Title:	Self-reconfiguration of a robotic workcell for the recycling of electronic waste
Acronym:	**ReconCycle**
Type of Action:	Research and Innovation Action
Grant Agreement No.:	871352
Starting Date:	01-01-2020
Ending Date:	31-03-2024

Deliverable Number:	D1.1
Deliverable Title:	ROS-based software architecture for disassembly cells
Type:	Report
Dissemination Level:	Public
Authors:	Timotej Gašpar, Rok Pahič, Mihael Simonič, Primož Radanovič, Sebastian Ruiz, Riccardo Persichini, Kübra Karacan, Matej Štefanič, and Aleš Ude
Contributing Partners:	JSI, all

Estimated Date of Delivery to the EC:	31-12-2020
Actual Date of Delivery to the EC:	08-03-2021

Contents

1 Executive summary

In this deliverable we describe the software architecture designed for ReconCycle. The system was developed to facilitate the implementation of automated disassembly solutions as proposed in the project. It is based on ROS (Robot Operating System), which enforces the modularity of the overall system. The deliverable describes the backbone architecture, the most important modules and services, and the interplay between different components of the system. Interfaces for the programming of robot movements by kinesthetic teaching and high-level task programming based on FlexBE are also described. The provided in-depth description of the software architecture and modules is important for all partners who need to integrate their solutions into the ReconCycle demonstrator, which is under development at JSI.

2 Software architecture based on ROS

The purpose of the software architecture is to ensure connectivity between the workcell modules in the context of data flow. In the developed cell, each module is connected to the same network in order to broadcast its data and receive information and instructions about what action to perform at any given time. An overview of the developed software system architecture is shown in Figure 1. Its constituent components are described in more detail throughout this deliverable.

The Robot Operating System (ROS) [13, 14] provides a suitable framework for developing various software components that need to share data over the shared network. The various tools and features that are available within ROS contribute to realizing the pursued software reconfigurability of the cell [6]. In our case, software reconfigurability means that it is possible to expand the cell's functionalities without disrupting the existing software architecture. New software components can be developed without the need to reprogram any of the existing ROS nodes (the definition of ROS nodes is provided below). This also eases the development and integration of new hardware components with their own ROS nodes. A requirement for the ROS system to function properly is that roscore runs on one of the computers in the network (denoted as ROS Master Computer in Fig. 1). Of the many features and tools provided within ROS, we use the following ones to achieve a high degree of software modularity in our system:

- *nodes* – any program (written in any programming language) that has connectivity to the ROS network and can therefore access to and publish data across it (i.e low-level hardware drivers, high-level state machines, trajectory generation, etc.)
- *topics* – a publish/subscribe table advertized by each ROS node that defines the data that can be provided by the said node, e. g. robot joint states, screwdriver torques, force-torque sensor data, etc.
- *messages* – a predefined structure to encapsulate data to be transferred across the ROS network for other nodes to read, e. g. robot joint states are written into *sensor_ msgs/ JointStates*, which is a predefined standard ROS message structure that can be sent across the ROS network.
- *parameter server* – used to store various static configuration parameters, e.g. controller gains, camera exposure parameters, kinematic models, etc.
- *services* – a request/response based Remote Procedure Call (RPC) interface that a node can expose in order to trigger short-running tasks from within the ROS network that do not require preemption or monitoring, e. g. visual quality control, pneumatic gripper actuation, tool exchange system lock/unlock, gravity compensation mode toggle, etc.
- *action servers* – similarly to *services*, a request/response RPC exposed by a node, however are used to trigger long-running preemptable tasks from within the ROS network that provide feedback throughout their execution, e. g. robot movement tasks, servo gripper grasping tasks, flexible fixture reconfiguration, etc.

Apart from ensuring software reconfigurability, the proposed architecture also allows us to control and monitor all the different modules in the workcell as well as the workcell as a whole. The developed system is designed in such a way that each module connects directly to the ROS network. This way we ensure that the data is structured and parsable by all of the software components within the developed system.

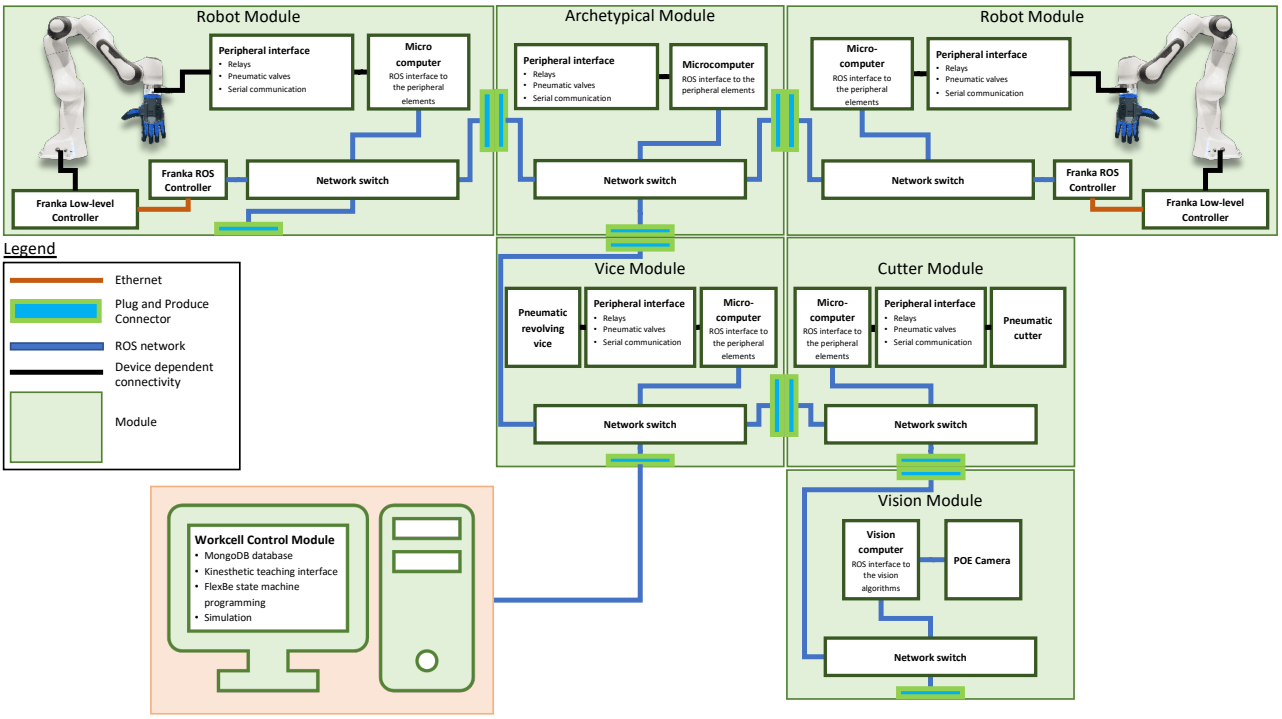


Figure 1: The ReconCycle software architecture.

An important feature of the proposed ROS-based software architecture is that we can program and exchange information between heterogeneous hardware modules within a single software architecture. Once the developer integrates a new module into ROS, the workcell programmer needs to know only which functionalities the new module exposes to ROS. No special knowledge about hardware-specific software is needed to start programming new workcell applications.

2.1 Integration through Docker containers

Although ROS provides a good framework for the development of robot cells, setting it up on a single computer still takes some effort and time. Our system is composed of several modules, each with their own computer. Setting up ROS and maintaining all of them would be very time-consuming. Moreover, the transfer of ROS code from one machine to another can be rather difficult. These difficulties arise if the developers do not properly define all the external resources (dependencies) that their code depends on. This is especially common in projects like this one, where different consortium members cooperate on software solution that should run on the same system. To solve these two issues, we decided to base our development and overall system on Docker containers [3].

A Docker container is an isolated environment that is build from a `Dockerfile`. In this file, we specify which Linux distribution the container is based on and what types of dependencies should be installed. The main advantage of this approach is that unlike virtual machines, Docker containers do not emulate the host's hardware but share it. This in turn means that, compared to a virtual machine, a Docker container uses fewer resources. Additionally, once the `Dockerfile` has been written, the image that is built from it will be the same regardless of

the platform it runs on. In terms of deploying ROS software on different modules, this means that the developer designs the code in such a way that it runs within the Docker container and thus removes the commonly encountered problem of unmet dependencies when transferring the code. An example of quick deployment of software by using Docker containers is the ReconCycle Raspberry Pi Docker image [15]. The details of how the Raspberry Pi integrates into the ReconCycle cell are provided in Section 3.3.

In terms of network connectivity, Docker containers can communicate between each other just like any other programs. This means that different software components running in different Docker containers can exchange data seamlessly. Therefore, using this technology does not hinder the overall ROS software architecture but it simplifies the set up of modules and the transfer of code.

2.2 Persistent data storage with MongoDB

All ROS nodes within the ROS network can access the data on the network by either subscribing to *topics* or by reading from the *parameter server*. However, as described at the beginning of this section, these two channels are meant to either provide data of the current state or some general parameters that can be used. To carry out a disassembly task, the robot has to move through various configurations in its workspace using motion generators described in Section 4.3. The data required by these motion generators need to be stored as persistent data and read during the disassembly process. It is also required that we are able to modify these data as the need arises, e.g. when the final pose of the robot at one step of the disassembly process changes.

To meet all these requirements, we decided to store the motion configuration data in the MongoDB database. We integrated this database in our ROS software architecture by using the already existing `mongodb_store` ROS package [11]. In our setup, the MongoDB database runs on the ROS master computer. All the data are saved as named entries into the MongoDB database. For point-to-point movements, the initial and final robot configurations are saved, whereas complex trajectories are saved as parameters of dynamic movement primitives (DMPs). It then becomes possible to define a high-level disassembly sequence that reads these named entries (robot configurations or DMPs) from the database and moves the robot accordingly. The high-level disassembly sequences are programmed using FlexBE (see Section 5). The poses and trajectories are saved in the database as ROS messages corresponding to each type of movement. Having these trajectories saved as named entries enables quick reconfiguration in terms of changing robot motion. It is sufficient to overwrite the entry in the database with a modified motion to update the disassembly sequence without changing the high-level disassembly sequence program.

3 Plug-and-Produce connectivity

When designing the ReconCycle modules we strove towards high modularity in order to support quick and efficient reconfigurability of the cell. Modularity needs to be supported both in terms of hardware and software. On the software side, the ROS-based architecture enforces modularity. To achieve modularity also on the hardware side, we designed each module as a self-contained unit that is equipped with all the resources it requires to fulfil its function. While this design does increase the overall size of the layout, it allows us to quickly add, remove or

Figure 2: Computer render and photograph of an archetypical module.

exchange the modules of the cell and therefore change its functionality. This approach enables an efficient preparation and adaptation of the cell for the disassembly of different electronic products.

3.1 ReconCycle archetypical module

Adding or removing modules and therefore reconfiguring the cell should not require the designer to dedicate a lot of time and attention to the software or hardware design and connectivity. Our cell is therefore made out of modules that are built following the same archetypical design:: a steel frame that provides rigidity, with aluminium work surface that allows for easy mounting of module-specific equipment. The frame is mounted on castors to make module transportation an easy task. Within the frame there is a basic electric wiring which distributes power to the module's electronics like network switches, low voltage DC power supplies, etc. There is also a network wiring that connects the devices contained within the module (computers, cameras, controllers, etc.) and exposes them to the rest of the cell's network through the "Plug & Produce" (PnP) connectors (see Section 3.2), which have been newly developed in the project. According to the requirements of each individual module, additional equipment (computers, cameras, controllers, etc.) can be added to achieve the module's desired functionality.

To achieve the desired software properties such as connectivity, each module within the cell can connect to the ROS network. Thus, all modules are equipped with sufficient computational hardware to run ROS nodes, thereby exposing each module's data and functionalities to the cell's ROS environment. This way the modules can be controlled by the top-level task scheduling software as soon as they are connected to the cell. Some modules may require more than just network connectivity in order to function properly, e. g. pneumatic air or electric power. To provide these capabilities, modules are connected to each other using the above-mentioned PnP connectors.

The computational hardware that every module in the cell is equipped with is a Raspberry

Figure 3: Electronics inside an archetypical module.

Pi 4 micro-computer. We mounted the so-called "PoE Hat" on each Raspberry Pi. This allows the Raspberry Pi to be powered via PoE, thus reducing the amount of necessary power supplies. There are other devices that can make use of the PoE connectivity (e.g. cameras). To connect them to the network and at the same time provide them with power, we installed a PoE-enabled network switch on each of the modules. These components are shown in Figure 3.

3.2 Plug-and-Produce (PnP) connector

The term "plug-and-play" carries an expectation of ease of use and reliable, full proof operation. A plug-and-play product, as its name suggests, can simply be connected and turned on – and it works. The practical extension of plug-and-play products, when applied to industrial automation, has given way to a new term: *Plug-and-Produce*. The Plug-and-Produce approach is the foundation of our standardized reconfigurable modular platform. It enables fast deployment of robotic cells, development of compatible specialized tools by third parties, and extremely fast, cheap and reliable work changeover. These properties are essential for a small batch production of highly variable products. This is even more true when it comes to recycling companies where products to be recycled constantly change and bring along different requirements for disassembly.

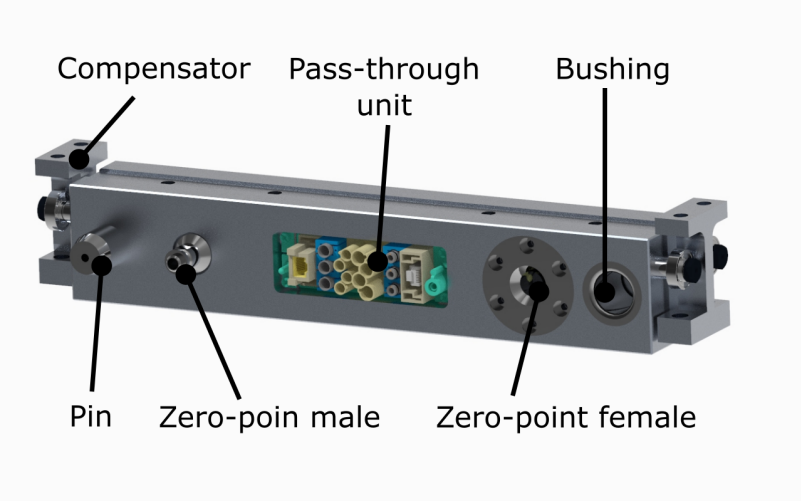


Figure 4: Plug-and-produce (PnP) connector.

3.2.1 PnP connector design

The enabling technology of the Plug-and-Produce production system is a Plug-and-Produce (PnP) connector (shown in Figure 4), which standardizes how a group of specialised individual modules are connected into a functional production system. Any module that is using the PnP connector is compatible with the modular platform. The new PnP connector developed in ReconCycle is the result of knowledge and experience gained through the use of the PnP connector developed in the ReconCell project[1], which was used to implement several industrial production processes. Some of the main characteristics of the newly developed connector are:

- cost of a connector pair is around 1600 €,
- repeatability is better than 0.05mm,
- maximum allowed forces: up to 1500N in all directions,
- maximum allowed torques: up to 1000Nm in all directions,
- transmission of electrical power: 6 x 3.5kW connections,
- transmission of compressed air: 6 x 6mm connections @ 6Bar,
- transmission of ICT signals: 2 x 8 pin Ethernet connections,
- coupling: achieved by pushing together two sides of the connectors, no unlocking needed,
- decoupling: automatically by pneumatic actuators, optionally can be aided with a decoupling force of 600N (@6bar),
- mechanical gland: compensation of inaccuracies up to ±15mm in height and ±5° in axis rotation,

[1] http://www.reconcell.eu/

- connector is "unisex", meaning that it has no male and female side,
- integrated microcomputer, which makes the connector IoT device.

The new version of the PnP connector has a universal shape, which means it no longer features a male and a female variant. Although this brings along a higher price tag, it significantly adds to the ease of use and flexibility of the system. The backbone of the connector is a single piece CNC machined aluminium body that ensures rigidity and stiffness as well as provides housing for all other components. There is a set of centering pins and bushings that facilitate proper alignment of the connector pair prior to the final coupling. The final mechanical coupling is achieved using custom zero-point clamping units that provide secure and repeatable connection of the connector pair. A big advantage of our zero-point system over most commercially available systems is that it does not require any unlocking action before the coupling. This eases the coupling process as the two connectors simply need to be pushed together with some force and the mechanism locks into place without the need for any control or actuation. The centre piece of the PnP connector is its power and data pass-through unit, which enables all modules in the cell to use power and share data among themselves. In our design, this is realized by using a commercially available modular connector unit, which can be assembled according to individual requirements and can provide pass-through of electrical power and data lines as well as pneumatic lines. An important advantage of the proposed design is the possibility of height compensation. This is achieved by integrating a height compensation unit that can even out height alignment errors of up to $\pm$15mm and angular alignment errors of up to $\pm5^\circ$. Of course, in this case the modules need to be equipped with a calibration system.

Although similar systems are commercially available, none of them come close to our solution when price/performance is taken into account.

3.3 Quick integration of new software modules on a microcomputer

Besides robot manipulation capabilities, the implementation of disassembly processes in the ReconCycle cell requires the availability of various support functions, which are provided by different auxiliary devices. For example, the robot module needs to be able to activate or deactivate the pneumatic tool changer mounted on the top of the robot. For modules that include an activation unit such as a clamp or a cutter, we need to be able to send activation signals and to check the state of the device. We selected Raspberry Pi 4 as the archetypical module's microcomputer. Raspberry Pi provides us with the ability to generate control signals and read the sensor values by attaching the auxiliary devices (equipment) to the GPIOs of the Raspberry Pi. When the auxiliary equipment is connected to the GPIOs of the Raspberry Pi, each GPIO in use must be properly configured by a suitable software library. Since the auxiliary equipment attached to each individual module needs to work in synchronization with the robots and the auxiliary equipment of other modules, we need to be able to control the equipment globally throughout the cell. Therefore we have prepared a ROS package [17] that wraps the developed software library for configuring and controlling GPIOs in a ROS node. This way we enable the configuration and control of auxiliary devices through ROS services. The cell programmer no longer needs to deal with GPIOs but can control and communicate with the auxiliary equipment through ROS interfaces.

For the operation of each module's auxiliary equipment, we implemented two ROS nodes running on each individual module's microcomputer. The first ROS node is "Equipment

Server", which configures control of the GPIOs connected equipment according to the module configuration file and forwards control commands from ROS services to the connected equipment. The second ROS node is "Equipment Manager", which allows a user to modify the module's configuration file via a ROS service.

When the "Equipment Server" node is started, it first reads the individual module's configuration file `actual_config`, which is stored locally on the module's microcomputer, and then configures the required GPIOs. The configuration file must contain information which additional equipment is attached to the module and to which GPIOs it is connected. Once GPIOs are configured, the "Equipment Server" creates a separate ROS service for each GPIO to control it. The names of the created ROS services are defined in the configuration file. In operation, the "Equipment Server" accepts the commands sent to its ROS services and controls the equipment connected to the GPIOs accordingly. Currently we have three different GPIO configurations and control interactions that "Equipment Server" can handle. The first possible configuration is a digital output where a service call can set the digital state of the GPIO, making it suitable for controlling devices such as pneumatic valves. The next configuration is a digital input that allows the digital state of devices such as digital sensors to be read on a service call. The last is a configuration that, according to the value in the service call, controls PWM signals that can be used to control devices such as step motors. The node also has a restart service that – when triggered – closes all active services, releases the pin's hardware interface, and reads the configuration file again. Then it starts with the newly read configuration.

When switching from one disassembly process to another, we often need to change, add or remove various auxiliary equipment attached to the modules. The "Equipment Manager" node allows us in these cases to quickly change the "Equipment Server" configuration according to the changes in the auxiliary equipment. We change the configuration by sending the new desired configuration to the "Equipment Manager" ROS service. When the Equipment Manager receives the new desired configuration, it overwrites the `actual_config` file and restarts the "Equipment Server" by calling the node's restart service. In this way, the "Equipment Server" reconfigures itself according to the new configuration file. Two additional "Equipment Manager" ROS services allow the user to read the current active configuration from the module and obtain an empty configuration template with default parameter values. The configuration files are of type yaml. They are human readable and can thus be modified manually.

Instead of writing or correcting configuration files manually, we created the ROS package [16] with a more user-friendly approach to handling configuration files. This package contains a client that can communicate with the "Equipment Manager". When we start the client, it opens a terminal window user-interface that guides the user through creating or modifying configuration files. At the beginning, the client searches for all "Equipment Managers" from the different modules in its reach and offers the user to select the one he wants to configure. In the next steps, the user can choose whether to modify the currently active configuration file or start over with a blank template. According to the selected option, the client then reads the correct configuration file from the "Equipment Manager". When changing the configuration, the user only needs to answer the questions about the various parameter values asked by the terminal guide. When the user is satisfied with the desired configuration, the client automatically changes the yaml file and sends it to the module "Equipment Manager".

To simplify the installation and process control of our ROS package on the module's microcomputer, we packed the ROS package into the Docker container [15] and prepared the automatic setup of the system [18].

4 Robot module

4.1 Franka ROS API

The robot control module is based on vendor-provided `franka_ros`[2], which integrates the robot's API `libfranka`[3] into ROS ecosystem.

The `franka_control_node` acts as a robot state publisher, which provides the current joint state and estimated external wrench to topics using standard ROS messages.

Robot states that are specific to the Franka Emika Panda robot arm are published using `franka_msgs`. These states include among others the current robot mode (kinesthetic guidance, robot motion using an external program, robot motion using internal controller, reflex, error recovery), which contact level is activated, whether or not collision threshold has been reached, and the compensated external load.

The ROS node `franka_control_node` also provides the following ROS services:

- `SetJointImpedance` and `SetCartesianImpedance` specify joint or Cartesian stiffness for the robot's internal controller (damping is automatically derived from the stiffness),
- `SetEEFrame` specifies the transformation from the Panda's flange frame to the tool centre point (TCP).
- `SetForceTorqueCollisionBehavior` and `SetFullCollisionBehavior` set thresholds for external forces in Cartesian and joint space to configure the collision reflex and contact detection.
- `SetLoad` sets an external load (e.g. caused by a grasped object) that the robot controller should compensate.

Most common gripper actions (`Grasp`, `Move`, `Open`) for the Panda hand gripper are also implemented in the `franka_control_node`.

4.2 `ros_control` and action servers

Controllers are implemented using `ros_control` on a dedicated computer running real-time Linux (Franka ROS Controller in Fig. 1). The `ros_control` framework provides a hardware abstraction layer (`RobotHW`) that enables standardized access to actuators and comes with a common interface (`ControllerBase`) to write robot-agnostic controllers [2]. The robot middleware is represented by the robot's hardware interface. For the Franka Emika Panda robot, such interface is implemented by the `franka_hw` ROS package using the `libfranka` library as shown in Fig. 5.

By following such a scheme, usage of standard ROS controllers and 3rd party tools (such as MoveIt!, Play Motion, or RQT joint trajectory controller GUI) is also possible.

Custom implementations of joint and Cartesian space impedance controllers (Section 4.4) expose action server interface for different robot motion modes (Section 4.3). The benefit of using ROS provided action servers to trigger robot motion is the ability to cancel the request during execution and to get periodic feedback about how the request is progressing. Upon

[3] `https://frankaemika.github.io/docs/franka_ros.html`

[3] `https://frankaemika.github.io/docs/libfranka.html`

acceptance, the action goal's status is set to active if there are no other action goals, e.g. motions, waiting for execution. If an action goal is preempted, the robot does not enter an emergency state and does not require any restart procedure. The client receives appropriate result messages in order to handle the preemption in its scheme and continue with another action if desired. This enables integration with the state machine framework presented in Section 5.

4.3 Motion generation

In order to achieve the desired robot motion, new desired joints have to be calculated at every sample time. We implemented various trajectory generation strategies to meet the most common robot motion needs in the context of automated disassembly:

- joint space point-to-point trajectory with trapezoidal velocity profile [10] (`JointTrapVel` action server in Fig. 5),
- Cartesian space straight line point-to-point motion & quaternion SLERP trajectory [20] with minimum jerk time evolution (`CartLinTask` action server in Fig. 5),
- joint space point-to-point trajectory with trapezoidal velocity profile, with the initial and final pose provided in Cartesian space and transformed into joint space using inverse kinematics (`JointTrapVelCartTarget` action server in Fig. 5),
- dynamic movement primitive (DMP) in joint space [8] (`JointDMP` action server in Fig. 5),
- Cartesian space DMP [24, 9] (`CartDMP` action server in Fig. 5),

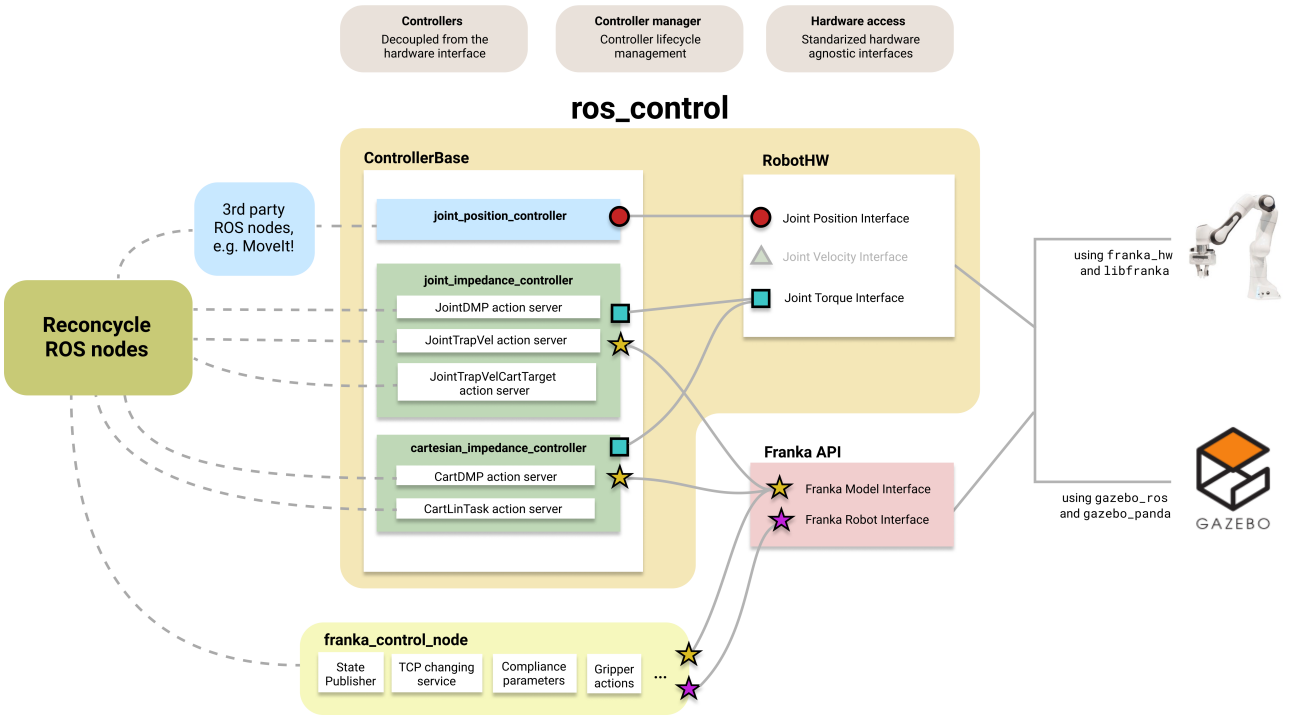


Figure 5: Integration of `ros_control` controllers into the ReconCycle arhitecture.

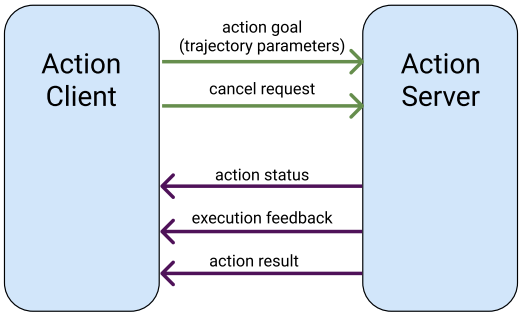


Figure 6: Clients can send action goals (motion parameters) or cancel them. Action server periodically sends status update and feedback information. When the execution is finished or interrupted, it reports the end result.

- direct joint angle control (e.g. using MoveIt! in Fig. 5).

To generate a motion according to the selected strategy, a new action goal (motion parameters) has to be sent to an appropriate action server.

`JointTrapVel`, `JointTrapVelCartTarget` and `JointDMP` action servers calculate joint configuration at each sample time. The underlying `joint_impedance_controller` calculates appropriate joint torques and sends them to the robot's low-level controller using `franka_hw` and `libfranka` as shown in Fig. 5.

`CartLinTask` and `CartDMP` action servers calculate task-space positions and orientations at each sample time. The underlying Cartesian impedance controller calculates appropriate joint torques and sends them to the robot's low-level controller using `franka_hw` and `libfranka` as shown in Fig. 5.

Using 3rd party motion generators or tools for direct joint angle control is also possible though the standard ROS interfaces.

4.4 Torque control

We rely on joint and Cartesian impedance controllers to calculate the desired joint torques $\boldsymbol{\tau}_d$. Joint impedance controller is based on the joint space dynamic model [21]. As the influence of friction and gravity is compensated by the internal Panda controllers, we can calculate the desired joint torques as follows:

$$\boldsymbol{\tau}_d = \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} + \boldsymbol{K}_q(\boldsymbol{q}_d - \boldsymbol{q}) + \boldsymbol{D}_q(\dot{\boldsymbol{q}}_d - \dot{\boldsymbol{q}}), \tag{1}$$

where $\boldsymbol{q}_d, \dot{\boldsymbol{q}}_d \in \mathbb{R}^7$ are the desired positions and velocities, $\boldsymbol{q}, \dot{\boldsymbol{q}} \in \mathbb{R}^7$ are the current joint positions and velocities and $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) \in \mathbb{R}^{7\times7}$ is the Coriolis and centrifugal matrix. $\boldsymbol{K}_q, \boldsymbol{D}_q \in \mathbb{R}^{7\times7}$ are diagonal matrices containing the gains that describe stiffness and damping per joint. Currently we neglect the term associated with the inertia matrix $\boldsymbol{M}(\boldsymbol{q})$, which is needed to realize the complete dynamic model.

To achieve impedance behaviour in the end-effector frame, the impedance controller equation using the desired Cartesian motion is specified as follows:

$$\boldsymbol{\tau}_d = \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} + \boldsymbol{J}(\boldsymbol{q})^{\mathrm{T}}(\boldsymbol{K}_x\tilde{\boldsymbol{x}} + \boldsymbol{D}_x\dot{\tilde{\boldsymbol{x}}}), \tag{2}$$

where $\tilde{\boldsymbol{x}}, \dot{\tilde{\boldsymbol{x}}} \in \mathbb{R}^6$ are the pose and velocity errors, $\boldsymbol{J}(\boldsymbol{q}) \in \mathbb{R}^{6\times7}$ is the Jacobian matrix, $\boldsymbol{K}_x, \boldsymbol{D}_x \in \mathbb{R}^{6\times6}$ are the gains describing stiffness and damping in the task space, and the rest of the

variables are as in Eq. (1). Just like in the case of joint space impedance controller, we currently neglect the term associated with the inertia matrix $\boldsymbol{M}(\boldsymbol{q})$.

The controllers are implemented as Simulink models. To enable integration with `ros_control`, Simulink Coder is used to generate standalone binaries with C++ API. The resulting plugins (`joint_impedance_controller` or `cartesian_impedance_controller` in Fig. 5) accept the following global input and output variables:

- `u`: Input variables (can be changed for every execution step).
- `p`: Parameters, that are given to the model just once at the beginning and remain constant during execution.
- `y`: Output variables (updated in every execution step).

The API consists of the following functions:

- `initialize()`: Reads parameters `p` once at the beginning.
- `step()`: Computes the output `y` given input `u` at 1ms step size.
- `terminate()`: Cleans the parameters when the computations are finished.

Configuration of the Joint impedance controller is as follows:

- Input, `u`:

  $\boldsymbol{q}$, $\boldsymbol{q}_d \in \mathbb{R}^7$: the actual and desired joint positions.

  $\dot{\boldsymbol{q}}$, $\dot{\boldsymbol{q}}_d \in \mathbb{R}^7$: the actual and desired joint velocities.

  $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) \in \mathbb{R}^{7\times7}$: Coriolis and centrifugal matrix.

- Output, `y`:

  $\boldsymbol{\tau}_d \in \mathbb{R}^7$: the required joint torque.

- Parameters, `p`:

  $\boldsymbol{K}_q$, $\boldsymbol{D}_q \in \mathbb{R}^{7\times7}$: diagonal joint stiffness and damping matrices.

The Cartesian impedance control law is encoded in the plugin as follows:

- Input, `u`:

  $\boldsymbol{T}_d \in \mathbb{R}^{4\times4}$: the desired transformation matrix of the end effector in base frame (used to calculate the pose error $\tilde{\boldsymbol{x}} \in \mathbb{R}^6$).

  $\boldsymbol{v}_d$, $\boldsymbol{\omega}_d \in \mathbb{R}^3$: the desired linear and angular velocities (used to calculate the velocity error $\dot{\tilde{\boldsymbol{x}}} \in \mathbb{R}^6$).

  $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) \in \mathbb{R}^{7\times7}$: Coriolis and centrifugal matrix.

  $\boldsymbol{J}(\boldsymbol{q}) \in \mathbb{R}^{6\times7}$: Jacobian matrix.

  $\boldsymbol{q}$, $\dot{\boldsymbol{q}} \in \mathbb{R}^7$: the actual joint positions and velocities.

- Output, `y`:

  $\boldsymbol{\tau}_d \in \mathbb{R}^7$: the required joint torque.

- Parameters, `p`:

  $\boldsymbol{K}_x$, $\boldsymbol{D}_x \in \mathbb{R}^{6\times6}$: diagonal stiffness and damping matrices.

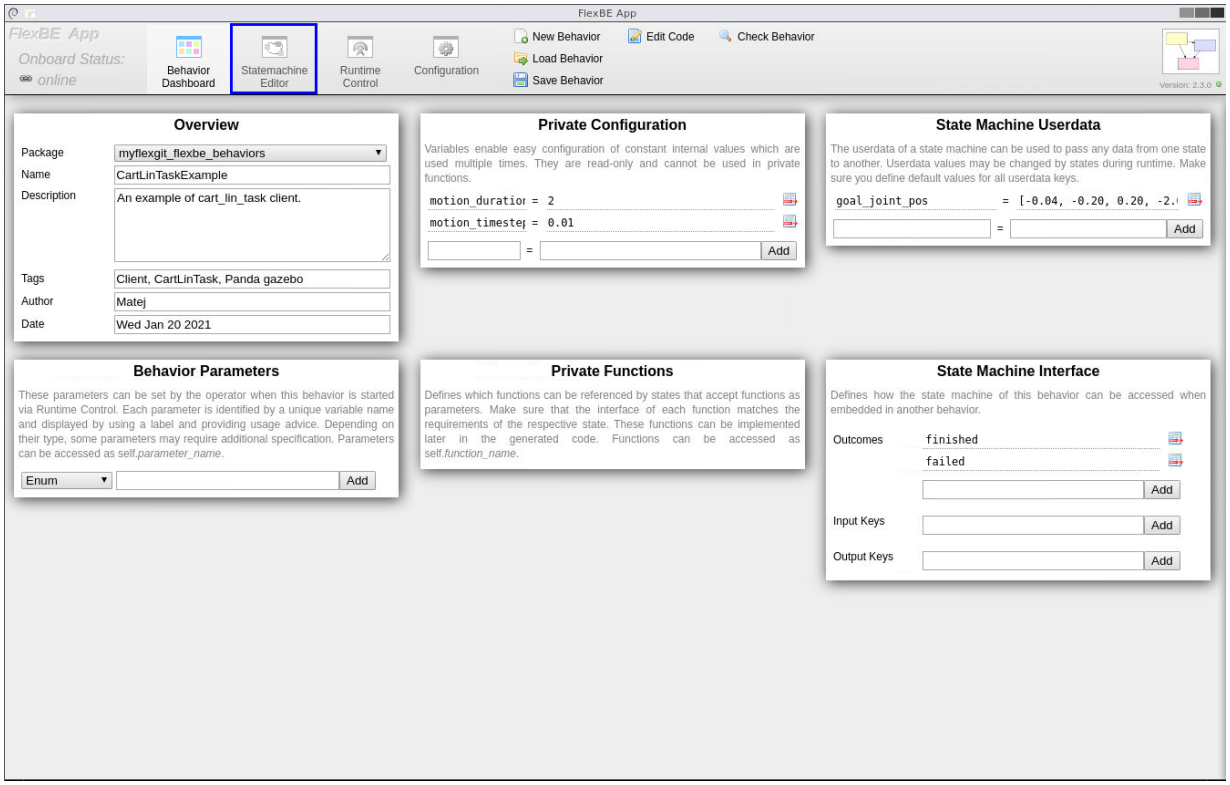


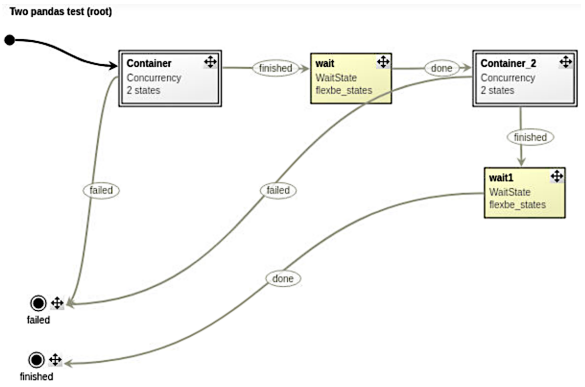
Figure 7: FlexBE application dashboard. Under the first tab ("Behavior Dashboard"), the configuration parameters of the state machines involved in the desired behavior are specified. Under the second tab ("Statemachine Editor" enclosed in a blue rectangle), the desired behavior is constructed. A behavior construction example is shown in Fig. 8.

5 Integration of FlexBE state machine editor

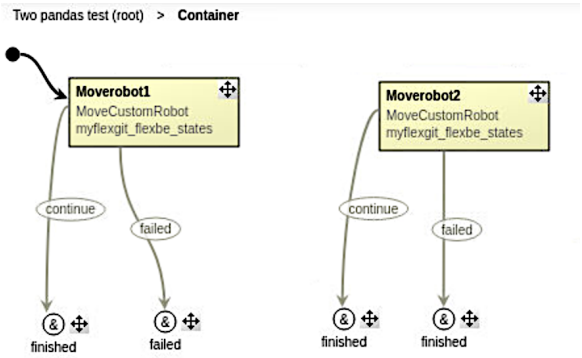
Manually coding a program that sends instructions in a specific order to all of the components of the cell (i.e. robots and peripheral equipment) requires a deep know-how of both the programming language and the underlying system. To facilitate this process, we integrated FlexBE in our workflow. FlexBE is a high-level behavior engine that supports creating, executing and monitoring complex robot behaviors [4]. When creating a FlexBE state machine (in FlexBE terms it is called “behavior”) the user can specify various parameters and constants that will be used during the execution in the Dashboard (see Fig. 7). It also provides a graphical user interface where each state is represented by a square and the transition between them are represented by arrows (Fig. 8b). This provides the user with an intuitive interface to change the state machine sequence by simply adding or removing states and creating connections between them.

Like many other tools that we used when developing the ReconCycle architecture, FlexBE is open-source. This is advantages because there are many ready-to-use states developed by the community. However, as expected, within the available community-developed states we did not find ones that would be a parfect fit for our system. Therefore, it was necessary to create states that allow us to control the execution of the disassembly within our cell.

We have created multiple custom-made FlexBE states that are used to execute robot trajectories, control the peripheral machinery, manipulate the process data, etc. Each FlexBE state



(a) FlexBE behavior specifying the process path through concurrency containers and event states



(b) State machines inside the concurrency container

Figure 8: Definition of a FlexBE behavior with concurrency containers that include two state machines specifying the concurrent motion of two robot arms. Both state machines in the container have to finish successfully in order to continue to the next stage in the execution pipeline.

is associated with a service or action server developed for ReconCycle. These states perform the following operations:

- Reading from and writing to MongoDB database, e.g. to store data acquired by kinesthetic teaching and to read data to initialize the desired robot movements (using `mongodb_store` interface).
- Controlling and monitoring the execution of robot motions using "Action Servers".
- Controlling the peripheral equipment, e.g. grippers, pneumatic vice, etc., that does not require preemption and continuous monitoring.
- Sensor data acquisition and processing, e.g. vision pipeline.

5.1 Parallel state execution

A sequential execution of states may not always be sufficient to specify an optimal task execution in a robotic cell thats contain multiple active components. Hence FlexBE also enables parallel (concurrent) execution of states.

State machines are a great representation for tasks that should be executed sequentially and where the decision which state becomes active after the previous one depends on the outcome of operations associated with previous states. For situations where multiple states should be executed at the same time (in parallel), FlexBE provides a *concurrency container* (Fig. 8a). A concurrency container delivers parallel execution and monitoring of multiple states (Fig. 8b), where each state specifies the motion of a different robot arm or an operation involving some other part of the cell.

6 Simulation of the recycling cell in Gazebo

Within the ROS community, one of the most often used software for simulating robots is Gazebo [7]. Aside from offering a good dynamic simulation and graphical interface, its main advantage – compared to other robot simulators – is its seamless integration into ROS. The robots simulated in Gazebo are controlled via the `ros_control` interface (see Section 4.2).

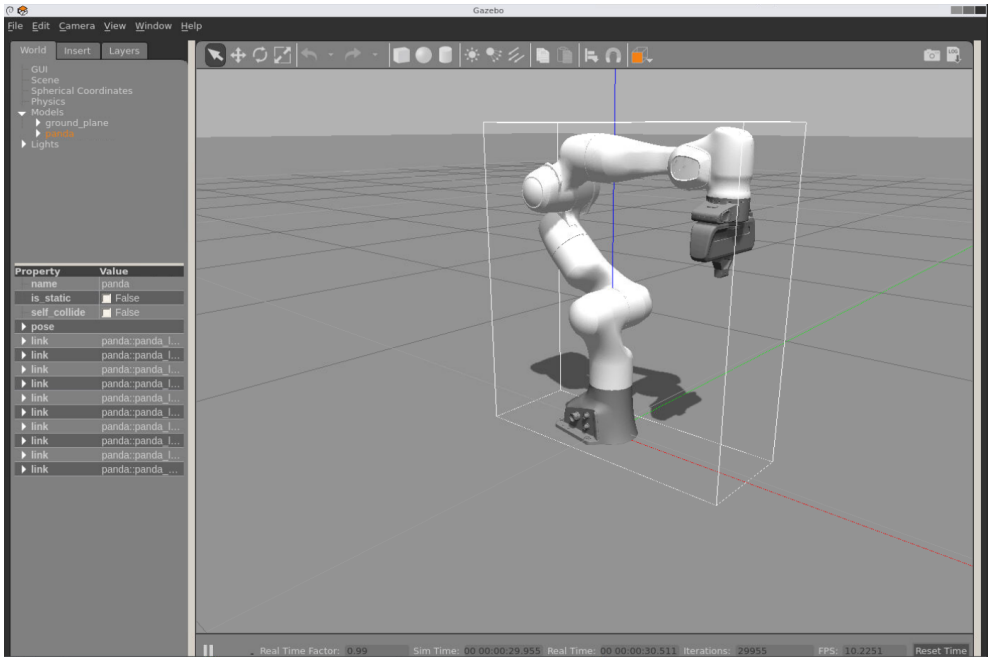


Figure 9: Simulation of the Panda robot in Gazebo

To simulate the Panda robot in Gazebo, a dynamic model of the robot needs to be created. The official software that is available for the Panda robot does not include a dynamic model that could be easily integrated into Gazebo. We therefore used a third-party software stack that is publicly available on Github [22]. This package provides the dynamic description of the robot, the 3D models of the robot's links and the hardware abstraction layer implemented in `ros_control` (see Fig. 5 in Section 4.2). Having the hardware abstraction layer implemented in `ros_control` allowed us to use the same controller plug-ins and action servers both for the simulated and the real robot. In practice this means, that it is possible to develop a top-level FlexBE program that carries out the disassembly in simulation, independently of the real system. Figure 9 displays the Panda robot in the Gazebo robot simulator.

6.1 Cell visualization in Rviz

As explained above, Gazebo provides the dynamic simulation of our robots and exposes the same control interfaces as for the real robot. However, the ReconCycle robot cell is not only composed of the robot arms but also of many other modules that are required for the disassembly of an electronic device (see Sec. 3). When developing the top-level FlexBE program, the developer needs to visualize the simulated robot in the entire robot cell. Since these modules are not expected to move as a result of external forces during the disassembly, we do no need to simulate them dynamically. We therefore decided to build a visual model of the cell in the *Unified Robot*

Description Format - URDF [25]. This format dictates how to describe a kinematic chain of links and allows us to append CAD meshes to them. To visualize this kinematic chain, we used Rviz [19].

Rviz provides a graphical interface where we can visualize the 3D model(s) described with the URDF file using various plug-ins within Rviz. Among those that we used is the "Robot Model" that displays the 3D meshes and transformation frames (*tf*) that display all of the coordinate frames available within our kinematic model. A visualization of two Panda robots installed on the modular ReconCycle cell is displayed in Figure 10. Rviz visualizes each robot by subscribing to its `joint_states` topic, which receives information either from the real robot or from dynamic simulation.

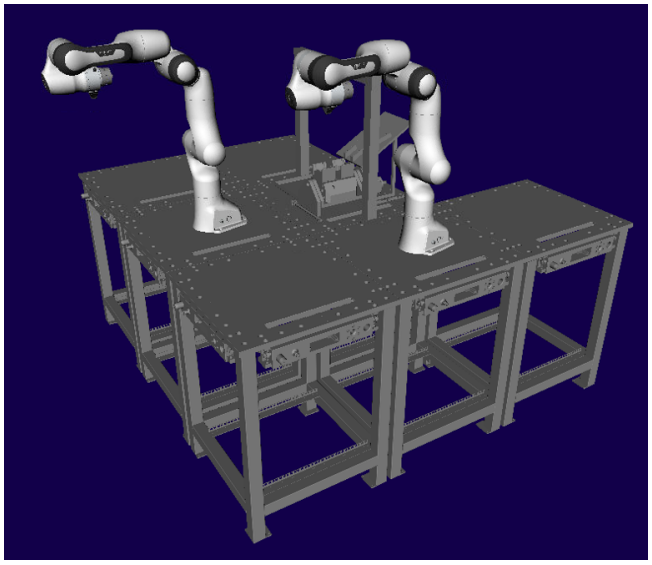


Figure 10: The ReconCycle cell visualized in Rviz.

7 Important services

7.1 Kinesthetic guidance & Helping Hand GUI

The definition of robot motions to carry out disassembly procedures can be a difficult and time consuming process even for experts, let alone non-expert users. Programming by Demonstration (PbD) provides a methodology to define such motions in a natural way rather than by coding complex programs in a robot programming language. In ReconCycle, PbD is based on kinesthetic guidance, which enables the user to move the robot through its workspace by physically guiding it along the desired path. Kinesthetic guidance is best implemented on robots with torque sensors in each joint, such as the Franka Emika Panda manipulator arms [5] utilized in ReconCycle. By utilizing the torque sensors and a model of the robot's dynamics, the robot control system can compensate for the effects of gravity. Thus the demonstrator can focus on task demonstration without needing to overcome also the robot's weight and the friction in its joints.

Compared to traditional robot programming approaches where the robot operator uses the so called teaching-pendant, kinesthetic guidance is much simpler to use. However, it comes

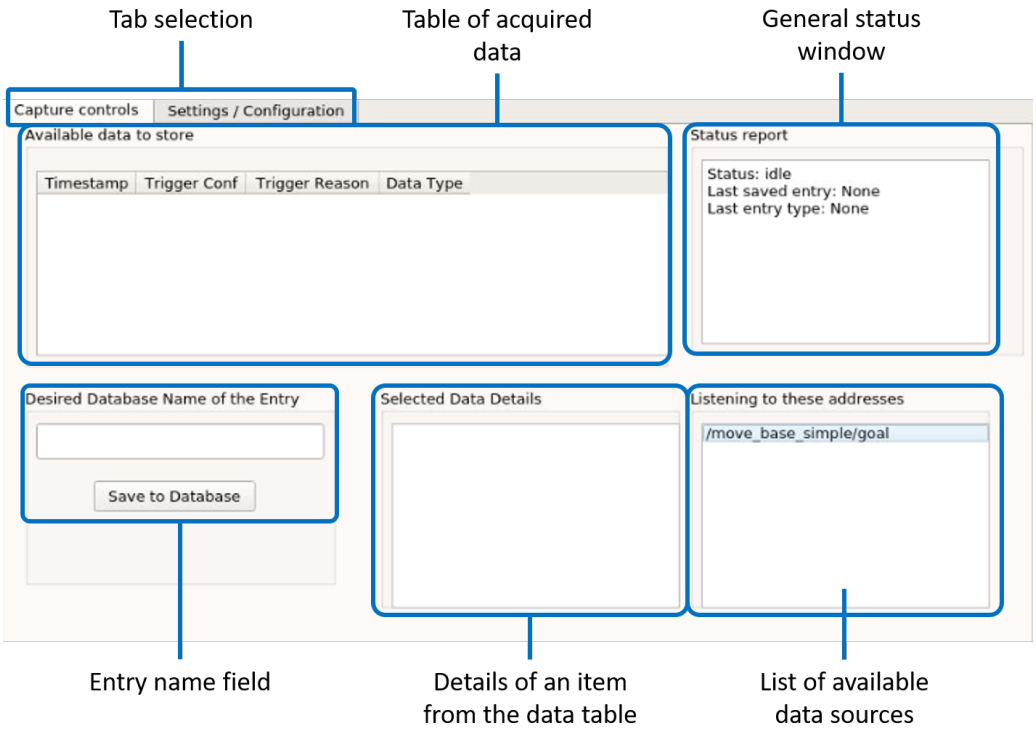


Figure 11: The "Capture controls" tab of the Helping Hand graphical user interface.

with a drawback – lack of an interface to move and store the taught poses. To overcome this drawback, we developed a graphical user interface (GUI) to help users storing said skills into the MongoDB database (Fig. 11). We named it Helping Hand. This user interface runs on a computer connected to the ROS network so it has access to all that is being broadcast on the network, including robot joint configurations and poses of their tool center point (TCP). This GUI is used to store either single configurations (joint or Cartesian space) or whole trajectories encoded as DMPs (joint or Cartesian space) as named entries into the database.

7.1.1 Using the Helping Hand GUI

The GUI has two main tabs: "Capture controls" and "Settings/Configuration". The first is used during the kinesthetic teaching process to store data into the database, while the second is used to configure the GUI itself.

The *Capture controls:* this tab is composed of the following fields (see Fig. 11):

- Available data to store – when a user triggers a signal defined in the GUI's configuration, the current robot posture (joint and Cartesian space) are saved in a temporary buffer and displayed in this field.
- Status report – this field is used to display some basic status information of the robot and the GUI.
- Desired Database Name of the Entry – to store some data into the database, the user writes the name under which the entry should be saved into the database, selects the

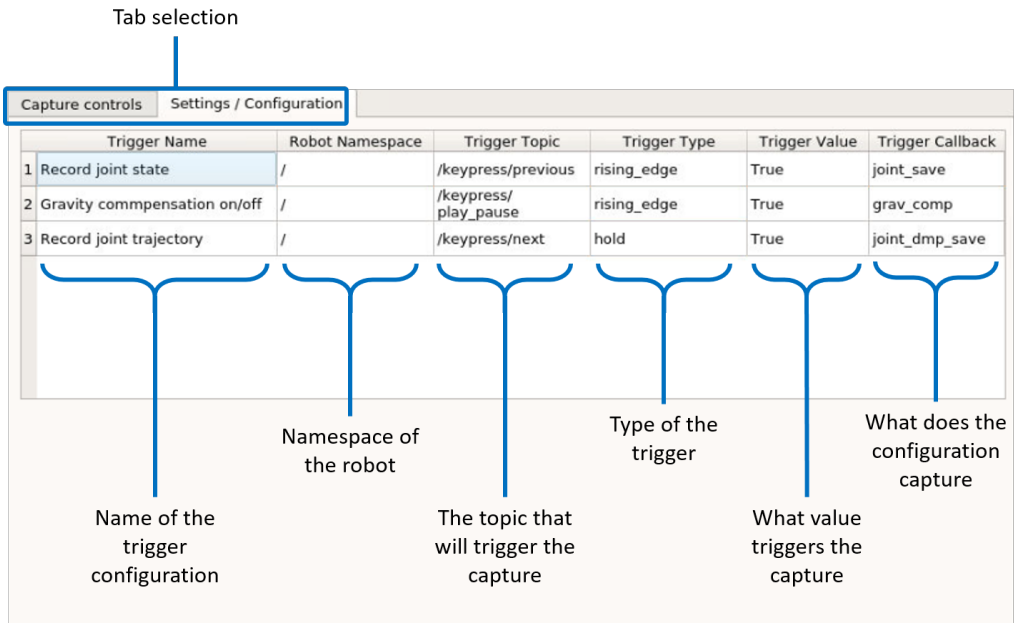


Figure 12: The "Settings/Configuration" tab of the Helping Hand graphical user interface.

single data entity from the "Available data to store" fields, and then clicks on the "Save to Database" button.

- Selected Data Details – Before saving the data into the database, the user can inspect the data details. These details are displayed for the data selected from the "Available data to store" field.

- Listening to these addresses – The topics displayed in this field inform the user, which robot status is available to be recorded.

Fig. 12 shows the *Settings/Configuration* tab. It presents the current configuration of the GUI:

- Trigger Name – The name of the configuration for the specific trigger.

- Robot Namespace – The namespace of the robot for the specific trigger.

- Trigger Topic – The name of the ROS topic (of the type `std_msgs/Bool`). Upon triggering it signals the GUI to save the data into the aforementioned temporary buffer.

- Trigger Type – The type of the trigger, this can be either `rising_edge`, `falling_edge` or `hold`.

- Trigger Callback – The storing function that is executed when a trigger is detected.

The configuration is loaded when the GUI is started. It is read from a YAML file.

To acquire and store the required information, the user first demonstrates the skill on the robot and then triggers a signal that stores this skill into a temporary buffer of the GUI. This buffer is used so the user can demonstrate multiple skills or multiple variations of a single

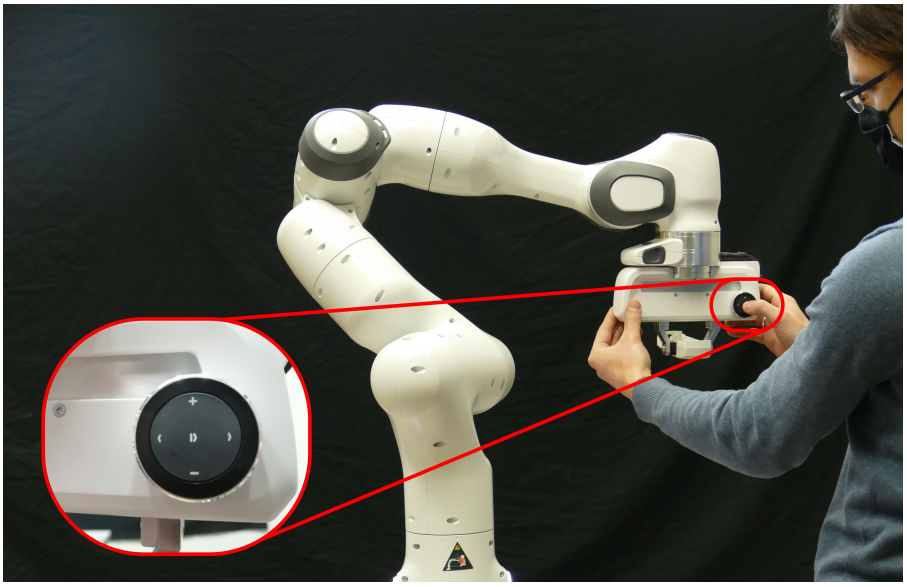


Figure 13: Bluetooth media controller is mounted on the robot arm to provide an easy-to-use interface to the Helping Hand GUI.

skill before s/he interacts with the GUI and thus saves time. After all the skills have been demonstrated and stored in the buffer, the user can evaluate which of them should be stored (the "Available data to store" field in Fig. 11) in the database and in what form (either single configurations or whole trajectories). The user then selects the data from the buffer, defines a name under which it will be saved in the database, and clicks on the "Save to Database" button. These named entries are then used by the top-level scheduler (see the details about the usage of FlexBE in Section 5) to generate robot motions.

To further improve on the intuitiveness and speed of kinesthetic guidance, we equipped the Panda robots with a button interface. The button interface is a small battery-powered media control device (Fig. 13) that connects to the computer via Bluetooth. Therefore, we either need a computer that has Bluetooth interface integrated or we must use a Bluetooth USB dongle. To fully integrate this button interface into the ROS architecture and therefore use it during the kinesthetic teaching process, a ROS node was developed that reads the key-press events and broadcasts this information on the ROS topics. Integrating it with the Helping Hand GUI was done by writing a configuration file that describes the buttons as triggers for the storing events explained above.

7.2 Vision integration

For the robot arm to interact with the electronic devices that are to be recycled, the position and orientation of the objects needs to be known. A high resolution camera by Basler is therefore mounted directly above the work surface for each of the modules where objects need to be detected. The finish of the archetypical module's surface is matt grey to ease vision processing. The Basler camera can return an image at 4k resolution, but for our purposes this has been limited to an image of resolution 1450 x 1450, which the camera can send to the computer at a frequency of 5Hz. Larger frame rates are not needed due to the fact that visual scene analysis only needs to be performed "point-wise" (situation assessment).

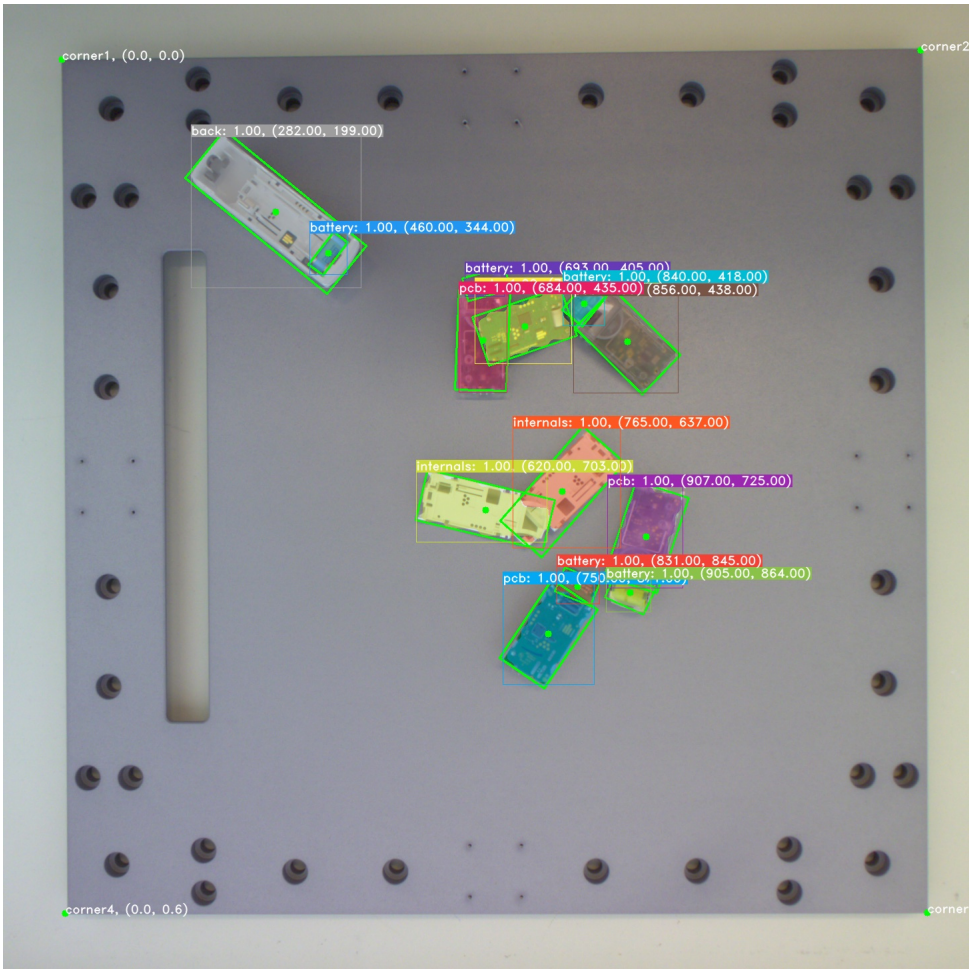


Figure 14: Example of the vision software detecting the pose of heat cost allocator components on the work surface.

In it's current state, the vision software works on the Kalo 1.5 heat cost allocator (HCA), but can be adapted to other devices without much effort. The objects that it can currently detect are:

- the HCA before dissasembly has occurred,
- the battery completely separated from the device,
- the battery with part of the PCB still attached,
- the PCB on it's own,
- internal components of the HCA such as internal plastics.

When the HCA is detected, the object's discrete orientation is also provided as part of the class name. The possible discrete orientations that can be detected are: front, side1, back, and side2. The orientation "front" is given when the face facing the camera is the "front" face, and similarly for the other 3 faces. "side1" is the side to the right of the front and "side2" the side to the left of the front face. An example of what the vision software detects is shown in Figure 14. For each detected object, the following information is given:

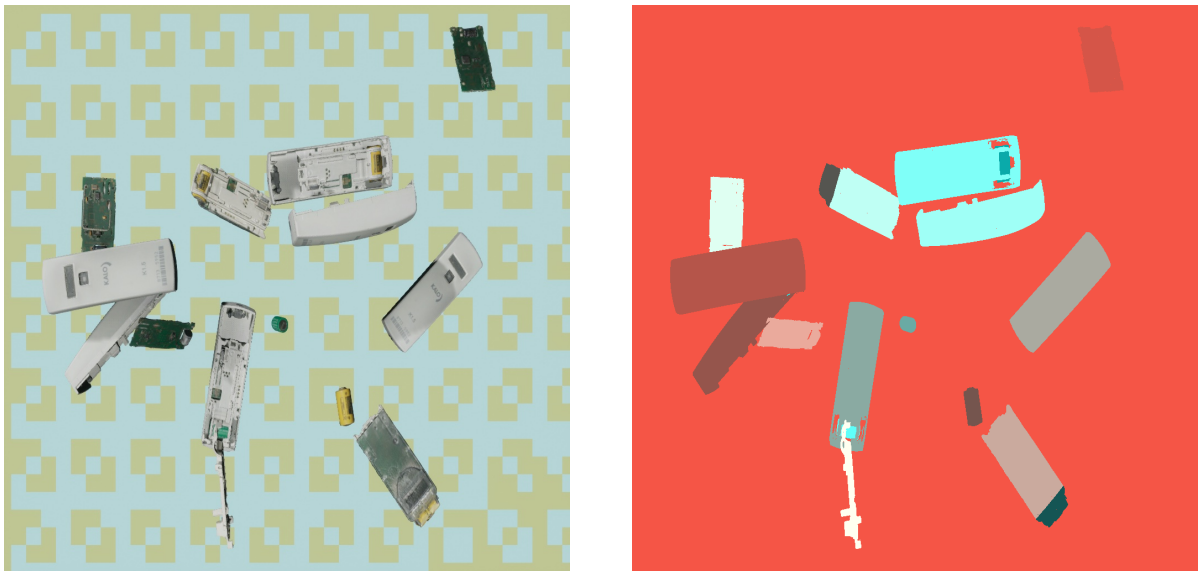
- the class name of the detection,
- the detection score, which is a real number in the interval $[0, 1]$ for how confident the model is in the prediction,
- the vertices of the oriented bounding box in a list of x, y coordinates,
- the x, y coordinates of the oriented bounding box center,
- the rotation quaternion of the oriented bounding box.

Note that all coordinates are in meters and are given relative to the work surface corners. This may in future be adjusted to be relative to certain screw holes on the work surface because they are easier to detect when modules are connected together.

The vision software comes with a ROS front-end. The results computed by the vision software can be published continuously at a rate of around 4 fps when Nvidia 1080ti graphics card is used for processing or around 3 fps when Nvidia 1060 is used. When publishing continuously, after publishing the results, the software takes the latest image received from the camera to process next. Alternatively, the data for the latest camera image can be published on receiving a service call. The vision software provides the following ROS nodes:

- `/camera/image_color` camera image (undistorted),
- `/vision_pipeline/image_color` labeled camera image (undistorted),
- `/vision_pipeline/data` JSON string containing a list of detections.

The vision software uses the Yolact neural network [1] for instance segmentation. Yolact uses ResNet at its core and has been chosen for its ability to perform real-time instance segmentation.



(a) Synthetic image (b) The corresponding instance segmentation

Figure 15: Sample from the synthetic dataset generated using Unreal Engine with the NDDS plugin.

Yolact requires a dataset to train on. A synthetic dataset has been created, consisting of 20.000 labeled images. An example is shown in Figure 15. It was generated using Unreal Engine with the NDDS plugin [23]. The 3D models have been created using a 3D scanner (Artec Space Spider). In addition, we have created a second dataset that consist of 80 hand labeled images from the real-world setup. Yolact is first trained on the synthetic dataset and then on the real dataset. The model trained on the synthetic dataset only can be used to detect the real world objects. However, there are many false positives on real images if the mode has been trained only on synthetic data. Thus the real data have been added to bridge the reality gap.

7.3 Integration of qb SoftHand Research

The *qb SoftHand Research* [12] is an anthropomorphic robotic hand based on soft robotics technology. With its under-actuated structure, the hand is able to replicate about 75% of the grasps of a human hand. It is able to naturally adapt to the objects it picks up without the sophisticated sensing technology. This simplicity and flexibility make it an excellent gripping device that can grasp a variety of different objects without any change in the control action. This makes it very suitable for use in the ReconCycle project, where the robot should be able to pick up objects for recycling that come in many different shapes, states of wear and tear, and in different configurations on the table. The position and orientation of the objects to be grasped is detected by vision, which also introduces some errors into the detected values. The gripping flexibility of qb SoftHand makes it possible to reduce the influence of these uncertainties and errors, which ensures more robust grasping.

In ReconCycle, the qb SoftHand is mounted on a Franka Emika Panda manipulator equipped with a commercial tool changer, as shown in Figure 16. The hand is connected through the tool changer with custom wiring that allows for quick mounting or dismounting of the hand on the robot.

The single-motor actuation makes the hand plug-and-play and easy to control. There is only one motor that accepts continuous percentage values for the opening & closing of the

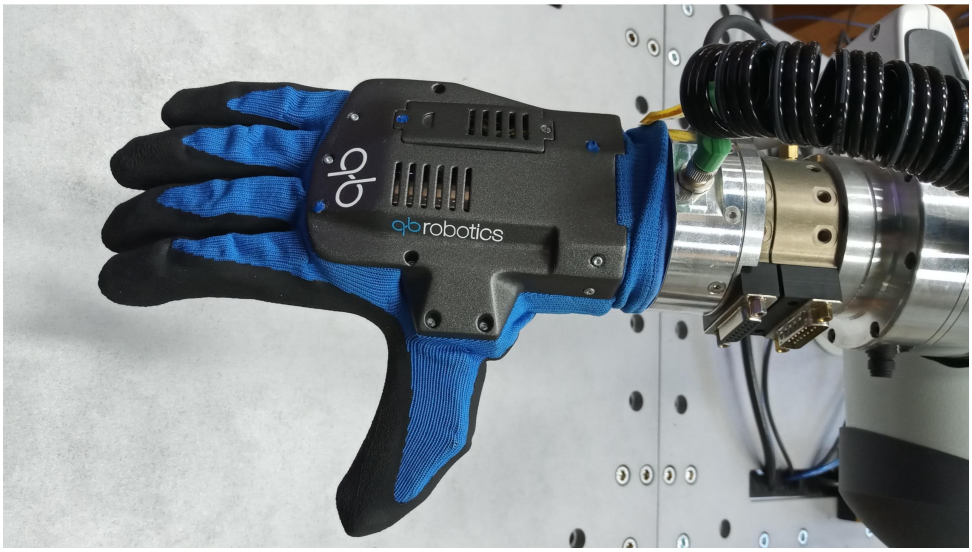


Figure 16: qb SoftHand Research soft five-finger gripper attached to the Franka Emika Panda robot arm

whole hand. On the low-level, the hand is controlled by dedicated driver electronics. On the high-level, the driver electronics connects to a computer with a USB cable. It is then possible to control the hand from the computer by sending the appropriate serialized signals. To integrate the hand into our ROS architecture, we use a modified ROS driver implemented as an "Action Server" that can be used to trigger the opening & closing of the hand. The action server receives the desired grasping values from the action client trough ROS and converts them so they can be sent to the low-level driver. The standardized ROS control interface used in the hand's action server also enables us to control the hand using standard motion generators and `joint_position_controller`, similar to the control scheme shown in Fig. 5.

We use Raspberry Pi 4 microcomputer, which is already part of every ReconCycle module, as the control computer. We integrated the required software for interaction with the hand, i.e. the hand's action server and the software drivers for the hand electronics, into the Docker container. This allows easy installation of the entire software set on the microcomputer and migration to a different microcomputer if required. It also prevents interference with other software on the microcomputer.

We have prepared a suitable FlexBE state machine to control the qb SoftHand Research within more complex tasks. The state takes as input the desired value of the hand's grasp and the desired time interval for this movement. The implemented FlexBE state allows easy integration of different hand movements for grasping through the entire disassembly sequence that runs in the ReconCycle cell.

References

[1] D. Bolya, C. Zhou, F. Xiao, and Y. J. Lee. "YOLACT: Real-time Instance Segmentation". In: *ICCV*. 2019.

[2] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodríguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke, and E. Fernández Perdomo. "ros_control: A generic and simple control framework for ROS". In: *The Journal of Open Source Software* 2.20 (2017), p. 456. DOI: 10.21105/joss.00456.

[3] *Docker documentation*. https://docs.docker.com/. Accessed: 2021-2-26.

[4] FlexBE homepage. http://philserver.bplaced.net/fbe/. Accessed: 2021-3-5.

[5] Franka Emika Panda Powertool; Robot Arm & Control. https://www.franka.de/technology/. Accessed: 2021-02-01.

[6] T. Gašpar, M. Deniša, P. Radanovič, B. Ridge, T. R. Savarimuthu, A. Kramberger, M. Priggemeyer, J. Rossmann, F. Wörgötter, T. Ivanovska, S. Parizi, Ž. Gosar, I. Kovač, and A. Ude. "Smart hardware integration with advanced robot programming technologies for efficient reconfiguration of robot workcells". In: *Robotics and Computer-Integrated Manufacturing* 66 (2020). DOI: 10.1016/j.rcim.2020.101979.

[7] Gazebo simulator. http://gazebosim.org/. Accessed: 2020-12-31.

[8] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal. "Dynamical Movement Primitives: Learning Attractor Models for Motor Behaviors". In: *Neural Computation* 25.2 (2013), pp. 328–373.

[9] L. Koutras and Z. Doulgeri. "A correct formulation for the Orientation Dynamic Movement Primitives for robot control in the Cartesian space". In: *Proc. Conference on Robot Learning (CoRL)*. Osaka, Japan, 2019, pp. 293–302.

[10] K. M. Lynch and F. C. Park. *Modern Robotics; Mechanics, Planning and Control*. Cambridge University Press, 2017.

[11] mongodb_store package. http://wiki.ros.org/mongodb_store. Accessed: 2021-1-19.

[12] qb SoftHand Research. https://qbrobotics.com/products/qb-softhand-research/. Accessed: 2021-3-5.

[13] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. "ROS: An Open-Source Robot Operating System". In: *ICRA workshop on open source software*. Kobe, Japan, 2009.

[14] M. Quigley, B. Gerkey, and W. D. Smart. *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. Sebastopol, CA: O'Rilley Media, 2015.

[15] Raspberry Pi 4 Raspi-ROS Docker. https://github.com/ReconCycle/raspi-reconcycle-docker. Accessed: 2021-2-25.

[16] Raspi-ROS-Client package. https://github.com/ReconCycle/raspi-ros-client. Accessed: 2021-2-25.

[17] Raspi-ROS package. https://github.com/ReconCycle/raspi_ros. Accessed: 2021-2-25.

[18] ReconCycle automatic initialization of Raspberry Pi 4 Raspi-ROS Docker. `https://github.com/ReconCycle/raspberry_reconcycle_init`. Accessed: 2021-2-25.

[19] Rviz. `http://wiki.ros.org/rviz`. Accessed: 2021-3-5.

[20] K. Shoemake. "Animating rotation with quaternion curves". In: *SIGGRAPH Computer Graphics* 19.3 (1985), pp. 245–254.

[21] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics, Modelling, Planning and Control*. Springer, 2009.

[22] The `panda_gazebo` package for simulating the Panda robot from Franka Emika. `https://github.com/justagist/panda_simulator/tree/kinetic-devel/panda_gazebo`. Accessed: 2021-3-3.

[23] T. To, J. Tremblay, D. McKay, Y. Yamaguchi, K. Leung, A. Balanon, J. Cheng, W. Hodge, and S. Birchfield. *NDDS: NVIDIA Deep Learning Dataset Synthesizer*. `https://github.com/NVIDIA/Dataset_Synthesizer`. 2018.

[24] A. Ude, B. Nemec, T. Petrič, and J. Morimoto. "Orientation in Cartesian space dynamic movement primitives". In: *IEEE International Conference on Robotics and Automation (ICRA)*. Hong Kong, 2014, pp. 2997–3004.

[25] Universal Robot Description Format - URDF. `http://wiki.ros.org/urdf`. Accessed: 2021-3-5.