we need to aim for more than 90 (FPS), otherwise it is not smooth and people can get sick. Since Unity can import these CAD objects directly, we can easily update the HAIVE models, we just need to update the created physics.

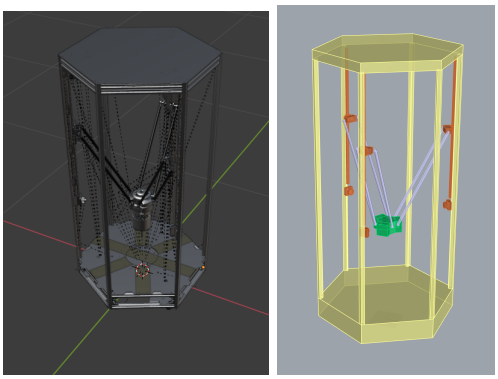

FIGURE 4.1: High and low quality HAIVE from C.A.D.

4.1.2 Physics

HAIVE headers are operated by 3 actuators that move 3 rods and all three are attached to the header. Considering the automation rules, we need as many degrees of freedom of the actuators as necessary. Here we only need 3 degrees of freedom because rotating the head makes no sense in the protocols. We only need X,Y,Z positioning, which we can achieve with 3 actuators.

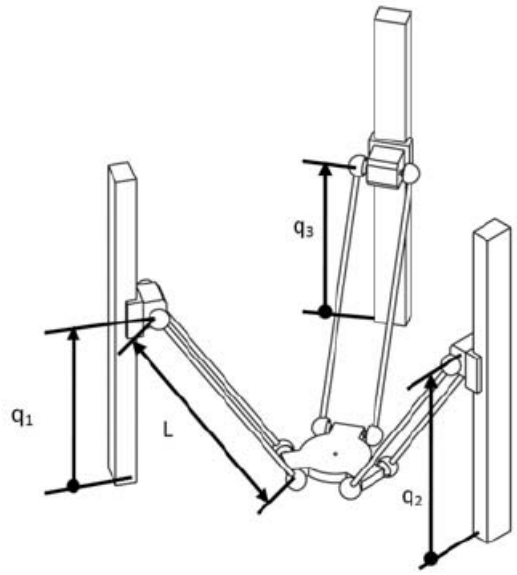

FIGURE 4.2: Delta scheme

Unity provides configurable joints and allows us to create any type of joint (spring, slider, ...), we just need to connect it to a RigidBody (that is, a Ga-

meObject that has the physics properties such as gravity, collisions, ... Once this connection is made to the correct RigidBody. Once this is done, the movement of these connected RigidBody will affect others through forces (and then create our inverse kinematics).

However, for several reasons, this solution was not enough for Unity. First, using physics is possible, but resource intensive.Second, we need to limit the operation.Indeed, if we are asked in a case where the head position is outside the operation limit the rod positions diverge and broke the Unity physics.

Although the operation limit is not so difficult to define. Let us call the minimum and maximum value of the actuators q_{min}, q_{max}.

The following lines give an idea of the operation space :

First, we define the intersection of 3 circles representing the range of each rod from each actuator :

$$\begin{cases} (x-x1)^2+(y-y1)^2+(z-z1)^2 = R1^2 \\ (x-x2)^2+(y-y2)^2+(z-z2)^2 = R2^2 \\ (x-x3)^2+(y-y3)^2+(z-z3)^2 = R3^2 \end{cases}$$

So from this, we can express the operation space by integration, we need to integratte all 3 spheres surfaces into a volume along z axis. Then the intersection of those 3 volumes is the operation space. So it can be written as this system :

$$\begin{cases} \int_{q_{min}}^{q_{max}} (x-x1)^2+(y-y1)^2+(z-z1)^2.dz1 = \int_{q_{min}}^{q_{max}} R1^2.dz1 \\ \int_{q_{min}}^{q_{max}} (x-x2)^2+(y-y2)^2+(z-z2)^2.dz2 = \int_{q_{min}}^{q_{max}} R2^2.dz2 \\ \int_{q_{min}}^{q_{max}} (x-x3)^2+(y-y3)^2+(z-z3)^2.dz3 = \int_{q_{min}}^{q_{max}} R3^2.dz3 \end{cases}$$

Then all tuples (x,y,z) satisfying this equation are inside the operation space

On these numbers 4.3, we can see the intersection of our 3 spheres in 2 and 3 dimensions. on 2 dimensions (z-projection), we can see the intersection, that is, the header operation boundary holding the entire HAIVE space (with the projection). When the z-question on the header is outside of $[q_{min}, q_{max}]$, the operation boundary changes and this "edge effect" can be observed in the 3-dimensional graph, and as we can see, it is more like an almond shape at the "edge". This shape must be integrated from q_{min} to q_{max} to get the whole operation space.

For both real header control and the Unity simulation, we need to retrieve from the header target position, actuators positions (q1, q2, q3), we can see at figure 4.2. Molcure has already developed the reverse kinematics and my tutor has implemented the reverse kinematics. This reverse kinematics is based on a model where the head is just the intersection between 3 spheres where the center of each sphere move vertically (on Z axis) according to q1,q2,q3 actuators values.

$$\begin{cases} (x-x1)^2+(y-y1)^2+(z-z1)^2 = R1^2 \\ (x-x2)^2+(y-y2)^2+(z-z2)^2 = R2^2 \\ (x-x3)^2+(y-y3)^2+(z-z3)^2 = R3^2 \end{cases}$$

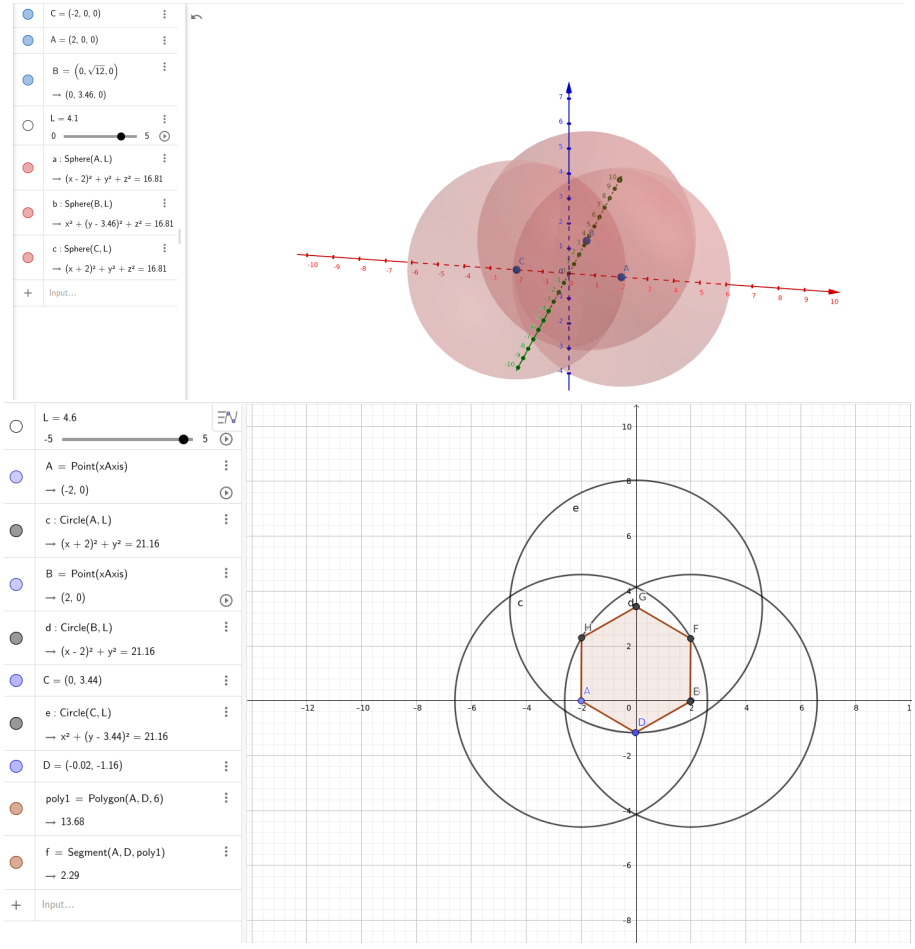

FIGURE 4.3: Operation space and intersection graph with Geogebra

From this model, we need to retrieve q1,q2,q3, which are actually respectively z1,z2,z3, and we already know x,y,z since it's our header position we will call x_h,y_h,z_h

$$\begin{cases} q1 = z_h - \sqrt{R1^2 - (x_h - x1)^2 - (y_h - y1)^2} \\ q2 = z_h - \sqrt{R2^2 - (x_h - x2)^2 - (y_h - y2)^2} \\ q3 = z_h - \sqrt{R3^2 - (x_h - x3)^2 - (y_h - y3)^2} \end{cases}$$

More than just the actuators position we need also the exact pose (position and rotation) of the object. A problem answered by reasoning with vectors. if the center of the object mesh cross the sphere center, then we just need to give the orientation of the V $= \overrightarrow{HS_{center}}$ where H is the header position (x,y,z) and HS_{center}, is the sphere center (xi, yi, qi), with i $\in [1, 2, 3]$

4.1.3 Container navigation

NavMesh

Containers move from slot to slot during a protocol experiment. However, layer 2 only gives a distance (how many slot he have to drive through) and turntable rotation while the layer 3 or upper plan to give order to containers such as Move to slot 7 on HAIVE4002. Layer 3 use layer 2 command to realise his command (ContainerMove, Turntable) but he need to take into account other container position or status since they can block their way.

Layer 3 or upper has not been computed yet but I tried to compute those layer command in Unity and the solution I brought was operating research and graph theory.

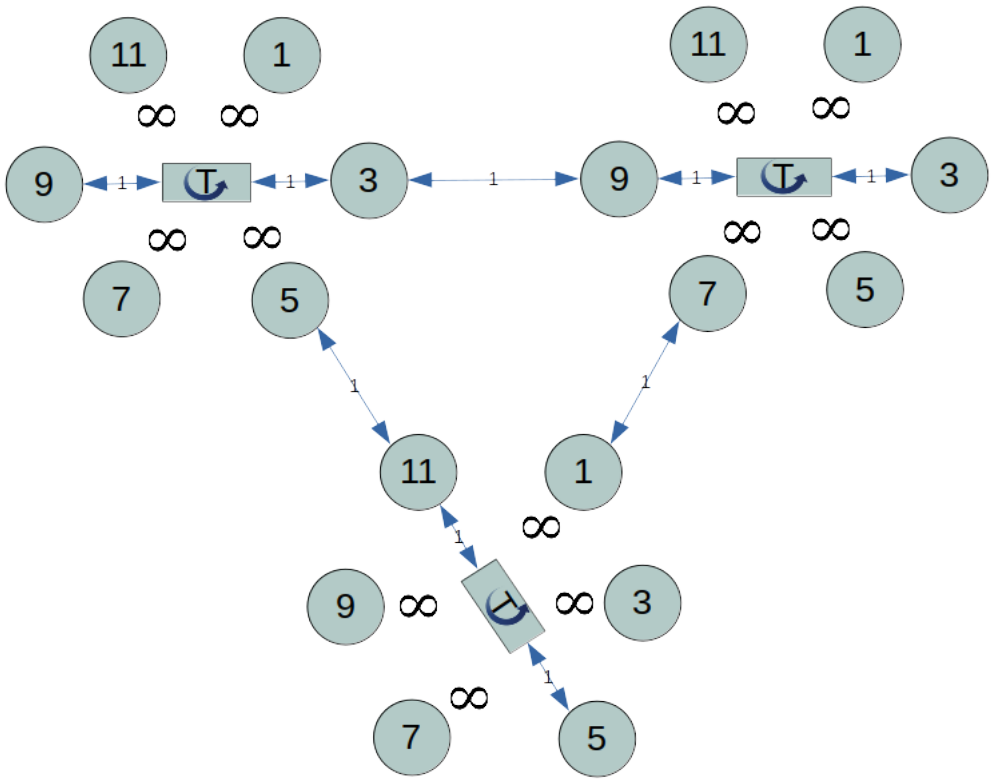

FIGURE 4.4: Graph node operating research

The HAIVE system can be easily modeled as a graph. Each parking slot and turntable between HAIVEs can be assimilated as a node connected with other parking slots.

They still remain one thing different and make this graph difficult : edges between nodes can change according to turntables position and current containers position. Indeed all knowned graph theory algorithm used "static" graph.I've tried to find some more adapted algorithm, and those articles gaved me some clues, however i didn't took the taking to dig deeply into this.

That's the reason why, I'd simplified the graph. we can see the turntable link

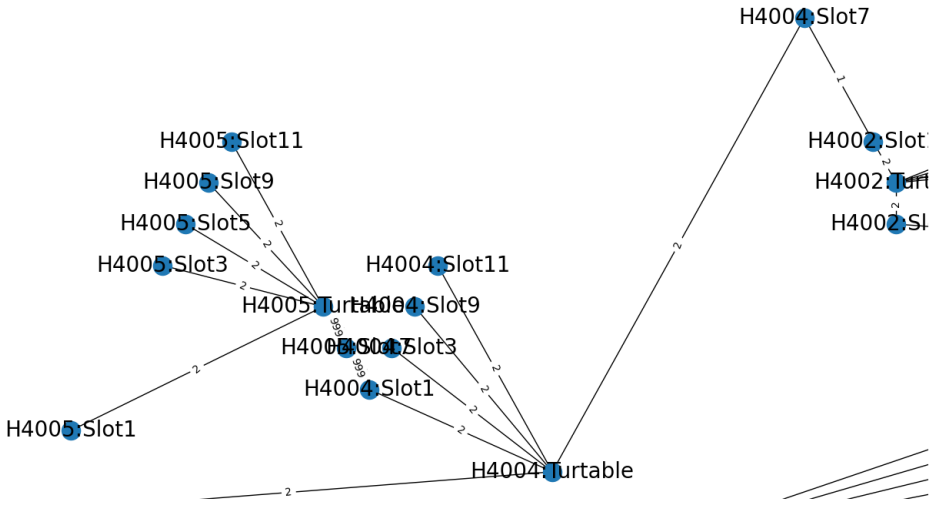

FIGURE 4.5: simplified HAIVE graph draw with networkx

to all other slots, making him the mediator between every HAIVE slot. Then this HAIVE can be linked to other HAIVE via slots.

So first,I've computed Dijkstra path algorithm considering every edges between turntable and parking slots with a weight of 2 (representing the two action to move between them, the turntable have to rotate and align with the container and then, the container move forward of 1 slot. As HAIVE parking slot are also link to other HAIVE since they are interconnected.parking slot to parking slot have a weight of 1. Finally is a container occupies a parking slot, then adjacent nodes are set with an infinite weight.This allow us to have the shortest path for each container.

Yet, we still have some issues because containers may "collide" and because some containers reach their destination, other containers can't execute their shortest path. During the shortest path algorithm, edges are changing value as

containers are moving and so the usual node labeling may not work. as we can see in the figure 4.5 H4005 slot 7 is linked to H4004 slot 1 and H4005 turntable. Their edge value is 999 (assimilate to infitity) because a container is at H4005 slot 7. Beyond edge changing value, the graph is different for every container point of view. In truth a container can't be blocked by himself, that's the reason why each this graph change according to each container point of view. I've tried to bring a solution with a custom algorithm that can ensure having the shortest path by taking at least into account the end position of containers despite this is not enough for a perfect path execution.

Below, the custom shortest path function I've computed. this algorithm aim to find a "global" shortest path for containers :

- We start with 1 container, find his shortest path,
- then we find the shortest path for the containers m+1 taking into account the end position of all previous containers
- Once we've find a path for each containers, we restart the process with a different container order. in fact we are trying all combination. @at the end we keep the couple of path (associated to a containers combination) with the shortest total length

Even though we don't take into account container position during the trip, since we know all path, if some of them have to cross, we just need to give the priority to the container with the higher priority, define by their position in the combination.

Johnson's algorithm runs in $O\big(V^2 \cdot \log(V) + |V| \cdot |E|\big)$ and since we are calling this function for each container combination, we will have $O\big((V^2 \cdot \log(V) + |V| \cdot |E|) * N * N_c!\big)$ complexity where N_c is the number of container, V the numbers of vertices and E the number of edges. This algorithm has a huge cost especially because of the $N_c!$ and can be improved.

For instance with a AMD Ryzen 5000, runtime execution are :

- 3 containers : 0.25s
- 4 containers : we have 25s
- 6 containers : we have 10 min
- 9 containers : undefined

These time record are huge and must be executed for each sequences that's why this solution is not possible to implement. Some simplifications could be using only this priority check for container group that need to do this. For example sometimes containers have no impact in other container path. We can maybe check whose containers have their path crossing. we can then reduced computation time, we can have 20 container as long as with their shortest path they cross no more than 4 containers path, then it may work more efficiently.

```
1     def custom_containers_path(self):
2
3         previous_container = []
4         self.containers_sim_position_dict = self.
    containers_position_dict.copy()
5         current_path= {}
6         res_length = inf*inf #inf
7         current_length = 0
```

```
        key_list = []
        res_path = []

        winner_combo = None
        for key, value in self.containers_position_dict.items():
            key_list.append(key)

        combo = [i for i in permutations(key_list,len(key_list))]
        for config in combo:
            self.containers_sim_position_dict = self.
    containers_position_dict.copy()
            for key in config:
                # Shortest path computation and updating the Graph
    according to containers position
                self.update_graph_Ci_turntable(config,self.
    containers_sim_position_dict.items(), previous_container)
                current_path[key] = self.shortest_paths_Ci(self.Ck[
    key])
                self.containers_sim_position_dict[key] = self.
    containers_target_dict[key]
                previous_container.append(key)

            for key in config:
                # lenght calculation
                for i in range(len(current_path[key][self.
    containers_position_dict[key]][self.containers_target_dict[key
    ]])-1):
                    node1 = current_path[key][self.
    containers_position_dict[key]][self.containers_target_dict[key
    ]][i]
                    node2 = current_path[key][self.
    containers_position_dict[key]][self.containers_target_dict[key
    ]][i+1]
                    current_length += self.Ck[key][node1][node2]["
    weight"]


            if current_length == res_length:
                res_path.append(current_path)
                winner_combo.append(config)

            if current_length < res_length:
                res_path = [current_path]
                res_length = current_length
                winner_combo = [config]

            current_length = 0
            previous_container = []

        for n in range(len(res_path)):
            print("___________PATH FOR CONFIG : " + str(
    winner_combo[n]) +"______________")
            for key, value in self.containers_sim_position_dict.
    items():
                print("custom path for "+ str(key)+ " is :")
                print(res_path[n][key][self.
    containers_position_dict[key]][self.containers_target_dict[key
```

```
    ]])
```

let's take an example with this following configuration :

We are starting with 5 HAIVE and 3 containers giving the graph in the figure 4.6

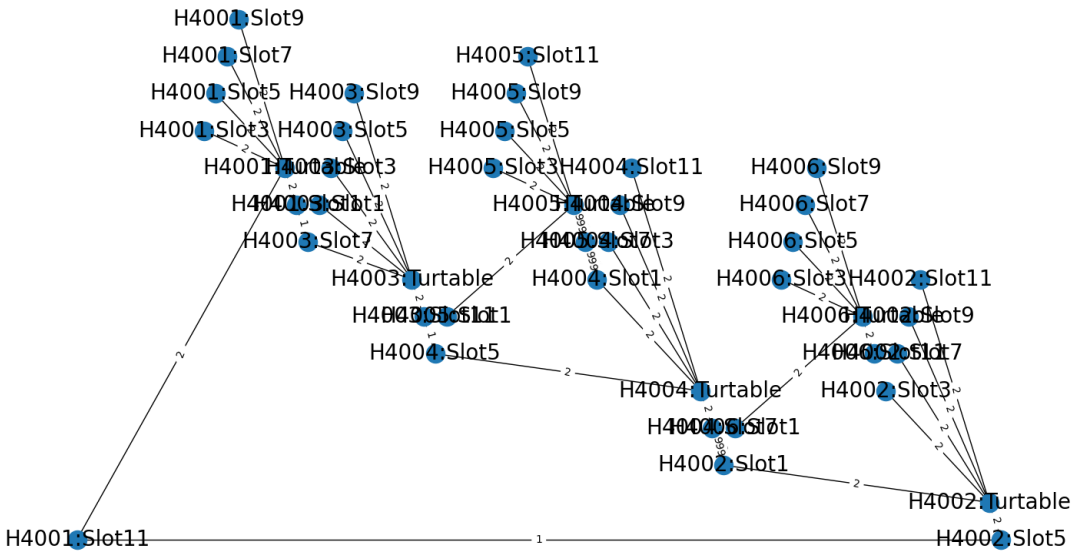

FIGURE 4.6: Networkx graph with HAIVE configuration

— HAIVE n°4001 and HAIVE n°4002 are linked
— HAIVE n°4002 and HAIVE n°4003 are linked
— HAIVE n°4001 and HAIVE n°4003 are linked
— HAIVE n°4002 and HAIVE n°4004 are linked
— HAIVE n°4004 and HAIVE n°4005 are linked
— C4001 : (Container n°4001) initial position at "H4004 :Slot1" (HAIVE N°4001, slot n°7)
— C4002 (Container n°4002) initial position at "H4001 :Slot3" (HAIVE n°4001, slot n°3)
— C4003 : (Container n°4003) initial position at "H4001 :Slot7" (HAIVE n°4001, slot n°1)

Then we will also define containers target :

— C4001 : (Container n°4001) target position at "H4001 :Slot1" (HAIVE N°4004, slot n°1)

— C4002 (Container n°4002) target position at "H4001 :Slot3" (HAIVE n°4002, slot n°3)
— C4003 : (Container n°4003) target position at "H4001 :Slot7" (HAIVE n°4005, slot n°7)

Then we execute our algorithm, giving the output in figure 4.7

```
__________PATH FOR CONFIG : ('C4002', 'C4003', 'C4001')_____________
custom path for C4001 is :
['H4001:Slot7', 'H4001:Turtable', 'H4001:Slot11', 'H4002:Slot5', 'H4002:Turtable', 'H4002:Slot1', 'H4004:Slot7', 'H4004:Turtable', 'H4004:Slot1']
custom path for C4002 is :
['H4002:Slot3', 'H4003:Slot9', 'H4003:Turtable', 'H4003:Slot3']
custom path for C4003 is :
['H4001:Slot9', 'H4001:Turtable', 'H4001:Slot11', 'H4002:Slot5', 'H4002:Turtable', 'H4002:Slot1', 'H4004:Slot7', 'H4004:Turtable', 'H4004:Slot1', 'H
4005:Slot7']
__________PATH FOR CONFIG : ('C4003', 'C4001', 'C4002')_____________
custom path for C4001 is :
['H4001:Slot7', 'H4001:Turtable', 'H4001:Slot11', 'H4002:Slot5', 'H4002:Turtable', 'H4002:Slot1', 'H4004:Slot7', 'H4004:Turtable', 'H4004:Slot1']
custom path for C4002 is :
['H4002:Slot3', 'H4003:Slot9', 'H4003:Turtable', 'H4003:Slot3']
custom path for C4003 is :
['H4001:Slot9', 'H4001:Turtable', 'H4001:Slot11', 'H4002:Slot5', 'H4002:Turtable', 'H4002:Slot1', 'H4004:Slot7', 'H4004:Turtable', 'H4004:Slot1', 'H
4005:Slot7']
__________PATH FOR CONFIG : ('C4003', 'C4002', 'C4001')_____________
custom path for C4001 is :
['H4001:Slot7', 'H4001:Turtable', 'H4001:Slot11', 'H4002:Slot5', 'H4002:Turtable', 'H4002:Slot1', 'H4004:Slot7', 'H4004:Turtable', 'H4004:Slot1']
custom path for C4002 is :
['H4002:Slot3', 'H4003:Slot9', 'H4003:Turtable', 'H4003:Slot3']
custom path for C4003 is :
['H4001:Slot9', 'H4001:Turtable', 'H4001:Slot11', 'H4002:Slot5', 'H4002:Turtable', 'H4002:Slot1', 'H4004:Slot7', 'H4004:Turtable', 'H4004:Slot1', 'H
4005:Slot7']
```

FIGURE 4.7: Command class object hierarchy python

This output tell us that 2 priority configurations allow to have the shortest path : ('C4003', 'C4002', 'C4001'), ('C4003', 'C4001','C4002') and ('C4002', 'C4003', 'C4001'), executing shortest path in this priority order ensure we do have the shortest path. Indeed in our graph HAIVE n°4004 and HAIVE n°4005 are linked by H4004 :Slot1 and H4005 :Slot7, if C4001 reach his target before C4003, C4003 won't be able to reach H4005 :Slot7. Therefore, all configuration with the C4003 executing before C4001 are the best. the C4002 have no impact on the shortest path computation.

4.2 Qt and Unity

Qt and Unity are two different processes and we need them to communicate data such as protocols and one of the more efficient method could be using Sockets. Sockets give the opportunity of communicating between different processes even though they have different languages or not they are not working on the same computer.
HAIVE UI has been developed in Python while Unity has been developed in C#. HAIVE UI need to send protocols data to Unity. Those protocols are just a list of command compute as a class object in a python list. Each command hold his own class object (ex : WaitSeconds,ContainerMove,...). Sockets send only binary data streams and once this is done, sockets can translate them as a type such as string, float, double, ...
However, class object contain many different information (ex : ContainerMove take in argument the resource concern, distance and speed). these data are held in class object properties.
We could create a string format in order to retrieve all the information correctly at the end. However I've learned something about JSON. JSON is a syntax for storing and exchanging data. JSON is text, written with JavaScript object notation. This format already gives us the opportunity to send entire class object with there properties through socket. JSON format convert all class object into dictionary where all data are stored. At the end , the endpoint socket can receive this class objects as long as it has already these class object defined somewhere, so that JSON package function can recognize this class object.

So, I first created these sockets and sent them to the Unity socket. But it could not work because there were different class objects and C could only store them in a list, so I was not able to call methods because "Object" has no command methods or properties. So I came up with the class object polymorphism to use. I created a child class object in UI named "JsonUniversalCmd". This class object inherits from all class objects in Python. Then I could store it in a list in Unity and then call any property or method I need. Below we have the JsonUniversalCmd definition :

So according to the "cmd" properties, we call the appropriate method and other properties. that also means that even though some command need for instance only 3 properties, in fact they hold all other properties with a "None" value.

In this figure, the arrow goes from the class object to its class object parent. So as we can see, all commands (WaitSeconds, ContainerMove, ...) have a common parent : CommandData. Then JsonUniversalCmd inherits from all command class objects. Actually it should be the other way around, i.e. JsonUniversalCmd should be a parent of every other command, but there were some conflicts in the code implementation. So this configuration allows to keep the upper method even though we have a JsonUniversalCmd object.

Once we are able to receive these commands from Qt to Unity, they are executed, as Figure 4.9 shows us. The protocols are controlled with threads.

First we have the sequence m, which creates P other threads (P is the number of resources). To start the next sequence, all resource threads must finish their task, so we use the .join() method. The Thread class provides the join() method, which allows a thread to wait until another thread finishes its execution. If t is a Thread object whose thread is currently executing, then t.join() ensures that t finishes before the next instruction is executed by the program. In this way, all instructions for each resources can be executed asynchronously and wait until they all finish. Within these P resource threads, the instructions are executed sequentially.

4.3 R.O.S. and Unity

R.O.S. and Unity is quite the same problem as Qt and Unity link, except that R.O.S. has its own serialized messages. Fortunately, since 2020 there's a library that allows communication between R.O.S. Unity via TCP communication and R.O.S. serialized messages, this library is constantly updated and I'm working on it too.

To communicate, Unity uses a server endpoint computed in R.O.S. with Python that subscribes or publishes all necessary topics and sends this serialized data to a server held by Unity in C. all custom messages must be referenced in Unity. The package provides the ability to create an R.O.S. project folder and then turn all custom messages into a class object that Unity can use and convert that serialized data (as we did for the json format).

All HAIVE sensor values are kept in namespaces so that each topic is unique, even if they have the same name. Since we do not have an odometer at the moment to get the position of the actuator. The node "random_motor_simulator" generates a random value every 2s.

4.4 Virtual Reality

A Virtual Reality (VR) version of the HAIVE is in progress. No pictures were taken because it was complicated to get them through the Occulus Quest 2. VR is a very good way to show and interest potential customers or investors. There are still many build dependencies and issues, VR development is not easy.

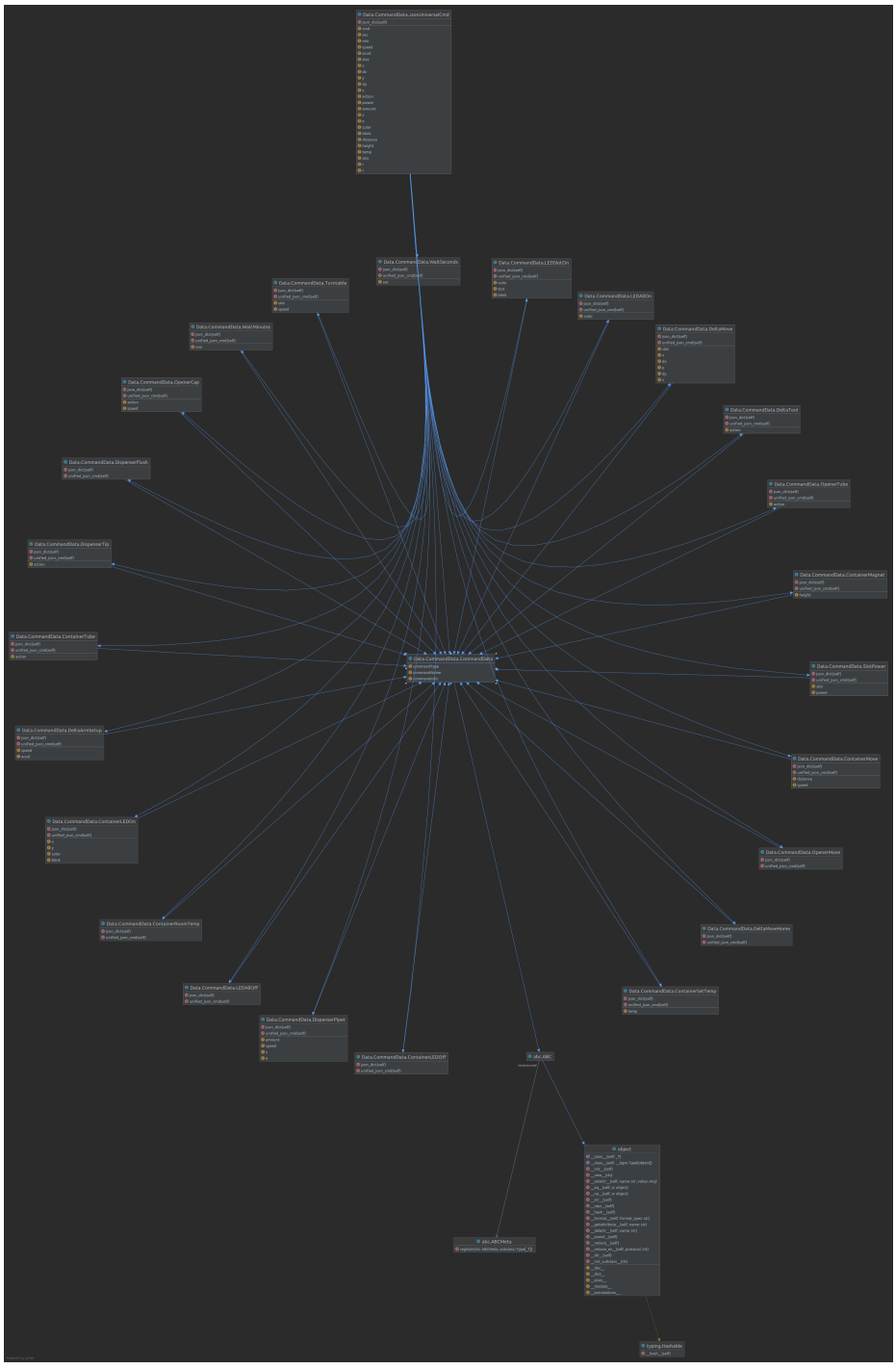

FIGURE 4.8: Command class object hierarchy python

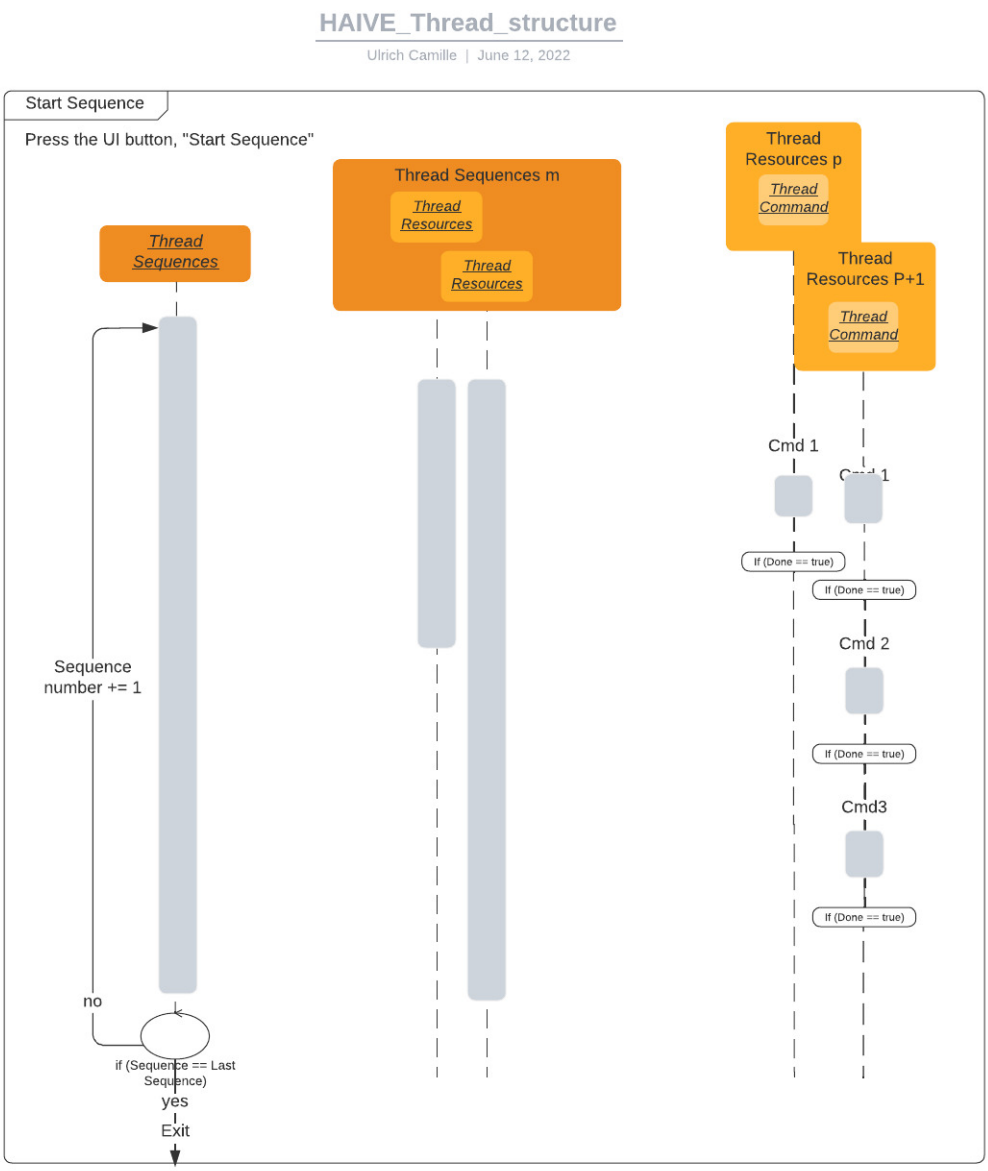

FIGURE 4.9: Thread structure

FIGURE 4.10: RQT graph R.O.S./Unity

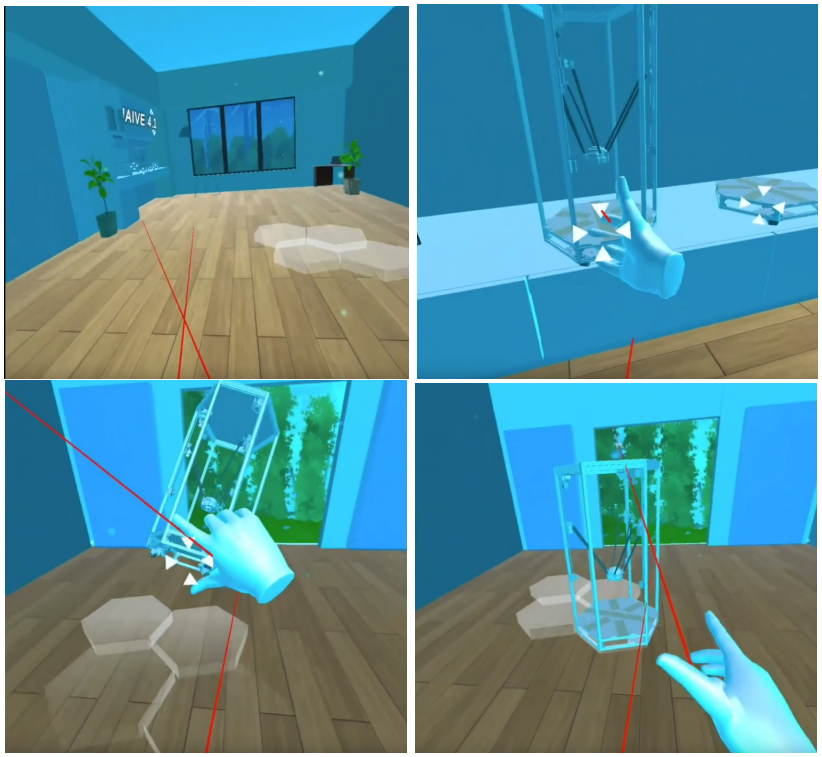

FIGURE 4.11: VR screenshots from Unity

Chapter 5

MyCobot

myCobot is the world's smallest and lightest six-axis collaborative robot, jointly produced by Elephant Robotics and M5STACK.

FIGURE 5.1: MyCobot from Elephant Robotics

MyCobot is a feature of the HAIVE system, and allow to move plate or lid from the system called Opentrons to any container.

5.1 Path planning

The path planning implementation is important for many reasons :

— As for many robot, we need to retrieve from the operation space target, the articular space target. the path planning will allow this

— Lids and plates are holding tube experiment and they need to be strictly horizontal to protect them. During path planning we will add conditions so that we can move plate plates keeping them horizontal. mycobot should also be able to avoid some obstacle during the path planning. Indeed, containers and other features can collide with mycobot during the motion, that why they must be integrated as a constraint in the path planning

These could have been computed ourselves and in fact they have been calculated by the team to have a working mycobot fast (first iteration). So what I've been working is the implementation of Moveit.

5.1.1 MoveIt

MoveIt ! is ROS's most advanced and flexible library for motion planning and manipulation tasks. It integrates state-of-the-art inverse kinematics solvers, path planning algorithms, and collision detection into a single, unified ROS interface.

The figure 5.2 illustrate the Moveit workflow. Moveit seems to be kind of black box with many dependencies and packages. In spite of this, the utilization for the user stay quite simple. the Worklow has only around 3 entries :

— URDF : Unified Robot Definition Format is an xml file that describe the geometry of a robot. URDF is a tree structure with one root link. The measuring units are meters and radians.

— Robot Sensor which are topic holding real state of the robot, or a fake state of the robot as a simulation.

— User space input from RVIZ plugin. There are numerous but in general User interface give target and select the option for the path planning.

these are the main parameters but many other can be changed or added, as for the path planning solver, or external plugin.

MyCobot has already some work with Moveit delivered with the MyCobot arm. However I had to dig more deeper into Moveit in order to reach our goal.

Beyond understanding Moveit at the beginning, my first iteration was create a simple path planning. from here, we had to dig into the entire Moveit workflow, and i find out "$MoveGroup$" holds most of method we need.

Second iteration, i had to implement the whole environment of MyCobot, that is to say, giving HAIVE, containers, Opentrons and gripper meshes. The gripper has been design by Molcure and adapted for their plate. For a precise evaluation/simulation we need to integrated the gripper to the URDF file .

```
1
```

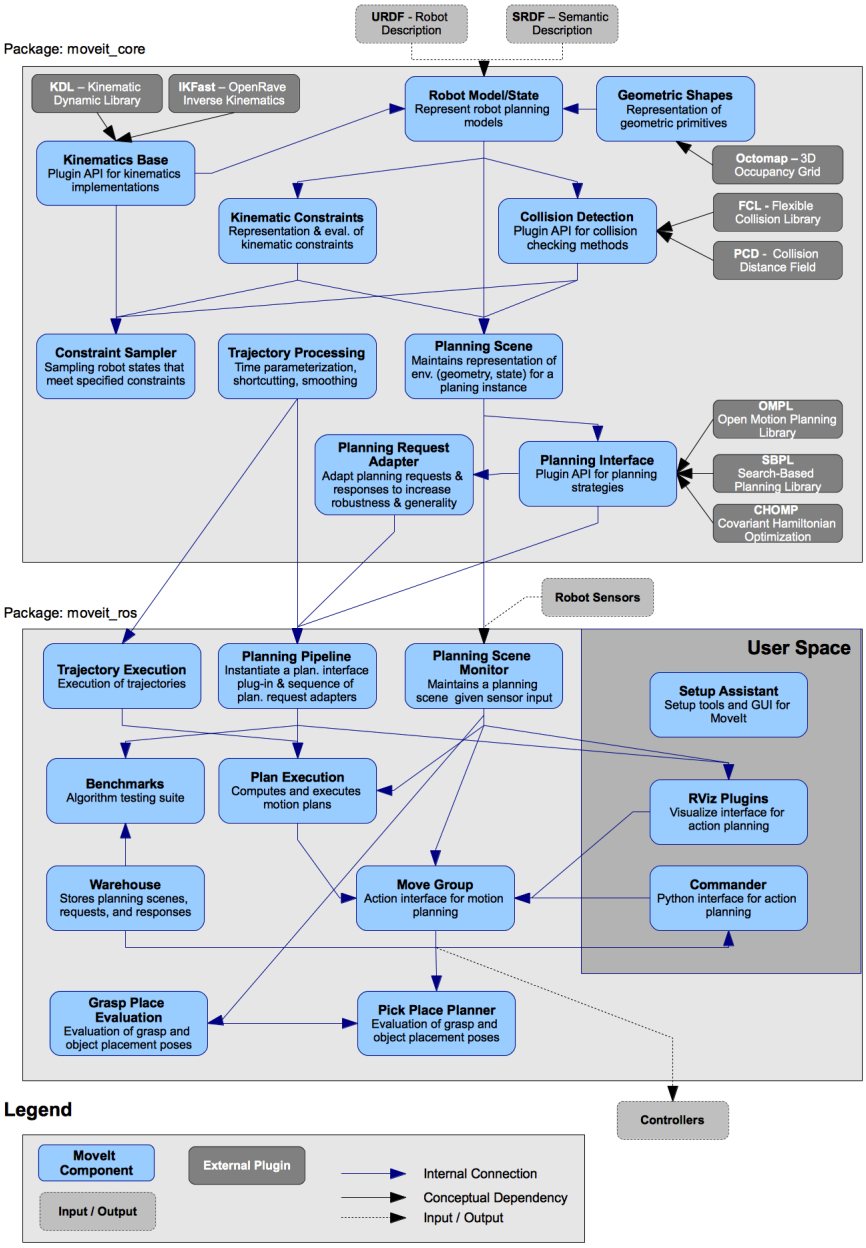

FIGURE 5.2: Moveit structure

```
<link name="gripper">
  <visual>
    <geometry>
```

```
5         <mesh filename="package://mycobot_description/urdf/320_urdf
    /PCR_HAIVE_3ddata/double_gripper.dae"/>
6       </geometry>
7       <origin xyz = "0 0 0 " rpy = " 0 -1.5708  0"/>
8     </visual>
9     <collision>
10      <origin xyz = "0 0 0.0415 " rpy = " 0 0 0"/>
11      <geometry>
12        <box size = "0.083 0.083 0.083" />
13      </geometry>
14    </collision>
```

These lines allow to give the visual information for RVIZ, as for a path planning, there's collision information and position.

During this iteration I'd to manipulate many file formats, and I understood the true difference between each of them. In truth, I switched between 2 or 3 formats (.dae, .blender, .stl). To clarify, dae keep object separated and keep hierarchy in mesh structure, while STL format will compile a file object define with vertices/triangles. STL seems to be a lighter object, but the tools from ROS only use .dae format, otherwise they can't be displayed in RVIZ, so we've to use both formats. To include these meshes in the path calculation, a simple move-group method is enough : $add\_mesh()$.

Since these meshes are only a prototype, they must first be validated before we move on to our third iteration, layout validation. With a hypothetical position of MyCobot in HAIVE and another hypothetical gripper, I defined reachability for each target.

First, I was able to use the Moveit and RVIZ user interfaces to give an idea of the reachability of each object, as shown in 5.3. MyCobot's arm turns red when it detects a self-collision, as we can see when MyCobot tries to access the nearest container. The second container appears to be accessible, as the second figure shows. Finally, container 3 is out of reach. We've done all also for the opentrons and it can be accessed.

Visual checks are quick and easy, but not accurate. Therefore, I implemented a configuration test that checks the reachability of each item entry. The figure 5.4 is the result of this test. The results of this test have been summarised in the figure. 5.5 After validation, the hardware team starts production.

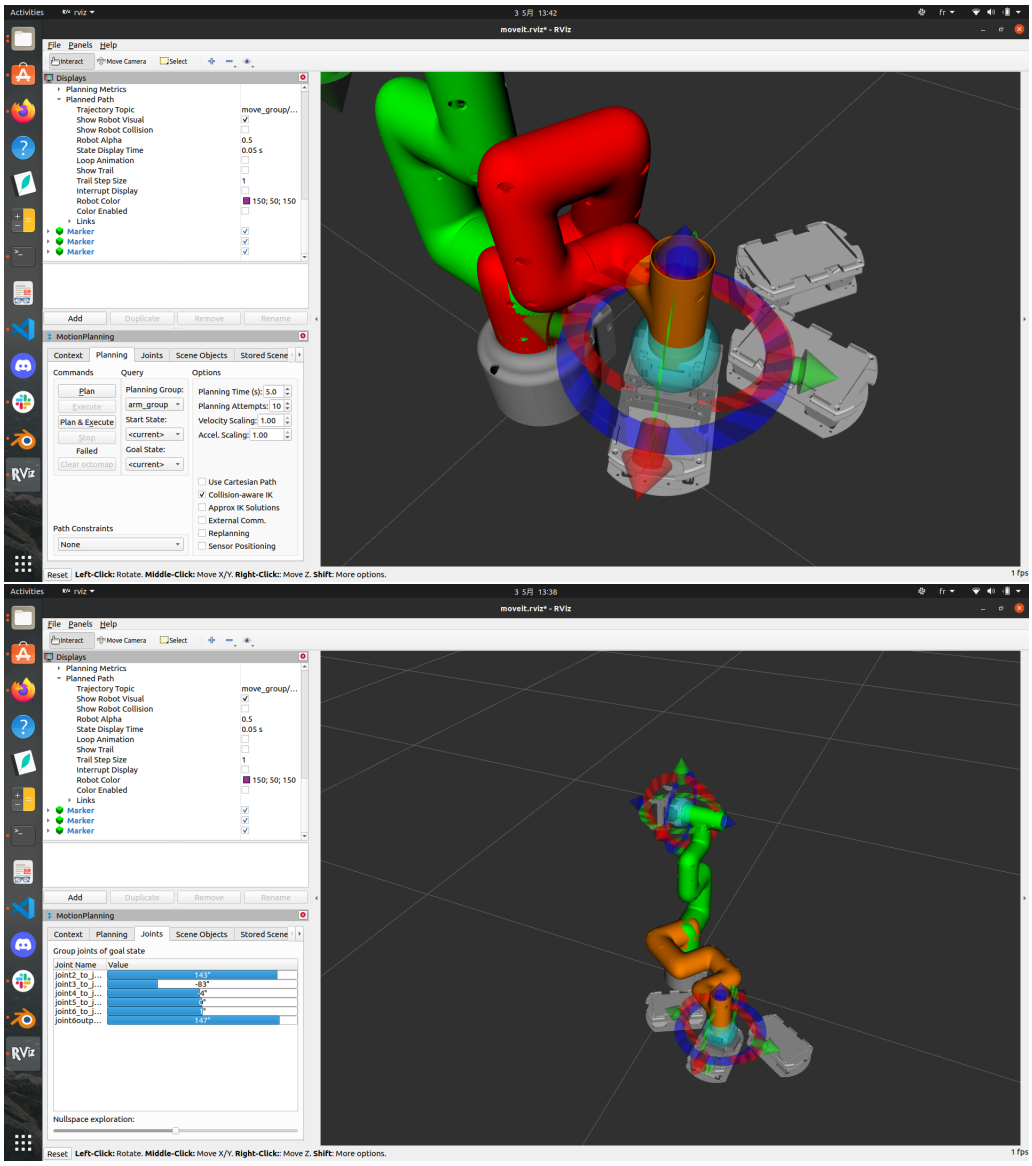

FIGURE 5.3: RVIZ and Moveit evaluation

5.1.2 User Interface

Then almost all the path planning tools are found. Still, no other user (except me) can write the Python script for the desired path planning. This is the reason why I was asked to create a UI. The figure 5.6 shows the UI concept I should aim for. Then the figure **??** is the current version of MyCobot UI. Designed

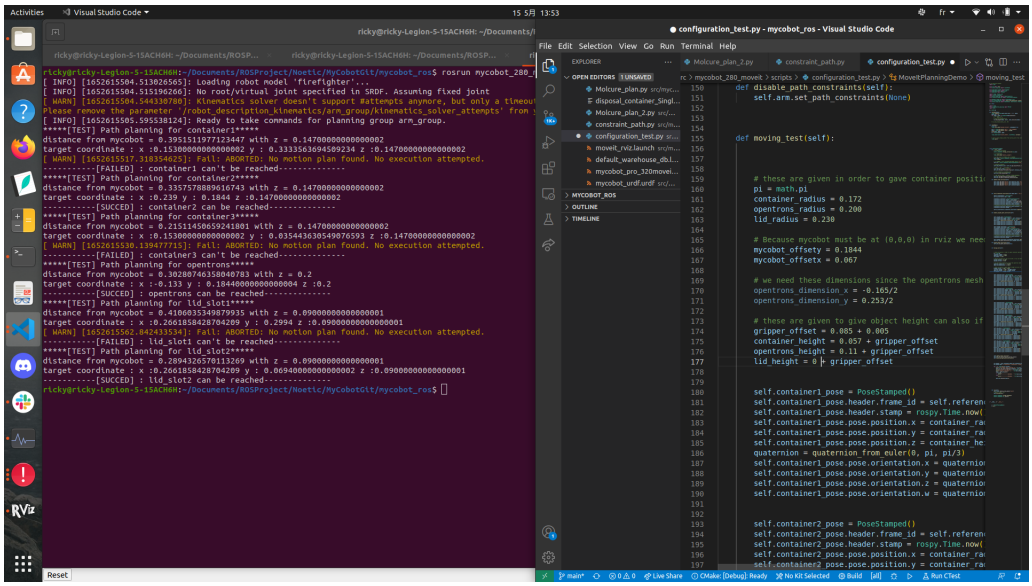

FIGURE 5.4: Moveit layout test

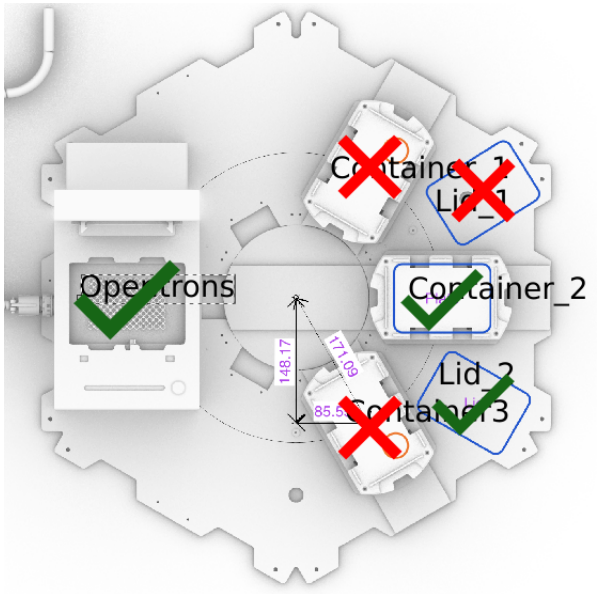

FIGURE 5.5: MyCobot User Interface

with Qt/python. Qt creator provides the ability to design a UI from scratch. Many other methods can be used with Python, and to give an example, my

tutor had a different method than me. he initially gave me a main file to start with, but I preferred to use my method. However, I was able to use what I was given and integrate it into my own UI. My method uses the Qt interface and outputs an xml file. This file is converted by a package called pyuic5. From this .py file, I use a new class object that inherits from the class object generated by pyuic5 and then adds all the required functions. This allows us to separate the frontend from the backend. We can update it separately, since the backend script is a different file than the one generated by pyuic5, we just import it into the main.py file.

The main problem was storing path planning data. This path data contains the joint position, velocity and acceleration. A JSON format was chosen to shop all this data.

In fact, my development helped validate the layout or raised some MyCobot issues, such as the lack of velocity and acceleration control, but my development was too long, so in the end another solution was found to perform these arm tasks. I couldn't meet the deadline.

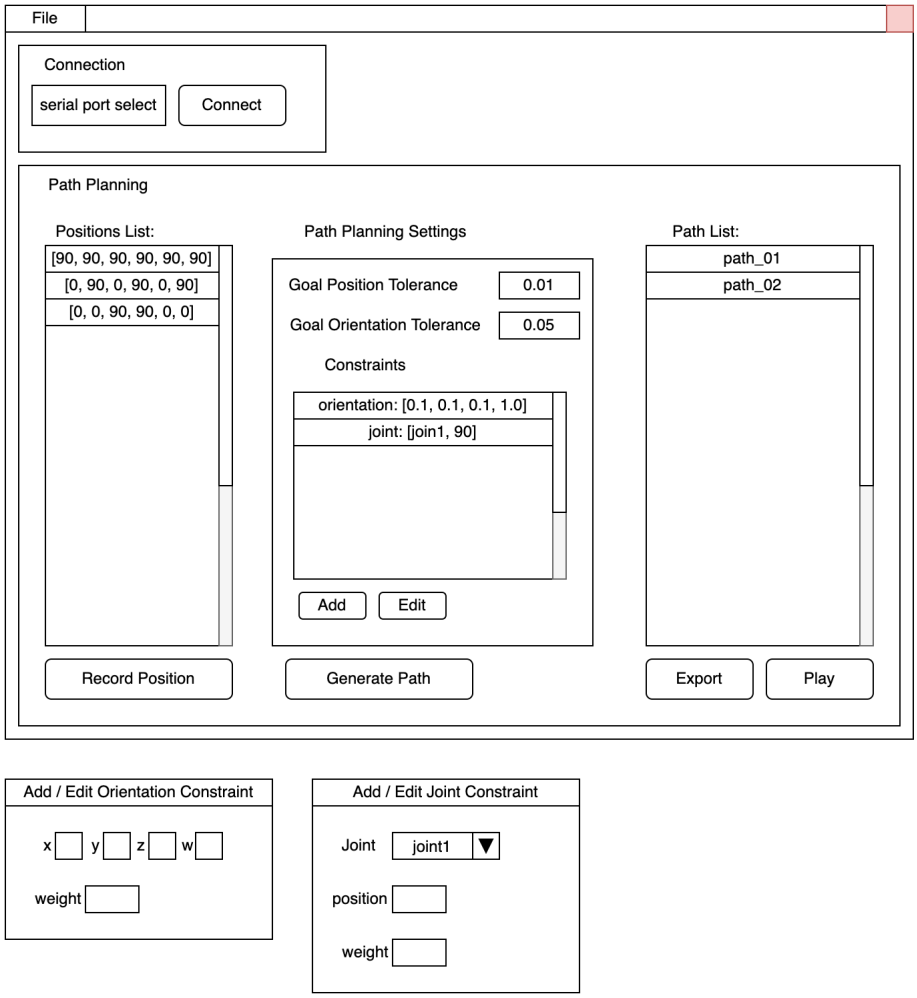

FIGURE 5.6: UI concept

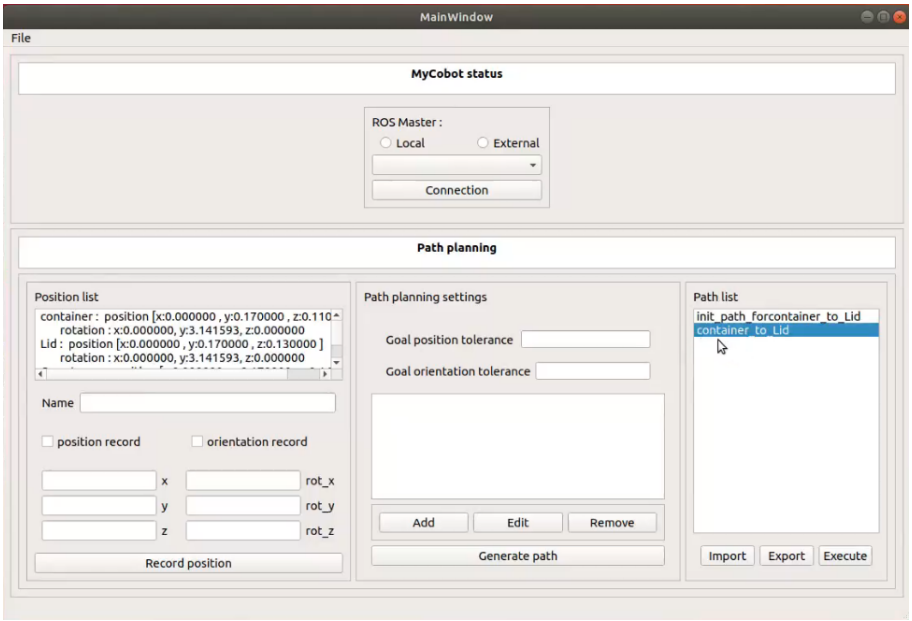

FIGURE 5.7: UI concept

Chapter 6

Conclusion

To conclude, I can say that my work at Molcure can be improved. The Unity software helps with some HAIVE showcases, communication is an important task to interest customers and investors. The VR version of Unity software is also very attractive, but also not ready yet.

Remote work for a long time is possible, but only part-time, otherwise I wouldn't stand it. Even though it's possible, I saw all the limits of remote work quite quickly. Not only because of the remote working conditions, but mainly because then I could work with the real MyCobot and HAIVE and save some time instead of working systematically with the simulation (the simulations are still important). Molcure has a really interesting company culture and I hope to work with them for a long time.

Bibliographie

[1] https ://moveit.ros.org/documentation/source-code-api/

[2] https ://brilliant.org/wiki/johnsons-algorithm/

[3] https ://www.lix.polytechnique.fr/ liberti/sppsurvey.pdf

[4] Gupta, M., Aggarwal, C.C., Han, J. (2011). Finding Top-k Shortest Path Distance Changes in an Evolutionary Network. In : , et al. Advances in Spatial and Temporal Databases. SSTD 2011. Lecture Notes in Computer Science, vol 6849. Springer, Berlin, Heidelberg. https ://doi.org/10.1007/978-3-642-22922-
$0_9 Mendes, N.D.etal. Estimatingattractorreachabilityinasynchronouslogicalmodels. Front. Physiol. 9, 116$
$Nilsen, "Near-OptimalDecrementalSSSPinDenseWeightedDigraphs, "2020IEEE61stAnnualSympos$
$1122, doi : 10.1109/FOCS46700.2020.00107.$

Annexe A

R.O.S.

```
import rospy
import roslibpy
from std_msgs.msg import String
from pymycobot.mycobot import MyCobot


PORT_MYCOBOT = '/dev/ttyAMA0'
BAUD_MYCOBOT =115200
IP_ROS_MASTER = '192.168.2.97'
PORT_ROS_MASTER = 9090


class MYCOBOT_NODE():
    def __init__(self):
        self.port = PORT_MYCOBOT
        self.mc = MyCobot(self.port, BAUD_MYCOBOT)

        print("Connection to ROS Master...")
        self.client = roslibpy.Ros(host=IP_ROS_MASTER, port=
    PORT_ROS_MASTER)
        self.client.run()
        print("Connected")

        # Publishers (roslibpy)
        self.pub_feedback = roslibpy.Topic(self.client, "/
    feedback_from_hardware", "haive_msgs/feedback_from_hardware")

        # Subscribers (roslibpy)
        self.sub_move = roslibpy.Topic(self.client, "mycobot/move",
     "std_msgs/String")
        self.sub_move.subscribe(self.move_callback)

    # TODO: These are just thoughts
    #  To implement move home we probably want to know where we
    currently are to execute the right path
    #  We should have a member like self.current_position
    #  self.current_position should be initialized in the
    constructor after MyCobot()
    #  It should be updated everytime in all move_* functions
```

```
35      def move_home(self):
36          print("!IMPLEMENT! Moving home")
37          self.mc.sync_send_angles([0, 0, 0, 0, 0, 0], 80)
38          self.pub_feedback.publish(roslibpy.Message({'
        device_identifier': "T4001", 'feedback_msg': 'DONE'}))
39
40      def move_container(self):
41          print("!IMPLEMENT! Moving to container")
42          self.pub_feedback.publish(roslibpy.Message({'
        device_identifier': "T4001", 'feedback_msg': 'DONE'}))
43
44      def move_lid(self):
45          print("!IMPLEMENT! Moving to Lid")
46          self.pub_feedback.publish(roslibpy.Message({'
        device_identifier': "T4001", 'feedback_msg': 'DONE'}))
47
48      def move_pcr(self):
49          print("!IMPLEMENT! Moving to PCR")
50          self.pub_feedback.publish(roslibpy.Message({'
        device_identifier': "T4001", 'feedback_msg': 'DONE'}))
51
52      def move_callback(self, msg):
53          cmd_dict = {
54              "HOME": self.move_home,
55              "CONTAINER": self.move_container,
56              "LID":self.move_lid,
57              "PCR":self.move_pcr,
58          }
59
60          cmd = msg['data']
61
62          try :
63              cmd_dict[cmd]()
64          except :
65              print("No function defined for this command")
66
67  def callback_node_shutdown():
68      print("\nNode %s is shutting down" % rospy.get_name())
69
70  if __name__ == '__main__':
71      rospy.init_node("MycobotNode")
72      Mc_node = MYCOBOT_NODE()
73      try:
74          print("MyCobot node ready!")
75          rospy.spin()
76      except rospy.ROSInterruptException:
77          pass
```

Annexe B

Unity code

B.1 JsonUniversal class definition

```
1    [JsonObject(MemberSerialization.OptIn)]
2    public class JsonUniversalCmd    {
3
4   public bool Done = false;
5
6        [JsonProperty("cmd")]
7        public string Cmd { get; set; }
8
9        [JsonProperty("sec")]
10       public int Sec { get; set; }
11
12       [JsonProperty("min")]
13       public int Min { get; set; }
14
15       [JsonProperty("speed")]
16       public int Speed { get; set; }
17
18       [JsonProperty("accel")]
19       public int Accel { get; set; }
20
21       [JsonProperty("slot")]
22       public int Slot { get; set; }
23
24       [JsonProperty("x")]
25       public int X { get; set; }
26
27       [JsonProperty("dx")]
28       public double Dx { get; set; }
29
30       [JsonProperty("y")]
31       public int Y { get; set; }
32
33       [JsonProperty("dy")]
34       public double Dy { get; set; }
35
36       [JsonProperty("z")]
```

```
37          public double Z { get; set; }
38
39          [JsonProperty("action")]
40          public string Action { get; set; }
41
42          [JsonProperty("power")]
43          public bool Power { get; set; }
44
45          [JsonProperty("amount")]
46          public int Amount { get; set; }
47
48          [JsonProperty("s")]
49          public int S { get; set; }
50
51          [JsonProperty("e")]
52          public int E { get; set; }
53
54          [JsonProperty("color")]
55          public string Color { get; set; }
56
57          [JsonProperty("distance")]
58          public int Distance { get; set; }
59
60          [JsonProperty("height")]
61          public int Height { get; set; }
62
63          [JsonProperty("temp")]
64          public int Temp { get; set; }
65
66          [JsonProperty("seq")]
67          public int Seq { get; set; }
68
69          [JsonProperty("r")]
70          public int R { get; set; }
71
72          [JsonProperty("c")]
73          public int C { get; set; }
```

B.2 Thread handle

```
1     public static void Main()
2     {
3         Thread myThread;
4         myThread = new Thread(new ThreadStart(ThreadSequence));
5         myThread.Start();
6     }
7     public static void ThreadSequence()
8     {
9   bool finish = false;
10            while (!finish)
11            {
12              Thread.Sleep(1000);
13    for (int seq = 0; seq < SeqArray.SeqSize ; seq++)
14      {
15      s = seq;
```

```
16      seqprint = seq + 1;
17      Thread CurrentResThread;
18      CurrentResThread = new Thread(ThreadRes);
19      Debug.Log("----------Sequence---------- : " seqprint);
20      CurrentResThread.Start();
21      CurrentResThread.Join();
22      }
23         finish = true;
24
25           }
26     }
27
28     public static void ThreadRes()
29    {
30      Thread.Sleep(100);
31  for (int res = 0; res < SeqArray.ResSize ; res++)
32      {
33      r = res;
34      Param myParam = new Param();
35      myParam.r = r;
36      Thread CurrentCmdThread = new Thread(new
    ParameterizedThreadStart(ThreadCmd));
37      resRunningThreads.Add(CurrentCmdThread);
38      Debug.Log("Thread res : " + r);
39      CurrentCmdThread.Start(myParam);
40
41      /*
42      Thread CurrentCmdThread;
43      CurrentCmdThread = new Thread(ThreadCmd);
44      resRunningThreads.Add(CurrentCmdThread);
45      CurrentCmdThread.Start();
46      */
47      }
48  foreach(var thread in resRunningThreads)
49  {
50      thread.Join();
51  }
52
53    resRunningThreads.Clear();
54
55    Debug.Log("finish all cmd in Sequence "+ seqprint);
56    Thread.Sleep(100);
57    }
58
59    public static void ThreadCmd(object data)
60    {
61    Param myParam = (Param)data;
62     Debug.Log(" in cmd : [r,s] = " +r + " , " + s);
63    if (SeqArray.Array3DCmd[myParam.r,s] != null)
64    {
65      foreach (int c in SeqArray.Array3DCmd[myParam.r,s])
66      {
67        Debug.Log("C = " + c);
68      Thread CMD = new Thread(()=>CmdList[c].execute_cmd());
69        //Debug.Log("Thread created ");
70    CMD.Start();
71        //Debug.Log("Thread start");
```

```
72      CMD.Join();
73          //Debug.Log("Thread joined");
74        }
75      }
76      else
77      {
78      Debug.Log("No cmd");
79      }
80
81  Thread.Sleep(100);
82      }
83
84
85  }
```

Annexe C

Python code

C.1 Graph theory

```
class HAIVE():
    def __init__(self,number):
        self.slots = ["H40" + str(number) + ":Slot1","H40" + str(number) + ":Slot3","H40" + str(number) + ":Slot5", "H40" + str(number) + ":Slot7", "H40" + str(number) + ":Slot9", "H40" + str(number) + ":Slot11"]
        self.slots_bis = ["H40" + str(number) + ":Slot1Bis","H40" + str(number) + ":Slot3Bis","H40" + str(number) + ":Slot5Bis", "H40" + str(number) + ":Slot7Bis", "H40" + str(number) + ":Slot9Bis", "H40" + str(number) + ":Slot11Bis"]
        # self.turntable_orientation = ["H40" + str(number) + ":Slot1", "H40" + str(number) + ":Slot7"]
        self.num = number


class GraphHAIVE():
    def __init__(self):
        self.Ck = {}
        self.C = nx.Graph()
        self.resources = [HAIVE("01"),HAIVE("02"),HAIVE("03"),HAIVE("04"),HAIVE("05")]
        self.external_link = [["H4001:Slot11", "H4002:Slot5"],["H4003:Slot11", "H4004:Slot5"],["H4001:Slot1","H4003:Slot7"],["H4002:Slot1","H4004:Slot7"]]
        self.containers_position_dict = {"C4001": "H4001:Slot1" ,"C4002" : "H4002:Slot1","C4003" : "H4002:Slot5"}
        self.containers_target_dict = {"C4001": "H4004:Slot5" ,"C4002" : "H4003:Slot1","C4003" : "H4002:Slot1"}
        self.canWait = False
        self.containers_shortest_path = {}

    ...

    def update_graph_Ci_turntable(self, config, container_pos, previous_container) :

```

```
24
25          for key_origin in config:
26
27              self.Ci = nx.Graph()
28
29              for a in self.external_link:
30                  w = 0
31                  for key, value in container_pos:
32                      if ((a[0] == value or a[1] == value) and
    key_origin != key):
33                          w = max(inf,w)
34                      else :
35                          w = max(1,w)
36                  self.Ci.add_edge(a[0],a[1],weight = w)
37
38              for res in self.resources:
39                  for i in res.slots:
40                      for key, value in container_pos:
41                          if (i == value and key in
    previous_container and key != key_origin):
42                              w = inf
43                          else :
44                              w = 2
45                          self.Ci.add_edge("H40" + str(res.num) + ":
    Turtable",i,weight = w)
46              self.Ck[key_origin] = self.Ci.copy()
47
48
49      def custom_containers_path(self):
50
51          previous_container = []
52          self.containers_sim_position_dict = self.
    containers_position_dict.copy()
53          current_path= {}
54          res_length = inf*inf #inf
55          current_length = 0
56          key_list = []
57          res_path = []
58
59          winner_combo = None
60          for key, value in self.containers_position_dict.items():
61              key_list.append(key)
62
63          combo = [i for i in permutations(key_list,len(key_list))]
64          for config in combo:
65              self.containers_sim_position_dict = self.
    containers_position_dict.copy()
66              for key in config:
67                  # Shortest path computation and updating the Graph
    according to containers position
68                  self.update_graph_Ci_turntable(config,self.
    containers_sim_position_dict.items(), previous_container)
69                  current_path[key] = self.shortest_paths_Ci(self.Ck[
    key])
70                  self.containers_sim_position_dict[key] = self.
    containers_target_dict[key]
71                  previous_container.append(key)
```

```
72
73              for key in config:
74                  # lenght calculation
75                  for i in range(len(current_path[key][self.
    containers_position_dict[key]][self.containers_target_dict[key
    ]])-1):
76                      node1 = current_path[key][self.
    containers_position_dict[key]][self.containers_target_dict[key
    ]][i]
77                      node2 = current_path[key][self.
    containers_position_dict[key]][self.containers_target_dict[key
    ]][i+1]
78                      current_length += self.Ck[key][node1][node2]["
    weight"]
79
80
81              if current_length == res_length:
82                  res_path.append(current_path)
83                  winner_combo.append(config)
84
85              if current_length < res_length:
86                  res_path = [current_path]
87                  res_length = current_length
88                  winner_combo = [config]
89
90              current_length = 0
91              previous_container = []
92
93          for n in range(len(res_path)):
94              print("___________PATH FOR CONFIG : " + str(
    winner_combo[n]) +"______________")
95              for key, value in self.containers_sim_position_dict.
    items():
96                  print("custom path for "+ str(key)+ " is :")
97                  print(res_path[n][key][self.
    containers_position_dict[key]][self.containers_target_dict[key
    ]])
```