DexBuddy – System Overview

1) Summary

DexBuddy is mainly a software capability demonstrator experiment. It shows the potential of the combination of 3D-vision, finger-based force sensing and wrist-based force sensing used with online grasp and motion planning as well as force-controlled motions. Furthermore, it shows how concepts of intuitive programming can be used to parameterize arm and finger motions for dexterous manipulation. The overall setup is demonstrated in the context of a dexterous industrial assembly use-case with cables.

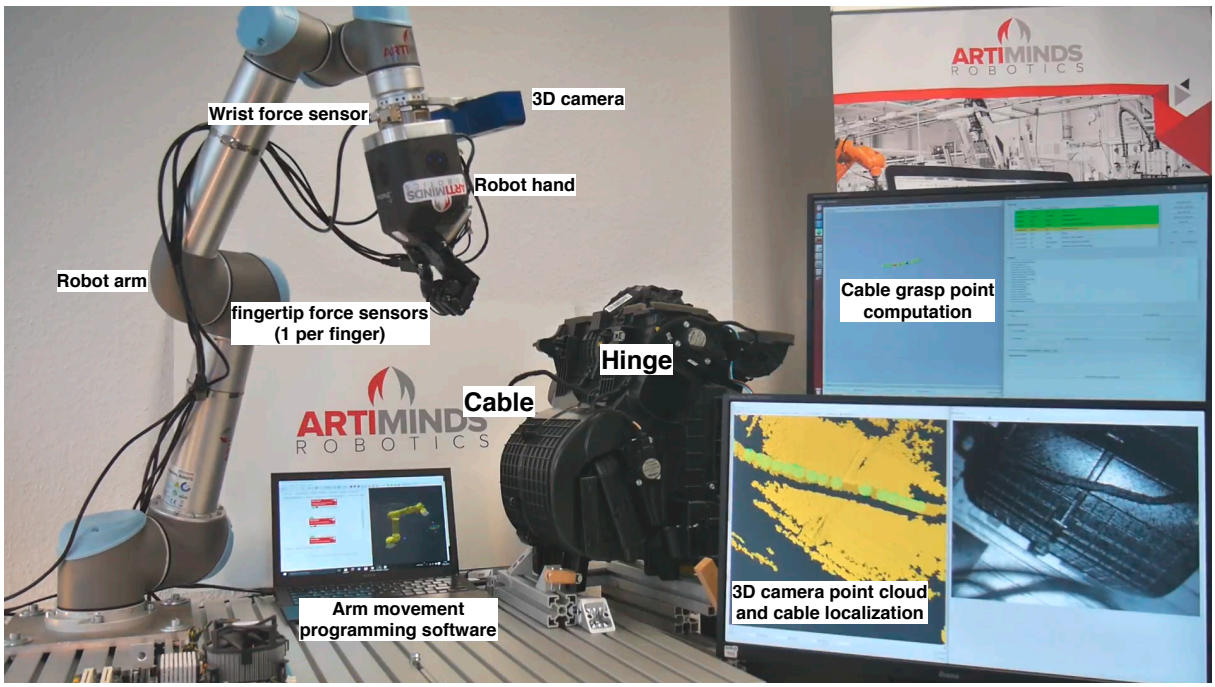
This documents provides a more detailed textual system description than the software integration or storyboard deliverable.

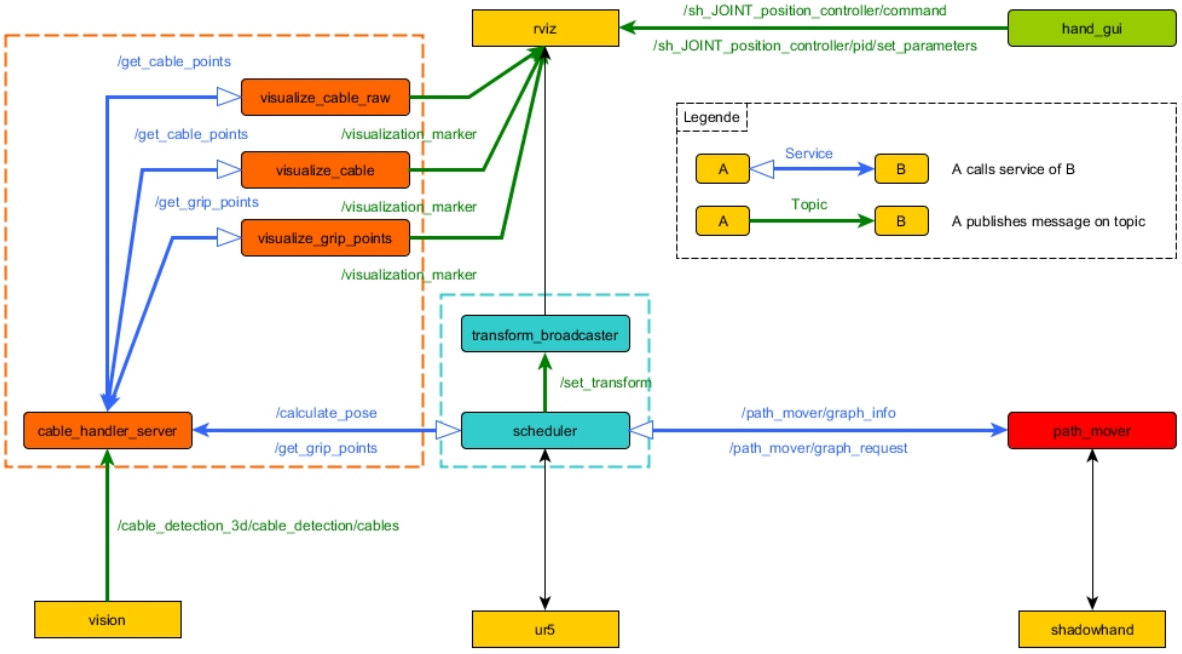
2) Deviations from project plan

The project was delayed because of the necessity to integrate force sensors into the fingertips of the robotic hand. This was not originally intended and could only be achieved from M12-M15. This delayed the final experiments and the final, full experiments could just be performed near the end of the project. However, full integration of 3D-vision-based grasp planning, finger-based force control and wrist-based force control could be achieved beyond the planned state. Adding fingertip-based force control to the system also increased the complexity of the software system. Thus, final experiments could not be performed as extensively as originally planned.

3) System overview

<TODO>

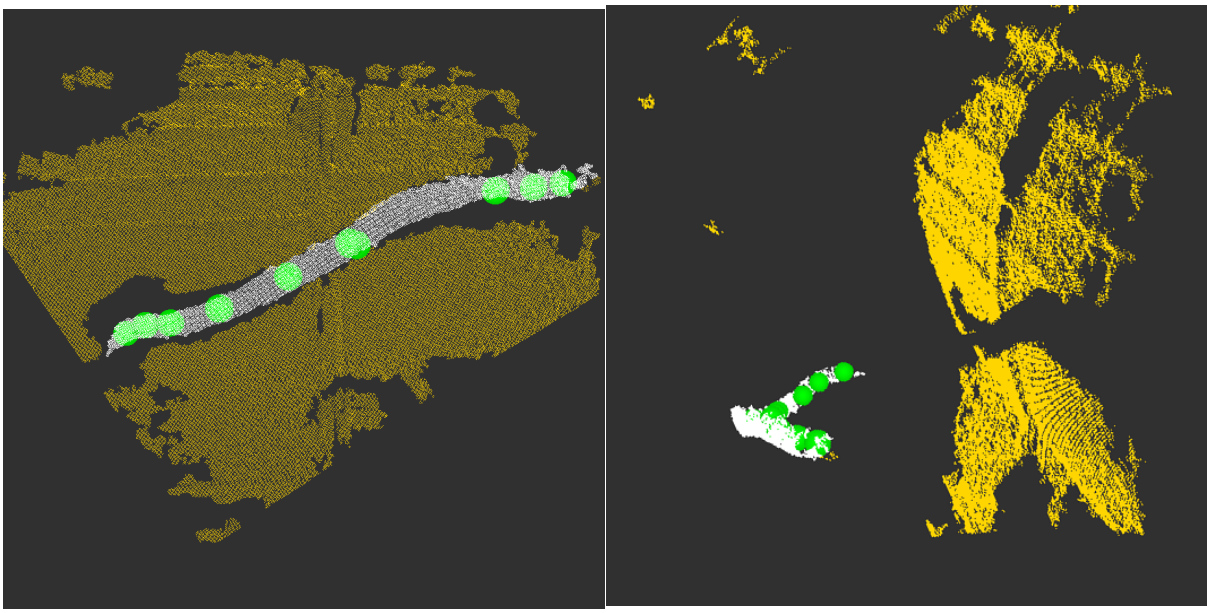



rviz
/sh_JOINT_position_controller/command
/sh_JOINT_position_controller/pid/set_parameters
hand_gui
/get_cable_points
visualize_cable_raw
/get_cable_points
/visualization_marker
visualize_cable
/get_grip_points
/visualization_marker
visualize_grip_points
/visualization_marker
Legende
A
Service
B
A calls service of B
A
Topic
B
A publishes message on topic
transform_broadcaster
/set_transform
cable_handler_server
/calculate_pose
/get_grip_points
scheduler
/path_mover/graph_info
/path_mover/graph_request
path_mover
/cable_detection_3d/cable_detection/cables
vision
ur5
shadowhand

4) Individual components

4.1) Cable pose and shape localization

Input are 3d point clouds given by an Ensenso N20 camera system. The Ensenso N20 contains a stereo camera setup with two 1.3 mega pixel cameras and a projector projecting a random but static pattern to bring more texture information to the scene. Not only the short aperture angle causes a small field of view, also the shallow depth of field of camera and projector cause a small depth range where 3d information is recognized with low noise. According to this restriction the cable, which should be recognized, has to be in this area.
Detecting the cable consists of various steps of preprocessing the point cloud, detecting cable candidates and validating those. The first step is clustering the point cloud and dismissing all clusters, which are too big for a cable or too small for a stable decision whether this cluster belongs to a cable or not. Then each remaining cluster is checked for a cable shape using RANSAC with a cylindrical model. All clusters fulfilling these criteria are used as seed points for the next step.
In the next step the process iteratively searches for cable segments starting at these seed points. For easier iteration steps, all points from the point clouds are shifted along their calculated normal in this way, so that all points from the cable surface are now at the cable center of mass. Thus, the diameter of the cable has to be known apriori. Now, the iteration proceeds as follows: first, the direction of the cable segment at the current point is determined by the last two points. At the initial step the direction of the cylinder is used. Now we are going along this direction to get an initial guess for the next point and enhance the position of this point by calculating the center of mass of all points located in a sphere around the initial guess. The resulting points now form a discrete representation of the center of mass of the cable segment. Together with the diameter of the cable segment they form a chain of spheres, which represents the cable segment.

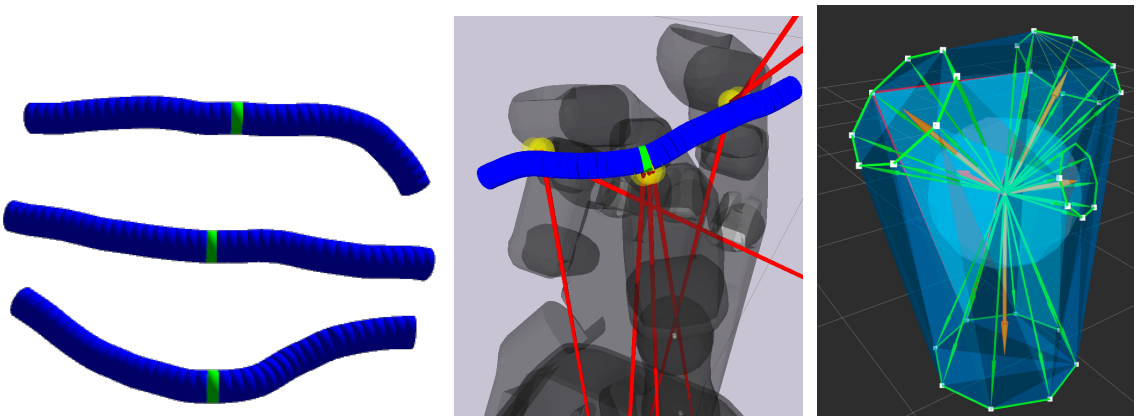


These cable center points have to be filtered to delete some possible false positives by using some constraints like the following: there have to be enough surface points in the initial point cloud around the found cable center points and the curvature of the segments has to be less than a maximum value.

By starting at some seed points and never checking whether two points belong to the same cable, the found cable can be represented by two or more independent segments of cable center points. These segments have to be merged together. This is done by checking the distance between every point of one cable with all others. If the distance is lower than a threshold, both points are deleted and a new point, calculated by the average of these points, is added to the final cable.
This approach works quite well detecting cables and determining the correct position, even when the cable does not have ideal characteristics like in DexBuddy project, where the profile of the used cable was not perfectly round and the surface consisted of fleece and was a bit ruffled. Even the full camera resolution was not needed and it was possible to reduce it to speed up the recognition a lot and so it was possible to process one point cloud in about 0.3 seconds.

4.2) Finger motion optimization

The task of this component is the optimization of grasps that haven been taught via the teaching component (see 4.6). It assumes that the human-taught finger motions are “good”, but that they may not be optimal. That is, by some slight variation of the joint angles a better grasp with the same characteristic as the base grasp might be achievable. Thereby, “better” refers to some kind of quality metric. In this case, the epsilon metric (i.e. the radius of the biggest sphere in the grasp wrench (or force) space induced by the grasp's contacts) is used. The optimization is done in simulation, using the robot simulator Gazebo.
The component currently uses the following ROS packages: cable_generation, contacts_utility, grasp_evaluation, grasp_optimization and grasp_optimization_msgs. It uses grasp execution functionality provided by the movement component (ROS package path_mover).
Package cable_generation: Used to generate random cable configurations so the grasp can be tested on several different cables. The cable is modeled as a chain of cylinders and double revolute joints between each cylinder. Parameters such as the standard deviation and the smoothing coefficients of the joint angles can be changed dynamically. The package provides a ROS service, which can be called in order to spawn a random cable at a given pose in the simulation. Currently the spawned cable is static, i.e. not moving (left picture).
Package contacts_utility: Some utilities such as collecting and visualizing the contact information from the simulation (middle picture, contacts are yellow, force vectors are red).



Package grasp_evaluation: Computes the quality of a grasp given its contact information. The package provides ROS services which take a grasp's contact information (i.e. the wrenches exerted by the grasp), construct the convex hull over either the force vectors (grasp force space, 3D) or the wrench vectors (grasp wrench space, 6D) using the library qhull, and compute and return the epsilon (ε, see above) and ny (v, volume of the grasp space) metrics of the grasp. Friction cones are used to model soft-finger-contacts (right picture, convex hull of four friction cones. The radius of the sphere in the center is the epsilon metric).
Package grasp_optimization: The core package, performs and controls the optimization process. The optimization process consists of the following steps:

1. Load the human-taught grasp.
2. Vary the cable pose and evaluate, choose the best cable pose.
3. Vary the grasp joint angles and evaluate, choose the best joint angles.
4. Vary the cable pose and evaluate (with best joint angles), choose the best cable pose.
5. Save the optimized grasp.

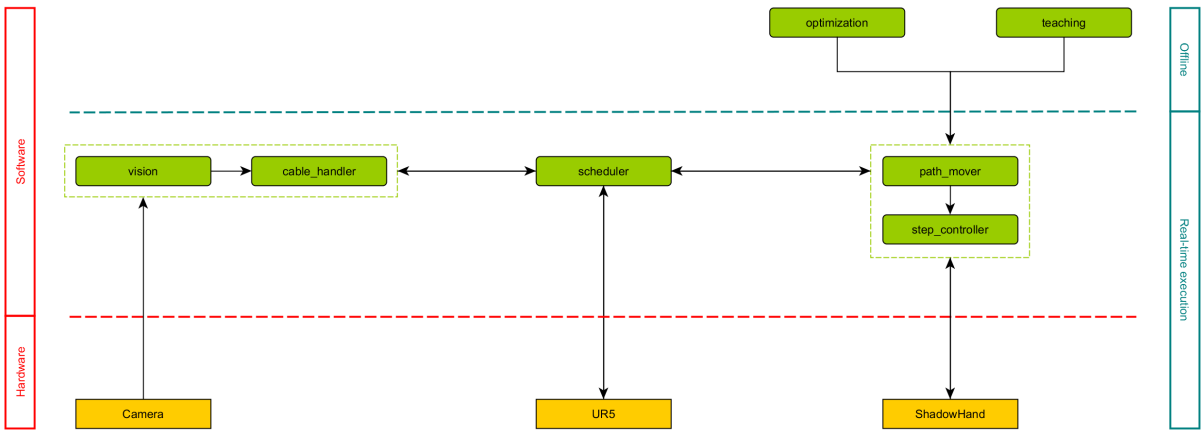
It uses ROS topics and services provided by the other packages.

4.3) Arm motion control

M

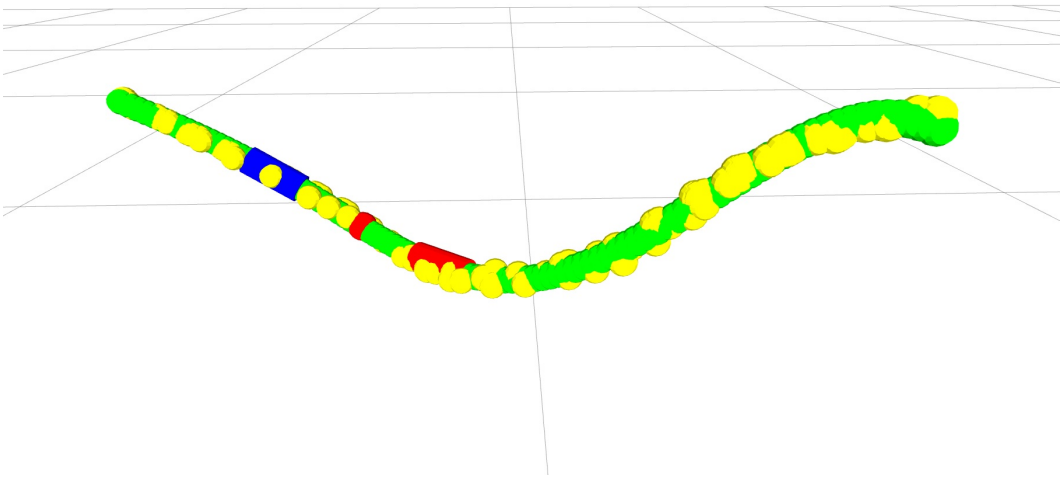
4.4) Finger motion scheduling and execution

The *scheduler* component plays a central role in the DexBuddy project since it is responsible for the coordination of and the interaction between the different components of DexBuddy. As a central component it interacts with all other components during the real-time execution of a grasping task, that is the camera, the ShadowHand and the UR5.



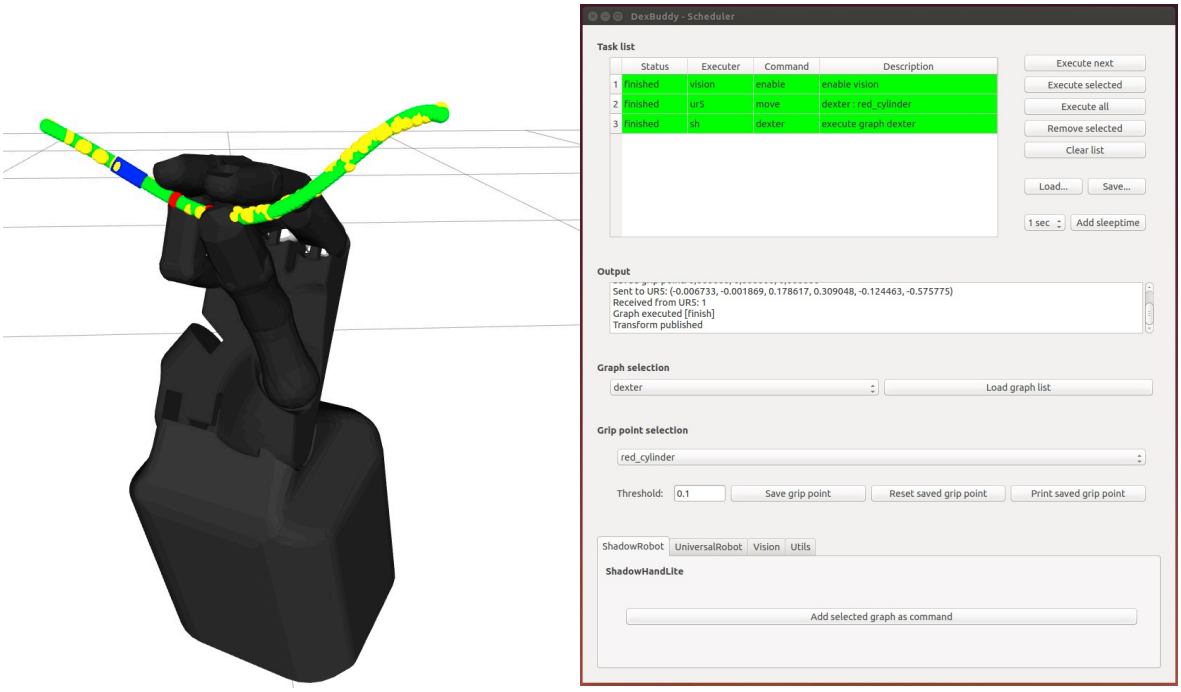
Interaction with the camera

For the DexBuddy project an Ensenso 3D stereo camera is used to record and gather information about the location of the cable to grasp. On the software side, the cable recognition is realized by the *vision* component of DexBuddy. Additionally, the *cable_handler* component is used to gather data recorded from the camera by the *vision* component and to generate an interpolated cable (see picture) from this data. Based on the generated cable, the grasp points and the respective pose of the ShadowHand can be calculated by the *cable_handler* and are offered to the *scheduler* via ROS service calls.



Interaction with the ShadowHand

The execution of a ShadowHand grasp is performed by the *path_mover* component of DexBuddy. For this purpose, the *scheduler* can retrieve information about the available grasps, so-called "*graphs*", and trigger the execution of a specific graph via ROS service call to the *path_mover*.



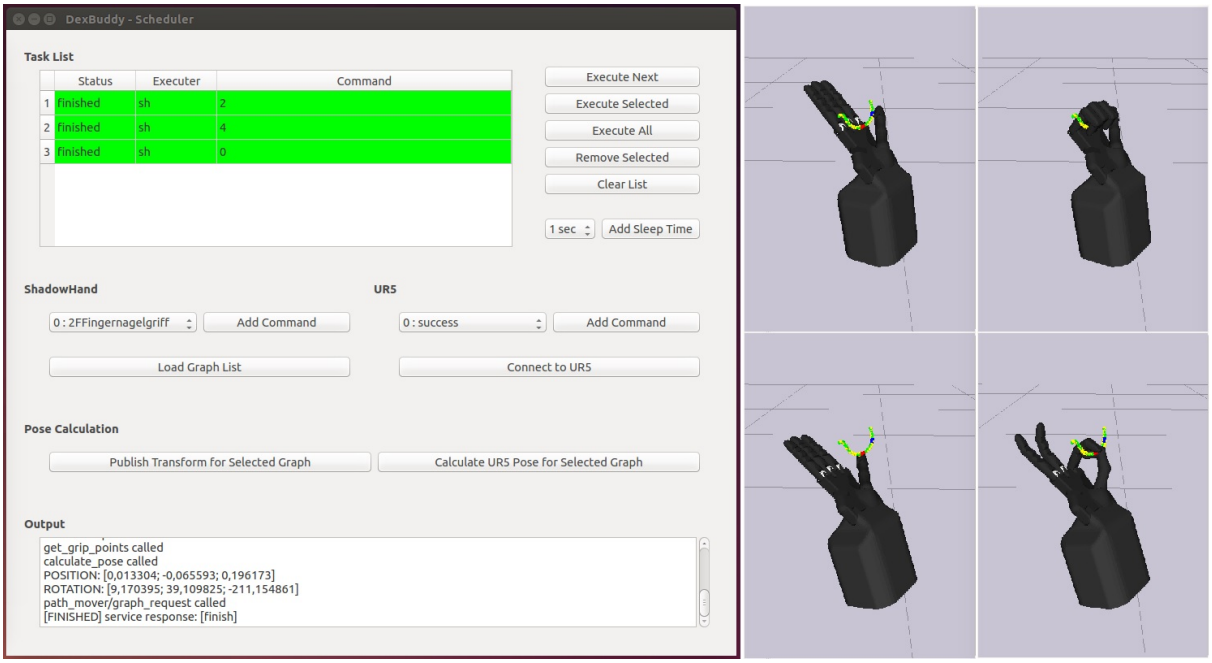
Interaction with the UR5

The communication to the UR5 is realized by establishing a TCP connection between the *scheduler* and the UR5. For this, the *scheduler* acts as a TCP server to which the UR5 can connect in order to exchange data. This data can be information about the location of the grasp point or other a priori agreed command codes to perform specific movements on the UR5.

Task execution

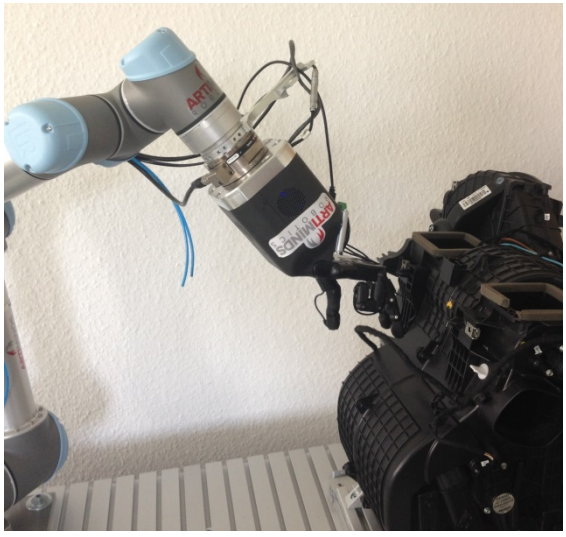
The overall grasping task can be divided into sub-tasks, each concerning different components. Therefor, the *scheduler* offers the functionality to create and add sub-tasks and

to automatically execute them in sequence. Depending on the executing task, the *scheduler* will interact with the respective component, e.g. to call the *path_mover* service to perform a specific grasp with the ShadowHand. Each task will either return a success or a failure code indicating whether a task was successful or if an error occurred during its execution. In case of a failure, the automatic execution will stop immediately to prevent further damage to the hardware components. Alternatively, the execution can also be performed step by step by manually selecting the next task to execute.



4.4) Finger motion control

The following chapter describes the act of grasping the cable in action by the robot shadow hand-lite. After the camera locates the cable and the point by which the shadow-hand should grasp, the robot arm combined with the shadow hand-lite moves to a calculated point.



On the picture, you can see the robot arm, which has reached the right position to grasp the cable at the calculated point. For grasping the cable, the shadow robot hand needs to change the position of the finger joints. A new grasp has to be taught before beginning the whole process of threading the cable into the hook. For teaching a grip, the finger-teach program, described in Section 4.6, is used, with which it is possible to save the position of every finger. For that you need to bring each finger of the robot-hand to the desired position. For changing the position of the finger joints you have to run the joints with position controllers. In the need of grabbing something with the humanoid robot hand you have to switch the controllers to effort controllers. If the joints are running in effort control the finger is able to push with a defined force against objects. That is the reason why the controllers of the fingers run in position control until the finger-sensors feel something or the finger joints have reached their position. If the finger-sensors feel pressure on their tips some defined joints of the hand-lite change to effort control and the other joints stay in position control. After gripping a cable and changing the different controllers from position control to effort control we experimentally found out which controllers need to switch their status and how strong the effort for the effort controllers should be. This does not have to be the best solution for the change between position and effort control. For this reason we do more research work for finding better solutions for gripping. This research deals with searching for the optimal position of the joints and finding out which controllers have to change and how strong they should be for the best grip. After a successful performance of the robot arm the controllers from the robot hand can switch from effort control to position control for opening the hand without overtaxing. Afterwards the robot-hand can start a new task.

4.5) Arm motion teaching

Arm motion teaching is performed by using the ArtiMinds RPS in it standard online teach-in mode.

4.6) Finger motion teaching

Teach graph

The Teach graph tab is used to teach a graph. The result will be a .graph file which can be used for imitating. See figure screens/tab-teach-graph.png.

Main

- Graph name: The name of the graph and also the corresponding .graph file. The graph name must be alphanumeric, but it may contain “-” and “_” in between letters and numbers. If a graph already exists, it can be overwritten (a warning message will appear in this case).
- Cable: The information of the cable.
  - Use cable: A checkbox to specify whether a cable should be used for this graph or not. The following bullets are only available when this checkbox is checked (they are grayed out if it is unchecked).
  - Position (x|y|z): The position of the cable with x, y and z coordinates.
  - Orientation (w|x|y|z): A quaternion w, x, y, z specifying the orientation of the cable.
  - Distance|Width: Distance to the cable and width of the cable.
- Nodes: A list of nodes which are relevant for this graph.

◦ A node can be added by selecting a node from the dropdown and pushing the Add selected button.
◦ A node can be removed by selecting a node from the list and pushing the Remove selected button.
◦ Please be aware that the order of the nodes in the list is arbitrary. The successor of a node is determined by the Next node property described in Teach node.
• Start node: Select a node from this dropdown to specifiy the node to start from. Please be aware that this dropdown is populated by the nodes specified in the Nodes list. This means that the Start node must be one of the added nodes.
Tools
• Save graph: Saves the graph. If the graph with the name specified in Graph name does already exist, a modal will appear asking the user whether the existing file should be overwritten or not.
• Reset: Resets all input fields (empties them).
Example
A user wants to teach a graph for the hand to form a fist. For this he has previously created the node file fist.node. This step by step guide will show how to accomplish this task.
Enter the name of the graph: Input fist in Graph name.
He does not need a cable to form a fist, so he leaves the checkbox for Cable unchecked.
From the dropdown below the Nodes list, he selects the node fist and pushes the Add selected button.
Since fist is the only node for this graph, there is only one possibility for the Start node. He therefore chooses fist as the Start node from the dropdown.
The user now pushes the Save node button to save his work.

Imitate

The *Imitate* tab is used to review and test nodes and graphes. See figure screens/tab-imitate.png.

Main

• *Imitatables*: A list of imitatables (i.e. nodes and graphes). An imitatable can be selected from the list.

Tools

• *Imitate selection*: Imitates the imitatable selected in the *Imitatables* list.

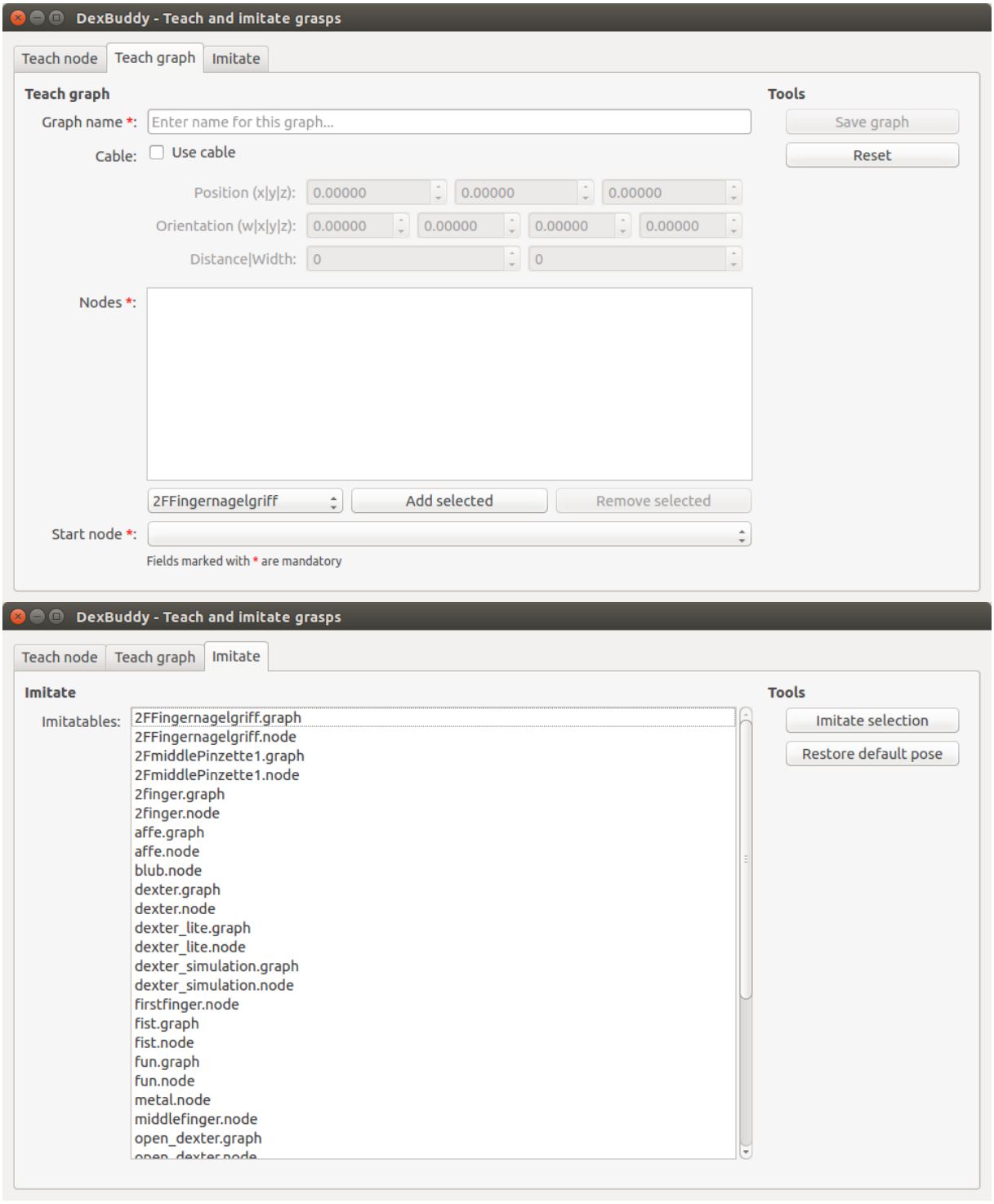
• *Restore default pose*: Restores the default hand state (a flat hand).

Example

The user has previously created a graph to from a fist (**fist.graph**). He now wants to test if it works as expected. This step by step guide will show how to accomplish this task.

1. The user selects **fist.graph** from the list of *Imitatables*.

The user pushes the *Imitate selection* button and reviews the result. (Alternatively, he could have double-clicked **fist.graph** from the list of *Imitatables*.


DexBuddy - Teach and imitate grasps
Teach node
Teach graph
Imitate
Teach graph
Graph name *:
Enter name for this graph...
Cable:
Use cable
Position (x|y|z):
0.00000
0.00000
0.00000
Orientation (w|x|y|z):
0.00000
0.00000
0.00000
0.00000
Distance|Width:
0
0
Nodes *:
2FFingernagelgriff
Add selected
Remove selected
Start node *:
Fields marked with * are mandatory
Tools
Save graph
Reset
DexBuddy - Teach and imitate grasps
Teach node
Teach graph
Imitate
Imitate
Imitatables:
2FFingernagelgriff.graph
2FFingernagelgriff.node
2FmiddlePinzette1.graph
2FmiddlePinzette1.node
2finger.graph
2finger.node
affe.graph
affe.node
blub.node
dexter.graph
dexter.node
dexter_lite.graph
dexter_lite.node
dexter_simulation.graph
dexter_simulation.node
firstfinger.node
fist.graph
fist.node
fun.graph
fun.node
metal.node
middlefinger.node
open_dexter.graph
Tools
Imitate selection
Restore default pose