Grant Agreement number: 288899

Project acronym: Robot-Era

Project title: Implementation and integration of advanced Robotic systems and intelligent Environments in real scenarios for ageing population

Funding scheme: Large-scale integrating project (IP)

Call identifier: FP7-ICT-2011.7

Challenge: 5 – ICT for Health, Ageing Well, Inclusion and Governance

Objective: ICT-2011.5.4 ICT for Ageing and Wellbeing

Project website address: www.robot-era.eu

D4.3

Final Domestic robotic platform prototype for the second experimental loop

Due date of deliverable: 31/08/2014
Actual submission date: 16/12/2014

Start date of project: 01/01/2012 Duration: 48 months

Organisation name of lead contractor for this deliverable: UHAM
Deliverable author: N. Hendrich, H. Bistry, F. Cavallo, T. Langner

Version: [1.]

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Service)	
RE	Restricted to a group specified by the consortium (including the Commission Service)	
CO	Confidential, only for members of the consortium (including the Commission Service)	

Revision History

Date	Rev.	Change	Author
2015.01.31	1.4	updated for 2015.02 Peccioli integration week	N. Hendrich
2015.01.21	1.3	description of new design and cover	I. Mannari
2014.12.22	1.2	updates after 2014.12 Angen integration week	H. Bistry
2014.12.18	1.1	document exekutor/action_exekutor and the meta-tuples PEIS interface	N. Hendrich
2014.12.15	1.0	revision and final version for submission	N. Hendrich
2014.11.08	0.4	revision after Ancona meeting	N. Hendrich
2014.08.31	0.3	updates for initial submission	N. Hendrich
2014.06.02	0.2	track changes from D2.6 and D2.8	H. Bistry
2014.04.01	0.1	created	N. Hendrich

Future Updates

This handbook documents the current state of the Robot-Era *Domestic Robot* including all main aspects of the robot hardware, software, and the integrating services architecture with connection to the PEIS middleware at the time of writing. Future updates and developments are expected to track the experiences gained with the robot during the second experimental phase of the project. Updated releases of the handbook will be uploaded to the project repository as the software and services evolve.

Contents

Executive Summary **1**

1 Overview **3**

2 Hardware **7**
2.1 Concept and General Robot Design 7
2.2 Updated design for the second experimental loop 8
2.2.1 New robot cover and design changes 8
2.2.2 Tilting handle 11
2.2.3 Updated sensors and computers 12
2.3 SCITOS G5 platform 13
2.3.1 SCITOS-G5 mobile advanced 13
2.3.2 Sick S300 safety laser-scanner 14
2.3.3 Hokuyo URG-04LX laser-scanner 14
2.3.4 Ultrasonic sensor ring 14
2.3.5 Platform mounted camera 14
2.4 Kinova Jaco manipulator 15
2.4.1 Kinova Joystick 15
2.5 Sensor head 16
2.5.1 Asus XtionPro 16
2.5.2 Camera DFK 31BF03 16
2.5.3 Camera DFK 21BF04 16
2.5.4 Directed Perception D46 pan-tilt unit 16
2.6 Human-Robot interface 17
2.6.1 Tablet-based Interface 17
2.6.2 Teleoperation Interface 17
2.6.3 Tilting Handle for Walking Support 17
2.7 Safety Features 18
2.7.1 Emergency-stop switches 18
2.7.2 Bumper ring 18
2.7.3 Safety laser scanner 18
2.8 Workspace analysis 19

3 Software **21**
3.1 Overview 21
3.1.1 ROS 24
3.1.2 Domestic Robot URDF 27
3.1.3 Coordinate-systems and tf 28
3.1.4 Launching the Domestic Robot 31
3.1.5 Running ROS on Multiple Computers 36
3.1.6 Teleoperation interface 37
3.1.7 Control Center teleoperation interface 38
3.1.8 Robot calibration 39
3.2 Robot localisation and navigation 40

3.2.1 MIRA . . . 40
3.2.2 Cognidrive . . . 41
3.2.3 MIRA-ROS bridge . . . 43
3.3 Sensing and Perception . . . 47
3.3.1 Overview . . . 47
3.3.2 Pan-Tilt Unit . . . 47
3.3.3 Camera System and Image Processing . . . 47
3.3.4 XtionPro RGB-D Camera and Point-Clouds . . . 52
3.3.5 MJPEG-Server 2 . . . 52
3.3.6 Object Detection and Pose Estimation . . . 54
3.3.7 Visionhub . . . 54
3.3.8 SIFT-based object recognition . . . 56
3.3.9 AprilTag Marker detection . . . 58
3.3.10 Tabletop segmentation . . . 59
3.3.11 Human Detection and Recognition . . . 60
3.4 Manipulation . . . 62
3.5 Kinova Jaco API and ROS-node . . . 63
3.5.1 Jaco DH-parameters and specifications . . . 63
3.5.2 Jaco Joystick and teleoperation . . . 63
3.5.3 Jaco .NET API . . . 64
3.5.4 Jaco ROS integration . . . 67
3.5.5 Jaco gripper . . . 70
3.5.6 Inverse Kinematics . . . 71
3.5.7 Traps and Pitfalls . . . 72
3.5.8 Collision map processing . . . 72
3.6 MoveIt! framework . . . 73
3.7 Manipulation Action Server . . . 77
3.8 Manipulation Tasks in Robot-Era Scenarios . . . 78
3.8.1 Domestic to Condominium Object Exchange . . . 78
3.8.2 Basket objects . . . 80
3.8.3 Cleaning Tasks . . . 80
3.8.4 Object Handover Tasks . . . 81
3.9 Robot Watchdog . . . 84
3.10 Simulation . . . 86
3.10.1 Domestic Robot in Gazebo . . . 87
3.10.2 Notes and version compatibility . . . 88
3.11 PEIS Integration . . . 90
3.11.1 PEIS-ROS TupleHandler architecture . . . 90
3.11.2 Using actionlib and feedback functions . . . 90
3.11.3 Synchronisation . . . 91
3.11.4 Structured data . . . 93
3.11.5 Writing a new TupleHandler . . . 93
3.11.6 ActionExekutor . . . 94

4 Services **99**
4.1 Low-Level Services . . . 100
4.1.1 EmergencyStop . . . 101
4.1.2 GetCameraImage . . . 102
4.1.3 GetKinectImage . . . 103
4.1.4 GetLaserScan . . . 104
4.1.5 GetSonarScan . . . 105
4.1.6 MoveTo . . . 106
4.1.7 Dock and Undock . . . 107
4.1.8 MovePtu . . . 108
4.1.9 RetractJacoArm . . . 109
4.1.10 ParkJacoArm . . . 110
4.1.11 MoveJacoArm . . . 111
4.1.12 MoveJacoCartesian . . . 112
4.1.13 MoveitJacoCartesian . . . 113
4.1.14 MoveJacoFingers . . . 114
4.2 Intermediate Services . . . 115
4.2.1 DetectKnownObject . . . 116
4.2.2 DetectUnknownObject . . . 117
4.2.3 GraspAndLiftKnownObject . . . 118
4.2.4 SideGraspAndLiftObject . . . 119
4.2.5 TopGraspAndLiftObject . . . 120
4.2.6 PlaceObjectOnTray . . . 121
4.2.7 PlaceObject . . . 122
4.2.8 DropObject . . . 123
4.2.9 GraspObjectFromTray . . . 124
4.2.10 HandoverObjectToUser . . . 125
4.2.11 HandoverObjectFromUser . . . 126
4.2.12 PourLiquidMotion . . . 127
4.2.13 MoveHingedDoor . . . 128
4.2.14 LookAt . . . 129
4.2.15 DetectPerson . . . 130
4.2.16 TrackPerson . . . 131
4.2.17 RecognizePerson . . . 132
4.3 High-level Services . . . 133
4.3.1 WalkingSupport . . . 134
4.3.2 SwipeSurfaceService . . . 135
4.3.3 CleanFloorService . . . 137
4.3.4 CleanWindowService . . . 138
4.3.5 CleanTableService . . . 139
4.3.6 BringFoodService . . . 140
4.3.7 CarryOutGarbage . . . 141
4.3.8 LaundryService . . . 142

5 Software installation and setup **143**
5.1 Ubuntu 12.04 and ROS Hydro 143
5.2 Software Installation Paths 143
5.3 MIRA and CogniDrive 144
5.4 PEIS 144
5.5 Kinova Jaco Software 145
5.6 ROS Hydro 145
5.7 GStreamer Libraries 146
5.8 Robot-Era ROS stack 146
5.9 Network Setup and Wired/Wifi Bridge 147
5.10 Remote Roslaunch Setup 149
5.11 Robot calibration 152

6 Summary **153**

References **154**

List of Figures

1 Domestic Robot prototype . . . 4
2 Designer concept of the Domestic Robot . . . 7
3 Sketch of the Domestic Robot with handle . . . 8
4 Updated design of Domestic Robot . . . 9
5 Updated colour scheme . . . 10
6 New soft headgear . . . 11
7 Updated tilting driving handle . . . 11
8 Updated Scitos-G5 platform . . . 12
9 Jaco arm and gripper . . . 15
10 Robot arms evaluated for the Domestic Robot . . . 19
11 Arm workspace analysis . . . 19
12 Software Architecture of the Domestic Robot . . . 22
13 Robot-Era ROS stacks and packages . . . 26
14 URDF of the Domestic Robot . . . 27
15 Coordinate frames of the Domestic Robot . . . 29
16 Base coordinate frames of the Domestic Robot . . . 30
17 Coordinate frame graph of the Domestic Robot . . . 30
18 Default rviz configuration . . . 35
19 Sony Sixaxis joystick . . . 37
20 MIRA-Center . . . 44
21 ROS-Cognidrive bridge . . . 45
22 Updated perception pipeline . . . 48
23 Object Coordinate Systems . . . 55
24 Object manipulation scene . . . 57
25 Visualisation of detected objects . . . 57
26 AprilTag marker detection . . . 59
27 SVM-based people detection . . . 61
28 Person detection and recognition . . . 61
29 Jaco arm home and retract positions . . . 64
30 Jaco finger positions vs. object size . . . 70
31 MoveIt! overview . . . 74
32 Reference objects . . . 78
33 Doro-Coro object exchange . . . 79
34 Grasping the basket object . . . 80
35 Setup for cleaning tasks . . . 81
36 Robot-to-human object handover . . . 82
37 States in robot-to-human object handover . . . 83
38 Multi-robot simulation in Gazebo . . . 86
39 Visualisation of multi-robot simulation . . . 87
40 Gazebo simulator architecture . . . 88
41 Actionlib client-server interface . . . 91
42 ROS actionlib states . . . 92
43 tuple_handler.h . . . 95
44 DemoService.h . . . 96

45 DemoService.cc (1/2) 96
46 DemoService.cc (2/2) 97
47 PEIS exekutor meta-tuples 98
48 Software installation paths 144
49 On-board computer network 147
50 Example network setup file with WiFi/wired Ethernet bridge. 148
51 Example *env.sh* environment file for remote roslaunch. 151
52 Example roslaunch *machine* definitions and usage. 151

Executive Summary

This robot handbook is part of the public deliverable *Final Domestic robotic platform prototype for the second experimental loop* (D4.3) of the European integrated research project Robot-Era, www.robot-era.eu. It documents the hardware, software, and services architecture of the *Domestic Robot*, a state-of-the-art indoor service robot designed to help elderly people with their daily activities. Together with the companion reports of the project (D3.3, D3.4, D5.3, and D6.3), the document describes the robot systems and ambient intelligence architecture to be used in the upcoming second experimental phase of the project.

The report is a revised and extended version of the original robot handbook D4.2 [58]. Based on user-feedback and experiences gained during the first experimental phase of the project in 2013, several S/T requirements were identified and documented in deliverable D2.8 [52]. The corresponding changes and improvements have all been implemented. The result is a more capable and more robust robot platform for the real end-user tests.

Hardware updates

- redesigned and improved outer appearance of the robot, motivated by analysis of the user-acceptance during the first experimental loop,
- minor hardware changes to improve robot capabilities, including updated arm and main-pillar positions and larger object transportation tray,
- additional sensors for manipulation tasks and robot docking,
- additional on-board computers for better performance.

Notable software improvements

- improved HRI and speech manager, see D5.3 [60],
- meta-tuples based PEIS interface (“exekutor”), see D3.4 [56],
- multi-modal object perception, combining PCL, SIFT, and AprilTags,
- PCL-based human tracking to complement the AmI user tracking,
- sensor-based robot-to-human object handover capability,
- collision-aware object manipulation based on the MoveIt! framework.

Improved Robot-Era services

- additional robot services implemented and exported to PEIS,
- reconfiguration of services using meta-tuples,
- Doro/Coro object exchange using custom objects and basket,
- improved implementation of laundry and object transportation tasks,
- initial version of cleaning tasks,
- services regression testing using multi-robot Gazebo simulation.


Robot
ERA

1 Overview

This handbook summarises the updated hardware, software, and service architecture of the *Domestic Robot*, short *doro*, of project Robot-Era. The report is a revised and extended version of the original robot handbook D4.2 [58]. It documents the changes made to the robot hardware and software based on the experiences gained during the first experimental loop of the project. For convenience, those parts of the text that have been added or are significantly changed are marked by vertical bars in the margin of the document.

Together with the companion reports D3.4 (Ambient Intelligence) [56], D5.3 (Condominium Robot) [60], and D6.3 (Outdoor Robot) [61], this handbook documents the overall system to be used and tested during the upcoming second experimental loop of the Robot-Era project.

See Fig. 1 on page 4 for a photo of the updated robot prototype. Only minor changes were made to the basic robot hardware. The main pillar with the head was moved backwards a bit to make room for the transport tray on the robot and the mount position of the arm was changed slightly. A new cover was designed for the robot based on user-feedback collected during the first experimental loop. The overall software architecture remains unchanged, but several improvements have been integrated to track developments of the ROS and MIRA frameworks, and to improve the robustness of the robot. Most of the development effort has concentrated on refining and implementation of the end-user driven *services* provided by the robot.

Please visit the project website at *www.robot-era.eu* for details about the Robot-Era project and its background and goals. The core objective of the Robot-Era project is to improve the quality of life and the efficiency of care for elderly people via a set of *advanced robotic services*. Going beyond the traditional concept of a single standalone robot, the project considers a set of different robots operating in a sensor-equipped intelligent environment, or *smart home*. Besides the actual design and implementation of the robot services, the project also monitors the feasibility, scientific/technical effectiveness and the social and legal plausibility and acceptability by the end-users.

Three robot prototypes are developed in the context of the project, each targeting different scenarios identified as relevant for the end-users in the initial phase of the project. The *outdoor robot* provides transportation, guidance, walking support and surveillance services, and the services of the *Condominium Robot* centre around transportation tasks. The *Domestic Robot* is a classical indoor service robot equipped with advanced sensors for environment perception and a robot arm and gripper to manipulate household objects.

As described in chapter 3.1, the overall software architecture for the Robot-Era services consists of several layers, where the PEIS system provides the *ambient intelligence* (*AmI*) that manages the sensor-network and the different robots in the system. The end-user requests services from the whole system, which implies that no advanced human-robot interface is required for the Domestic or Condominium Robots. Details of the *AmI* layer and software have been documented in the project reports D3.1, D3.2, and D3.3 [53, 54, 56].

Figure 1: The Domestic Robot combines the SCITOS-G5 differential-drive platform, the Kinova Jaco 6-DOF arm with integrated 3-DOF hand, and a pan-tilt sensor-head. Sensors include one front and one rear laser-scanner, two high-res cameras equipped with different lenses, and the Asus XtionPro RGB-D camera. Voice input is possible via the XtionPro microphones or additional microphones. The handle on the right carries the iPad tablet-computer that provides a touch-screen interface and additional sensors.
The picture shows the Y3 redesign of the robot with the new cover, foldaway moving handle, and the object transportation tray.

Outline

This report is part of the public project deliverable D4.3, which consists of the actual *Domestic Robot prototype*, and provides the tutorial and handbook information about the physical robot and the software developed to control it. The handbook is structured into three main chapters, followed by reference information about software installation and setup:

- Chapter 2 summarises the *hardware* of the Domestic Robot, starting with a short summary of concept studies and the aspects of user-friendliness and acceptability that guided the design of the robot in section 2.1. Section 2.3 describes the SCITOS-G5 mobile differential-drive platform selected as the mobile base of the *Domestic Robot*, while section 2.4 summarises key data of the Kinova Jaco 6-DOF arm and integrated gripper selected as the manipulator on the robot. Section 2.5 describes the movable (pan-tilt) sensor head equipped with one Asus XtionPro RGB-D depth-camera and two standard RGB cameras. The head also includes microphones as part of the XtionPro device. Section 2.6 sketches the hardware devices used to control the robot; a standard iPad tablet PC provides a friendly user-interface to the end-users, while a joystick interface allows expert users to tele-operate the robot.

- Chapter 3 describes the *software architecture* designed for the Domestic Robot, which is based on the ROS middleware, and the integration into the intelligent environment. A general overview of the software is presented in section 3.1, followed by sections describing the key components of a service robot, namely *navigation* 3.2, *environment and object perception* 3.3, *object manipulation* 3.4. Additional information about the Kinova Jaco robot arm is collected in section 3.5 and the integration into the MoveIt! motion planning framework is described in 3.6. The complete ROS/Gazebo simulation model of the robot is explained in 3.10. Finally, section 3.11 motivates and explains the design of the PEIS-ROS bridge, which integrates the Domestic Robot into the ambient intelligence and the multi-robot planner of the smart home.

- Chapter 4 provides the complete *list of all services* of the Domestic Robot. The list is subdivided into three groups of increasing complexity, starting with a set of *basic robot skills* in section 4.1. These skills are then combined and nested to provide the *intermediate services* described in section 4.2. These services form the basis for the first experiment phase of project Robot-Era. The last section 4.3 summarises the advanced *high-level robot services* that form the core of the scenarios developed by the project. Each service corresponds to a complex task that requires autonomous operation of the Domestic Robot in close interaction with the ambient sensor network and the end-users.

- Chapter 5 provides reference material about download, installation, and set-up of the major software components for the Domestic Robot.

- The handbook concludes with a short summary and the list of references.

Domestic Robot Updates for the 2nd Experimental Loop

Following analysis of the first experimental loop [62], no major changes were required to the basic robot design. However, evaluation of the user-feedback resulted in a couple of changes to improve the visual appearance and acceptability of the Domestic Robot:

- new robot cover and brighter colours. The material is now robust plastic, easier to clean and more rugged than the 2012 soft cover,
- enlarged removable tray for object transportation,
- different mount positions for the Jaco arm and the main robot column,
- updated robot head with a "cap" for friendlier appearance,
- additional gyroscope to improve localisation and navigation,
- additional camera to help precise-docking and manipulation tasks,
- additional on-board computers for extra I/O and performance.

See deliverable D3.4 [56] for details of the updated ambient intelligence software and the speech/tablet based user interface. Several updates of the Domestic Robot software were designed and implemented during 2014, in order to improve the usability and the robustness of the robot software during the 2nd experimental loop. The following list summarises the most significant changes to the Domestic Robot software:

- refined service-architecture with better feedback and status messages from the robot,
- supervisor node on the robot to schedule and manage incoming service requests, to reduce the amount of robot state in the AmI layer and planner,
- watchdog node on the robot to monitor all required ROS nodes, to protocol failures, and to restart nodes automatically when required and possible,
- partitioning of ROS nodes between several computers (on-board and off-board) for load-balancing and better performance,
- rewritten launch files and new helper programs for fast software start-up and easier re-start during the experimental tests,
- integration of the AprilTags fiducial marker [1] recognition software into the perception architecture,
- improved perception architecture with parallel execution of the point-cloud clustering, SIFT-based object detection, and fiducial marker based detection pipelines,
- improved manipulation capabilities including pick&place of known object, object transportation, use of the robot tray,
- support for object handover tasks using visual and force sensing,
- initial implementation of cleaning motions,
- improved (marker-based) docking of the Domestic and Condominium Robots,
- support for Kinect-based 3D collision-aware navigation,
- preparation of a simplified tele-operation interface for the *real testing* (in user's homes) of the second experimental loop.

2 Hardware

This chapter documents the hardware components of the Domestic Robot. See Fig. 1 on page 4 for a photo of the completed robot. Please refer to project report D4.1 [57] and two conference papers [7, 10] for additional details and the explanation of the design process including the selection of the sensors and the Jaco manipulator.

2.1 Concept and General Robot Design

See Fig 2 for an early artists' rendering of the Domestic Robot. The robot itself is a fairly typical mobile service robot, combining a wheel-based mobile platform with a robot arm and gripper to perform manipulation tasks. The sensor setup is also fairly common, with laser-scanners on the mobile base for localisation and obstacle-detection, and a moving head with cameras and microphones.

Unlike many service robot prototypes, which are designed for industrial environments or research laboratories, the Domestic Robot is meant to help elderly people in their daily lives, moving and operating in close proximity with the users. End-user acceptance of the robot is therefore a major concern of the project, and studies and questionnaires have been used to characterise the properties required for the robot [49]. Several aspects were identified as crucial for the acceptability of a service robot in elderly care scenarios, including the *affordances* offered by the robot, the *safety guarantees*, and last but not least the *aesthetics* and *friendliness*. In short, the robust must be capable of the tasks expected by the users, but must be non-obtrusive and integrate into the living environments.

One direct result of this study was the selection of the manipulator. Most robot arms are designed for performance and have a clearly industrial look, even if the outer appearance is

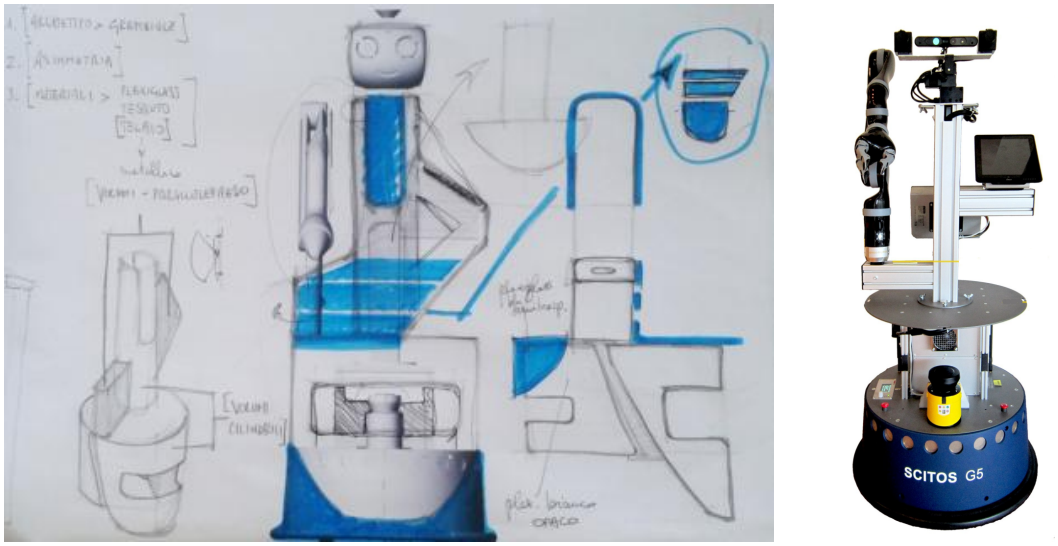


Figure 2: Designer concept of the Domestic Robot and a photo of the first prototype. Without the cover, the main hardware components are clearly visible.

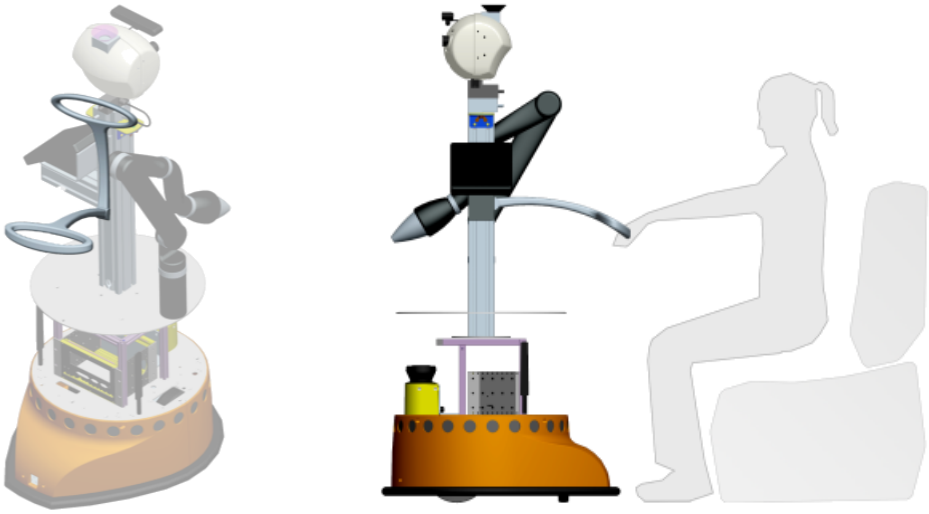


Figure 3: Concept sketches of the Domestic Robot with a tilting handle.

kept smooth (e.g. KuKA light-weight robot). In addition to its certification for wheel-chair tele-operation and therefore safe operation around humans, the smooth outer appearance and low operational noise of the Kinova Jaco arm were clear selling points. The first two prototypes of the Domestic Robot have been equipped with only one arm each, but the design and payload of the Scitos-G5 platform would also allow the installation of two arms on the robot. This would enable the robot to perform a larger set of manipulation tasks, but at a significantly higher price point and with much more complex software.

2.2 Updated design for the second experimental loop

The overall hardware of the robot performed well during the first experimental loop [62], but a couple of minor changes were implemented after analysis of the user-evaluations. See deliverable D2.8 [52] for the description of updated requirements and the proposed list of changes to the robot systems.

2.2.1 New robot cover and design changes

The most obvious change of the robot is the new design and cover, motivated by analysis of the user-feedback from the first experimental loop. A very important issue, already known and strongly confirmed by the answers of the interviewees, concerns the too large dimensions of the two indoor robots, and especially of Domestic Robot: according to most of the people interviewed, in fact, the robot wouldn't able to integrate into their homes because of its large size. To the question "*If you were a technical developer and you could change anything you want regarding the robot, what are the things you would change?*", many interviewees gave answers such as: "*I would reduce the size of the robot*" or "*bulky size*" or "*smaller dimensions*" or even "*change appearance, it looks obese (bottom part)*".

Figure 4: Photos of Domestic Robot for the first experimental loop (left) and sketch of the updated design for the second experimental loop with the new arm position and changed iPad tablet orientation highlighted (right).

Since it is not possible to decrease the overall size of the robot for technical reasons (it would be impossible to change or reduce the size of the mobile base or the arm of Domestic Robot), we tried to lighten its outer face. For example, to the question: "*Do you have any doubt about the look of the robot?*" one interviewee said "*I do not like the cover below, the material is an ugly dress*". To this effect, the covers of both robots have been redesigned with more rounded and enveloping shapes, in order to mitigate the sense of dissimilarity shown by the interviewees between the lower and the upper part of the two robots. We tried to achieve this by reducing the use of soft parts and to realise the two covers in a single shell front and rear by the process of thermoforming plastic (ABS and PMMA) with specially shaped wooden mold. In this way, as shown in figure 4 the robot has a more linear and compact appearance, in agreement with the feedback detected by interviews with users.

Also, the internal structure of the robot has been changed in order to succeed in what has been explained so far. The middle plate of Condominium Robot has been redesigned according to the shape of the mechanical system of objects exchange (see figure 5); the Domestic Robot has adopted the same new form of plate, to simplify the production and standardisation of the various components of the two robots. The central pillar of the frame of the Domestic Robot has been moved to the back, the same as it is currently shaped the Condominium Robot. In this way we obtained a tray (removable and washable as in the first version of the robot) with a form more congenial to the movements of the arm, see figure 4.

In the user evaluations, the Condominium Robot received the better feedback probably because of its more linear and symmetrical appearance. Some interviewees considered the arm size of the Domestic Robot too big and the colour too dark; they also pointed out that two arms would have been better than one. For example, to the question: "*If you were a technical developer and you could change anything you want regarding the robot, what are the things you would change?*" two users answered "*arm more stable and not black, two*

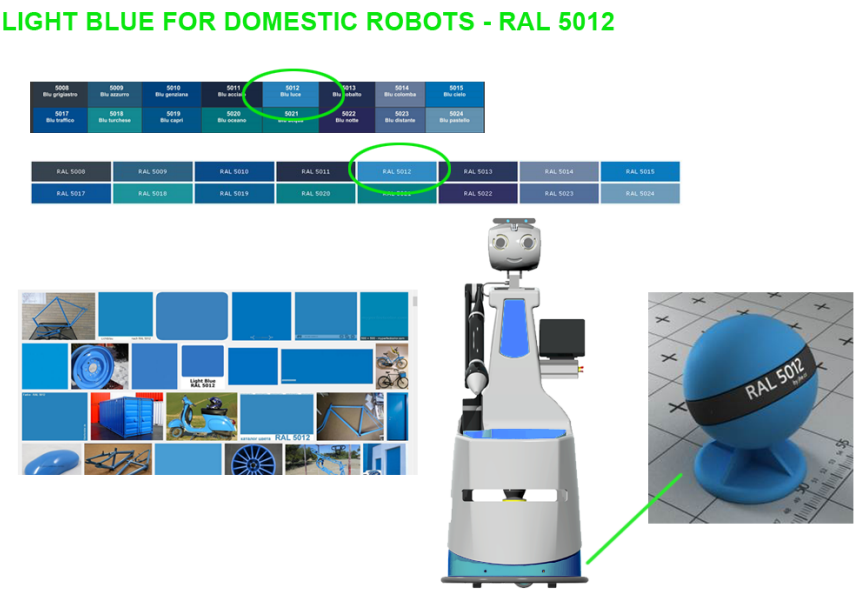


Figure 5: New and lighter colour scheme based on RAL 5012 (right).

arms, because it seems amputee" and "*tablet front, put both arms*". To the question: "*Do you have any doubt about the look of the robot?*" some answers were: "*arm too dark*" or "*arm bulky*" or even "*better for Doro two arms and front screen*". We contemplated the idea to colour the arm of a lighter colour but this is very difficult from a technical and economic point of view; we also discarded the idea of inserting a second arm without functions on the opposite side of the robot. Finally we were able to redesign the new cover only trying to incorporate the arm in a better way. Moreover, as in the first version of the robot, we tried to balance the figure with the presence of the tablet on the other side of the robot, but rotated 90 degrees and, therefore, with the interface visible from the front (see figure 4).

Another reason that explains some preference for the Condominium Robot is probably the yellow colour of the tie and of the tray, namely a warmer colour and garish rather than the blue used for same parts of Domestic Robot. Regarding this fact, during interviews some respondents made statements like: "*Doro colours too gloomy*" and "*´Doro should have brightest colours*". For this reason, we decided to keep the grey colour for the main cover and for the head, the yellow colour for the tie and the tray of Condominium Robot and change the colour of the tie and the tray of Domestic Robot: according to all partners, the final choice was a brighter and lively colour like light blue (figure 5).

Finally, the Asus XtionPro camera has been fixed better than before (figure 6). We also created a sort of coloured headgear. We thought this additional accessory to better characterise the Domestic Robot. In the design of the first version of the robot we tried to combine form and function deciding to *dress it up* with a uniform: the housemaid. Despite our best efforts, during the trial, the users who participated have not jumped at these aspects. For this reason and because it is important that the appearance of the robot communicates its basic nature to reach the highest degree of acceptability, we thought to develop one headgear ad hoc with soft materials (figure 6).

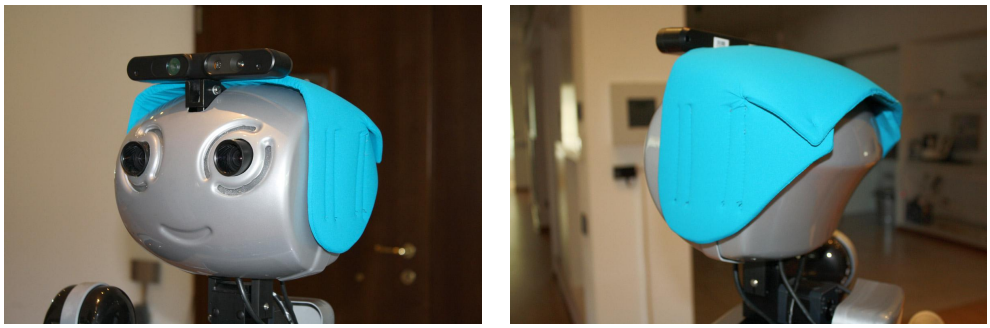


Figure 6: Photos of the redesigned robot head with improved Asus XtionPro camera holder and the soft-cloth headgear.

2.2.2 Tilting handle

While the original design concept already included a tilting handle (see figure 3), a simple fixed handle was used during the first experimental loop. The handle is designed to helping users when trying to get up from a chair or the bed, and is the means to grasp and control the robot during the indoor walking-support scenario, see figure 3.

While the fixed handle provided a simple and very robust solution, the handle reduced the ability of the robot to navigate in narrow spaces and increased the risk of collisions or accidents when the handle was not used for walking-support.

Therefore, a new handle was designed and implemented that is aesthetically and functionally compatible with the rest of the covers of the Domestic Robot. The handle remains in vertical position when now used, thanks to a specially designed return spring (figure 7). The main components of the handle are square aluminium rods (20x20 mm^2) which are clothed with removable, washable and customisable foam and fabric.

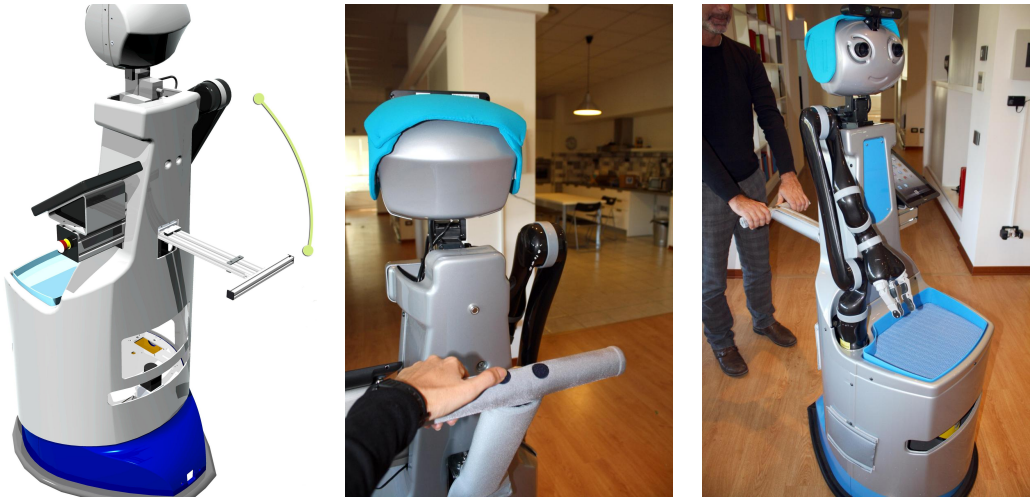


Figure 7: Design sketch and two photos of the redesigned tilting handle of Domestic Robot.

Fully autonomous operation of the robot is not planned for the walking-support scenario. Instead, switches or strain-gauges are installed on the handle, allowing the user to drive the robot around. The sensors for moving the robot by the user are inserted within the foam. Pushing the buttons, the user can drive the robot forward, left and right. During driving, the robot software checks the laser-scans from the robot sensors and automatically stops the robot to avoid collisions.

2.2.3 Updated sensors and computers

In addition to the changes of the outer appearance of the Domestic Robot, three minor functional changes have been applied.

- A gyroscope sensor was added to the Scitos-G5 platform to continuously measure and correct the robot orientation. The sensor drastically improves the localisation precision of the robot since most of the localisation errors are caused by wrong rotation readings caused by drift and wheel-spin on slippery floor.
- Two additional computers (Intel NUC D5250WYK) were installed on the robot to provide extra CPUs and memory for the complex robot software. The overall system performance is increased roughly 3× over the original (single-PC) design. The NUC computers also provide USB3 and USB2 connections for additional peripheral devices.
- An additional camera has been installed on the side of the robot, looking towards the floor. The camera is used for precise docking and manipulation tasks, e.g. detection and localisation of objects on the floor and for the *Garbage Transportation* scenario.

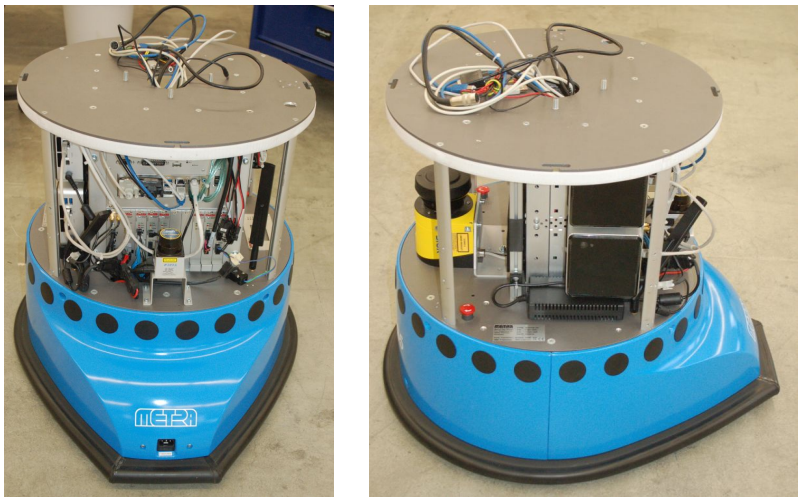


Figure 8: Updated Scitos-G5 robot platform with gyroscope sensor and two compact Intel NUC modules to increase processing power.

2.3 SCITOS G5 platform

2.3.1 SCITOS-G5 mobile advanced

Differential-drive robot base:

- 582 mm x 7537 mm x 617 mm (H x L x W)
- two driving wheels, one caster wheel
- high-torque EC gear motors
- 24 ultrasonic range finders, range 15..300 cm, 100 ms sampling time
- gyroscope sensor
- bumper ring with mechanical emergency stop
- tools and slot nuts for fast mounting of additional devices

Batteries and power-supply:

- lead-acid gel batteries, 24 V, 1.008 Watt-hrs
- integrated battery charger
- floor-contacts for automatic recharging
- on-board power supply: 2x 24 VDC (unregulated), 2x 12 VDC, 2x 5 VDC

Main Computer:

- Industrial embedded PC, Intel QM57 Express chip-set
- CPU: Intel Core-i7-620M (2 x 2,66 GHz, max. 3.333 GHz, 4 MB Cache)
- RAM: 1 x 2 GB PC8300 SODIMM, HDD: at least 250 GB, SATA II
- WiFi IEEE 802.11a/b/g, 4x SATA (3 free)
- 1x PCI (occupied), 1x Mini-PCI-E (occupied), 1x PCI-E(x1)(free)
- 1x VGA, 2x DVI/DVI-D, 1x 18/24 bit LVDS
- 2x 1000 BaseT Ethernet, 7x USB 2.0, 3x Firewire
- 1x PS/2, 1x LineOut, 1x Line-In, 1x Microphone, 2x RS232
- 15” touch-screen TFT display, 1024x768 pixels (unused in Y3 robot setup)
- Linux Ubuntu 12.04 / Fedora 14 (pre-installed and configured)
- MIRA and CogniDrive for navigation and localisation

Additional Computers:

- 2x Intel NUC D5250WYK
- CPU: Intel® Core™ i5-4250(2 x 1.3 GHz, max. 2.6 GHz).
- RAM: 8 GB, HDD: 80 GB, infrared sensor
- 1x VGA, 1x DP/HDMI, 2x USB 3.0, 2x USB 2.0
- Linux Ubuntu 12.04
- ROS software for perception and manipulation planning

2.3.2 Sick S300 safety laser-scanner

- scanning range 270 deg (reduced to 180 deg by the cover)
- angular resolution 0.5 deg
- distance measuring range up to 30 m
- support for user-defined safety zones
- Linux driver

2.3.3 Hokuyo URG-04LX laser-scanner

- scanning range 270 deg (reduced to 180 deg by the cover)
- angular resolution 0.35 deg
- distance measuring range from 0.2 m to 6 m
- USB connection, Linux driver

2.3.4 Ultrasonic sensor ring

- scanning range 360 deg
- distance measuring range from 0.15 m to about 3 m
- data more noisy and less reliable than laser-scanner data
- sensors are disabled for the 2nd experimental loop

2.3.5 Platform mounted camera

- standard, small high-res web-cam (Logitech C910)
- 5 MPixel @ 10 Hz, USB-2 connection
- object detection on the floor (especially boxes)
- used for manipulation scenarios

2.4 Kinova Jaco manipulator

Figure 9: The 6-DOF Jaco arm with integrated 3-finger gripper, and a close-up of the three-finger gripper.

- 6-DOF robot arm
- 3-DOF robot gripper
- 9 high-torque DC motors, planetary gears
- max. payload 1.5 kg, 50 W power
- cartesian speed limited to 20 cm/sec. for safety
- underactuated fingers close around small objects
- user-specified home and retract positions
- no-brakes, robot falls down on power-loss
- USB connection
- Windows .NET drivers and application code
- Linux Mono wrapper for Kinova DLLs

2.4.1 Kinova Joystick

- robust 3-axis joystick (x,y,twist)
- 2-axis or 3-axis control modes
- intuitive cartesian (x, y, z) hand translation
- intuitive cartesian (ϕ, ψ, θ) hand rotation
- drinking mode, user-specified IK params
- 2-finger and 3-finger grasping

2.5 Sensor head

2.5.1 Asus XtionPro

- PrimeSense RGB-D projector and
- 640x480 RGB
- 640x480 depth-image
- 30fps (colour)
- USB connection
- OpenNI driver

2.5.2 Camera DFK 31BF03

- 1/3” CCD
- 1024x768
- 30fps (mono), 15fps (colour), Progressive Scan
- IEEE1394 (DCAM 1.31)
- C/CS-mount

2.5.3 Camera DFK 21BF04

- 1/4” CCD
- 640x480 Progressive Scan
- 30fps (colour)
- IEEE1394 (DCAM 1.31)
- C/CS-mount

2.5.4 Directed Perception D46 pan-tilt unit

- payload 1.5 kg
- pan-range
- tilt-range
- RS-232 connection, 19200 b/s

2.6 Human-Robot interface

2.6.1 Tablet-based Interface

- Apple iPad3 tablet
- 2048x1536 pixel RGB touch-screen
- WIFI 802.11a/b/g
- menu-based selection of Robot-Era services
- image and video playback from robot cameras
- emergency stop for the robot(s)

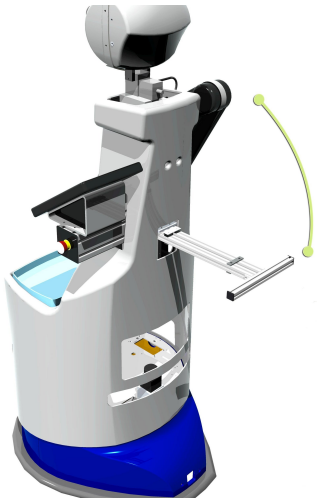
2.6.2 Teleoperation Interface

Sony PS3-Sixaxis controller

- 2 analog joysticks
- 4 analog buttons (trigger)
- 3-axis accelerometer
- 10 buttons
- wireless (Bluetooth) or cable (USB)
- *ps3joy* ROS stack

2.6.3 Tilting Handle for Walking Support

- custom design
- Tilting handle
- help the user to sit-down or get-up
- walking support
- integrated drive switches
- stop, forward, left, right
- ROS software checks laser-scanners
- robot stops in front of obstacles



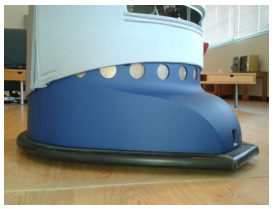
2.7 Safety Features

2.7.1 Emergency-stop switches

A red emergency switch button is located on right side of the Domestic Robot, under the tablet location. The switches currently only stops the SCITOS platform, not the PTU nor the Jaco.

2.7.2 Bumper ring

The bumper ring is located on the base of SCITOS platform. When the bumper is hit, the motor stops.



2.7.3 Safety laser scanner

The SICK S300 Safety Laser Range Finder (described in 2.3.2) is certified to detect obstacles including lying-down humans inside its working range.

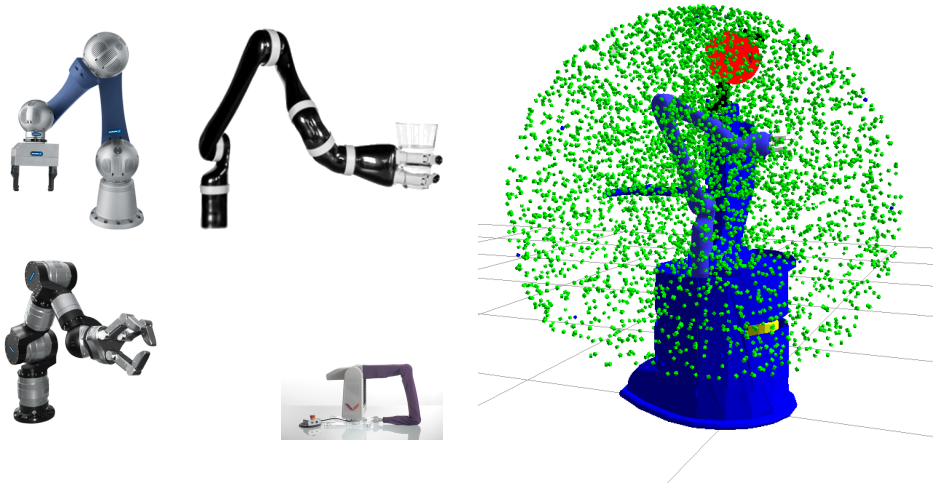


Figure 10: Left: The robot arms evaluated for the Domestic Robot due their combination of weight, payload, reach, mobile use, and costs. (a) Schunk Powerball arm, (b) Kinova Jaco, (c) Schunk modular arm, (d) BioRob arm. Images courtesy of the vendors.
Right: Overall workspace of the Jaco arm on the Domestic Robot (ROS rviz). The arm is mounted so that the fingers can just reach the floor.

2.8 Workspace analysis

The robot workspace analysis performed during 2013 (figure 10) was repeated during 2014 to take the changed mount-position of the arm and the new robot cover into account. See figure 11 for a few examples. As the arm was moved backwards by about 15 cm with respect to the Scitos-G5 platform, the workspace in front of the robot has decreased correspondingly. The best reach is just to the right of the robot, and the figure shows the reachable workspace when trying to grasp objects from the floor and an example scenario suggested by project partner YOUSE.

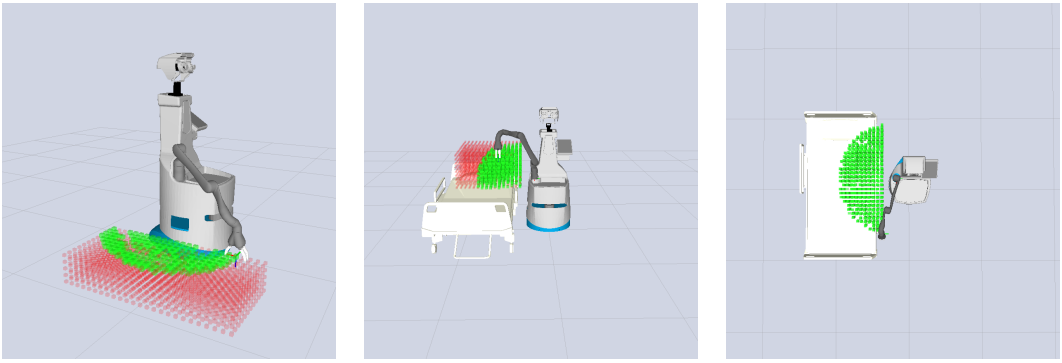


Figure 11: Examples of arm workspace analysis from the updated arm position.

3 Software

This chapter summarises the overall software architecture for the Domestic Robot and provides tutorial information about all key software components. For detailed information including the complete API and implementation notes, please refer to the Robot-Era wiki and the SVN repository. See chapter 5 for detailed instruction on how to download, install, and set-up the major components of the Domestic Robot software.

First, section 3.1 presents an overview of the software architecture, which is built around the ROS *robot operating system* [35] framework and communication model. A short introduction of the major aspects of ROS is given in section 3.1.1 while section 3.1.2 summarises the URDF robot model created for the Domestic Robot.

Next, section 3.2 explains the main software components for localisation and navigation of the mobile robot, which uses a combination of ROS and the MIRA/CogniDrive software. A MIRA-ROS bridge developed within the project creates the seamless interface between CogniDrive and ROS.

Section 3.3 describes the sensing and perception architecture, including the low-level interfaces to the cameras and the XtionPro RGB-D camera, camera calibration, and image processing and the planned object-recognition and object pose tracking modules.

Section 3.4 sketches the overall concept for object manipulation and the core software modules available within the ROS framework. The Kinova Jaco robot arm is then presented in section 3.5, including a short description of the hardware, the original Windows-based software, and the details of the current ROS interface for the Jaco arm. For more advanced manipulation tasks, collision- and context-aware motion planning is required. An overview of the Domestic Robot manipulation action server and the MoveIt! framework is sketched in sections 3.7 and 3.6. The next section 3.10 describes the simulation model of the Domestic Robot created for the Gazebo [24] simulator.

Last but not least, section 3.11 explains the interface layer between the PEIS ambient intelligence network and the Domestic Robot software. The interface is based on dedicated *exekutor* ROS nodes (previously known as tuplehandler) that register themselves with the PEIS network, listening for incoming commands and parameters and providing feedback and execution monitoring.

3.1 Overview

As explained in the previous project report *Domestic Robot Specification* [57], the overall software consists of three major modules, with the PEIS ecology managing the whole multi-robot system and sensor network. The ROS framework was chosen as the core of the robot control while MIRA/CogniDrive is used for 2D-navigation and localisation. See Fig. 12 for a block diagram that highlights the main components of the software.

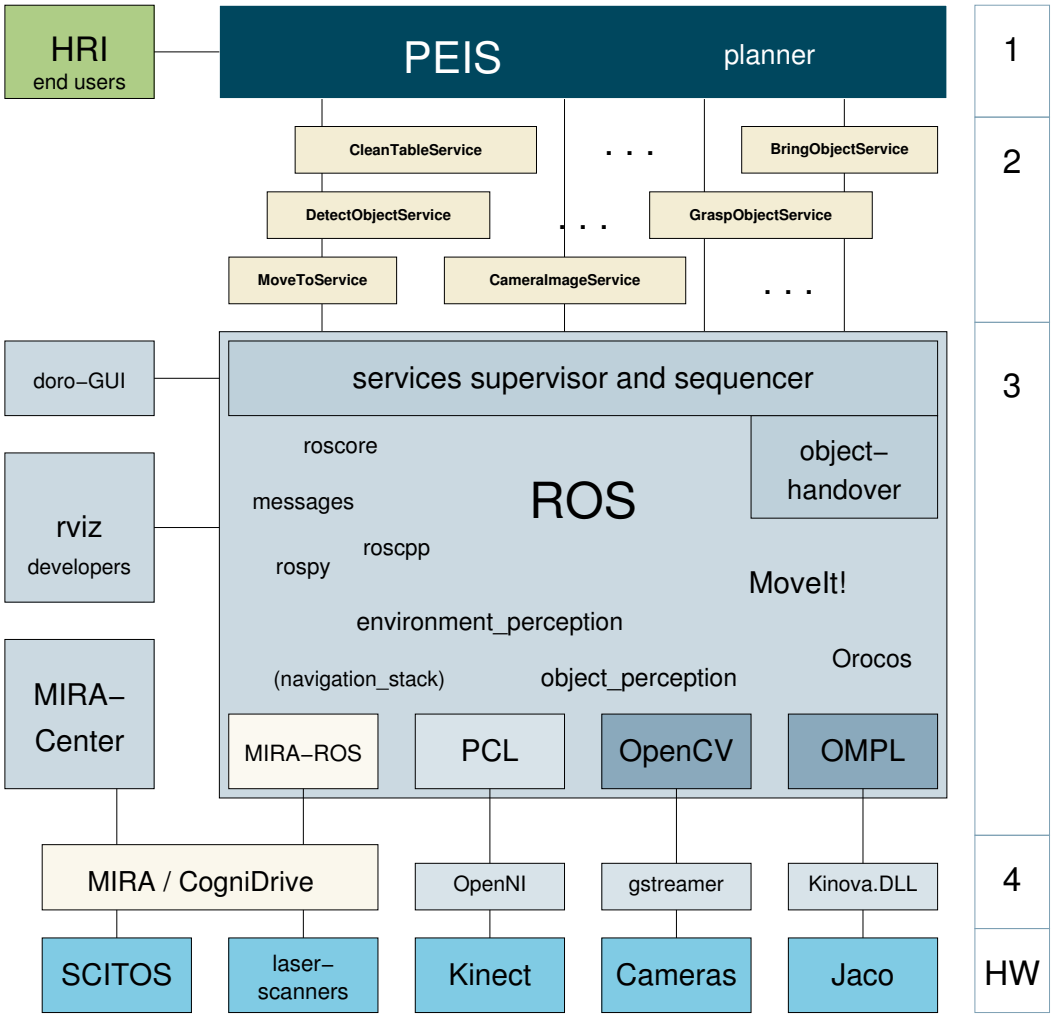


Figure 12: Software architecture of the Domestic Robot with the five main layers and key modules. The topmost layer consists of the user-interface and the PEIS infrastructure, which provides access to the ambient sensor network and other robots. It also includes the multi-robot planner.
A collection of *services* implemented as PEIS-ROS *exekutors* forms the software interface between PEIS and the Domestic Robot; see chapter 4 for a list of the implemented services. All perception, navigation, and manipulation planning for the Domestic Robot is performed by a large number of ROS software modules. A set of device drivers encapsulates the actual hardware actuators (SCITOS drive-train, PTU, JACO arm) and sensors (laser-scanners, Kinect, Xtion-Pro, cameras).
The modules on the left show the different user-interfaces, where HRI indicates the main human-robot interface for the end-user, while the *rviz* and *MIRA-Center* modules are targeted towards expert-users and software developers.

On the conceptual level, the architecture can be divided into five main layers:

1. the *PEIS* framework manages the complete Robot-Era system, including the different robots and the sensor network in the smart home. It also provides the multi-robot planner and interfaces with the main user-interface (*HRI*) which allows the end-users to request services from the system.

2. the second layer consists of the *Robot-Era services* provided by the robots. They correspond to and implement the abstract services that were extracted from the user-questionnaires and are described in detail in the project scenario reports.

   For each service, an *exekutor* process is created that listens to PEIS messages and triggers the required robot skills. Note that new services can be added to the system very easily, and existing services can be improved by corresponding changes in the ROS layer, but without changes to either the PEIS nor the device-driver layers. See chapter 4 for a list and description of the services planned and implemented so far.

3. the *ROS* framework is at the heart of the actual robot control. Our concept is heavily based on the ROS setup for the PR2 robot, where the main changes are due to the different control of the Jaco arm. Among others, we will share OpenCV and PCL for image and depth-image processing, and the manipulation stack and OMPL for pick-and-place tasks.

4. a set of device-drives that control the actuators and sensors of the robot. Several drivers are available within ROS or the standard Linux installation, while the Mono run time is used to wrap the Windows DLLs required for the Kinova Jaco. The MIRA/CogniDrive [43] software manages navigation and localisation of the SCITOS-G5 mobile platform.

5. the fifth layer in the diagram consists of the hardware devices installed on the robot, including the motors and odometry sensors on the SCITOS-G5, the front and rear laser scanners, the camera-head with pan-tilt unit, and the Kinova Jaco arm.

The figure also sketches the different user-interfaces, namely the green blocks on the left part of the diagram. The topmost block labelled *HRI* (human-robot-interface) summarises the main end-user interface, which of course includes the interface to the PEIS system and sensor-network as well as the service-request interface to the robots. This interface includes speech in addition to the graphical user interfaces and is described in detail in a separate technical report [54].

The three blocks below are targeted towards expert-users and software developers rather than towards the end-user. The ROS nodes in the *doro_control_gui* and *doro_teleop* packages provide a simple dashboard-style user-interface and the joystick-based teleoperation interface for remote control of the robot and debugging of the software. The *RViz* and *MIRA-Center* modules are the standard user-interface for the ROS and MIRA/CogniDrive frameworks.

3.1.1 ROS

Within just five years since its introduction, the ROS framework or *robot operating system* has established itself as one of the favourite middleware solutions for robot integration [35]. Due to the flexibility of the software and the liberal open-source licensing, ROS has also been selected by several vendors of robot hardware and sensors for their own products. Visit the ROS website at *www.ros.org* for an overview and the ROS Wiki at *www.ros.org/wiki* for the list of currently supported software, documentation, and tutorials. By now, literally hundreds of software modules are available, ranging from low-level device-drivers via sensor data processing up to software for symbolic planning and human-robot interaction. This includes several key software libraries, for example, the OpenCV computer vision library, the PCL point-cloud processing library, and the OpenRAVE and OMPL motion-planning frameworks.

Regarding the context of Robot-Era, ROS has been chosen as the core control framework for several leading service robots, notably the PR2 from WillowGarage and the Care-o-bot series designed by Fraunhofer. Additionally, the so-called *MoveIt* manipulation platform integrates a large set of open-source software for constraints- and collision-aware motion-planning and grasping, with optional back-ends supporting tactile-sensor based grasping. For details, see section 3.6 below. This provides a unique base for the complex manipulation tasks targeted by the Robot-Era services. As UHAM owns one PR2 robot and has long used ROS for several other projects, the selection of ROS as the main control framework for the Domestic Robot was an easy choice.

Despite the catchy name, ROS is not an *operating system* itself, but rather creates an easy-to-use *communication middleware* on top of existing operating systems. However, ROS provides a set of tools to manage large software projects, including a file-system structure consisting of *stacks* and *packages*, a build-system capable of tracking and resolving software dependencies. The software can be built and installed on several operating systems, with Ubuntu Linux as the main developer platform, but other variants of Linux are supported as well. There is also (partial) support on Microsoft Windows and on top of Android. However, due to the large number of software packages and dependencies, building the framework on the less well supported platforms is a huge task, and for now only Ubuntu Linux (12.04 LTS) can be used for the Domestic Robot.

ROS nodes The central paradigm underlying ROS software development is a system of largely independent but interacting software processes, called *ROS nodes*. That is, there is no centralised single control level or monolithic master process. Instead, ROS nodes can be added to a system at any time, allowing for the easy integration of new hardware components and software modules.

Unlike some other frameworks, ROS is mostly language-neutral, and bindings are provided for C/C++, Python, LISP. Additional language bindings are available as third-party software, including the Ros-Java interface.

Roscore and parameter server In a typical ROS system, there are only two centralised processes, namely the *ROS core* process and the *parameter server*. The *roscore* process acts are the central registry for all software nodes, either on the local system or distributed over a local network. It provides a look-up-service that allows other nodes to query the existing list

of processes and to establish point-to-point communication between nodes. The *parameter server* is used as the central repository of node parameters; it supports different namespaces and provides an easy means to store and retrieve software parameters without having to change (and recompile) code.

ROS topics and services There are two basic paradigms for communication between ROS nodes, namely *topics* and *services*. The so-called ROS *topics* provide a unidirectional communication channel between one or many *publisher*s and an arbitrary number of *subscribers*. A newly created ROS node advertises all topics it wants to publish with the *roscore* lookup service. Clients that want to subscribe to a topic first query the roscore, than negotiate a point-to-point communication with the corresponding publisher.

The base ROS system already defines a large set of standard *messages*, but one of the real strengths of ROS is the ability to define a hierarchy of user-defined messages on top of the available messages. The ROS build infrastructure automatically resolves the dependencies and creates the header/class files required for easy access to message contents from within the ROS nodes. For example, to specify the 6D-pose of an object, the *geometry_msgs/PoseStamped* message is used, which consists of a *std_msgs/Header* and a *geometry_msgs/Pose*. The header in turn is built up from a *uint32* sequence number, a *time* timestamp, and a *string* for the name of the coordinate frame (if any). The *Pose* consist of one *Point* with three *float64* (x, y, z) coordinates and one *Quaternion* with four *float64* (x, y, z, w) values. This mechanism is very powerful and significantly reduces the effort to define structured data-exchange between different nodes.

The second communication mechanism in ROS are the so-called *services*, which implement the request-response paradigm for communication between clients and a single server. Again, the messages to be exchanged between the client and the server are defined using the hierarchical ROS message format. Once a request has been sent, the client must wait until the server responds, without any control of timeouts. This is also a frequent source of deadlocks, as clients may wait indefinitely for a service not yet started or crashed. The newer *actionlib* infrastructure provides a way around this problem, as an actionlib-service *goal* request can be cancelled by the client at any time. Also, the server can provide a periodic *feedback* to indicate progress to the client before the original service goal has been reached.

Stacks and packages Apart from the core run-time functionality, ROS also suggests a specific file-system structure for its components, organised into *stacks* and *packages*. This is backed up with a set of command-line tools for navigation and a complex build infrastructure that automatically traverses the inter-package dependencies declared in the *manifest.xml* files and recompiles missing or outdated packages and messages. The overall setup of the Robot-Era ROS software is shown in Fig. 13. There are several stacks, with one stack for the Domestic and the Condominium Robot each, while the common perception and navigation functions are collected in the *robot_common* stack.

Build system To manage the compilation of hundreds of software packages, ROS provides its own build system, with the *catkin* and *rosmake* tools. When configured accordingly, *rosmake* can detect, download, and install missing system dependencies automatically with help from the *rosinstall* tools. The search path for the different stacks and packages is configured using the all-important *ROS_PACKAGE_PATH* environment variable. However,

catkin_ws	catkin (Hydro) root of the project repository
domestic_robot	ROS stack for the Domestic Robot
doro_description	Domestic Robot model and launch files
doro_gazebo_plugins	Gazebo simulation utilities
doro_handbook_Y3	robot documentation (Y3 version)
doro_msgs	robot specific messages
doro_teleop	joystick/tele-operation tools
doro_peis	PEIS services for the Domestic Robot
...	
JacoROS	alternative Kinova Jaco arm ROS stack
jaco_api	Kinova API and CSharp-wrapper
jaco_description	Jaco arm model and launch files
jaco_driver	Jaco joint-level control node
...	
condominium_robot	ROS stack for Condominium
condo_description	robot model and launch files
...	
robot_common	common robot software
cognidrive_ros	MIRA-ROS interface
peis_ros	Y1+Y2 PEIS services
...	
exekutor	Y3 PEIS interface
action_exekutor	actionlib interface
look_exekutor	pan-tilt unit interface
moveit_hand_exekutor	Jaco Moveit PEIS
...	
peis	PEIS kernel and tools
...	

Figure 13: Robot-Era software repository structure with ROS stacks and packages.

the implementation and details of the build system have changed with every major release of ROS so far. This is one major obstacle when trying to upgrade existing ROS software to a new release. For the second experimental loop, the Robot-Era Domestic Robot software has been tested with version *Hydro* of ROS. All software components have been ported from the deprecated *rosbuild* system to the newer *catkin* build system.

Real-time robot control ROS also includes support for real-time robot control, based on the control architecture designed for the PR2 service-robot. The *pr2_controller_manager* architecture defines the interface between higher-layer software and low-level controllers, for example joint-position controllers with their PID parameters loaded from YAML configuration files. This architecture is expected by several ROS packages, including manipulation stack. On the PR2, the controllers access the hardware via the EtherCAT bus at 1 kHz sample rate. This is not possible on the Domestic Robot, where the default cycle time of the Jaco arm is just 10 Hz.

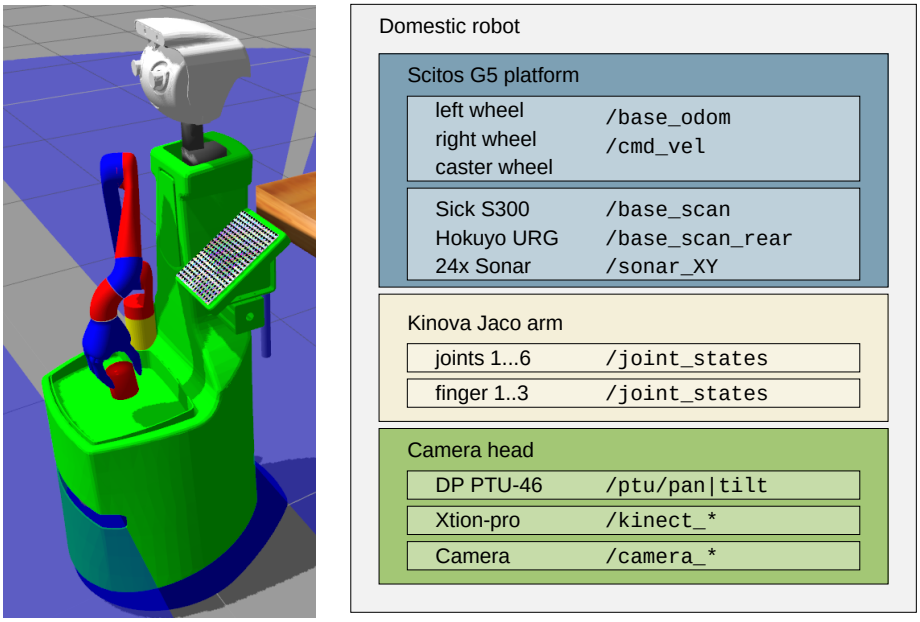


Figure 14: The basic structure of the URDF robot description for the Domestic Robot, consisting of the SCITOS-G5 mobile platform with the wheels and navigation sensors, the Kinova Jaco arm, and the camera head.

3.1.2 Domestic Robot URDF

A full URDF or *universal robot description format* model is the first step to integrate a robot into the ROS framework. The model specifies the kinematics structure of the robot parts (called *links*) and describes the joints, actuators, and sensors of the robot. To simplify the description, the *xacro* preprocessor can be used to code parts of the robot with macros, which can also simplify the geometric description by calculation of simple mathematical equations in the macros.

Fortunately, individual URDF/Xacro descriptions already existed for several parts of the Domestic Robot, including the Kinova Jaco arm and several sensors (XtionPro, cameras, laser-scanners). A model of the SCITOS-G5 robot was converted from the existing MIRA description and the datasheets. The resulting URDF model of the Domestic Robot is shown in Fig 14. It consists of a modular structure that mirrors the main parts of the robot, namely the SCITOS-G5 platform with motors and sensors, the Kinova Jaco arm, the Directed Perception PTU-46 pan-tilt unit, and the Asus XtionPro and Firewire cameras on the sensor head.

In addition to the geometry, the full URDF model of the robot also includes the weight and the inertia properties of all components. The weight of the main platform was taken from the SCITOS-G5 datasheets, while the inertia parameters were estimated based on a cylindrical model of the mobile base. For the other parts of the robot, realistic estimates of the components masses are used, but the inertial terms are only simplified. In particular, the inertial parameters of the distal joints of the Jaco arm and fingers are larger than in reality,

which does no harm on the real robot but helps to keep the simulation model stable.

Regarding the sensor setup of the robot, the existing models of the Sick S-300 and Hokuyo URG-04LX laser-scanners provided by ROS were used, with the mounting position on the mobile base taken from the SCITOS-G5 datasheet. For use in simulation, the ray geometry and deadband settings were adapted to the current mounting positions as well. The sonar sensors are also included, with the sensor model backported from the Gazebo *ray_sensor* plugin. The sensor model should be accurate enough for single sensors, but does not model the inevitable crosstalk when running a ring of 24 sonar sensors at the same time. ROS also includes the URDF models for the Asus XtionPro depth-camera and the standard cameras mounted on the sensor-head of the Domestic Robot.

So far, the default parameters are used for the intrinsic calibration of the cameras in the URDF model; actual calibration data for the cameras is stored in external files as required by the ROS camera drivers (openni2, gstreamer).

3.1.3 Coordinate-systems and tf

All geometry calculations in ROS are based on a right-handed coordinate system. For the Domestic Robot, the base coordinate system was chosen according to the usual convention, with the x-direction towards the front, y to the left, and z upwards. The actual origin is at the floor ($z = 0$) and halfway between the two driving wheels. While this coordinate system is difficult to measure from the outside, the choice of origin is typical for differential-drive robots and simplifies the 2D-navigation calculations.

Managed by the ROS *tf* transformation library, a separate coordinate system is attached to every part (*link*) of the robot as defined in the robot URDF model. See Fig. 15 for a screen-shot of the robot in the *rviz* visualisation tool, with the tf coordinate-system markers enabled and overlaid on the semi-transparent robot model. For each marker, the red, green, and blue arrows correspond to the x, y, z directions respectively.

See Fig. 17 for a cut-out of the whole coordinate frame graph showing the most relevant coordinate systems, including the wheels, Jaco arm with hand and fingers, and the sensor head. The tf graph can be visualised at runtime using the tf *view_frames* command,

```
rosrun tf view_frames
```

which generates a snapshot of the tf graph, and saves the result in a file called *frames.pdf*.

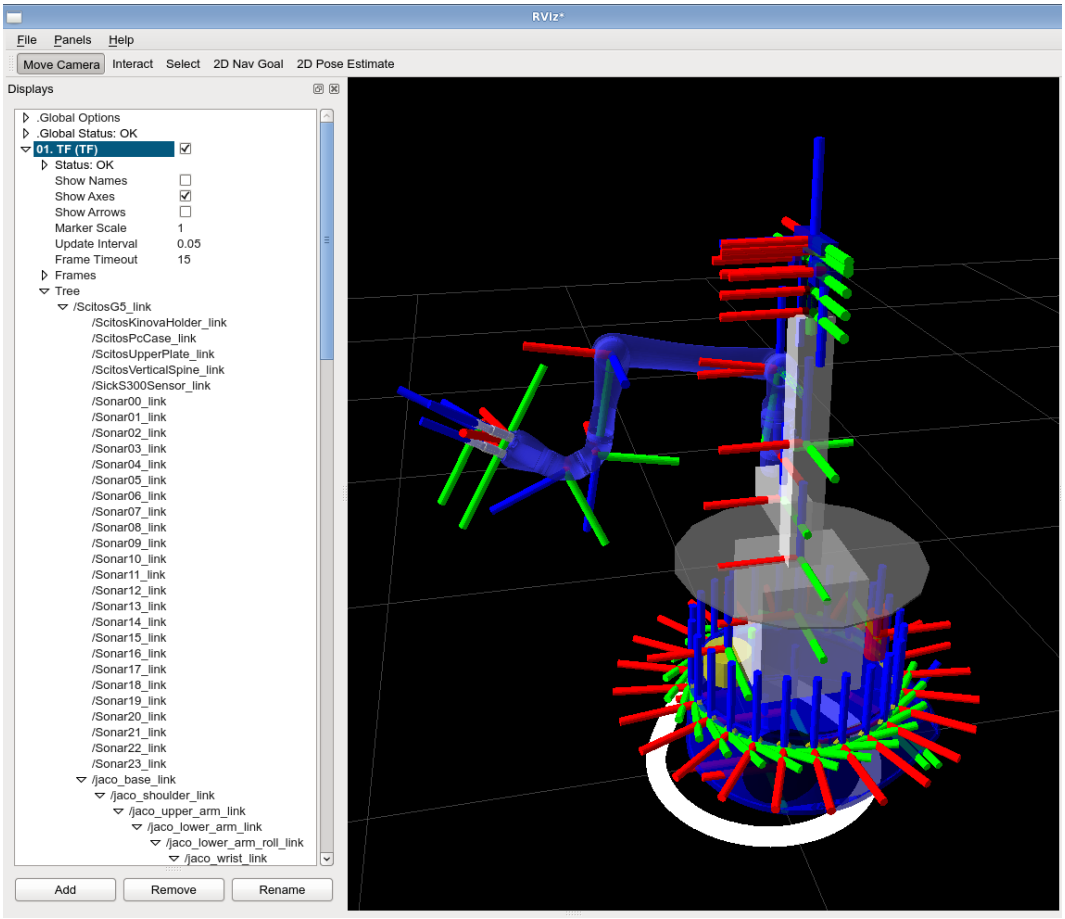


Figure 15: Coordinate frames of the Domestic Robot. The figure shows the rviz visualisation of all tf coordinate frames used in the Domestic Robot URDF model (red: x, green: y, blue: z). The main coordinate system of the platform has positive x towards the front, y to the left, and z upwards. In the default orientation, the pan-tilt angles are both 0, and the x-axes of the Kinect/Xtion and the cameras point forward.
Also note the coordinate systems for the segments of the Jaco arm in the current mount position. As shown, the *jaco_shoulder_yaw_joint* is at $-\pi/2$ radians. Moving the shoulder-yaw joint of the arm to its zero position results in a self-collision with the central pillar of the robot, and must be avoided. The ring of coordinate system indicators around the platform base corresponds to the sonar sensors.

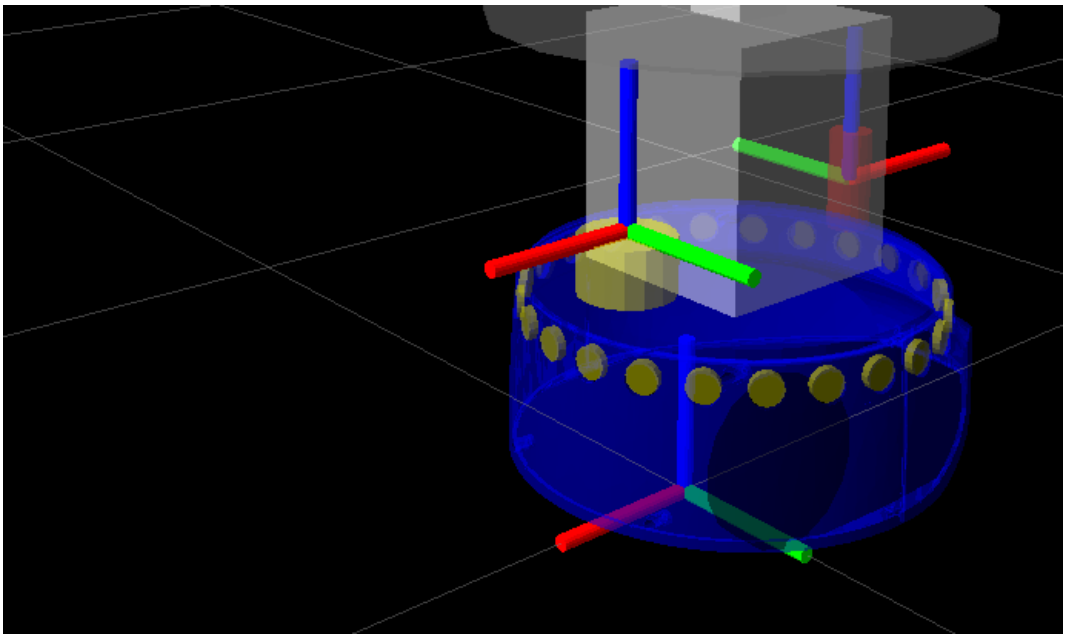


Figure 16: Base coordinate frames of the Domestic Robot. The figure shows the base coordinate system *base_link*, halfway between the driving wheels, and the two coordinate systems for the front and rear laserscanners.

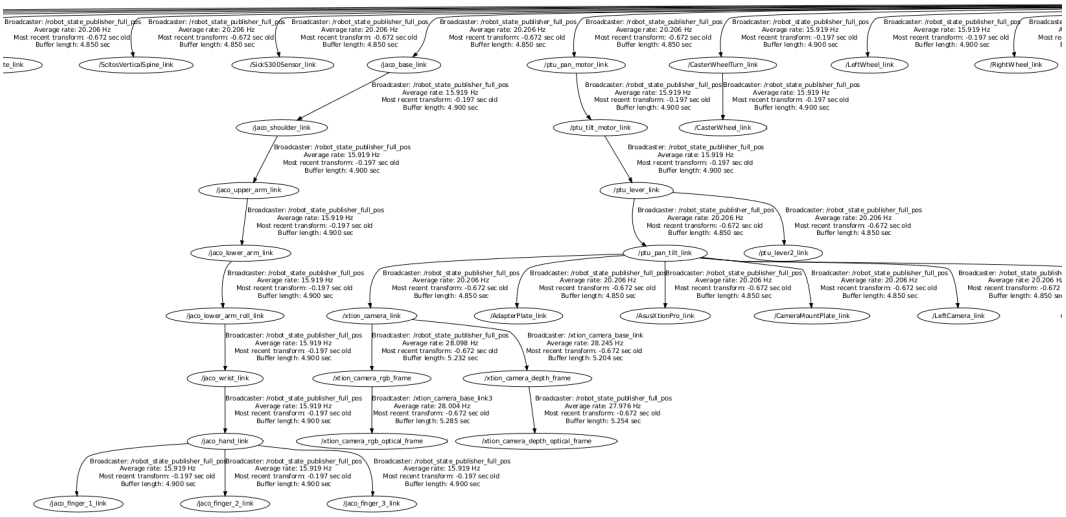


Figure 17: Coordinate frame graph of the Domestic Robot. The figure shows a zoomed-in region of the total graph with the most relevant tf coordinate frames of the robot, including the wheels, the Jaco arm with hand and fingers, and the sensor head with pan-tilt unit and the cameras.

3.1.4 Launching the Domestic Robot

To simplify operation of the Domestic Robot, the start up of the different required device drivers and ROS nodes is managed by a set of ROS launch files. At the moment, the start up files do not include all required nodes. It is recommended to run the launch files and programs in different terminals for easier control and debugging:

```
shell-1> roslaunch doro_description domestic_bringup.launch
shell-2> rosrun rviz rviz
shell-3> roslaunch doro_moveit_config move_group.launch
shell-4> roslaunch visionhub start_perception.launch
shell-5> roslaunch doro_peis domestic_services.launch
```

Domestic Robot bringup The first step required for robot startup is running the *domestic_bringup.launch* launch file. This file starts the different essential low-level processes that are needed for robot operation. Therefore, this launch file is required, while the other launch files are optional and may be skipped. For example, manipulation is not available on the Condominium Robot, but the platform navigation and other Robot-Era services can be started exactly as on the Domestic Robot.

In the current version, the bringup launch file integrates the following functions:

- uploads the Domestic Robot URDF to the *robot_description* parameter onto the parameter server.
- starts the MIRA software for control of the SCITOS platform, powering up the different sensors and the pan-tilt-unit, and enabling the front and rear laser-scanners.
- starts the *cognidrive_ros* bridge to interface the MIRA localisation and navigation functions.
- starts the *jaco_node* for joint-level control of the Kinova Jaco arm and hand.
- starts the *doro_ptu46* node for controlling the pan-tilt unit.
- runs the *cameras.launch* file which in turn starts the ROS nodes for the XtionPro RGB-D camera and the firewire cameras.
- starts a set of utility nodes. This includes the *doro_joint_state_merger* node and the *robot_state_publisher* required for providing the *tf* transformation library with up-to-date robot joint-state data.
- starts the *joy* nodes for ROS tele-operation.
- starts the *mjpeg_server* webserver that allows clients to access the camera-images as an MJPEG-format stream from any web-browser.

The Jaco arm should be powered-on and in its *retract* position (see section 3.5 on page 63 for details) before running the launch script. If initialisation of the Jaco arm fails, or if the Jaco node needs to be restarted, it is possible to restart the arm using the provided *jaco_node.launch* file:

```
rosnode kill jaco_node
roslaunch doro_description jaco_node.launch
```

Do not launch the original Kinova *jaco_node/jaco_node.launch* file, which loads a wrong robot configuration.

Depending on the system- and MIRA-configuration, several devices (e.g. laser-scanners, PTU) may only be powered-up when MIRA is started, and it takes some time until the devices have completed their own initialisation sequence. In particular, the PTU node is known to crash sometimes, when the PTU initialisation is triggered and takes too long. You can restart the PTU node easily,

```
rosnode kill ptu
roslaunch doro_ptu46 doro_ptu.launch
```

but this will not restart the PTU calibration sequence. If necessary, power-cycle the PTU using the small power-switch on the PTU controller, to ensure that the PTU is in its zero position before restarting the PTU node.

To visualise the current ROS node graph, including the interconnections via ROS topics (but not services), run the *rxgraph* utility,

```
rxgraph -o rxgraph.dot
dot -T png -o output.png rxgraph.dot
dot -T pdf -o output.pdf rxgraph.dot
```

ROS nodes started during bringup The following list documents the key ROS nodes started as part of the above launch sequence,

```
rosnode list
/cognidrive_ros
/diag_agg
/doro_joint_state_merger
/doro_telnet_server
/jaco_node
/left_camera
/right_camera
/floor_camera
/mjpeg_server
/ptu
/ptu_action_server
/robot_state_publisher_full_pos
/rosout
/rossink_1365097477302051982
/xtion_camera/depth/metric_rect
/xtion_camera/depth/points
```

```
/xtion_camera/depth/rectify_depth
/xtion_camera/depth_registered/metric_rect
/xtion_camera/disparity_depth_registered
/xtion_camera/driver
/xtion_camera/ir/rectify_ir
/xtion_camera/points_xyzrgb_depth_rgb
/xtion_camera/register_depth_rgb
/xtion_camera/rgb/debayer
/xtion_camera/rgb/rectify_color
/xtion_camera/rgb/rectify_mono
/xtion_camera_nodelet_manager
```

where the `/cognidrive_ros` node provides the platform control and navigation, while `/jaco_node` controls the arm, and `/ptu` controls the PTU. The laser-scanner data and localisation is published by `/cognidrive_ros`, while `/left_camera`, `/right_camera`, `/floor_camera` and `/xtion_camera/*` are the controller nodes for the RGB- and Xtion-Pro RGB-D cameras.

ROS topics published after bringup Once the basic robot bringup-sequence has been completed, almost 100 ROS topics are active on the Domestic Robot. The following list documents the key ROS topics published as part of the robot-bringup launch sequence, sorted alphabetically,

```
rostopic list

/base_odometry/odom
/base_scan
/base_scan_rear
/battery/server2
/cmd_abs_finger
/cmd_abs_joint
/cmd_rel_cart
/cmd_vel
/diagnostics
/diagnostics_agg
/doro/scitos/wheel_states
/hand_goal
/hand_pose
/initialpose
/jaco/joint_states
/jaco_finger_1_joint_controller/command
/jaco_finger_2_joint_controller/command
/jaco_finger_3_joint_controller/command
/jaco_joint_trajectory_action_controller/joint_trajectory_action/goal
/jaco_kinematic_chain_controller/follow_joint_trajectory/cancel
```

```
/jaco_node/cur_goal
/joint_states
/left_camera/camera_info
/left_camera/image_raw
/left_camera/...
/right_camera/...
/floor_camera/...
...
/map
/map_metadata
/move_base/cancel
/move_base/feedback
/move_base/goal
/move_base/result
/move_base/status
/move_base_simple/goal

/ptu/ResetPtu/goal
/ptu/SetPTUState/goal
/ptu/cmd
/ptu/joint_states
/rosout
/rosout_agg
/tf
/xtion_camera/depth/camera_info
/xtion_camera/depth/disparity
/xtion_camera/depth/image
/xtion_camera/depth/image/compressed
...
/xtion_camera/depth/image_rect_raw
/xtion_camera/depth/points
/xtion_camera/depth/rectify_depth/parameter_descriptions
/xtion_camera/depth/rectify_depth/parameter_updates
/xtion_camera/depth_registered/camera_info
/xtion_camera/depth_registered/disparity
/xtion_camera/depth_registered/image
/xtion_camera/depth_registered/image_rect/compressed
/xtion_camera/depth_registered/points
...
/xtion_camera/driver/parameter_descriptions
/xtion_camera/driver/parameter_updates
/xtion_camera/ir/camera_info
/xtion_camera/ir/image_rect/compressed
/xtion_camera/rgb/image_color/compressed
...
```

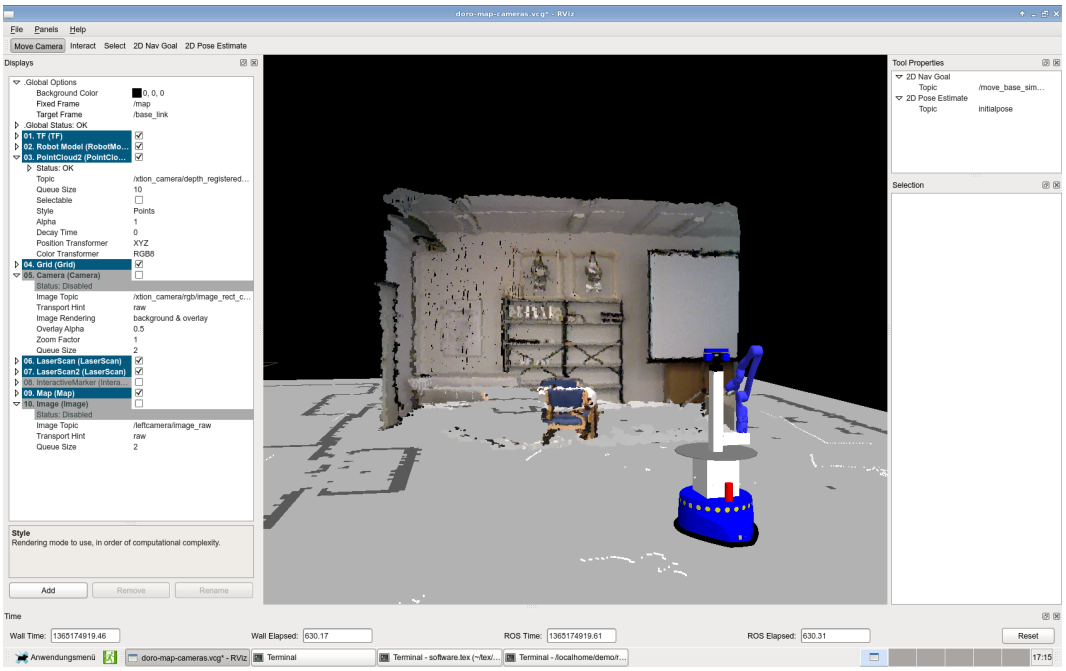


Figure 18: An example rviz configuration for control of the Domestic Robot. The *robot_description* and *tf* topics should be selected, with *map* as the fixed-frame and *base_link* as the root link of the robot. The 3D view in the centre shows the robot localised on the map, with overlaid laser-scans and coloured point-cloud from the XtionPro on *xtion_camera/depth_registered*. Initial pose-estimates are provided on *initialpose* and interactive 2D navigation via publishing to *move_base_simple/goal*.

rviz configuration The ROS *rviz* visualisation tool provides the main user interface for the Domestic Robot software developer. The 3D-View included in the tool generates an integrated view of the environment and map, the robot state including its position in the environment and the pose of the robot arm and pan-tilt-unit, and visualisation of the incoming sensor-data overlaid onto the 3D world model.

Depending on the developers' needs, different sensor-data and plugins can be selected, enabled, and configured. A default *rviz configuration* file is provided as part of the *doro_description* ROS stack. This selects the robot-description, tf frames, the environment map, laser-scanner and camera data, and 2D-navigation topics to control the SCITOS 2D-motions. See Fig. 18 for a screen-shot of rviz using this configuration.

Alternatively, the MIRA-Center software provides an easy-to-use interface for the navigation and localisation tasks on both the Condominium and the Domestic Robot (see Fig. 20 on page 44 for a screen-shot).

Launching manipulation To start the ROS/MoveIt manipulation stack adapted for the Domestic Robot and Kinova Jaco arm, please check that the robot bringup launch was successful, and that the Jaco arm is in the home position. Then start the *move_group.launch* file,

```
roslaunch doro_moveit_config move_group.launch
```

and wait until all nodes and services have been started (The Arm in RViz matches the actual position and OMPL is displayed as the planning library).

See section 3.6 below for an overview of the ROS manipulation implementation and a detailed description of the different ROS nodes and services that provide the robot with collision-aware motion planning.

Launching Robot-Era services See chapter 4 for an overview of the Robot-Era service architecture and a list of the services. To connect the robot to the PEIS ambient intelligence network, including the multi-robot planner and human-robot interface, just launch the *domestic_services.launch* file. This assumes that the robot bringup and manipulation launch files have been started and all services are running,

```
roslaunch doro_peis domestic_services.launch
```

3.1.5 Running ROS on Multiple Computers

In many cases, the raw compute power provided by a single computer will not be sufficient to run advanced algorithms with many nodes. Fortunately, ROS provides a very simple and convenient way to distribute computation across multiple machines, because nodes running on different computers can communicate seamlessly using ROS messages and services. The key idea is to start the *roscore* lookup-service process on one selected machine, which then acts as the master and provides the node, topic, and service lookup for all machines in the network. The master returns the hostnames and ports used for communication on the requested topics and services, and the nodes establish the direct network connection to exchange data between them. Typically, either the on-board computer of the mobile robot or a fast machine is acting as the ROS master. Note that there is also ongoing work on multi-master ROS systems, but this is beyond the scope of the handbook.

In some cases, the communication with the *roscore* process is possible, while actual data-transfer between nodes on different computers is not working. A typical situation is that *rostopic list* returns the full list of topics managed by the *roscore* process, while running *rostopic echo* on one of the listed topics does not return any data. This occurs when the hostname lookup on the different machines in the network is not working properly. However, ROS requires the mutual hostname lookup to work in order to exchange data between different machines. The easiest solution is to check the */etc/hosts* files and to explicitly add the names of all required computers:

```
cat /etc/hosts
127.0.0.1    localhost
192.168.0.33 scitos scitos.informatik.uni-hamburg.de
192.168.0.44 laptop
...
```

Figure 19: The Sony Sixaxis joystick used in the *doro_sixaxis_teleop* node. The labels indicate the axes and button numbers used by the *ps3joy* joystick driver for ROS.

Once the network and hostnames have been set-up on all participating computers, the *roscore* process is started on the machine that should act as the master. On all other machines, the *ROS_MASTER_URI* environment variable is set to point to the master machine, and subsequent attempts to launch or contact ROS nodes will then be redirected to the given master:

```
export ROS_MASTER_URI=http://scitos:11311
roslaunch ...
```

3.1.6 Teleoperation interface

The ROS nodes in the *doro_teleop* package provide a basic tele-operation interface for interactive robot control. Apart from basic maintenance and test, the tele-operation command interface can be used by expert users to recover from situations where the autonomous robot control software is stuck (e.g. backing up from an obstacle). As the moment, three ROS nodes are available:

- *doro_keyboard_teleop*
- *doro_sixaxis_teleop*
- *doro_telnet_server*

The *doro_keyboard_teleop* node allows us to drive the robot around via the keyboard (a,s,d,w) keys. It directly publishes to the */cmd_vel* topic. Additional commands for controlling the arm and PUT are planned, but not implemented yet.

The *doro_sixaxis_teleop* node reacts to user input on a Sony Sixaxis joystick, either connected via USB cable or wireless via Bluetooth. This requires the *ps3joy* package. As the code is based on the PR2 joystick tele-operation node from WillowGarage, the installation instructions on *www.ros.org/wiki/pr2_teleop* may be helpful to set the software up.

motion	activation	execution
driving:	hold button 10	use left stick
pan-tilt:	hold button 11	use right stick
shoulder:	hold button 8	use left stick
elbow:	hold button 8	use right stick
wrist:	hold button 9	use left stick
fingers:	hole button 9	use right stick

Table 1: Default mapping of the joystick buttons to robot motions.

The usage of the Sixaxis joystick is illustrated in Fig 19. For safety reasons, the user has to hold down one of the trigger-buttons to enable the corresponding motions via the left- and right joysticks.

The *doro_telnet_server* node starts a simple telnet-style server that subscribes to the *joint_states* topic and connects to the various ROS nodes for execution of joint-level trajectories, PTU motions, and *cmd_vel* for moving the platform. Once started, the server accepts connections from telnet-style clients, for either interactive use via the command-line or for use by user-written scripts and programs.

```
telnet localhost 7790
telnet> help                              %      list of commands
telnet> get-joint-angles                  %  current joint angles
telnet> get-min-angles                    %    lower joint limits
telnet> movej to -90 0 0 10 20 30         %    joint-space motion
telnet> movej by 0 0 0 0 -5 0             % relative joint motion
telnet> fingers to 0 30 45                %    Jaco finger motion
telnet> ptu to 90 -45                     %  pan-tilt unit motion
telnet> ...
telnet> disconnect
```

3.1.7 Control Center teleoperation interface

One key aspect of the upcoming second experimental loop of the project is the testing of the whole Robot-Era system in *realistic* and *real* settings. In particular, no researchers or programmers will be close to the robot during the experiments. Instead, the robot control and any attempts at error recovery will be performed by non-expert people in the remote control center that supervises the experiments.

3.1.8 Robot calibration

No calibration is required for the SCITOS base platform. The location of the driving wheels is fixed and the gear-ratios of the differential drive are known. Any errors in wheel odometry (e.g. due to wheel slip) are handled by the AMCL localisation when fusing the laser-scanner and odometry data in MIRA. The front and rear laser-scanners and the sonar sensors are mounted fixed onto the platform, and their positions are documented in the SCITOS robot descriptions (MIRA XML and ROS URDF). However, any systematic modelling errors are hard to check, because Metralabs does not define reference points on the platform. Note that the curved outer shape of the SCITOS platform makes it rather difficult to estimate the *base_link* and mount positions of the sensors precisely.

For the Jaco arm, no calibration tools are provided by Kinova, and the factory calibration is assumed to be correct. Automatic full-robot calibration by matching camera images to arm movements is possible, but has not yet been implemented on the Domestic Robot. Also, there is no accuracy data available for the Jaco arm from the vendor. While the human user automatically compensates small errors when tele-operating the arm, any such errors may compromise the manipulation capabilities under autonomous control. Experience gained throughout the project experimental phases will show whether any additional modelling is required.

Regarding the base position of the arm, the documentation from Kinova seems to be inaccurate, but corrected positions are used in the robot URDF model. Note that the base position used in the URDF should be checked carefully against the actual mount point of the arm. A set of calibration jigs might be useful to verify arm poses, but so far neither Kinova nor Metralabs do provide any such objects.

The pan-tilt unit is driven by stepper-motors and performs an automatic self-calibration sequence when powered up. Afterwards, position is tracked by counting motor steps, which is highly accurate. Note that the PTU ROS node should be started when the PTU is in its zero position.

Camera calibration, file formats, calibration file locations, etc., see section 3.3 below.

3.2 Robot localisation and navigation

This section summarises the localisation, collision-avoidance, and navigation algorithms implemented on the Domestic Robot. The robot acts in a known indoor environment with level floors, which greatly simplifies the problem because well-known 2D localisation and path-planning methods and a static map of the environment can be used. See [37] for a review of the relevant basic algorithms.

As explained in the earlier project report D4.1 *Domestic Robot platform specification* [57], the MIRA framework with the CogniDrive module will be used for the control and sensor-interface of the SCITOS-G5 mobile platform, while a simple MIRA-ROS bridge interfaces to the ROS framework. This architecture was decided on after careful evaluation of the ROS *navigation_stack*, which basically provides the same functionality as the pair of MIRA and Cognidrive. However, using ROS here instead of MIRA would require us to rewrite the low-level drivers to the SCITOS platform with little other benefit.

See Fig. 12 on page 22 for the main software blocks of the Domestic Robot. The components concerned with navigation are located in the lower-left corner of the diagram, namely the MIRA framework with the hardware drivers for the SCITOS-G5 motors and odometry sensors, and the interfaces to the Sick and Hokuyo laserscanners. Robust localisation, collision-avoidance and path-planning is performed by the CogniDrive software, and the MIRA-Center user-interface allows the expert user to control the robot motions. The navigation combines a static map of the environment with a dynamic occupancy grid map generated from the laser-scanner data.

The material in the next two sections of this chapter is a shorted summary of the MIRA description already provided in [57] (chapter 4). It is repeated here to make this handbook self-contained and to motivate the design of the MIRA-ROS bridge explained in section 3.2.3.

3.2.1 MIRA

The MIRA framework is a robot middleware that targets a modular software development process built around a set of communicating processes or modules. See the webpage at *www.mira-project.org/MIRA-doc-devel/index.html* for documentation. The overall approach and goals are therefore similar to ROS, but several design decisions have resulted in a rather different implementation. For communication, the MIRA framework offers message passing by implementing the publisher/subscriber pattern as well as Remote Procedure Calls (RPC). Beside this communication, the MIRA base and framework provide much more functionality, including visualisation of the data flow and the data passed between modules, error monitoring and tracking and identifying problems with the modular application.

MIRA provides a middleware that handles the communication between the modules, or respectively the *units*, and ties these units together to compose a complex application. The MIRA core is divided into the following software components:

- *base*: commonly used classes, algorithms, and helpers.
- *framework*: publisher/subscriber communication.

- *GUI*: classes, widgets and tools for visualisation, Rich Client Platform for modular GUI design.
- *packages*: collection of components, dependency information.
- *toolboxes*: algorithms and classes used by other components.
- *domains*: one or more units to be used by other components.

Similar to ROS, the MIRA system supports the robot developer on several levels.

- *component level*: managing executables and shared libraries, with dependency information encoded in manifest files.
- *computation graph level*: managing *units* communicating via typed and named *channels*, support for remote procedure calls (RPC).
- *runtime level*: executables and share libraries.
- *filesystem level*: package, toolboxes, and domains.
- *repository level*: support for SVN and FTP repositories, source- and binary-code distribution, and software packages.

MIRA is designed to allow for fast and easy creation and testing of new distributed software modules. The interface is very lightweight and fully transparent and it hides implementation details like data-locking, usage of threads and cyclic processes, and the location of senders and receivers within the same process, a different process, or a remote process.

A detailed comparison between MIRA and ROS was included in the previous project report D4.1 [57], where MIRA was shown to have significant advantages in several important areas. On the other hand, ROS has a larger user-community and many more software packages are available for ROS.

3.2.2 Cognidrive

The *CogniDrive* software from Metralabs has been selected for the navigation and localisation capabilities of both the Domestic and Condominium Robots. See chapter 5 of the previous report D4.1 [57] for a detailed description of the CogniDrive software.

Instead of providing only the standard *drive-to* command, the motion-planning in CogniDrive is based on a set of *objectives*, which enable a fine-grained control over the robot motion. Several objectives can be active at the same time, with different weight factors adjustable from the high-level (application or user) interface.

Motion requests are scheduled as *tasks* which can be subdivided into several *sub-tasks*, each of which is then guided by the active objectives. Additionally, CogniDrive explicitly provides one of the key functions required by the Robot-Era services, namely the capability to navigate in a multi-map environment, e.g. several floors in a building that are connected by an elevator.

CogniDrive supports a variety of different requirements such as:

- support for non-holonomic robots of different sizes
- navigation with high precision (e.g. Docking, handling of narrow passages)
- fast path planning and online dynamic replanning
- taking moving obstacles into account
- consideration of traffic rules (e.g. forbidden areas and speed limits)

For task processing, the motion planner and the objectives play a major role. Each objective is a separate software module specialised for certain tasks like following a person, driving at a certain speed or direction, etc. The objectives are realised as software plugins. This allows us to add new objectives easily when new tasks are necessary without changing other parts of the navigator. The output of the objectives is then used by the motion planner to generate motion commands that are then sent to the robot motor controllers. Some objectives require additional information from other navigational modules such as localisation and mapping algorithms or modules for user interaction like person trackers.

Each sub-task can be parametrised by numerous tasks specific options, including: goal point to drive to, map to drive to preferred driving direction of the robot (backward, forward or both), accuracy for reaching a goal point, accuracy for the orientation angle at a goal point, maximum allowed driving distance (e.g. during exploration). By specifying a combination of sub-tasks and their parameters the robots navigational behaviour can be completely modified at runtime. For example the complex task *"Drive backward to the destination (10, 0) in map Floor2 with an accuracy of* ± 0.5 *m and turn to the orientation of 70° with an accuracy of* $\pm 15°$*"* is easily handled by CogniDrive.

Internally, CogniDrive manages a grid map of the environment, where cell covers a certain area of the velocity space and corresponds to a certain velocity command. For motion planning in CogniDrive, a cost function is computed for each cell and therefore the velocity command that yields the smallest cost is chosen. In the original Dynamic Window Approach [33] that cost function is composed of three different functions, called objectives. One objective yields large costs when the robot would get too close to obstacles by choosing that certain action. The second objective prefers actions that lead to high speeds and the third one takes care of the robots orientation. Additionally, each objective can forbid a certain action completely by marking it as "not admissible" when a certain requirement, like the minimal distance to an obstacle, is not met. If at least one objective marks an action as "not admissible", the action is excluded from the set of allowed actions.

After all active objective were processed for all cells in the dynamic window and the costs of all cells were computed based on the weighted sum, from all admissible cells, the cell with the lowest cost value is chosen and the corresponding action is sent to the motor controllers in terms of a velocity command. Afterwards, the whole processing cycle is repeated until the current task and the specified goal is reached. Several different objectives are supported:

- *distance objective*: responsible for avoiding collisions by calculating the distance between the robot and obstacles along the predicted trajectory. The objective also takes the braking distance of the robot into account.

- *path objective*: the default objective when trying to reach a given target pose. The algorithm is based on a Global Dynamic Window Approach [34], where the cost value for the objective is taken from the navigation function of the path planner. This map contains a value that resembles the distance to the goal, and the path objective there prefers actions that lead the robot closer to the specified target. The standard E*-algorithm is used for the actual path planning process.

- *speed and no-go objective*: allows the application-level to request a speed-limit for the motion, and to avoid forbidden areas. The speed-limit can be encoded in a separate grid-based map, so that different speed-limits are possible along the robot path.

- *heading objective*: used to control the orientation of the object once the final position has been reached. Typically given a small weight, so that intermediate poses along the robot path are not influenced by the final heading.

- *person follow objective*: this implements one of the key requirements and tasks for the Domestic Robot. It can be parametrised to follow a person while taking privacy into account by keeping a given minimum distance between the robot and the users. The object will turn the robot to face the user.

- *user objective*: manual remote control of the robot motion.

- *additional objectives*: can be added easily due to the modularity of the CogniDrive system. For example, an *explore objective* could reward actions that explore the given map.

3.2.3 MIRA-ROS bridge

The block diagram of the interface software is sketched in Fig 21. The MIRA/Cognidrive framework and the ROS navigation-stack use very similar internal representations for the environment map, the robot pose and pose-updates, and the laser-scan data used for localisation.

When running on the real robot, MIRA controls the SCITOS-G5 platform including the front and rear laserscanners, the sonar sensors, and the wheel-odometry and motors. It forwards the laserscan and wheel data directly to Cognidrive, which is then responsible for localisation and robot path-planning. The MIRA-ROS bridge in turn converts the laserscan data and the calculated robot pose estimation and covariance data into the message formats used by the ROS navigation-stack, and then publishes the data on the corresponding ROS messages. Incoming goal-pose requests are converted into the MIRA data-format and given to Cognidrive for execution of the motion request.

When running in the Gazebo simulator, the trajectories received from Cognidrive are used to drive the robot around in the simulated world, and the reached robot-pose and wheel-angles are calculated by the simulation engine. Additionally, the laserscan and sonar sensor data are estimated by calculating the distance between the robot and the nearest objects in the virtual world. The laserscan data is then taken by the MIRA-ROS bridge, converted into the

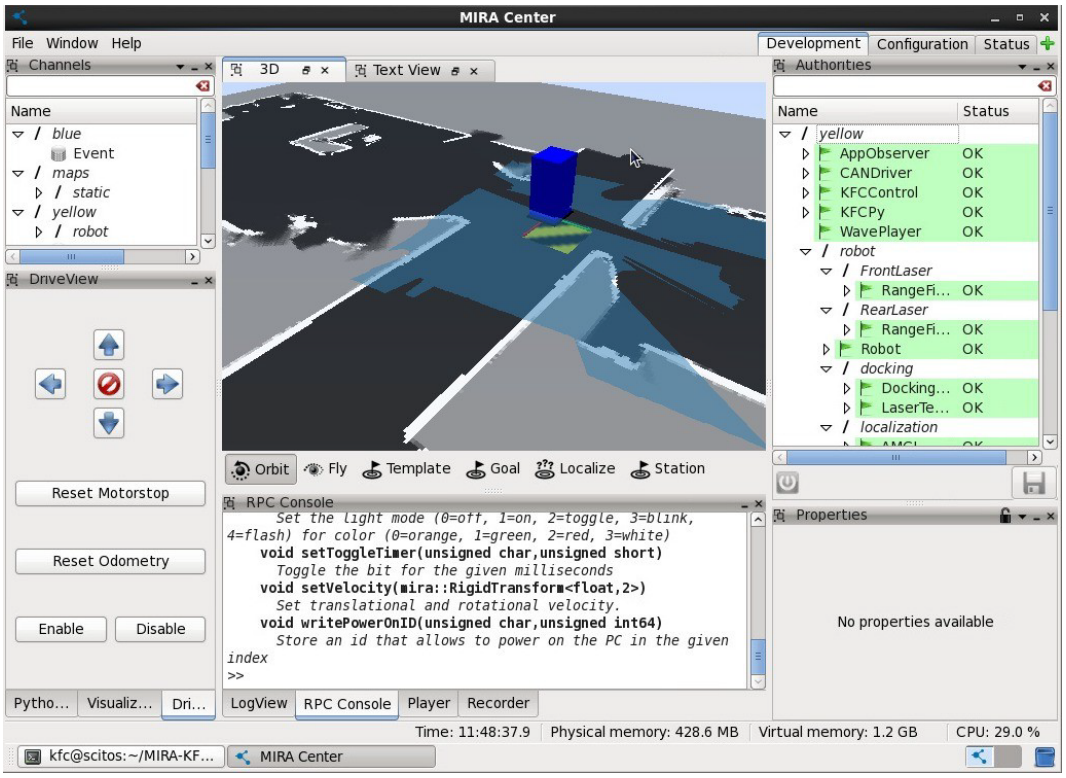


Figure 20: The user-interface of the MIRA-Center software with visualisation of the robot and laser-scanner data inside the map.

data format used by MIRA, and Cognidrive is then able to calculate the robot localisation base on the simulated laser scans.

To summarise, connecting MIRA/CogniDrive with ROS comes down to:

- subscribing MIRA channels and publishing that data using ROS topics.
- subscribing ROS topics and publishing that data using MIRA channels.
- forwarding transforms from one framework to the other.
- offering an actionlib-interface like *move_base* to MIRA's *task-based navigation*.
- allowing direct driving (bypassing CogniDrive) by subscribing to the ROS */cmd_vel* topic and forwarding the *geometry_msgs/Twist* to the robot's wheels.

On the Domestic Robot, the laserscanners, drives, encoders and battery are connected to MIRA, so their data needs to be forwarded into the ROS world when the robot is used in real application. During simulation, however, the virtual devices are created in Gazebo (ROS), so their data needs to be forwarded to MIRA to embed cognidrive into the simulation. So, the direction in which cognidrive_ros converts between the frameworks is determined by setting a *–simulation* flag on startup. The code was developed and tested on a Metralabs Scitos G5 robot running Ubuntu 12.04 LTS 32bit and ROS Fuerte.

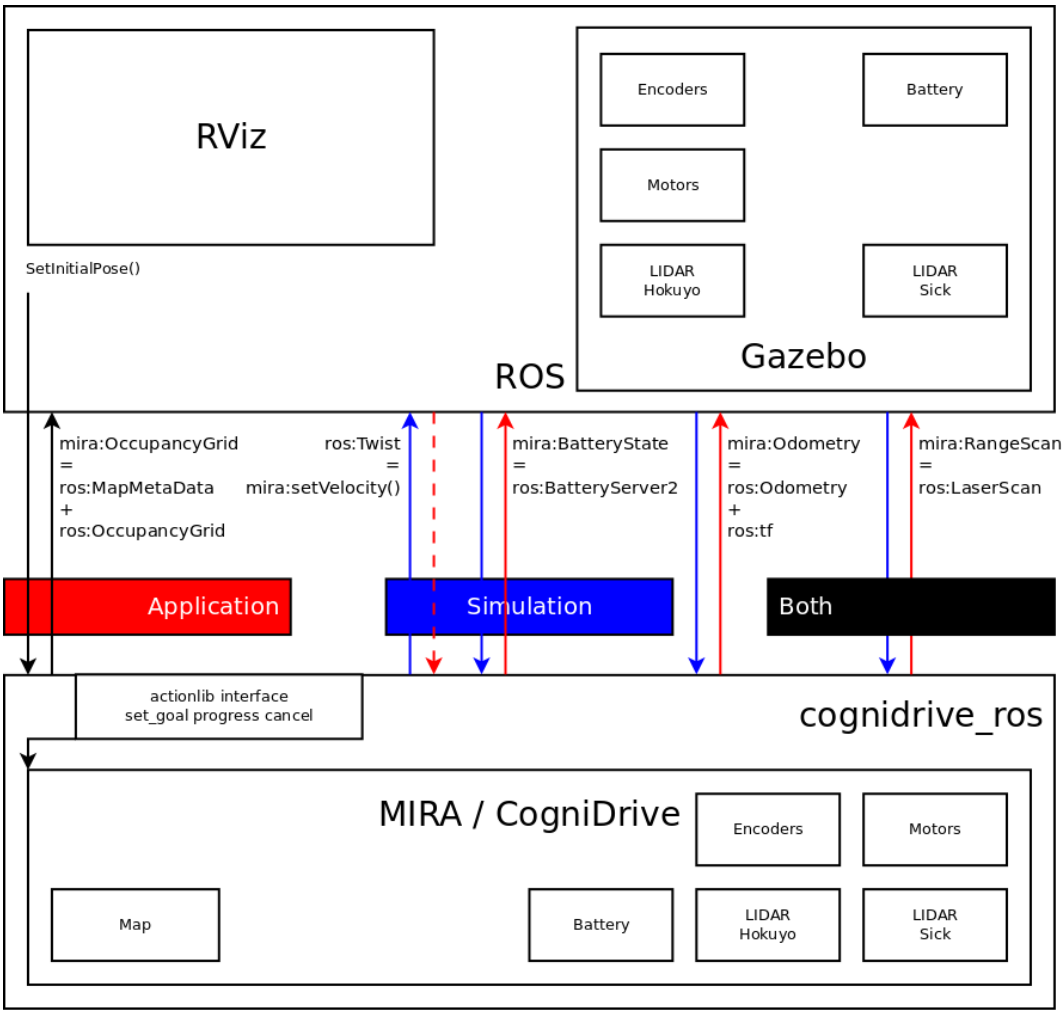


Figure 21: Architecture of the ROS-Cognidrive bridge. See the text for details.

When *cognidrive_ros* starts (as a ROS node), it also starts a complete MIRA framework, forwarding all command-line arguments. If you pass

- `-c | --config miraconfig.xml`, the contained MIRA framework will start in-process, loading all other MIRA units.
- `-k | --known-fw host:port`, the contained MIRA framework will connect to an already-running MIRA-framework on the given host:port.

Right now, the `-c` argument is disabled, because running MIRA and ROS in the same process leads to crashes. This is because MIRA uses the system's version of opencv (2.3) and ROS uses its own (2.4), but these versions are not binary compatible.

A typical way to use ROS with CogniDrive on the robot is as follows:

- change to the directory containing the MIRA configuration file (e.g. DomesticNavigation.xml),

- then start `mira -c DomesticNavigation.xml`. In this file, MIRA is instructed to listen on xml:root -> communication -> port (e.g. port 1234).
- start `cognidrive_ros -k 127.0.0.1:1234`, so that the MIRA functions in cognidrive_ros will connect to the instance of MIRA you started above.
- start `cognidrive_ros -k 127.0.0.1:1234 --simulation` when running in the Gazebo simulator, so that MIRA is getting data from the simulator.
- start `rosrun rviz rviz` to visualise laser scans and transforms, set pose estimates or 2D navigation goals.
- start `miracenter` and connect to the MIRA framework at address 127.0.0.1:1234 to see the same things in MIRA.

Setting the initial robot pose When the robot is first started, it may not be localised correctly. While the AMCL algorithm is capable of localising the robot after some movements, it is usually safer to specify the (approximate) initial robot pose. This can be done by publishing a pose to the */initialpose* topic, for example when starting from a fixed, known position of the robot.

Alternatively, start *rviz*, then enable both the */map* and the laser-scanner data */base_scan* and optionally */basescan_rear*. Press the *2D Pose Estimate* button from the button bar, click-and-hold the mouse at the approximate (x, y)-position of the robot, then drag the mouse to specify the robot orientation Θ, and release the mouse to adopt this position. Repeat, until the laserscan data matches the map.

Setting the navigation goal To move the robot to a given position and orientation, simply publish a pose goal to the */move_base_simple/goal* topic. Again, this can also be done interactively in *rviz* via the *2D Nav Goal* button in the button bar, then using click-and-hold to specify the (x, y) position of the target position, and finally using mouse-drag to select the Θ orientation of the robot. The robot motion will start as soon as the MIRA planner has calculated an obstacle-avoiding motion plan, which can take a few seconds.

Creating the map The map required for localisation can be drawn by hand, or can be created by driving the robot around and using SLAM to build the map incrementally. See the MIRA tutorials and reference manual for details. When updating the map file in the MIRA configuration xml file, also check to adjust the offset and orientation of the map.

3.3 Sensing and Perception

3.3.1 Overview

The Domestic Robot platform provides several different sensor systems.

- two laser range finders
- sonar sensors
- Asus Xtion Pro (RGB-D camera, comparable to Microsoft Kinect)
- RGB camera with tele-lens (firewire)

All different sensor systems are integrated in ROS in order to achieve:

- unified interface
- sharing devices between multiple subscribers.

3.3.2 Pan-Tilt Unit

The pan-tilt unit itself is an actuator system, but closely related to the sensory systems of the Domestic Robot, as it is used to change the direction of the Kinect- and RGB-cameras.

- ptu/cmd
- ptu/joint_states

```
rostopic pub -1 ptu/cmd sensor_msgs/JointState
  "{ header: { stamp: now },
     name: ['ptu_pan_joint', 'ptu_tilt_joint'],
     position: [1.57, 0], velocity: [0.5, 0.5] }"
```

3.3.3 Camera System and Image Processing

GStreamer-ROS-Adapter Due to several drawbacks in the gscam-node, we implemented an advanced ROS-GStreamer adapter

The open-source multimedia framework GStreamer [44] is used by many multimedia applications under Linux. Many functions needed for these applications are already implemented in GStreamer, like format conversion, image resizing, encoding, decoding, timing and network data transmission. The GStreamer framework is plugin-based, so the functionality can be expanded by new elements that can also define their own data types. The elements are connected to a processing pipeline, so that many operations can manipulate image data consecutively.

There are several reasons why we consider GStreamer as a suitable framework for handling high bandwidth multimedia data on a robot system. These are mainly:

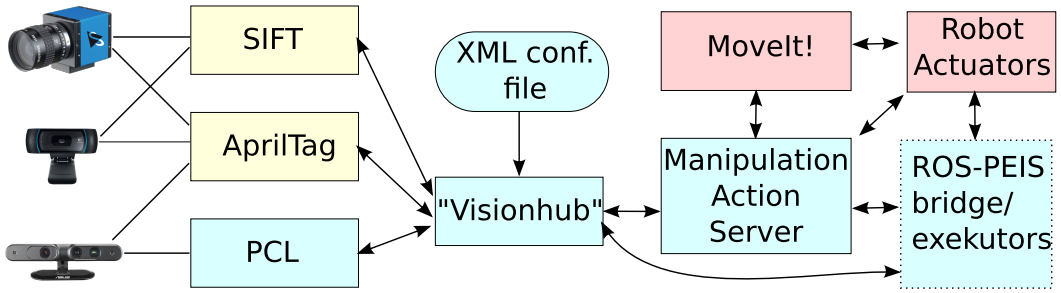


Figure 22: Overview of the updated perception pipeline. The data from the cameras on the robot including the Xtion sensor is forwarded to the SIFT-based recognition of known objects and a marker-based pose-recognition algorithm. In parallel, depth-images and point-cloud data from the Xtion sensor is processed using the tabletop_object_recognition stack to find clusters that correspond to (graspable) objects. The recognition results of the different pipelines are then fed into the multi-modal vision-hub note that performs data fusion and outputs a list of detected objects together with the object pose information and covariance estimates.

- efficiency of implementation
- large amount of available functions
- flexibility in the setup of pipelines

One important development objective of GStreamer is to generate as little overhead as possible. The most important principle applied is the “zero-copy” paradigm. The elements mainly exchange pointers to buffers containing the actual data. But GStreamer goes a step beyond this paradigm and allows downstream buffer allocation. This technique allows to “ask” the next element for a buffer (e.g. a mapped memory region from the video card) where the frame is directly rendered into. Exactly this technique makes GStreamer ideally suitable for developing adapters to arbitrary frameworks like ROS, as it allows GStreamer components to directly access the memory regions of these frameworks.

GStreamer allows the construction of various types of pipelines. Beside standard linear pipelines that consecutively apply filters to the sensor data, it is possible to construct branched pipeline graphs. Even in this case, no unnecessary copies of data are made. Only if an element wants to apply “in-place” data manipulation, a copy is created automatically if other elements also use this data buffer (i.e. copy on write). It is possible to implement different types of elements for data transfer. For the previous element in the pipeline, it makes no difference whether the following element for example writes the data to disk, sends it via TCP or transmits it via a framework like ROS.

Timing issues can be analysed by the so-called timestamps that every unit of data (e.g. one image of a video-stream) provides. We set this value to the current NTP timestamp directly after the image was captured. In different stages of the processing pipeline, the latency can be determined by comparing the timestamp to the current system time. Therefore, we have to synchronise all systems to an NTP timeserver. In a local area network, the achievable accuracy is better than 1 ms.

The "rossink"-element will act as a sink for image data inside the GStreamer framework. During the start sequence of a pipeline, this element will advertise a camera in the ROS-framework. Usually, the following sequence of actions is performed for each frame:

- An upstream element requests a buffer.
- "rossink" creates a ROS "sensor_msgs/Image"-message and provides the pointer to its payload to the upstream element.
- The upstream element renders the image into the buffer and passes it to the "rossink".
- The "rossink" will look up and publish the "sensor_msgs/Image" based on the pointer address

The "rossrc"-element acts as a source inside the GStreamer framework. It behaves like elements that allow access to camera hardware and provides a sequence of image buffers. At the start of a pipeline, this element will subscribe to an arbitrary ROS-topic. This element performs the following sequence of actions:

- When a new message is received, a callback function is called.
- In the callback function, a GStreamer buffer is created and the memory address is pointed to the payload of the "sensor_msgs/Image"
- The GStreamer buffer is sent to the next element.

The "rossrc"-element will store properties of certain GStreamer elements on the ROS parameter server. These parameters can be chosen manually. During runtime the element watches the parameters for changes on the parameter server and propagates them to the corresponding element. This check is also performed vice versa.

One feature of our implementation is that it becomes possible to access cameras that are integrated in ROS. Even simulated cameras (e.g. from Gazebo) can be use and the video stream can be handled in different ways (published via an RTSP server, recorded, displayed).

Installation of the GStreamer-ROS libraries While for the first evaluation loop the Gstreamer libraries had to be compiled separately from the ROS workspace, they are now included in the *catkin* workspace, are built within the *catkin_make* procedure and do not need to be installed in the systems' lib directories.

Usage The pipelines are started with the launch files, located in the gstreamer_pipelines directory. They contain methods to run GStreamer Pipelines, for example the launch file *doro_right_camera.launch* starts the following pipeline:

```
gst-launch dc1394src
! queue leaky=2 max-size-buffers=1 ! ffmpegcolorspace
! "video/x-raw-rgb , bpp=24, framerate=15/4"
! timestamper t1=-1 t2=1
! rossink topic=right_camera frame_id=RightCameraLens_link
    camerafile=/etc/rightcamera.txt sync=0 peeralloc=1
```

It is possible to adapt the frame-rate by changing the value 15/4 (=3.75 fps) to 15/2 (=7.5 fps) or 15/1 (=15 fps). The camera calibration file is */etc/right_camera.txt*. Also, it is configured that the images will be published on the topic *right_camera* and the frame_id is RightCameraLens_link (the latter parameter matches the URDF file)

Calibrating the Camera In order to (re-) calibrate the camera, first the camera needs to be started.

Terminal1: roscore
Terminal2: domestic_bringup
Configure the file-name where the camera calibration files will be stored, you need to have write access it they should be updated automatically.

(on startup, the system may complain that it did not find the calibration file)

Type: "rostopic list"

gives you something like /left_camera/image_raw

You can check this by running:

```
rosrun image\_view image\_view image:=/left\_camera/image\_raw
```

With the calibration pattern found in Örebro living lab:

```
rosrun camera\_calibration cameracalibrator.py
   --size 8x6 --square 0.02986
   image:=/left\_camera/image\_raw transport:=compressed
   camera:=/left_camera
```

With the PR2 calibration pattern pattern:

```
rosrun camera\_calibration cameracalibrator.py
   --size 5x4 --square 0.0245
   image:=/left\_camera/image\_raw camera:=/left_camera
```

Run/rerun the command from above:

- Move the checkerboard to all corners, turn it around, until about 50 images are captured.
- The green "Calibrate" button will be available - press it
- Be patient - window will hang for about 30 seconds, but it is NOT crashed !!
- Adjust the image size with the slider
- **either** save - saves raw data to /tmp - look at the console output (you manually have to copy the txt file to */etc/left_camera.txt*)

- **or** commit: advises the camera node to store the value internally - overwrites the camera config file */etc/left_camera.txt*
- if you get an error message, probably the Gstreamer-ROS has no write access to the file (see note from above)

For additional info, see *http://www.ros.org/wiki/camera_calibration*

In the current OpenCV libraries from ROS Groovy, there is a bug leading to a crash in the calibration routine. Here is a workaround: NOTE: This Only needs to be done once, and only, if camera_calibration crashes. It may be necessary to reinstall OpenCV after an update of the OpenCV libs (e.g. after "sudo apt-get upgrade")

Install OpenCV from the scratch... overwriting the previous...

- find out which OpenCV version is installed
  - cd /opt/ros/groovy
  - find . | grep libopencv
  - the output is something like 2.4.4
- get the 2.4.4 Source...(in case you have installed the latest groovy updates, otherwise 2.4.3)
- unpack the sources
- backup your ROS folder:
  - cd /opt/ros
  - tar czf groovy_backup.tar.gz groovy/
- cmake -DCMAKE_INSTALL_PREFIX:PATH=/opt/ros/groovy .
- sudo make install

This will overwrite the OpenCV libs installed from the ROS repository.

Publishing Rectified Images In order to use publish rectified image, the "image_proc" node is started in the cameras.launch file with the call:

ROS_NAMESPACE="/left_camera" rosrun image_proc image_proc

Check "rostopic list" in order to see the additional topics.

Example: Display the Rectified image:
rosrun image_view image_view image:=/left_camera/image_rect_color

For additional info, see http://www.ros.org/wiki/image_proc

3.3.4 XtionPro RGB-D Camera and Point-Clouds

If

```
roslaunch openni\_launch openni.launch
```

can not be executed, we have to re-install it separately. sudo apt-get install ros-hydro-openni-launch

Starting the tabletop segmentation:

```
roslaunch tabletop\_object\_detector tabletop\_segmentation.launch
   tabletop\_segmentation\_points\_in:=/xtion\_camera/depth/points
```

Check whether the object detector works:

```
roscd tabletop_object_detector
bin/ping_tabletop_node
```

The return values have the following meanings:

- Error 1: No cloud received
- Error 2: No Table detected
- Error 3: Other Error (see console output of tabletop_segmentation.launch console, problems with TF are always a hot candidate for errors)
- Return Value 4: No error

3.3.5 MJPEG-Server 2

ROS provides the possibility to stream arbitrary video topics directly to a browser (including mobile devices). The MJPEG-Server has been extended in order to support modern browsers and to save and load snapshots of the current scene. The package is called '"mjpeg_server2'" and is located in the Catkin-Workspace of the Robot-Era software repository.

Manual start:

rosrun mjpeg_server2 mjpeg_server2 in order to start the Webserver on port 8080.

In order to start it on a different port, run:
rosrun mjpeg_server2 mjpeg_server2 _port:=8081

You can connect to the video streams using your web-browser and opening the address:
http://$ROBOT:8080/stream?topic=/left_camera/image_raw

Where $ROBOT has to be replaced with the hostname or IP of the robot ("doro").

The command above will make the webserver subscribe to the Topic /left_camera/image_raw, you can use it the same way for other topics. In order to change the resolution or quality (in order to save bandwidth), you can use

http://$ROBOT:8080/stream?topic=/left_camera
/image_raw?width=320?height=240?quality=40
Quality can be chosen from 1 (lowest) to 100 (highest).

doro.informatik.uni-hamburg.de:8081/stream?topic=/xtion_camera/depth/image

MJPEG-Server2 - Additional Features Due to a long-standing bug in Firefox (https://bugzilla.mozilla.org/show_bug.cgi?id=984362) and due to a regression of not supporting raw MJPEG streams any longer in Webkit based browsers (https://productforums.google.com/forum/#!topic/chrome/PslmTeeM8Fo), the MJPEG-server had to be adapted. If the stream is embedded into a web page, it is displayed correctly with both browsers. When accessing the address

http://$ROBOT:8080/embeddedstream?topic=/left_camera/image_raw

a simple HTML-website is delivered with an embedded stream. The parameters to configure the image topic and resolution is exactly the same as explained on the project website. Internally, from this website the browser will then call

http://$ROBOT:8080/stream?topic=/left_camera/image_raw

while passing through all your parameters.

Additional functions:

Sometimes it might be useful to store and load images, for example when detecting an event like an intruder. There are two different options how the images can be saved: in RAM using slot 0..9 or persistently to files.

Saving a snapshot to slots (RAM):
http://$ROBOT:8080/store?slot=1?topic=/left_camera/image_raw

Saving snapshot to a file:
http://$ROBOT:8080/store?topic=/left_camera/image_raw?file=/path/to/filename.jpg

Loading images stored on a slot (If no image has been saved on this port you will get a 404 error page.):

http://$ROBOT:8080/load?slot=1
Loading images from a file:
http://$ROBOT:8080/load?file=/path/to/filename.jpg

It is possible to adjust the width and height of the delivered image, append these parameters to the request, separated by '"?'" (default: native image size, only valid if both width and height are specified) It is also possible to adjust the image quality this way (quality)

Example:
http://tams115:8080/embeddedstream?topic=/leftcamera/image_raw
?width=100?height=80?quality=40

Embedding the stream into a web page: It is possible to embed one or multiple (some browsers limit the number) streams into a web page, for example the UI for the robot. Sample HTML code:
<html><body>
<img src="http://$ROBOT/stream?topic=/camera/image_raw">
</body></html>
Of course, the image size can be adapted like described above.

Note: It is only necessary to start one instance of the MJPEG-server to publish all available ROS-integrated cameras.

3.3.6 Object Detection and Pose Estimation

The Domestic Robot platform provides three main object detection and localisation algorithms:

- SIFT-based object detection
- PCL-based object detection (including SIFT based classification). In this case SIFT is only used to classify the defected point cloud by finding the best match in the object database.
- AprilTag Markers

The different methods and their results are managed by a component called *Visionhub*. The concept of this ROS node is shown in figure 22.

3.3.7 Visionhub

This is a ROS node that acts as an interface between nodes for image processing (currently AprilTags and SIFT is supported), manipulation/planning nodes. It has been built in order to interface to an ontology or context awareness module that defines certain objects, but has also built in functionality. The reason why this node has been implemented is that on a higher level (grasping/planning) you just want to know which objects have been detected and what pose they have, but you want to hide the complexity of the underlying detection process. So it is irrelevant for the action-planning-nodes doing manipulation whether the detection has been done using SIFT or other algorithms or which cameras have taken the image.

Basically the Visionhub node provides the following function to publish which objects are detected and how they can be grasped. Therefore this node reads from a database, that contains the following information about objects:

- dimensions of the bounding box
- surfaces (image files) and their exact position
- AprilTag Markers and their size and position
- information on grasps (grasp-points maximum force)

Visionhub: Defining Objects The long term plan is to support different types of databases. Currently the objects need to be specified by an XML file. The file format should be self-explaining, a commented sample file is provided in the repository under:

The top level entity is *object*. This is usually an object (including other robots), or also a room (in order to implement initial self localisation). Beside the robot name, a box can be defined. This can be either used as a bounding box for collisions aware motion planning or

also for calculation of grasps. When defining this box, one should be aware of the coordinate systems attached to objects.

The selected orientation of the coordinate system complies with the coordinate system that has been chosen for the robots: -X axis pointing in the forward driving direction -Z axis pointing to the sky (given that the robot/object is in a upright position) -Y axis according to a right handed coordinate system (pointing to the left side from the looking forward view of the robot)

Surfaces: It is possible to support an arbitrary number of (sub-) surfaces, each with its own pose and size. In usual cases one might want to specify exactly the six surfaces of a box (front, back, left, right, top, bottom) or possibly only the front surface. For modelling/detecting the six surfaces, it is sufficient to generate border-less images of the surfaces, orientation according to figure 23, and use the <side>front</side> syntax. This will internally define the transformation from a coordinate system of the surface to the coordinate system in the centre of the object. Tags poses can also be described this way when attached according to figure 23. After <side> has been specified, it is still possible to overwrite the position (in order to describe offsets). When manually defining the transformation, it is important to know the coordinate attached to surfaces (and tags). For surface coordinate systems the X and Y axis comply with the x-pixel-value-axis (width) and the y-pixel-value-axis (height), but the origin is in the centre of the image (pixel coordinates {(width/2.0),(height/2.0)} while the image CS has the origin in the top left corner.

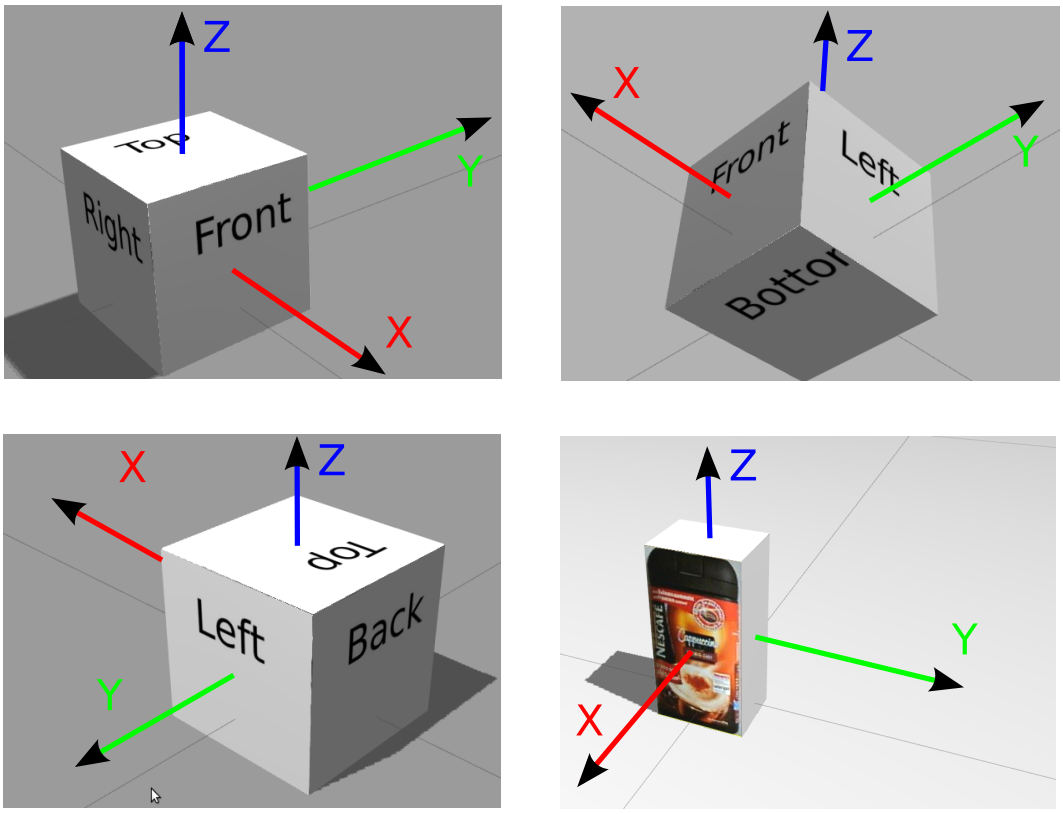


Figure 23: The orientation of the coordinate system that we assign to objects.

Visionhub Services

- get_detected_objects()

Returns a vector of string elements containing the name of the detected objects

- get_possible_grasps(objectname, max_gripper_opening, min_object_width)

For a certain object, this function provides a list of grasp points, that may be auto-generated based on the geometry of the object, manually defined or learned ones. If the maximum applicable force is defined in the XML-file/database, this value will be also be provided.

In addition to each Grasp Pose the Transformation of that Grasp Pose to the ground surface is stored. This information can be used when a manipulator needs to place an object onto another surface.

- place_announce:

With this service it is possible to notify visionhub, that an object has be placed somewhere in the environment or on the robot (or even on a different robot). The main intention of this service is that at the new pose the object may probably not be detected anymore by the robots camera systems, but it shall still be able to **grasp it from the tray**, **drop objects into a bin** that it currently cannot see (maybe currently placed on the tray) or suggest to **drive the robot** to a place where a desired object may be detected again (if the robot has seen it before)

- last_known_pose:

The "'last_known_pose"' principally acts as the "'get"' operation for the list of the known objects. It is possible to ask visionhub, where the object has been detected the last time. The client to query certain coordinate systems of the object, that will not be published on the /tf topic in order to save bandwidth. This can be used to drive to the object in order to grasp it, or to drop something in/onto the object.

3.3.8 SIFT-based object recognition

This function will detect objects and publish visualisation markers and stamped poses containing detected objects. It works well for bigger boxes, and may also work for other shapes. For latter, better rely on point clouds for pose and take the result of this algorithm only for classification. The SIFT-based object detection works the following way: All the images of the objects can either be placed in a folder or can be loaded using a query to the *visionhub* node. The images should be taken from a perpendicular angle. The border should be cropped, or the width number should be adjusted accordingly.

In order to calculate the pose, the calibration of the camera needs to be considered. Currently only the file-format of OpenCV is supported. The calibration is already ready and does not need to be repeated.

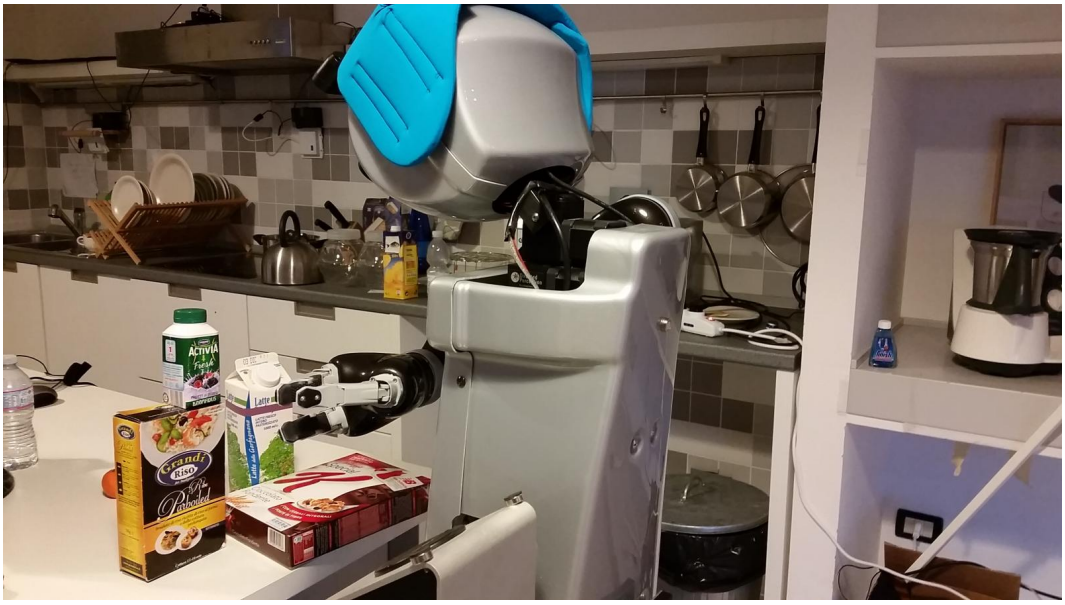


Figure 24: Object manipulation test in the DomoCasa living lab, Peccioli. Doro is looking at a cluttered table scene with objects that have been trained with the SIFT object recognition back-end. Note that the Jaco arm is still in the safe home position used for driving around and as a start position for manipulation. The arm is just outside the camera image.

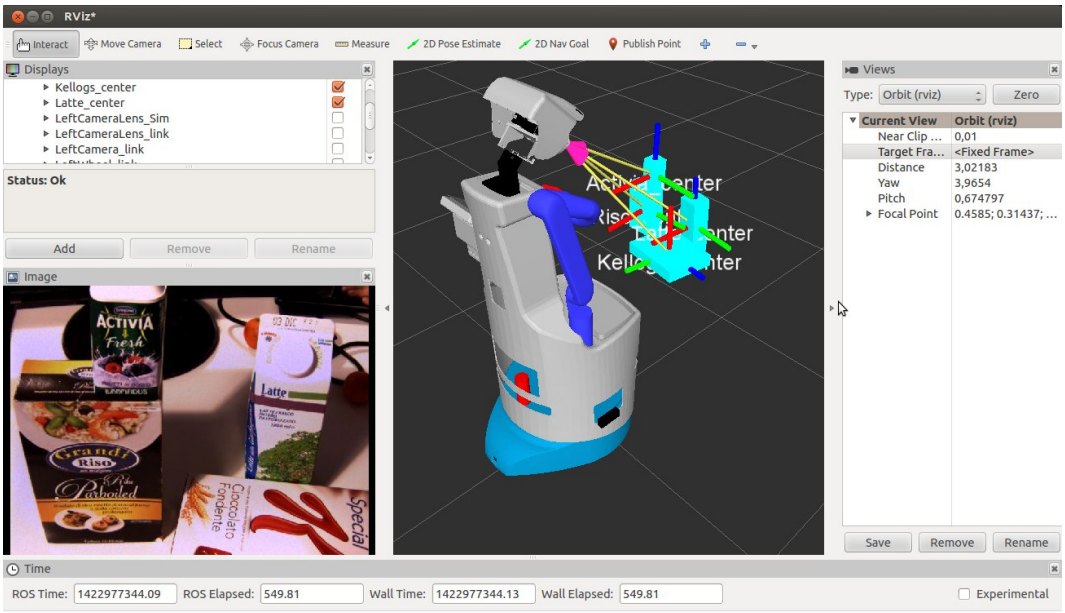


Figure 25: This screen-shot shown the object detection using visionhub and the SIFT back-end. All four objects visible in the camera image have been detected on the cluttered table scene, and their 6-DOF poses are estimated correctly. The objects are modelled in the database with their dimensions. Based on the object markers, grasps can be calculated.

The node publishes on the topic "siftobjects", type <visualisation_msgs::Marker> that puts out ALL detected objects for visualisation in Rviz. If an object is no longer detected, it publishes one "DELETE"-message according to the specification of the message type.

"siftobjects/objectname", type <geometry_msgs::PoseStamped> one topic for each object. NOTE: only advertised if the object is detected the first time. No delete message is sent, so please look at the timestamp.

"tf" type <tf/tf Message> This type of message describes the transformation between the camera- and the object-coordinate frame.

Using the tf-system of ROS it is easy to obtain other transformations like robot_base to object or gripper to object.

The SIFT implementation is included in the catkinized gstreamer_pipeline directory, internally for example the *start_siftserver_right.launch* file starts the following pipeline:

```
LD_PRELOAD=/opt/ros/groovy/lib/libimage_transport.so
  gst-launch rossrc topic=right_camera/image_raw
    ! queue leaky=2 max-size-buffers=1 ! ffmpegcolorspace
    ! siftextractor
    ! rossiftserver
        caminfotopic=right_camera/camera_info
imdbsrv=/get_image_sizes
    ! fakesink
```

This node can be configured as follows:

- *imdbsrv=/get_image_sizes*
  specifies a ROS-service (Visionhub) to look up which image files should be used for matching and what physical size the surfaces have.
- *caminfotopic=right_camera/camera_info*
  specifies the camera-info topic to retrieve the calibration parameters for the selected camera automatically.
- alternatively, you can specify the directory where the calibration parameters (*Distortions.xml* and *Intrinsics.xml*) are stored via this parameter:
  unddirectory=/localhome/demo/camera_calib_data/cal_April2013

3.3.9 AprilTag Marker detection

The integration of AprilTags has been done using an existing ROS node that has been extended in many ways. The following additional features have been implemented:

- automatically read out camera calibration from the corresponding topic
- support for markers with different sizes (configured through the “visionhub” node)
- high performance tracking of detected markers
- performance improvements and bug fixes

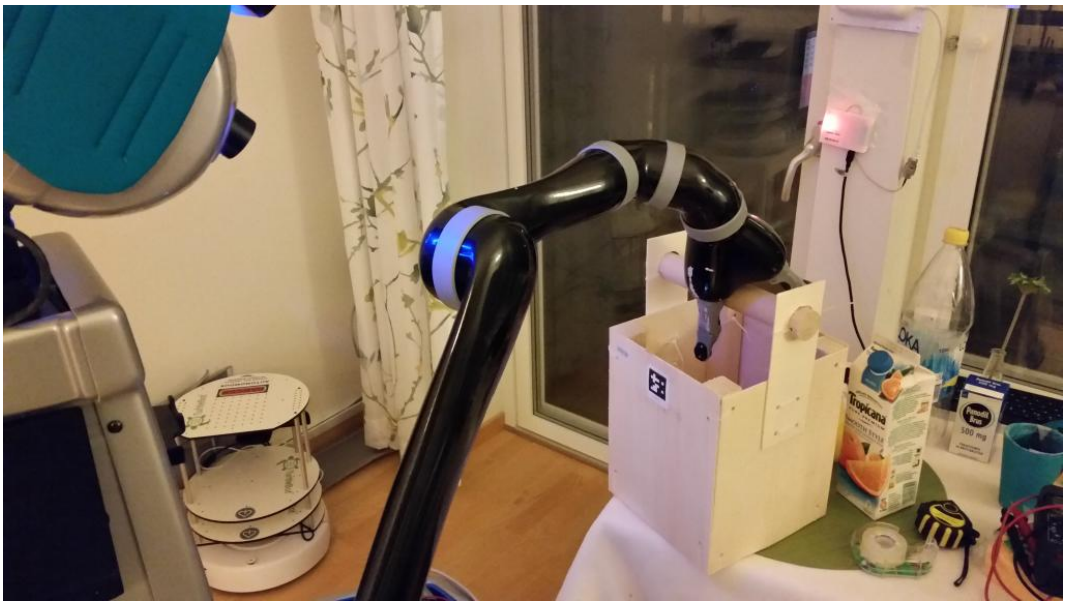


Figure 26: Object manipulation test in Angen, Örebro. Doro is grasping the prototype box for the garbage and laundry scenarios. As the wood lacks any visual texture suitable for SIFT feature matching, an AprilTag fiducial marker has been placed on box. Despite the small size of the marker tag, marker detection and identification are very reliable, and pose estimation errors are usually well below 1 mm position error.

3.3.10 Tabletop segmentation

In the first step of the perception pipeline, the table (dominant flat surface) is identified from the incoming sensor data using the RANSAC algorithm, listening on the */cloud_in* (*sensor_msgs/PointCloud2*) topic. The *Table* message contains the pose and convex-hull of the detected table.

Before performing the tabletop segmentation step, the robot should be close to the table, with the sensor head oriented so that a large part of the table and all interesting objects are in view. If possible, the robot arm should be moved sideways, to minimise the impact by self-occlusion of the table and objects.

Once the table is known, all point-cloud points above the table are assumed to belong to graspable objects, and a clustering step is applied to combine multiple points into larger clusters, which are then considered as the individual objects. The minimum inter-object distance used for the clustering is specified via the *clustering_distance* parameter, and defaults to 3 cm. To speed-up the table-detection and clustering process, additional parameters are provided by the software; default values are set in the launch files for the Domestic Robot. The point-cloud clusters found by the *TabletopSegmentation* service are indexed and returned in the result value of the service. Additionally, markers corresponding to the clusters are published on the *markers_out* topic and can be visualised in rviz.

3.3.11 Human Detection and Recognition

For the second experimental loop, human detection and recognition software will be tested on the Domestic and Condominium Robots. The detection modules have be encapsulated as PEIS exekutor services, and the information can be merged with the ambient sensor data (e.g. chair switches, wearable sensors) by the configuration planner. See section 4.2.15 on page 130 for details and the tuple definitions.

As people tracking is a common tasks, several independent ROS packages are available and have been evaluated:

- The *person_detection* stack uses functions from the PCL library and a pre-trained SVM to detect people in the pointclouds generated by the Kinect/Xtion sensor. The algorithm publishes the head positions of detected persons as well as their estimated motion with respect to the robot. The stack has been tuned and refined by UHAM to improve robustness and the detection of sitting people.

  This algorithm is currently used on Domestic Robot for the find_user service. Work has been started to also include face detection and face recognition algorithms to further improve recognition rates.

- The *openni_tracker* package is based on the people movement tracking originally developed for the Microsoft Kinect sensor. The current implementation is based on the NiTE tracking library from PrimeSense, and ready-to-run installers exist for ROS Hydro. The software directly accesses the depth-image from a Kinect/Xtion sensors and detects and tracks up to six persons simultaneously. Once the so-called Psi-pose is detected for a person (standing with the upper arms extended and the lower arms pointing upwards), a calibration sequence starts and afterwards the library tracks the users' head, arms, and legs. The corresponding poses are published as individual transformations to the ROS tf system.

  The tracking detects standing and sitting people, and can track moving people even with occlusions. However, the tracker is not suitable for people that are lying on a bed or on the floor. The ROS node only publishes arm and leg information after calibration has been performed for a person.

  Unfortunately, the original (openni) software is not compatible with the XtionPro sensor mounted on the Domestic Robot. While a replacement library (*NiTE2, openni2*) is available, the new library requires exclusive access to the sensor, so that image and point-cloud data is no longer available to other ROS nodes.

- The ROS *people detection* stack consists of several independent modules optimised for different usage scenarios. The *leg_detector* searches for legs in 2D laser scans, and *people_tracking_filter* in turn uses the candidate legs to identify and track moving people. The *face_detector* uses OpenCV image processing to find faces in 2D camera images. See wiki.ros.org/people for details. Unfortunately, little documentation is available for the different software modules.

- The *cob_people_detection* stack combines the OpenNI person tracking with face-detection and face-recognition (Eigenfaces, Fisherfaces) to identify known persons.

To add new persons to the databases, the robot simply looks at a person and then performs the steps to generate and managed the Eigenface data required by the recognition algorithm. See wiki.ros.org/cob_people_detection for details. At the time of writing, the stack was not available for ROS Hydro.

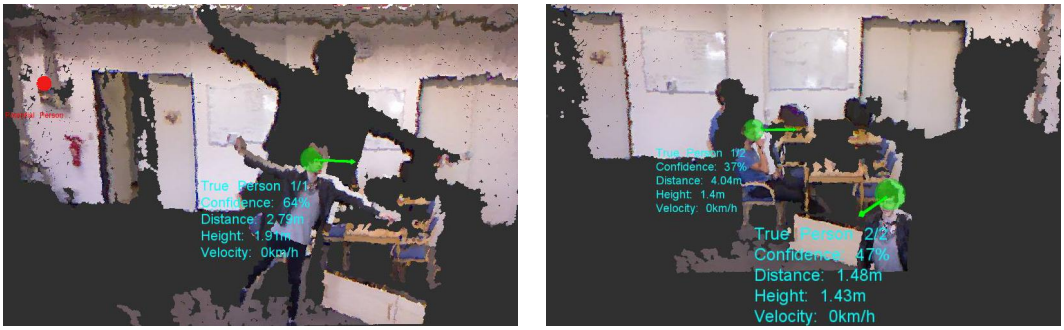


Figure 27: Typical output of the ROS person_detector module. The algorithm first performs a ground-plane detection to identify the floor and then searches for people-sized clusters in the point-cloud generated from the Kinect/Xtion camera. A pre-trained SVM is used to classify the clusters, and the estimated head position (x, y, z) and motion are published to ROS. The algorithm detects standing as well as sitting people. [12].

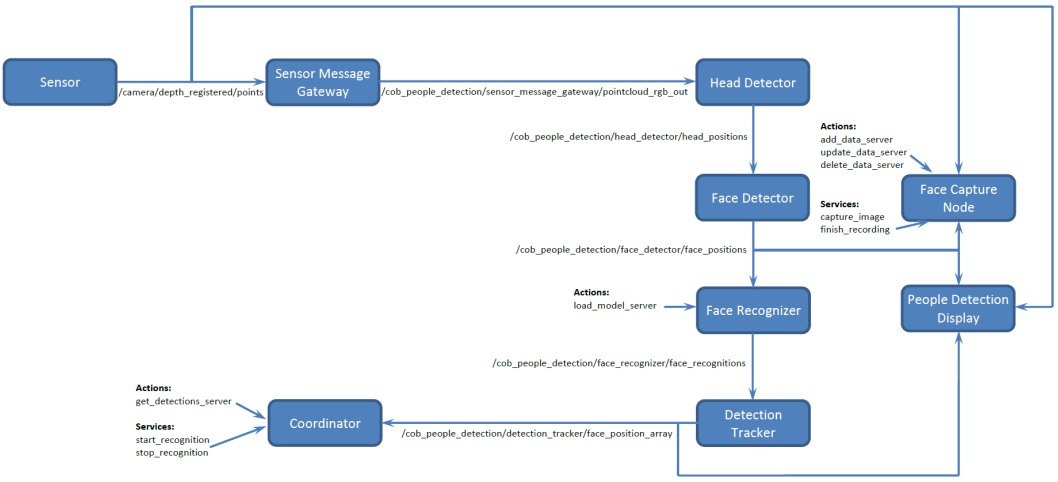


Figure 28: Main ROS nodes in the *cob_people_detection* module for people detection and face recognition. (Image from wiki.ros.org/cob_people_detection).

3.4 Manipulation

Object manipulation with robots is standard industry practice by today, but the typical factory solution is characterised by a strictly controlled environment and highly regular tasks. The robots are equipped with grippers and tools matched to the particular target objects, robot and object positions are known, and trajectories may be hard-coded. Sensor feedback, if used at all, is only required for slight corrections and adaptations to initial object positions.

In the context of service robots, however, manipulation is still unsolved and remains one of the key research challenges. Even for apparently simple tasks like picking up an object, several complex sub-problems must be tackled. First, perception algorithms must be able to detect the object in a potentially cluttered environment, and then to estimate the object pose in regard to the current robot position. As full scene understanding is far beyond the reach of computer vision, simplifications are required.

Second, to reach the target object a collision-free robot arm trajectory must be calculated, leading to complex motion-planning problems depending on the number and complexity of obstacles. Picking up a single mug from a table is easy, but putting the same mug into a dishwasher full of other objects is very challenging. Once the object is grasped, the motion-planning must take the object into account when calculation new trajectories.

Third, to grasp an object the robot gripper or hand must be configured to reach suitable contact points on the object, and to apply forces that allow to lift the object against gravity and stabilise the grasp against disturbances in free space. Fourth, manipulation often involves moving an object in contact to other objects, applying forces depending on the task context. For example, swiping a table, opening a door, or turning a screw requires highly precise motions that take the kinematic structure of the environment into account.

According to the tasks described in the Robot-Era scenario storybooks, all of the above problems need to be tackled and implemented on the Domestic Robot. As explained in chapter 4 below, an incremental approach is taken. First, simple reach and grasp motions re implemented for the robot, which are then refined and integrated with perception to more complex tasks. As force-control is not available on the Kinova Jaco arm, it remains to be seen to which extent actual manipulation motions can be realised.

The next section 3.5 first summarises the key features of the Kinova Jaco robot and the software architecture to control the arm and gripper. Next, section 3.6 sketches the recent MoveIt! framework, that is used for the manipulation tasks in the second experimental loop and has completely replaced the previously used manipulation stack on the Domestic Robot. MoveIt! is under heavy development, but we expect to track the progress and use the advanced manipulation capabilities, adapted to the Kinova gripper, for the Domestic Robot.

In section 3.7 the service node is described that actually connects to the MoveIt and Visionhub and provides higher level services (e.g. grasp object), that are then called by the executor nodes, which then export the functionality to the PEIS framework.

3.5 Kinova Jaco API and ROS-node

As explained in the previous project report D4.1 [57], the Kinova Jaco arm was selected for the Domestic Robot, due to its proven record in wheelchair applications, acceptable payload, the low-noise operation and pleasing outer appearance, and last but not least the availability of suitable software including the experimental ROS interface. The nominal payload of the arm is 1.5 kg at the gripper, but cannot be applied continuously. Depending on the payload, the arm needs rest periods to avoid overheating of the motors. Fully extended, the reach of the arm is about 90 cm from the base to the grasp position between the fingers.

From the mechanical point of view, the Jaco is a fairly standard 6-DOF robot arm with an integrated 3-finger gripper. The joints are driven by high-performance electrical DC motors with planetary gears, where the lower three joints use larger motors and gearboxes for higher torque. All joints are specified for pretty high speed, but are limited in software to slow motions that are considered safe around humans. Please see the documentation from Kinova for details and the exact specification of the arm and hand.

Warning: no brakes Unlike most industrial robots, the Jaco arm does not include brakes on the joints, and the gearboxes are not self-locking. When powered down, the arm will collapse under its own weight, unless it has been parked in a suitable rest position. This can damage the arm, any payload carried during power loss, and of course also objects and humans near the arm.

Warning: no emergency stop There is currently no emergency stop on the Jaco arm, neither via mechanical emergency buttons nor via software. The current ROS node allows us to cancel an ongoing movement, and commanding the current position stabilises the robot. Note that the emergency-switches on the SCITOS platform do NOT affect the Jaco arm in the current version of the robot.

3.5.1 Jaco DH-parameters and specifications

So far, Kinova releases key parameters of the Jaco arm only to licensed customers, including the DH-parameters of the arm kinematics and also the joint and motor specifications. Therefore, this data cannot be included in this (public) report. Please refer to the Kinova documentation for technical details and specification about the arm geometry, joint-limits and joint-torques, and operational limits of the motors.

3.5.2 Jaco Joystick and teleoperation

One of the biggest assets of the Jaco arm is the included high-quality Joystick together with its different operation modes. See page 15 for a photo of the three-axis joystick (left/right, up/down, twist) with a set of buttons and LEDs for visual feedback of the robot control mode. Depending on the skills of the user, either 2-axis or 3-axis is possible, with different modes to move the arm in cartesian space (translation), to orient the hand (orientation), and to control the fingers. Pressing the yellow-button will move the arm back to its two

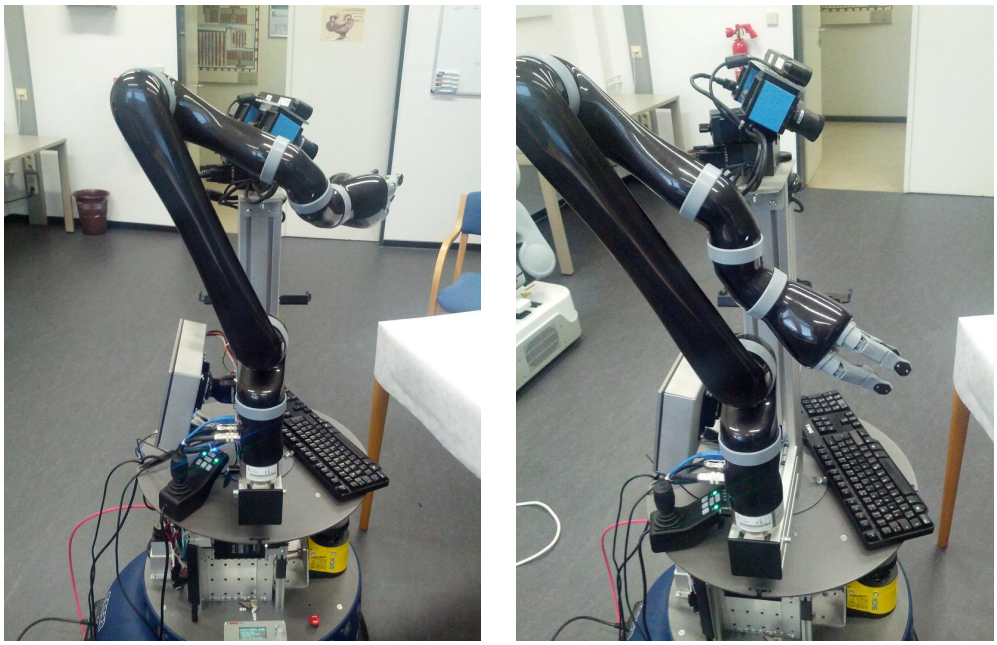


Figure 29: The *home* (left) and *retract* (right) positions of the Jaco arm. These are the two reference positions of the arm which can be reached by pressing (and holding) the yellow button on the Kinova joystick. Note that the *jaco_node* ROS node can only be used when the arm initialises into the Home position.

reference positions, namely the *retract* and *home* positions. The *drinking mode* provides a specific hand-rotation, where the IK solver keeps the rim of a user-defined glass or cup stable while rotating the hand.

Please see the Kinova documentation for usage information about the Joystick and the mapping between buttons and movements. Also check the Kinova API documentation about how to change the default *retract* and *home* positions used by the arm, and for the definition of safe-zones for arm movements.

3.5.3 Jaco .NET API

The Jaco arm was originally developed for use in rehabilitation, where the Jaco arm is teleoperated by the users via the supplied Joystick. This works well and can be done without any additional software, because the arm controller also includes the built-in IK solver.

However, the arm connects via USB to a PC, and Kinova offers software for arm configuration as well as a programming interface for remote control of the arm. The supplied software is written for the Windows .NET platform and distributed as a set of .NET DLLs. At the moment, the following libraries are included, which correspond roughly to the .NET packages defined in the API:

- Kinova.API.Jaco.dll

- Kinova.DLL.Data.dll
- Kinova.DLL.Tools.dll
- Kinova.DLL.USBManager.dll
- Kinova.DLL.TestData.dll
- Kinova.DLL.CommData.dll
- Kinova.DLL.TcpConnector.dll
- Kinova.DLL.SafeGate.dll

Kinova also supplies a Jaco arm control and configuration program. Its main user-interface provides access to the arm parameters and allows the user to configure the joystick, the safe regions to be avoided by the arm during movements, and the maximum speed. There is also a simple self-test, and a small set of example programs for C# and VBASIC. However, the example programs only use a very small subset of the API included by the Kinova runtime libraries.

Fortunately, the *mono* software environment can be used to wrap the .NET libraries on Linux. This avoids the additional complexity of using a separate control PC or a virtual machine for controlling the Jaco arm from the Domestic Robot and its ROS system. You need to install *mono-devel* and *mono-gmcs* software packages and their dependencies. It may also be required to install the latest version of *libusb-devel* for reliable USB communication with the arm controller.

At the moment, Kinova Release 4.0.5 (April 2010) is installed on both Domestic Robot prototypes; an update to the recently released research version 5.0.1 (February 2013) is planned. A large part of the Kinova API is dedicated to configuration functions required for tele-operation via the joystick, in particular the mapping from joystick buttons and axes to cartesian movements of the arm and gripper. Please see the Kinova *Jaco User Guide* and the *Jaco API Programming Guide* for details.

The *CJacoArm* structure is the basic abstraction of one arm, and is initialised via a call to the arm constructor, which expects the license key provided by Kinova as the password. Once initialised, the *CJacoArm* structure provides access to several members, namely the *ConfigurationsManager*, *ControlManager*, and *DiagnosticManager*. Before calling any other API function, the *JacoIsReady()* checks whether the arm is initialised and running, and should be called before any other function of the API.

The *ConfigurationManager* can be queried for the current arm configuration and parameters, e.g. the *MaxLinearSpeed* of the arm. Most arm parameters can be set by calling the corresponding *Set* functions. The *ConfigurationManager* is also used to define the *ControlMappings* and events for the joystick and any *ProtectionZones* that the arm is forbidden to enter. The *DiagnosticManager* is used for debugging, maintenance, and allows resetting the arm configuration to the factory defaults.

The *ControlManager* provides the functions relevant to autonomous robot control. Only joint position control and cartesian position control are supported at the moment. There are no tactile sensors on the arm and fingers, and force control is not supported. However, a rough estimate of joint-torques is available via measurement of the motor currents.

Important functions calls (and relevant parameters) are:

- *GetAPIVersion* — Kinova API software version
- *GetCodeVersion* — Jaco DSP software version
- *JacoIsReady* — true if working
- *GetClientConfiguration* — arm parameters
- *GetCControlMappingCharts* — joystick/button mapping
- *CreateProfileBackup* — safe configuration to file
- *GetPositionLogLiveFromJaco* — complete robot readings

- *GetCPosition* — voltage, accelerometer, error status
- *GetJointPositions* — CVectorAngle (joint angles)
- *GetHandPosition* — CVectorEuler (cartesian pose)

- *StartControlAPI* — start software control of the arm
- *StopControlAPI* — stop software control
- *Send JoystickFunctionality* — fake joystick events
- *SetAngularControl* — switch to joint-elvel mode
- *GetForceAngularInfo* — current joint-torques
- *GetPositioningAngularInfo* — current joint-angles
- *GetCommandAngularInfo* — current joint-positions
- *GetCurrentAngularInfo* — motor currents

- *SetCartesianControl* — switch to cartesian mode
- *GetCommandCartesianInfo* — current hand pose and fingers
- *GetForceCartesianInfo* — cartesian forces

- *GetActualTrajectoryInfo* — check current trajectory
- *GetInfoFIFOTrajectory* — check current trajectory
- *SendBasicTrajectory* — CPointsTrajectory
- *EraseTrajectories* — stops ongoing motion

Kinova does not specify the maximum rate allowed for calling the different functions, but the provided examples typically sample the joint-positions at 10 Hz. This should be sufficient for the first experiments in pure position control, but a much higher sample-rate and command-update rate will be required for fine motions and pseudo-force control implemented on top of a position-control loop.

Note that the software control of the arm is considered the lowest priority of all control methods of the Jaco. If the USB connection is lost, or if the Joystick is used, the software control is disabled similar to a call to *StopControlAPI*.

The *CPosition* structure includes some low-level information that can be useful for improved control. It contains the age of the robot since manufacture, the error status flag, the *laterality* flag (right-handed or left-handed arm), the *retract* state, the current supply voltage, built-in accelerometer readings, and several overload detection flags.

The default coordinate system used for Jaco cartesian control is right-handed with the x-axis to the left, y-axis to the rear (negative y is to the front), and z-axis upwards. This has been deduced experimentally when commanding poses with respect to the *jaco_base_link* link.

Therefore, the orientation of the coordinate system is different from the basic ROS coordinate system, and care must be taken when converting between (x, y, z) and $(X\Theta, Y\Theta, Z\Theta)$ angles for the Jaco *GetHandPosition*, *GetCommandCartesianInfo* and *SetCartesianControl* functions and ROS.

3.5.4 Jaco ROS integration

Despite its beta-status, the Kinova ROS stack already provides all major components for use of the Jaco arm in ROS. The *jaco_description* package contains the URDF model of the arm, the *jaco_api* package builds a C++ library that wraps the Kinova .NET DLLS, and the *jaco_node* package defines the *jaco_node* ROS node that communicates with the arm for real-time control. The stack also includes a set of launch and configuration files for ROS manipulation stack.

The *jaco_node* is the most important component. At the moment, the node initialises the following subscribers and publishers.

Subscribers:

- jaco_node/cur_goal (geometry_msgs/PoseStamped)
- hand_pose (geometry_msgs/PoseStamped)
- joint_states (sensor_msgs/JointState)
- jaco_kinematic_chain_controller/follow_joint_trajectory/result
- jaco_kinematic_chain_controller/follow_joint_trajectory/feedback
- jaco_kinematic_chain_controller/follow_joint_trajectory/status

Publishers:

- hand_pose (geometry_msgs/PoseStamped)

- cmd_abs_finger (jaco_node/FingerPose)
- cmd_abs_joint (jaco_node/JointPose)
- cmd_rel_cart (geometry_msgs/Twist)
- jaco_kinematic_chain_controller/follow_joint_trajectory/goal (control_msgs/FollowJointTrajectoryActionGoal.msg) Note that the current implementation of this publisher ignores the velocity, acceleration and time values information.
- jaco_kinematic_chain_controller/follow_joint_trajectory/cancel
  This publisher is not working (no implementation)

To move the arm to a joint-space position via the command line, just publish the corresponding joint angles (in radians, starting from the shoulder_yaw joint) the `/cmd_abs_joint/` topic:

```
rostopic pub -1 cmd_abs_joint jaco_node/JointPose
  "joints: [-1.7, -1.5, 0.8,-0.6, 1.5,-2.8]"
```

The joint-pose is published on */jaco/joint_states* and is also available as part of the global */joint_states* message,

```
rostopic echo /jaco/joint_states
```

To move the fingers to a given position (in radians):

```
rostopic pub -1 cmd_abs_finger jaco_node/FingerPose
  "fingers: [0.5, 0.5, 0.5]"
```

To move the Jaco arm to an absolute position in cartesian space (using the given ROS coordinate system, e.g. the arm base *jaco_base_link* or the robot base *base_link*):

```
rostopic pub -1 hand_goal geometry_msgs/PoseStamped
 '{ header: { frame_id: "base_link" },
      pose: { position: { x: 0.23, y: -0.23, z: 0.45},
              orientation: { x: -0.62, y: -0.3, z: -0.3, w: -0.65 }}}'
```

The absolute position of the arm is published on the *hand_pose* topic, but this seems not to be reliable in the current software version:

```
rostopic echo /hand_pose
```

Setting the relative position in cartesian space is also documented, but does not yet work:

```
rostopic pub -1 cmd_rel_cart geometry_msgs/Twist
  "{linear: {x: 10.0, y: 0.0, z: 0.0},
      angular: { x: 0.0, y: 0.0, z: 0.0} }"
```

The *doro_description/scripts* directory collects a set of small utility shell-scripts that encapsulate the verbose *rostopic pub* messages documented above. For example,

```
rosrun doro_description jaco_home.sh
rosrun doro_description jaco_retract.sh
rosrun doro_description jaco_fingers.sh 0 0 0
rosrun doro_description jaco_joints.sh -1.57 0.04 -1.1 -0.84 1.3 3.0
rosrun doro_description jaco_xyz.sh 1.45 -0.40 0.38
```

will move the arm to the home position using joint-space interpolation, open the fingers, move the arm to the given joint-level position, move the arm to the given (x, y, z) pose keeping current orientation using Kinova inverse-kinematics.

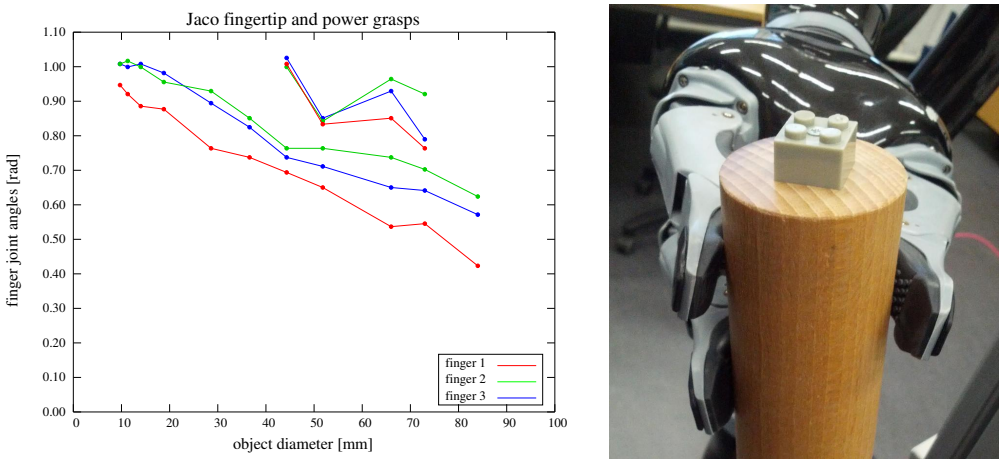


Figure 30: Jaco finger positions as a function of object size. The plot shows the joint-angle in radians for the fingers of the Jaco hand when grasping objects of known diameter using either the *power grasp* (solid lines) between the proximal links of the fingers, or the *fingertip grasp* (dash-dotted lines) between the distal links. Note that power grasps can only be performed for objects of medium diameter, with a nonlinear behaviour due to the underactuated distal links wrapping around the objects. The data is from human tele-operation using symmetric three-finger grasp positions with finger 1 acting as the thumb.

3.5.5 Jaco gripper

The Jaco gripper has three identical fingers which are actuated by one motor each. Each finger has two joints and two degrees of freedom, where the proximal joint is moved directly by the motor and the joint position is measured by the software. To allow stable grasping of medium sized objects, a second underactuated distal finger joint is provided on each finger with a spring-mechanism inside the finger. When the proximal link of the finger touches the grasped object, the distal link can continue to close, thereby *wrapping* the object. Unfortunately, the mechanism is not documented by Kinova at all. Also, the underactuated spring-driven joint is not modelled in the current ROS URDF model of the Jaco arm, which uses rigid fingers without the distal joint. This also implies that wrapping grasps can not be simulated in the Gazebo simulator.

The fingers are made from plastic, with a concave part covered by black rubber material on each of the proximal and distal links. This provides two suitable stable grasp positions for either *power grasps* between the proximal parts of the fingers with optional touching the palm and wrapping of the object via the distal joints, and also *fingertip grasps* between the concave parts of the distal links. When trying to pick up very small objects, it is also possible to use the outer ends of the fingertips for grasping. The numbering scheme is as follows. Finger #1 corresponds to the thumb, while Finger #2 is the index finger, and Finger #3 the remaining (ring) finger.

The relation between finger joint position and grasp is not specified by Kinova. When using the Jaco arm in tele-operation mode, the human user selects the grasp and closes the fingers until the object is grasped. To avoid overly high grasp-forces, a mapping from estimated

object-size to finger position is required. We performed a set of experiments under human tele-operation control, with both the power-grasps (proximal links) and fingertip grasps (distal links) on a set of objects of known size. The first results are shown in Fig. 30, which provides a first approximation to the grasp-planning for given target objects.

3.5.6 Inverse Kinematics

So far, Kinova does not provide a standalone software function to calculate forward- or inverse-kinematics of the Jaco arm. When using the Kinova joystick to control cartesian motions, an iterative IK-solver running directly on the arm controller is used. According to discussions with Kinova, the algorithm is considered valuable IP by the company and will not be released or implemented as a library function soon [41].

As described above, it is possible to request cartesian motions via the Kinova API, but any such function call will directly start the corresponding arm motion. The newest release of the Kinova API also includes a function to check the requested goal position for singularities, but in general it is impossible to predict whether the requested motion will execute properly or will be too close to a kinematics singularity.

On the other hand, the ROS manipulation stack requires both and forward- and inverse-kinematics (FK and IK) solver as prerequisite for the collision-aware arm-motion and grasp planning. The ROS interface expects the following four services:

- *get_fk* (kinematics_msgs/GetPositionFK)
- *get_ik* (kinematics_msgs/GetPositionIK)
- *get_fk_solver_info* (kinematics_msgs/KinematicSolverInfo)
- *get_ik_solver_info* (kinematics_msgs/KinematicSolverInfo)

where the *info* services provide the client with information about the solvers, namely the supported base and target coordinate frames. Here, the forward kinematics can be calculated easily from the robot URDF model and the given joint-angles. As described in the above section, the Jaco gripper has two preferred grasp positions, corresponding to the inner ("power") and outer ("fingertip") rubber covers of the fingers. The final IK solver for the Domestic Robot will be designed to solve for both positions.

The well-known analytical solvers for typical 6-DOF robot arms (e.g. PUMA, Mitsubishi PA10-6C) cannot be used on the Jaco, because the wrist design is not based on the standard approach with three intersecting orthogonal wrist-axes. The current release of the OpenRave motion planning software [27] includes the *FastIK* module, which is claimed to generate inverse-kinematics solvers for any given robot arm. The tool operates in several steps. It first parses an XML-description of the robot kinematics structure including joint-limits, and then derives symbolic equations for the given kinematics. In the third step, those equations are simplified based on a set of heuristics. Next, the resulting equations are converted and written to a C/C++ source file that implements those equations. The resulting file is then compiled and can be used either standalone or as a library function. While OpenRave 0.8.2 succeeded to generate a FastIK source-code for the Jaco arm, the resulting function seems to be invalid and does not find solutions.

Without an analytical solver and without a working FastIK module, the current backup is to rely on the common slower iterative inverse kinematics solvers. Note that the forward kinematics is directly available in ROS based on the tf-library and the existing URDF model of the Jaco arm.

3.5.7 Traps and Pitfalls

Mixing ROS and Kinova joystick operation This is not possible. Whenever the Kinova Joystick is used while the Jaco ROS node is running, the ROS node is disabled. You need to restart the *jaco_node* in order to regain control via ROS.

Bugs in jaco_node Unfortunately, the current Jaco ROS node is not very robust, and first-time initialisation may fail. Also, during initialisation, the node toggles between the Jaco arm rest position and the home position. Manipulation is only possible if the arm initialises into the home position. If necessary, kill and restart the node until the arm initialises into the correct position.

Finger position The finger position is not correctly reported after Jaco node startup. A work-around is to send a suitable finger pose to the */cmd_abs_finger* topic.

jaco_node initialisation and re-start When starting the ROS node, the Jaco arm moves to either its home or the rest position, using joint-space motion with default velocity. Depending on the current position of the arm, this can result in collision of the arm and hand with other parts of the robot. It is recommended to use the Kinova Joystick to carefully move the arm to its rest position before (re-) starting the Jaco ROS node.

3.5.8 Collision map processing

When operating in real-world environments, the robot must be aware of potential collisions between itself and objects in the environment. In addition, when carrying or manipulating objects, the object(s) must be included in the collision checks. Within the ROS manipulation stack, the *TabletopCollisionMapProcessing* service performs the following tasks:

- managing a collision environment using an Oct-tree representation.
- adding the objects (point-cloud clusters or 3D-meshes) identified by the *table-top_object_detector* to the collision environment. For un-identified objects the bounding-box of the point-cluster is added to the collision environment.
- combining multiple sensor data (e.g. Kinect, stereo-camera, tilting laser) into the common collision environment.
- performing self-filtering to avoid adding moving parts of the robot to the collision environment.

The *TabletopCollisionMapProcessing* service returns a list of *GraspableObjects*, which can then be sent to the object pickup action. Check the ROS wiki for a complete example about how to use the perception part of the object manipulation pipeline: *www.ros.org/wiki/-pr2_tabletop_manipulation_apps/Tutorials/Writing a Simple Pick and Place Application*.

3.6 MoveIt! framework

The MoveIt! framework is set to replace the original manipulation stack and has been the main motion-planning tool for the Domestic Robot since 2013. The software is under very active development, please check the website and Wiki at *moveit.ros.org/wiki* for documentation and recent updates. The block-diagram in Fig. 31 presents an overview of the planning architecture and the interface to ROS. Currently, the ROS Hydro version of the MoveIt! toolchain is used on the Domestic Robot.

One of the key ideas of MoveIt! is to enrich the existing kinematics description (URDF files) with semantic information using a new file format called *Semantic Robot Description Format* or SRDF. The SRDF describes the links of the robot used for robot motion and grasping objects, labels groups of joints belonging to the robot arm vs. the hand, and includes information about self-collisions between parts of the robot.

After the MoveIt! tools have been installed, the *MoveIt Setup Assistant* is run once to create the configuration files for a given robot. It provides a simple step-by-step user-interface, where the user first selects the URDF for the target robot, then selects the base and target coordinate systems, selects grasp and approach directions for the gripper, and configures the robot self-collision checks:

1. export LANG=en_US
2. *roscd doro_description/urdf*
3. *rosrun xacro xacro.py DomesticRobot.urdf.xacro > doro.urdf*
4. *roslaunch moveit_setup_assistant setup_assistant.launch*
5. choose mode: create new MoveIt Configuration Package
6. select the doro.urdf created in step 2
7. self-collisions: select high sampling density
8. virtual-joints: select *base_link* as the child link and *planar* as the joint-type, corresponding to the SCITOS base moving on the floor of the Peccioli/Lansgarden labs. Choose suitable names, e.g. *virtual_joint* and *virtual_frame*.
9. planning-group, and the following steps: select the six Jaco arm links as one planning group called *arm*, and the three Jaco fingers as another planning group called *gripper*.
10. robot-pose: enter the Jaco home position.
11. end-effectors: create a group called *gripper* that is a child of the *jaco_hand_link* parent and the *arm* parent group.
12. passive-joints: leave blank
13. select the output-directory and create the configuration files.
14. add the output-directory to your ROS_PACKAGE_PATH.

When successful, the assistant creates a bunch of configuration files in the given output-directory. Depending on your default locale, some files may be damaged due to invalid number formatting. If necessary, repeat the process with your LOCALE/LANG environment variables set to English (EN_US). Otherwise, use a text editor to replace invalid floating-point values with the syntax acceptable for C/C++/YAML files.

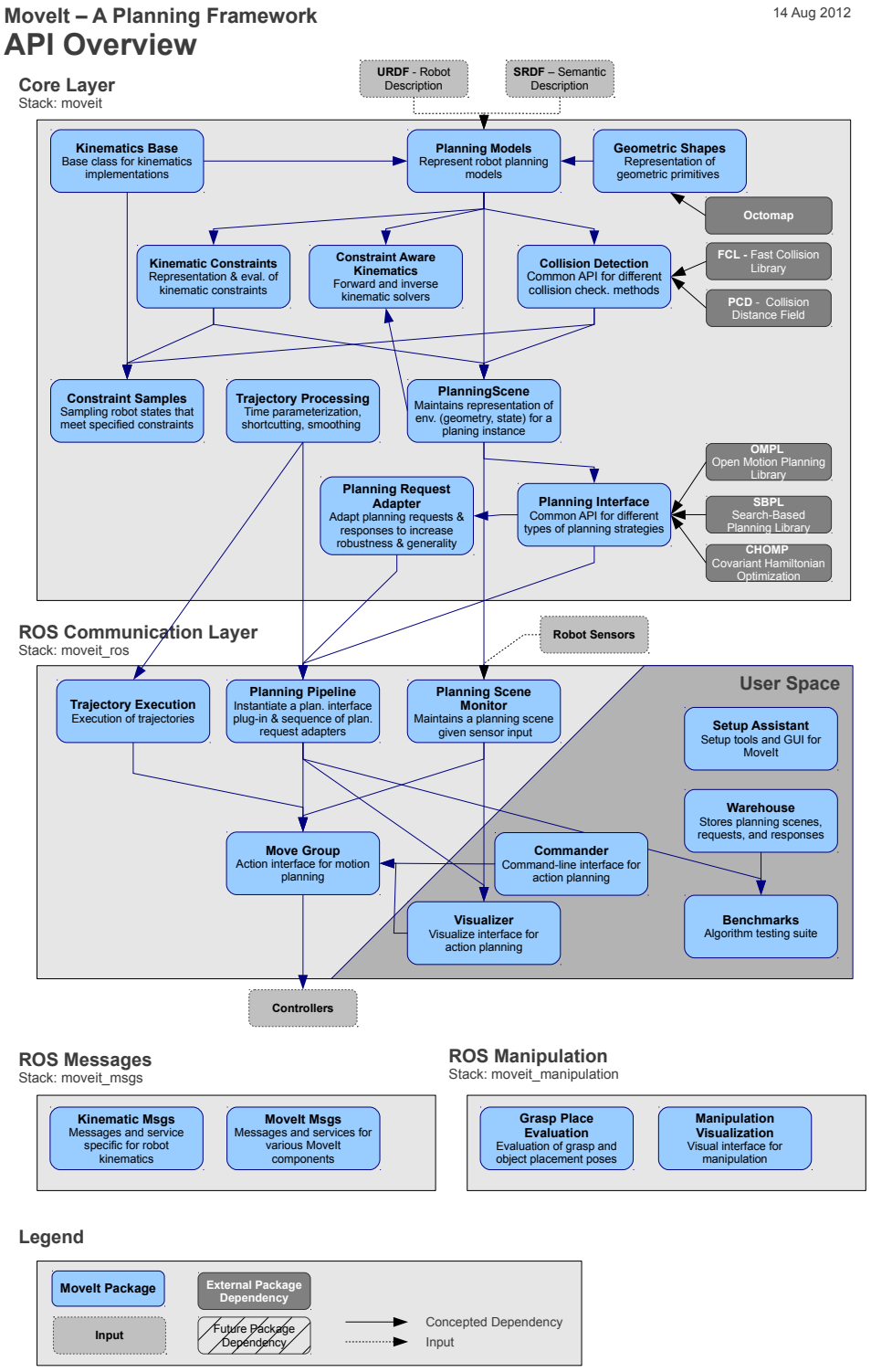


Figure 31: Block diagram of the main components in the MoveIt! framework. Copied from *http://moveit.ros.org/doxygen/pdfs/moveit_api_diagram.pdf*.

For each planning group, MoveIt! expects controllers that offer the *FollowJointTrajectoryAction* actionlib interface. Therefore, a configuration file *controllers.yaml* needs to be created which defines the corresponding controllers:

```
controller_manager_ns: jaco_controller_manager
controller_list:
  - name: jaco_arm_controller
    ns: follow_joint_trajectory
default: true
  joints:
    - jaco_wrist_roll_joint
    - jaco_elbow_roll_joint
    - jaco_elbow_pitch_joint
    - jaco_shoulder_pitch_joint
    - jaco_shoulder_yaw_joint
    - jaco_hand_roll_joint
```

The required launch file has been auto-generated by the assistant, but in the current software version ends up empty. Edit the file *jaco_moveit_controller_manager.launch*, where the last line must be adapted so that the parameter references the configuration file created in the last step above:

```
<launch>
  <arg name="moveit_controller_manager"
       default="jaco_moveit_controller_manager/MoveItControllerManager" />
  <param name="moveit_controller_manager"
         value="$(arg moveit_controller_manager)"/>

  <arg name="controller_manager_name" default="jaco_controller_manager" />
  <param name="controller_manager_name"
         value="$(arg controller_manager_name)" />

  <arg name="use_controller_manager" default="true" />
  <param name="use_controller_manager"
         value="$(arg use_controller_manager)" />

  <rosparam file="$(find moveit)/config/controllers.yaml"/>
</launch>
```

In the current software version, the generated *moveit_controller_manager.launch* file references a *MoveItControllerManager* which is not included in the default ROS installation.

The following example program demonstrates a simple ROS node that subscribes to the */move_group/result* topic. Every time that the MoveIt! planner has generated a motion, the *chatterCallback* function is called with a *MoveGroupActionResult* object as the parameter. The code than fills a *FollowJointTrajectoryActionGoal* object and publishes this on the Jaco *follow_joint_trajectory/goal* topic, which starts the corresponding Jaco joint-level trajectory:

```
#include <sstream>
#include <ros/ros.h>
#include <std_msgs/String.h>
#include <geometry_msgs/PoseStamped.h>
#include <control_msgs/FollowJointTrajectoryActionGoal.h>
#include <moveit_msgs/MoveGroupActionResult.h>
#include <jaco_api/jaco_api.hpp>

class trajectoryForwarding {
private:
  ros::NodeHandle nh_; ros::Subscriber sub_ ; ros::Publisher pub_;
public:

trajectoryForwarding(ros::NodeHandle &nh)
{
  nh_  = nh;
  pub_ = nh_.advertise<control_msgs::FollowJointTrajectoryActionGoal>(
    "/jaco_kinematic_chain_controller/follow_joint_trajectory/goal",10);
  sub_ = nh_.subscribe("/move_group/result", 10,
    &trajectoryForwarding::chatterCallback, this);
}

void
chatterCallback(const moveit_msgs::MoveGroupActionResult::ConstPtr& msg)
{
  // ------FollowJointTrajectoryActionGoal------
  control_msgs::FollowJointTrajectoryActionGoal fJTAG_msg;
  fJTAG_msg.header.stamp    = ros::Time::now();
  fJTAG_msg.header.frame_id = "/jaco_base_link";
  fJTAG_msg.goal_id.stamp   = ros::Time::now();
  fJTAG_msg.goal_id.id      = "goalID";

  // read planned_trajectory from MoveGroupActionResult and fill
  // control_msgs::FollowJointTrajectoryActionGoal with it.
  fJTAG_msg.goal.trajectory
    = msg->result.planned_trajectory.joint_trajectory;

  // at current state Kinova jaco ignores JointTolerance[] path_tolerance
  // and JointTolerance[] goal_tolerance
  // duration goal_time_tolerance
  fJTAG_msg.goal.goal_time_tolerance = ros::Duration(10.0);
  pub_.publish(fJTAG_msg);
}
};

int main(int argc, char** argv)
{
  ros::init(argc, argv, "jaco_trajectory_forwarder");
  ros::NodeHandle n;
  trajectoryForwarding trajFor(n);
  ros::spin();
}
```

```
<launch>
  <arg name="planning_plugin" value="ompl_interface_ros/OMPLPlanner" />
  <arg name="planning_adapters" value="
    default_planner_request_adapters/AddTimeParameterization
     default_planner_request_adapters/FixWorkspaceBounds
      default_planner_request_adapters/FixStartStateBounds
      default_planner_request_adapters/FixStartStateCollision
       default_planner_request_adapters/FixStartStatePathConstraints" />
  <param name="planning_plugin" value="$(arg planning_plugin)" />
  <param name="request_adapters" value="$(arg planning_adapters)" />
  <param name="start_state_max_bounds_error" value="0.1" />

  <rosparam command="load"
            file="$(find moveitDomestic)/config/kinematics.yaml"/>
  <rosparam command="load"
            file="$(find moveitDomestic)/config/ompl_planning.yaml"/>
</launch>
```

3.7 Manipulation Action Server

The Manipulation Action Server server provides a ROS node, that is specific to the Domestic Robot and provides the higher level actions that need to be carried out by the platform. It requires both MoveIt and Visionhub to be started on the robot system. Depending on their complexity and duration the functions are provided either as ROS services or as Actionlib servers.

Like the name says, the main purpose of this node is to provide services to grasp or handover objects. The services are exposed using the ROS Actionlib library and therefore support cancellation of actions and feedback messages. When a manipulation action is called, this nodes retrieves a list of possible grasp poses from the Visionhub node and uses ROS Moveit in order to calculate the collision aware trajectory. Object handover to the user is implemented featuring force controlled handover procedure. The robot uses its force sensors to detect when the user applies a stable grasp and then releases the object. A simple state machine ensures that only actions can be carried out that currently make sense (e.g. handover is only possible when an object is grasped).

A "place" service is also implemented, allowing the caller to specify how the currently grasped object is treated:

- drop at a defined place (the most simple operation), can also be defined relatively to another object (e.g. a bin)
- place operation, where the footprint of the currently grasped object is put on top of a surface specified in the call (on top of another object, on the tray of the robot)
- optionally the constraint can be defined to keep the object upright

3.8 Manipulation Tasks in Robot-Era Scenarios

3.8.1 Domestic to Condominium Object Exchange

In the overall Robot-Era system, the Domestic Robot stays in the users' apartment, while the Condominium Robot takes care of transportation tasks within the different floors of the building. Therefore, object exchange between the Domestic and Condominium Robots is a necessary part of several Robot-Era scenarios, notably the *shopping*, *laundry*, and *garbage* scenarios.

The object exchange between the Outdoor and the Condominium Robots is performed by moving the objects using the motorised roller trays on both robots. This only works for objects with a clean planar bottom and requires a precise alignment of both robots, see D5.3 and D6.3 for details [60, 61]. Therefore, a cardboard box of size $0.2 \times 0.2 \times 0.2\text{m}^3$ has been selected as the reference object and will be used during the second experimental loop.

The same cardboard box is also used for object exchange between the Domestic and the Condominium Robot. AprilTags markers have been put onto the box, and the pose recognition accuracy is usually better than 1 mm position error (see section 3.3.9). The box has been equipped with a cylindrical handle on the inner sidewall of the box. The handle nicely fits the rubberised parts of the Jaco thumb, so that the fully loaded box can be lifted without problems. This allows the Domestic Robot to grasp the box precisely with the Jaco arm, where localisation errors of about 10 cm and several degrees between the Domestic and the Condominium Robots can easily be tolerated. See figure 33 for photos of the object-exchange process.

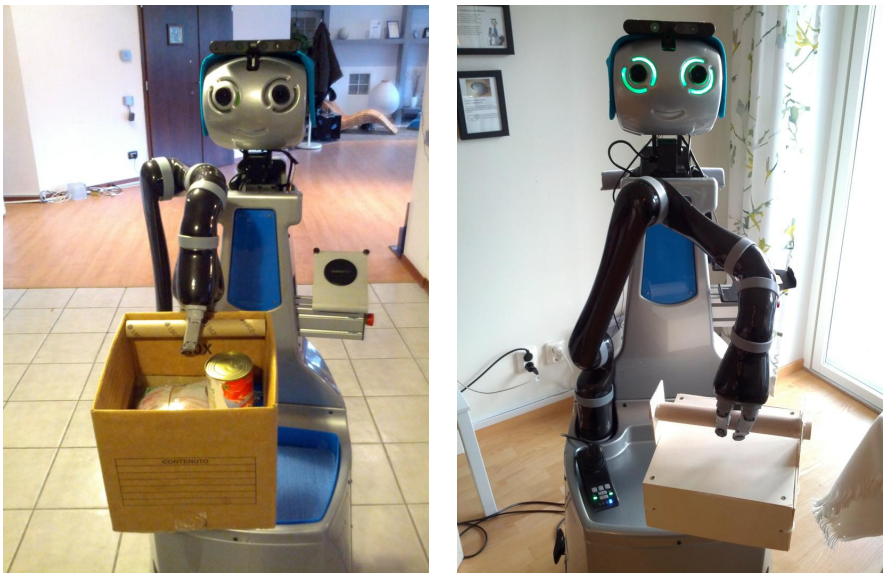


Figure 32: The picture shows the reference box that will be used for the object-exchange between the three robots. The cylindrical handle placed on the side wall of the box allows a stable grasp with the Jaco hand (left). Prototype of the lunch-box object to be used in the table cleaning scenario with a centered cylindrical handle.

Figure 33: Autonomous object exchange between the Domestic and the Condominium Robot. The Condominium Robot has a passive role. One or both robots move until they are close together, where position and alignment errors of several centimetres can be compensated by arm movements. The object is detected using either AprilTag markers or SIFT-features, and collision-aware grasp and place motions are planned by Moveit. The photos also show the prototype cardboard box that fits all three Robot-Era robots.

3.8.2 Basket objects

To improve the realism of the garbage scenario, basket-like objects of different sizes have also been designed. AprilTag markers are used to provide the robot with accurate object identification and pose estimation. To avoid payload torque limitations on the Jaco fingers, a central cylindrical handle is used on the basket objects. Different handle diameters have been tested, and a rather large handle diameter (approx. 5 cm) works best for power-grasps (closing the underactuated finger-joints) on the Jaco gripper. See figure 34 for a photo of Domestic Robot taking the prototype basket. In this setup, the basket is detected by the new camera added on the robot side, so that the head can point straight ahead.

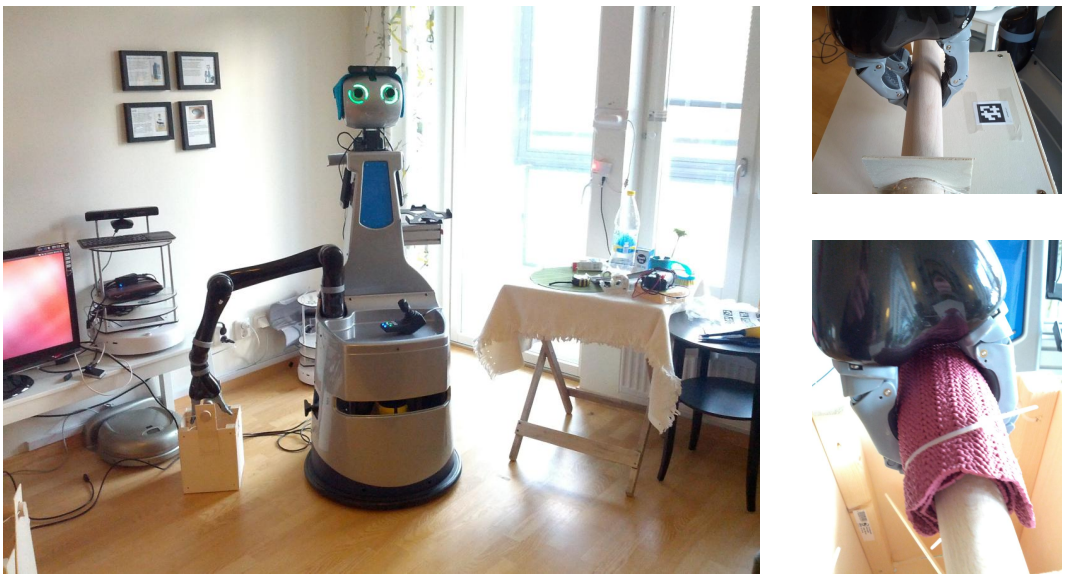


Figure 34: The photo shows grasping the prototype basket that will be used for the garbage scenario (left). AprilTag markers are put onto the object to improve object identification and pose estimation. In this case, the new side camera is used for detection, and the head is idle and can point straight ahead. Closeup of a soft cylindrical handle that is a good fit for power-grasps with the Jaco hand (right).

3.8.3 Cleaning Tasks

Given the clear user demand, cleaning tasks will be a key capability of future service robots. This in turn requires robot motion that keeps contact with the surfaces to clean and maintains adequate force and torque constraints throughout the motion. Unfortunately, neither cartesian- nor joint-space force-control is available on the Jaco arm yet. While measuring the motor-currents provides a rough estimate of applied force, the values are plagued by friction and hysteresis, and the sensor sample-rate is too low for maintaining controlled contact. Also, the Jaco arm is not waterproof, and the use of water and cleaning fluids is considered to dangerous for autonomous operation close to end users in the real experimental tests.

Therefore, only simplified cleaning tasks will be demonstrated and evaluated during the second experimental loop. In particular, dry brushing and swiping tasks have been defined

and implemented. For details and the service interface to the planner, see section 4.3.2. Using a set of carefully selected brushes and sponges, robust grasps can be executed using the Jaco gripper, while the elasticity of the tool provides the compliance to protect the Jaco arm and the surfaces during position-controlled motions. Despite the simplification, the task is still state-of-the-art and demonstrates that the robot can recognise, grasp, and use tools to perform meaningful motions and tasks. Where necessary, a custom tool interface similar to [4] will be used for robust tool grasping.

While the Moveit! framework provides functions for Cartesian motions, the combination of sampling-based planners with the slow iterative inverse-kinematics solver results in long planning times and low success rates for typical cleaning motions. The proposed solution combines collision-aware reach and retract motions planned by MoveIt! with fast local motions generated by the Reflexxes type-II library [5, 6]. The planned motions are then sent to Jaco for trajectory-execution, while a watchdog process monitors the position errors and motor-torques of the arm. The underlying assumption is that the surfaces to clean are smooth and free of large obstacles. The tool compliance ensures that small motion deviations can be tolerated.

Figure 35: Selected tools will be used to demonstrate cleaning tasks during the second experimental loop. In particular, the elasticity of hand-brushes and sponges provide the compliance to safely execute swiping and brushing tasks using position-control on the Jaco arm. Motion-planning combines collision-aware Moveit planning with fast trajectory execution using the Reflexxes library.

3.8.4 Object Handover Tasks

The object transportation tray allows the Domestic Robot to carry several small objects or one of the large reference object (object-exchange box, lunchbox, basket), where the user can pick them up as needed. However, direct robot to human object handover is desirable in some scenarios, e.g. the drug reminding or drug delivery, when the user is sitting down or lying on a bed.

Several handover strategies have been tested and implemented. As full tracking of the users' hand is not yet available, the handover sequence uses pre-defined fixes handover poses.

Assuming that the Jaco has grasped an object, the robot-to-human handover consists of the following steps:

- the planner drives the robot to a position close to the human,
- the planner selects a suitable pre-defined handover pose and executes a (collision-aware) Jaco arm motion,
- the robot speaks to ask the user to take the object,
- the robot samples the Jaco motor-torques to detect when the user has touched and grasped the object,
- the robot opens the Jaco fingers and lets the user take the object,
- the arm is retracted to a safe position.

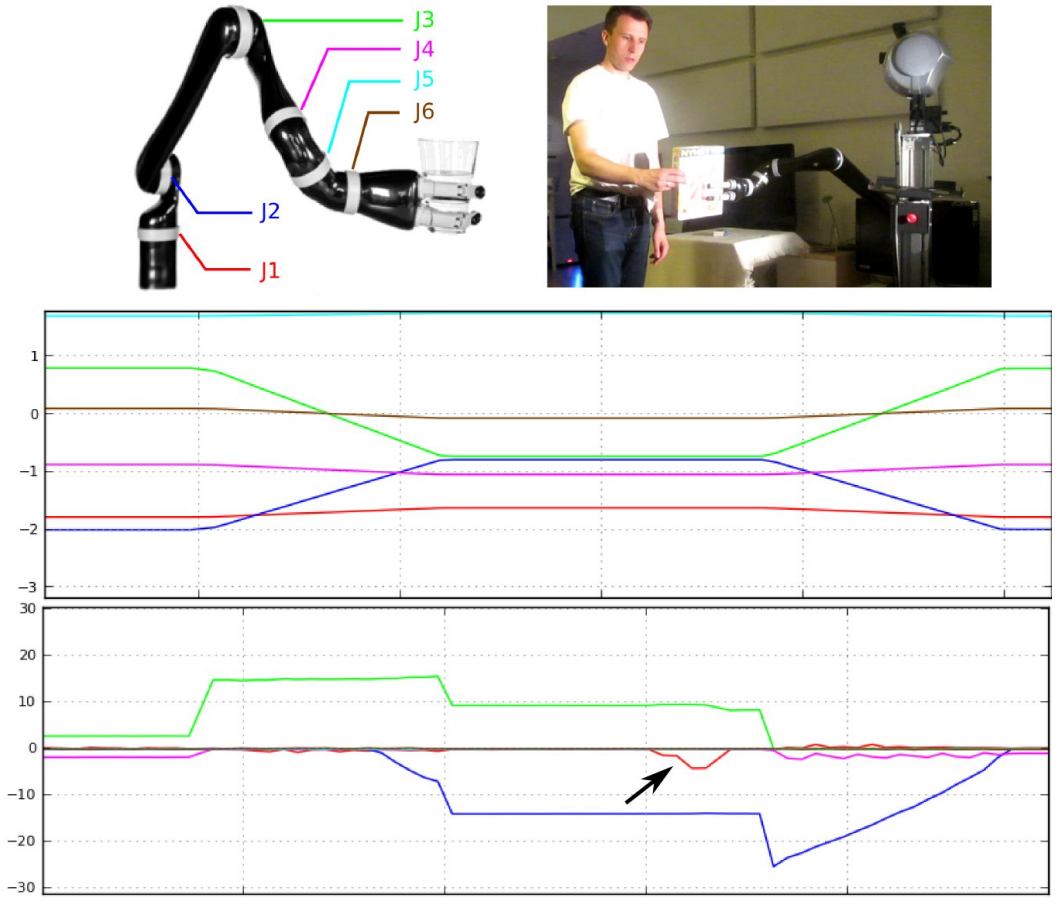


Figure 36: Robot-to-human object handover using the Jaco arm. Using the updated JacoROS driver, motor torque information is available from the arm and can be used to trigger object handover from the robot to the human. Experiments were conducted to match force thresholds to object size and weight for best user acceptance.

For the second experimental loop, this quasi-static handover will be tested, where the force thresholds are adapted to the size and weight of the object to ensure that the handover feels natural for the users [8,9]. Typical joint torques recorded by the Jaco are shown in figure 36.

Alternatively, the arm forces can also be sampled continuously, allowing the user to take the object while the robot arm is still moving. This has been validated using the KuKA LWR available in Hamburg, which has suitable torque-sensors and fast reactive motions (figure 37).

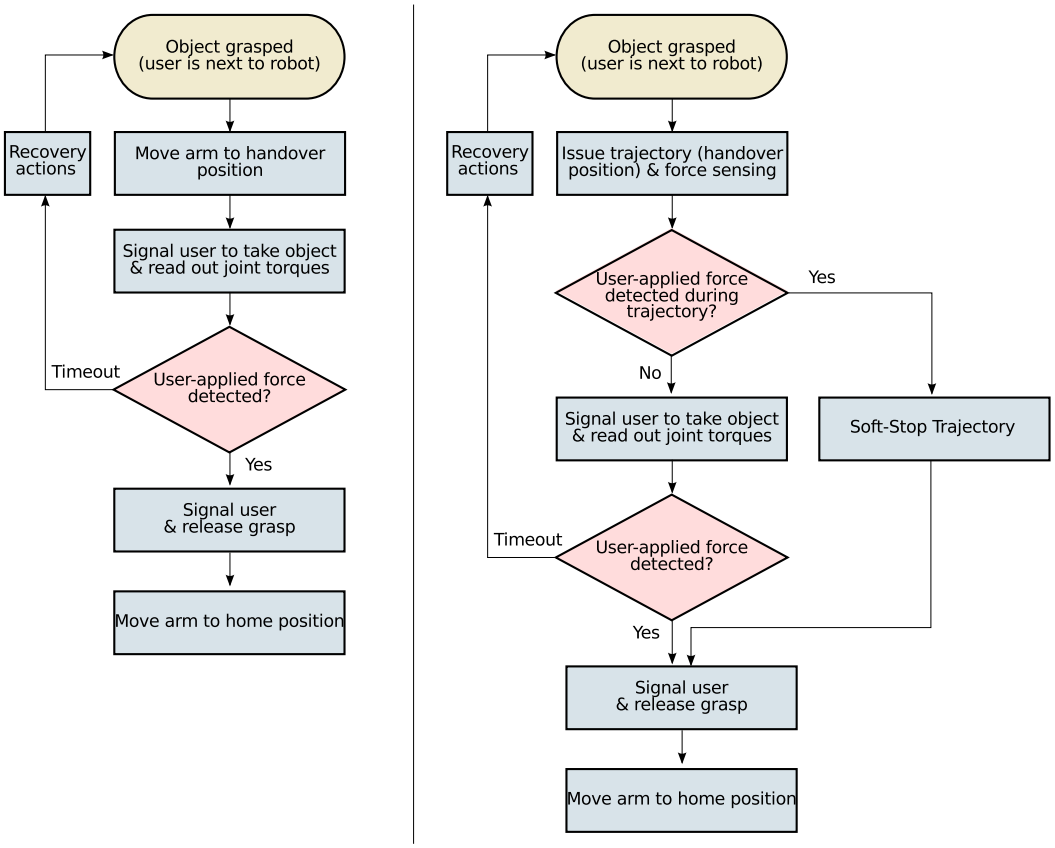


Figure 37: Flow diagram for robot-to-human object handover using the Jaco arm.

3.9 Robot Watchdog

While the hardware of the Domestic Robot performed as expected and no component failure occurred during the first experimental loop, several minor problems and a few design issues were identified in the overall Robot-Era architecture and the robot software. Some failure mechanisms were easy to track down, with flooded Wifi connections to the robots one of the obvious causes. When running all Robot-Era components, significant packet loss and latencies of up to 30 secs have been observed. This triggers message retransmissions, which in turn worsens the issue, and the long delay often invalidates timestamped ROS messages, making the received data useless for further processing. Better wireless network infrastructure will be installed in Angen-Lab and DomoCasa for the second experimental loop.

The simple problem of an overloaded main robot computer was addressed by installing additional processors connected by an on-board network. Two Intel NUC modules were selected due to good processor speed and their compact size, increasing the available on-board performance by about a factor of three. In particular, the very high amount of data streamed by the XtionPro camera is now handled by one of the Intel NUC modules, with a corresponding significant reduction in processor load and USB bandwidth on the main Scitos-G5 computer. While the main computer is still heavily loaded due to running MIRA and Cognidrive (platform sensors, localisation and navigation) as well as running the *roscore* ROS master process and the *JacoROS* arm driver, the load is significantly reduced and is expected to stay below 100%, so that some spare CPU cycles are available when needed.

Regarding the robot software, the situation is more difficult. Given the complexity of the distributed ROS software running on each of the three robots, and the intentional lack of synchronisation in the *roslaunch* tool, it was not always easy for the experimenters to recover from failures. For example, a single crashed ROS node will stop sending messages or service replies expected by other nodes, and small failures can therefore quickly spread to the whole network of interconnected ROS nodes. Even identifying the source of the problem can be difficult, but once the failed ROS node has been pinpointed, restarting single nodes may be impossible due to interdependencies and parameters set by complex chains of nested roslaunch-files.

In this situation, a complete shutdown and reboot of the whole robot system was often the quickest solution. Of course, this is unacceptable from a reliability and service-uptime point of view, and must be avoided for the second experimental loop. Therefore, the roslaunch files used for robot bringup have been re-factored for better modularity, allowing the operator to restart parts of the ROS and MIRA/Cognidrive system at runtime.

Work has been started to implement a *watchdog* program that monitors key components of the overall ROS system to help with the detection and analysis of problems and failures. As implemented now, the watchdog node simply subscribes to a set of key messages, keeping track of timestamps and cross-checking selected message data against expectations. Functions and safety checks include:

- periodically querying the *roscore* master process for the list of running nodes and services, to check both the health of the *roscore* process and the status of all required ROS nodes.

- periodically executing a system command (`ps -auxwww`) via *ssh* on all three on-board computers to query the Linux kernel for a list of running processes, again checking status of all required ROS nodes and comparing processor load and memory usage against a set of (currently hand-coded) thresholds.
- periodically running the `iftop` command to monitor network load, packet losses, and latencies.
- subscribing to all sensor messages (front and rear laser-scanners, cameras, XtionPro, gyro, battery, arm) and checking expected data-rate and timestamps on incoming messages.
- checking battery status.
- reading front and rear laser-scan data and checking for collisions.
- reading PTU data and checking position error to detect possible collisions with environment and self-collision with the arm.
- reading Jaco data and checking driver status, joint torque estimations, and joint temperatures.
- so far, only diagnostic messages are generated, but it would be straightforward to also restart nodes automatically or to shutdown the PTU or Jaco driver in case of detected collisions.

A list of additional useful watchdog functions has been compiled and will be implemented during the experimental phase. Also, the processor and network load and the relative importance of ROS nodes changes according to the current service performed by the robot, and this should be respected in the analysis. For example, navigation and localisation are most important while the robot platform is driving, while motion planning and arm control are key components during manipulation. As the watchdog already logs process status and load, this can be correlated with ongoing service requests and will be used to build the expected activity patterns to be checked in future runs.

3.10 Simulation

The *Gazebo multi-robot simulator* is capable of simulation a population of robots, sensors, and objects in a 3D world. Based on the ODE [45] physics engine, it generates both realistic sensor feedback and physically plausible interactions between objects, at least for rigid-bodies. See *www.gazebosim.org* for details and documentation.

A block-diagram of the Gazebo simulation framework is shown in Fig. 40 on the next page. The simulation kernel *gzserver* maintains the whole world model and the simulation time, updating the positions and motions of all objects based on the rigid-body physics calculated by the ODE physics engine. The key parameters for the physics engine are specified as part of the world-model specified when starting the server, but can also be changed during a simulation. Using a plugin-mechanism, additional code can be integrated into the simulation process, with full access to the data-structures of the simulator. This approach is currently used to implement the various sensor models, including distance-sensors, cameras, and tactile sensors. Additional user-provided plugins can be used, but must be recompiled for the specific version of Gazebo used.

The *gzclient* program provides a 3D-viewer onto the simulated world and also allows basic interactions, including pausing and restarting an ongoing simulation, adding new objects into the world, and interactively updating the pose of objects. Alternatively, the ROS *rviz* tool can be used to watch and control the simulated world. Both tools use the Gazebo network interface to communicate with the simulation engine.

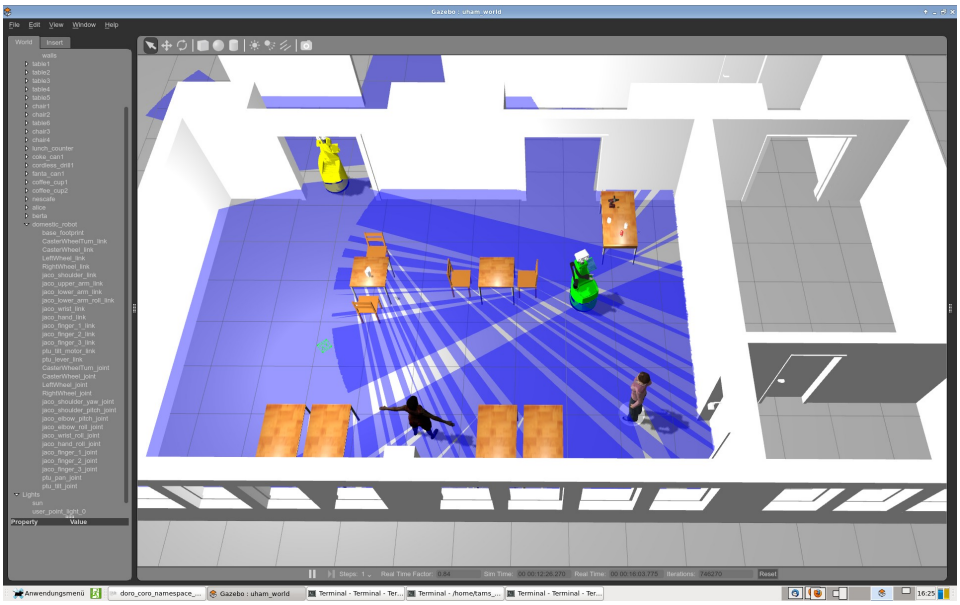


Figure 38: Realistic multi-robot simulation in Gazebo. Using the robot simulation models developed during Y3, realistic physics simulation of the whole Robot-Era system is possible (except for the speech interface). The Domestic, Condominium, and Outdoor Robots are simulated with all their sensors and actuators. The simplified human models can move and allow testing person detection as well as robot obstacle avoidance and evasive behaviours.

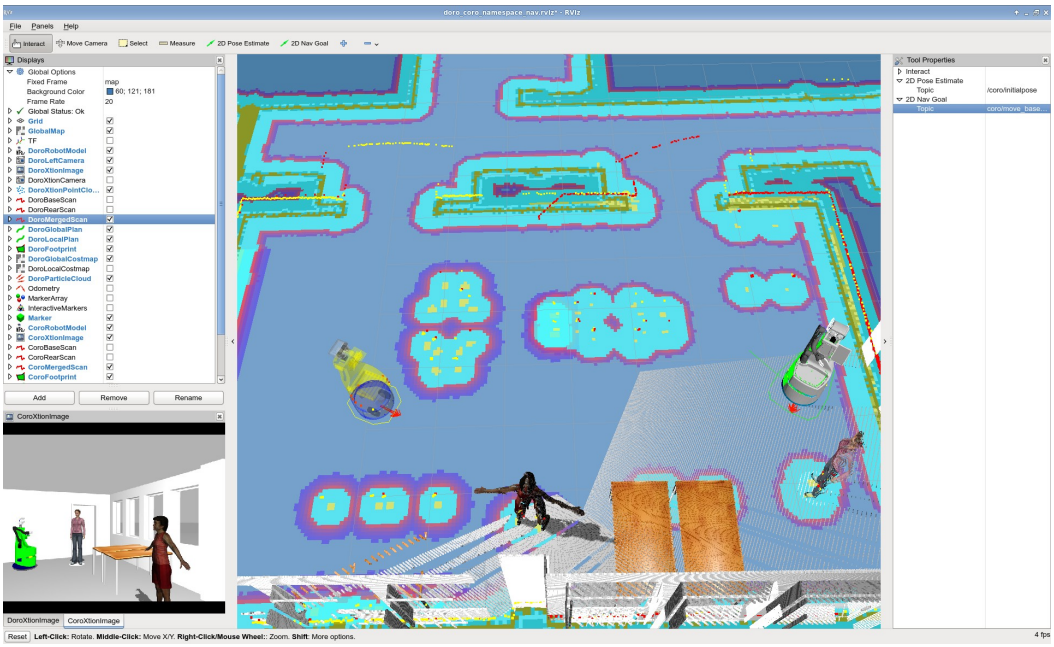


Figure 39: Screenshot of the ROS rviz visualisation tool for the setup shown in figure 38. All sensor data (laser-scanners, cameras, Xtion/Kinect) as well as the robot navigation and manipulation plans can be visualised. The areas on the floor indicate the obstacle cost-maps built by the robot navigation from the robot laser-scanner data.

3.10.1 Domestic Robot in Gazebo

Due to the support of the URDF file format, any robot model from ROS can be loaded into a running Gazebo simulation with a simple *spawn_object* call, either interactively from the command-line or from programs or launch-files. The easiest way is to first start Gazebo with an empty-world or any of the predefined world files, and then to add objects and robots into the world. This can also be done from the user-interface, where *pause*ing the simulation during interactive placement and alignment of new objects is recommended. Multiple robots are supported by Gazebo from the start, but care must be taken to use namespaces in order to keep the joint- and sensor-data from different robots separate.

To experiment with the Domestic Robot in Gazebo, please check-out the latest version of the ROS/Gazebo robot models and simulation interfaces from the Robot-Era SVN repository. Both the *doro_description* and the *doro_gazebo_plugins* packages are required, but you may want to also download, install, and rosbuild the remaining packages in the *domestic_robot* stack. In particular, the *doro_teleop* package provides a telnet-based server for joint-level motions and the joystick interface for interactive control of the robot.

Assuming that Gazebo and ROS are installed correctly, just run rosmake in the Domestic Robot stack, then launch the simulation server *gzserver*, optionally run the Gazebo 3D viewer *gzclient*, optionally load a world model with walls and furniture, and then spawn the robot and any other objects. In addition to the Gazebo 3D viewer, the ROS rviz tool is also a great help to watch and control the simulation.

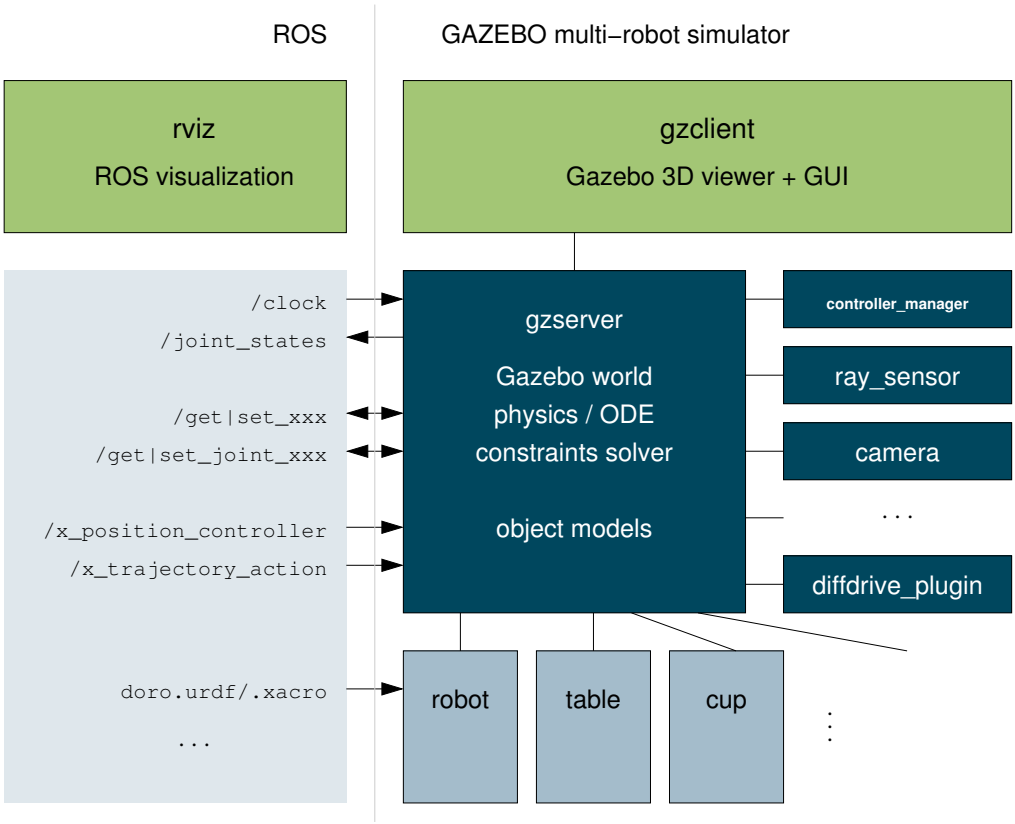


Figure 40: Block diagram of the main components of the Gazebo multi-robot simulator and the integration into the ROS framework. The *gzserver* process maintains the world model and calculates object motion based on the ODE physics engine. The server can load models described in its native SDF file format, but also from ROS URDF robot descriptions. Several plugins are available to model typical robot sensors, including cameras, depth-cameras, laser-scanners, sonar sensors, as well as tactile- and force/torque sensors. Additional plugins can be loaded into the simulator, where the *controller_manager* provides the interface to real-time controllers running within ROS. Either the ROS *rviz* tool or the Gazebo 3D viewer *gzclient* can be used to watch and control the simulation interactively. A network interface provides program calls that allow to query and modify the world state and to control the simulation engine.

3.10.2 Notes and version compatibility

Please note that the Gazebo simulator is under very active development right now, as a special version of Gazebo has been selected by DARPA for the *drcsim* project. Unfortunately, the main SDF (simulation description format) file format has undergone several incompatible changes, improving functionality and fixing severe shortcomings, but breaking simulation models developed for older versions of the simulator.

In the transition from ROS Fuerte to ROS Hydro, the interface from ROS to Gazebo was redesigned completely. In particular, the special version of Gazebo 1.02 updated for use

in ROS was dropped. Instead, ROS interfaces are provided to the standalone installation versions of Gazebo.

At the time of writing, Gazebo 1.9 is the official version for ROS Hydro, and this version has been used for the development and test of the Robot-Era robot simulations models during 2014. Later versions of Gazebo should work when the corresponding plugins are installed, but this has not been tested. All sensors and actuators are modelled on the Domestic, Condominium, and Outdoor Robots, including the laser-scanners, all cameras, the Xtion/Kinect RGB-D sensors, as well as the wheel odometry. The sonar-sensors on the Scitos-G5 platform are modelled as well, but are currently not used on either the real robot nor the simulation models. However, as Gazebo does not support sound, neither the microphones nor the speakers for the Robot-Era speech interface are modelled. The updated version of Gazebo and the use of updated 3D meshes for all three robot also fixes the bug with undetected self-collisions, that was documented in D4.2.

Unfortunately, not all race-conditions in simulator startup have been fixed yet. Instead, a new C++ class called *doro_coro_gazebo_launcher* is provided as part of the *doro_utilities* package. The launcher serialises the spawning of robot simulation models in the Gazebo world and takes care to start navigation and manipulation plugins in the correct order. The launcher also includes utility functions to populate the simulated world with the simulated humans and static simulation objects in either ROS (URDF) or Gazebo (SDF) file format.

The best way to launch the multi-robot simulation is therefore to rely on the provided predefined launch files, e.g.

```
roslaunch doro_utilities Y3_doro_coro_hamburg.launch
roslaunch doro_utilities Y3_doro_moveit.launch
```

3.11 PEIS Integration

This section describes the interface between the PEIS ecology layer and the several ROS software components on the Domestic Robot. As described above (see Fig. 12 on page 22), the interface layer consists of a set of largely independent modules, each of which provides one specific service from the Robot-Era storyboards. During Y3 of the project, the use of PEIS meta-tuples was suggested to simplify the task of the Robot-Era planner. A short overview of the corresponding *exekutor* class hierarchy is given in section 3.11.6.

3.11.1 PEIS-ROS TupleHandler architecture

The basic idea of the PEIS-ROS interface is very simple. It consists of a set of ROS nodes, called *TupleHandlers*, which first register themselves with ROS, subscribing and publishing topics, and announcing their ROS services. Next, every *TupleHandler* registers itself in the PEIS network with a naming pattern that matches the required tuple-names. Whenever one of the matching tuples is created or changed, a callback function is called and the TupleHandler analyses the tuple and performs the corresponding action. For upstream information exchange, the TupleHandler will modify the data field of the relevant tuples, and may also create new tuples.

While a TupleHandler will be called on all tuple-changes matching its tuple-name pattern, most of those changes will be silently ignored. The TupleHandler is only triggered when the *command=ON* change arrives, at which time the corresponding service is started. Most TupleHandlers will wait until this time to access the remaining tuples and read the command *parameters*. It is expected that the value of all required parameters is valid at the time of the *command=ON* change. Of course, it is also possible to write a TupleHandler that updates its internal state (e.g. parameters) on every subscribed tuple-change, but this requires the management of internal state and may be more complex than simply deferring reading parameters until the start of the activity.

3.11.2 Using actionlib and feedback functions

The basic ROS *services* are very useful for fire-and-forget tasks, where a request is submitted to the server and the client can wait (and must wait) until the reply arrives. However, this architecture is not suitable for the high-level control of the Domestic Robot, because several services are long-running tasks, and PEIS and the multi-robot planner cannot be blocked until the tasks finishes. Also, it may be necessary to cancel an ongoing action. This is exactly the functionality provided by the ROS *actionlib* architecture, which provides service-calls that send period feedback message about their progress until the action has completed and the final reply is returned. Also, actionlib services can be cancelled. See *www.ros.org/wiki/actionlib* for details and documentation.

Therefore, most PEIS-ROS TupleHandlers will support an *actionlib interface* to their services, including definition of the required ROS messages for the *goal*, *status*, *feedback*, and *result* of the action. Technically, the corresponding *feedback()* and *result()* callback functions need to be implemented in the TupleHandler class. At runtime, the TupleHandler

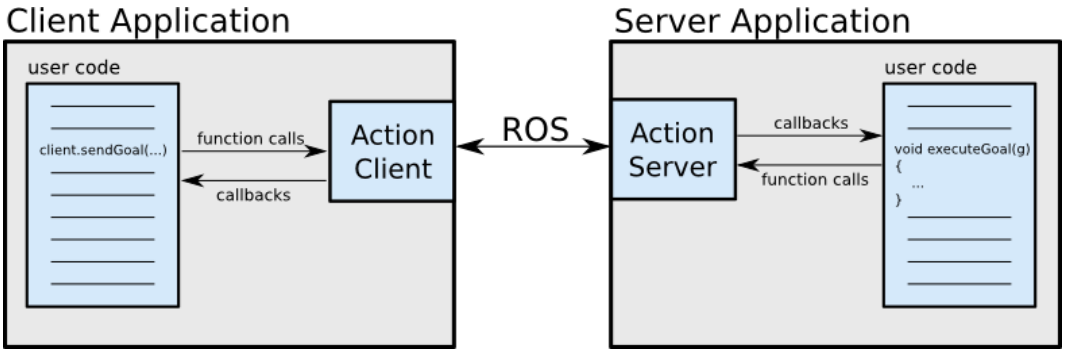


Figure 41: The interface between actionlib clients and servers. The PEIS-ROS TupleHandler classes are written as action clients, where incoming PEIS commands are converted to action goals. Corresponding PEIS tuples are created and/or updated when the action server sends new status and feedback data. The actual robot service is provided on the ROS server application side.

node will subscribe to the *goal* and *cancel* topics, and publish the *feedback*, *status*, and *result* messages. In addition to publishing the progress to ROS, the TupleHandler will also update the values of the corresponding PEIS tuples.

In addition to the geometry, the full URDF model of the robot also includes the weight and the inertia properties of all components. The weight of the main platform was taken from

See Fig. 42 for a state-machine diagram that shows the common states for the PEIS-ROS actionlib implementation. When starting the corresponding ROS node, the service initialises itself connecting to topics and ROS services, and registers itself with PEIS. It then enters the *SLEEP* state, waiting to be started. Once triggered, the service enters the *ACTIVE* state, and will provide periodic feedback about its completion status. Once the service has completed its goal, the state changes to *COMPLETED* (either *SUCCEEDED* or *ABORTED*) and PEIS is notified. Should the service receive a *cancel*-request, the state changes from *ACTIVE* to *PREEMPTING*, and then to *PREEMPTED*. Actual state and progress from the service is sent back to PEIS via the corresponding *STATUS*-tuples.

3.11.3 Synchronisation

The PEIS configuration planner (CPM) is responsible for the scheduling of actions in the PEIS network, including the synchronisation of service requests to the Domestic Robot. As described above, whenever the CPM wants to start a service, it changes the value of the *command* tuple to *command=ON*, which triggers the corresponding *TupleHandler* and starts the requested action.

At the moment, no robust mechanism for pre-empting active services exists. Triggering the *emergency-stop* service will stop all ongoing activity, but the robot may not be able to continue with the interrupted task.

For all services implemented via actionlib, the CPM planner is expected to poll the periodic feedback callbacks from the ongoing service and to wait until the service has sent its result

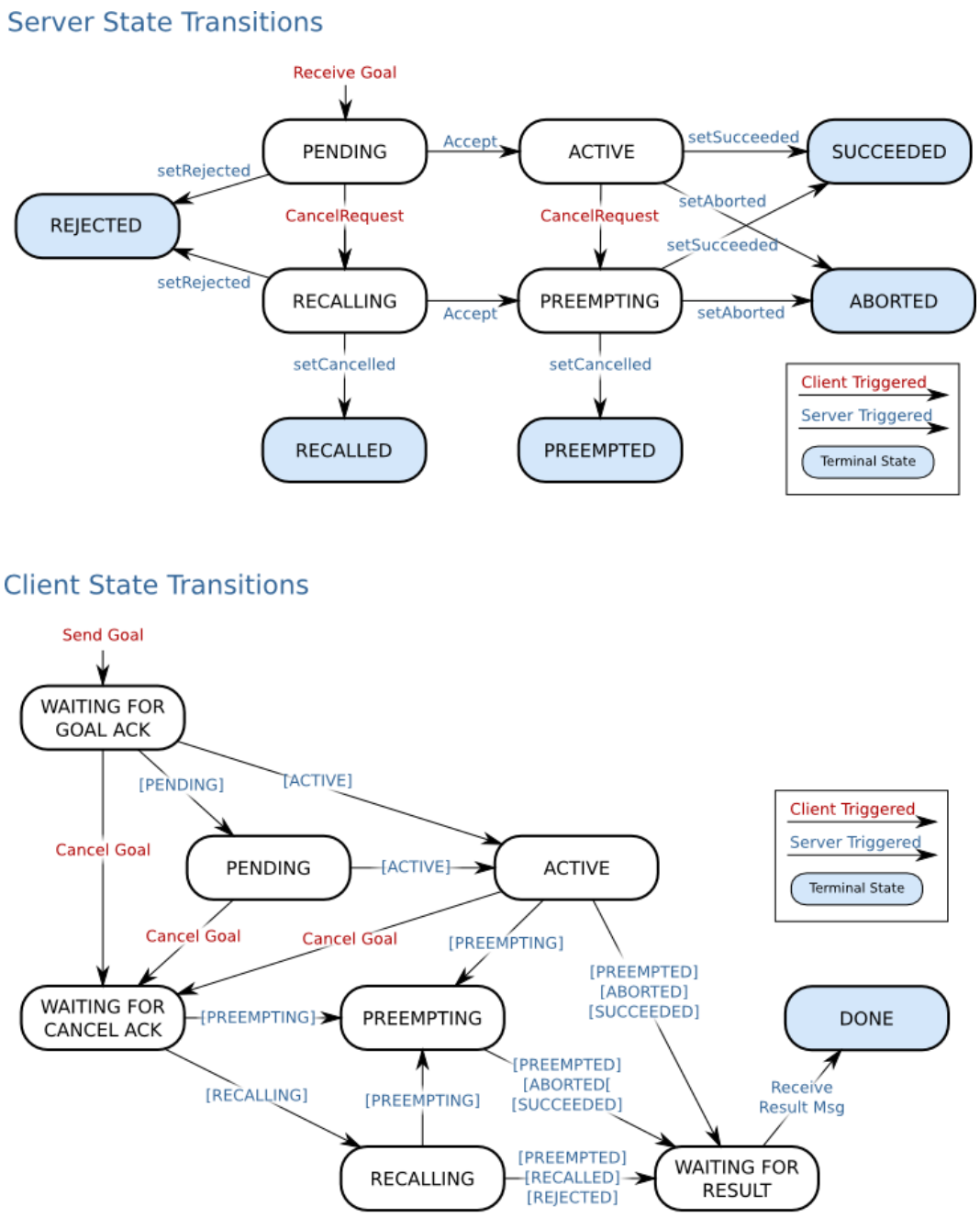


Figure 42: The server and client states for the ROS actionlib stack. Those states are also used in the PEIS-ROS bridge. See the text for an explanation. (Used with permission from www.ros.org/wiki).

message. Sending the *cancel* request should be acknowledged by the ongoing service, but the CPM planner still has to wait for the result. A new service can only be requested after the previous action has completed. Even short running tasks will be converted to actionlib, so that the high-level interface to the services looks the same from PEIS.

3.11.4 Structured data

In the PEIS framework, the payload of a tuple is supposed to be just an array of bytes. Optionally, the MIME-type and character encoding of the payload can be specified. Helper functions are provided to create tuples from null-terminated C-style strings. Unlike ROS and MIRA, the system has no support for multimedia data like JPEG images or complex structured messages, e.g. *Quaternion*, *JointTrajectory*, *PointCloud*. While such data can be passed into and transported via PEIS as an opaque array of bytes of the required length, the system itself has no notion of the data-type, and *application/octet-stream* should be used as the MIME-type. All respective tuple users are therefore responsible to encode and decode the data themselves.

Within PEIS-ROS, whenever the transport of complex messages is necessary, an additional tuple will be created with *key=*.*.*.ROS_MESSAGE_TYPE* and the ROS message class name as the *value*, e.g. *value=geometry_msgs/PoseStamped*. A client can first query PEIS for the existence of the *ros_message_type* tuple, which marks a ROS-based data-type, and then read the corresponding type from the tuple payload. Next, the client reads the *parameter* tuple, and retrieves the binary data. Finally, the client uses the ROS API and helper functions to decode the binary data.

Alternatively, some services on the Domestic Robot will simply use URLs or URIs (*unique resource identifiers*) as their return values. Strings are handled easily in PEIS and the *tuple-view* tool, and the client can then use a webbrowser or other tool to retrieve the information from the Domestic Robot. See the service descriptions in chapter 4 below for the documentation of any URL/URI messages used by the Domestic Robot.

3.11.5 Writing a new TupleHandler

The API for writing the TupleHandlers is still not stable, but the main ideas are highlighted in the following short code sequences. The basic idea is to have ROS nodes which also register themselves with the PEIS ecology. See Fig. 43 for the C/C++ header file that describes the base class of the TupleHandler/Service hierarchy. The base class contains a reference to a ROS *NodeHandle* and a *PeisSubscriberHandler* each. The *registerCallback* functions are called from the *init*-method and register a user-written callback method for a given tuple key with PEIS, where the pure virtual *getPattern*-method returns the tuple pattern (or key) that is registered with PEIS. For example, *robotname.MoveTo.*.COMMAND* would ask PEIS to call the given callback function whenever the value/state of the *COMMAND* tuple changes in the ecology. A separate *init*-method is required, as class initialisation in C++ forbids to call derived methods from a superclass constructor.

The *processTuple* method is the place where the actual processing of incoming tuples is performed. In reaction to a *COMMAND=ON* change, the service would then read the contents

of the corresponding *PARAMETERS* tuple, parse the given parameters, and start execution of the corresponding activity.

A very simple example of a derived class is shown in Fig. 44 and 46. Here, the service class inherits from *TupleHandler* and provides the actual implementation of the *getPattern()* method as well as the *processTutple()* method. The *main* method first calls *ros::init* and then the service constructor, which in turn initialises PEIS and the ROS subscriptions and publisher for the newly created ROS node. The next call to *init* registers the ROS node callback with PEIS, and then enters the endless *spin*-loop, where the ROS node reacts to incoming PEIS tuples as well as its own ROS subscriptions.

3.11.6 ActionExekutor

While the PEIS-ROS interface described above works fine, the use of hard-coded tuple-names also implies that the overall system is rather inflexible. In particular, restarting a service requires to reuse and modify existing tuples, with the corresponding need to lock and synchronise tuple access between multiple nodes on the PEIS network.

Therefore, the use of PEIS *meta-tuples* was suggested by ORU during Y3. A meta-tuple retains the normal key attribute, but the tuple payload is linked at runtime to the value attribute of another (normal) PEIS tuple. This basically doubles the number of active tuples on the system, but increases flexibility, as the planner or other nodes can dynamically update and change the payload of existing meta-tuples. This also simplifies online testing of the system using the tupleview tool.

To reduce the implementation and re-factoring effort required to introduce meta-tuples, the *exekutor* class library encapsulates the core functionality of accessing meta-tuples from ROS software. To implement a new service on the robot, a subclass of *exekutor* is created. When instantiated, the class will automatically create and subscribe to a set of input and output meta-tuples with standardised names given to the constructor of the exekutor sub-class. The current setup uses the *command* and *parameters* input tuples on every exekutor, while output tuples include the *state* (idle, running, success, failed), *result*, and *progress* (percentage of completion). The exekutor than subscribes to any ROS topics it needs for its tasks. Once activated (by receiving *command=ON*), the exekutor processes the incoming ROS data, calls ROS services and publishes data to ROS as required. To simplify access to actionlib service calls, the *action_exekutor* base class already provides most of the necessary interface code. The concept is very similar to the PEIS-ROS bridge described above.

Please refer to deliverable D3.4 for details about meta-tuples and see section 4 for a list and documentation of the different exekutor services implemented for the second experimental loop of project Robot-Era. See figure 47 for a screenshot of the tupleview tool during a test of the Robot-Era shopping scenario.

```
#ifndef TUPLE_HANDLER_H
#define TUPLE_HANDLER_H

#include <string>
#include <boost/thread/mutex.hpp>
#include <ros/ros.h>
extern "C"{
#include <peiskernel/peiskernel_mt.h>
#include <peiskernel/peiskernel.h>
}

class TupleHandler
{
protected:
  ros::NodeHandle         nodeHandle;
  PeisSubscriberHandle    peisSubscriberHandle;
  std::string             tuplePattern; // robotName.MoveTo.*.COMMAND
  std::map<std::string,PeisTuple*> cachedTuples;
  boost::mutex            mutex;
public:
  TupleHandler( int argc, char ** argv ); // initialises ROS and PEIS
  ~TupleHandler( void );
  virtual void init();  // registers the callback function

  // return the PeisTuple-Key-Pattern you're interested in processing,
  // for example, "doro1.MoveTo.*.COMMAND".
  virtual std::string getPattern() = 0;

  // processTuple() will be called with incoming PeisTuples with keys
  // matching your getPattern().
  virtual bool processTuple(PeisTuple* t) = 0;

  // register the callback function used to process incoming tuples.
  // Signature is "void callbackFunction(PeisTuple* t, void* arg)"
  PeisSubscriberHandle registerCallback(const int owner,
      const std::string& key,
      void *userData, PeisTupleCallback *callbackFunction);
  PeisSubscriberHandle registerCallbackAbstract(const int owner,
      const std::string& key,
      void* userData, PeisTupleCallback *callbackFunction);

  // You can only getTuple()s that you have subscribe()d beforehand.
  PeisTuple* getTuple(const int owner, const std::string& key,
      const bool subscribeBeforeReading=false,
      const int timeoutMillis = 1000);
  ...
  virtual int getID(); // the Peis ID of this TupleHandler
};
#endif
```

Figure 43: tuple_handler.h

```
#ifndef DEMO_SERVICE_H
#define DEMO_SERVICE_H

#include <doro_peis/tuple_handler.h>

class DemoService : public TupleHandler
{
private:
  // add variables for your service here

public:
  DemoService( int argc, char ** argv );   // initializes both ROS and PEIS

  ~DemoService( void );

  std::string getPattern(); // doro.demo_service.*.COMMAND

  bool processTuple( PeisTuple * t );
};
#endif
```

Figure 44: DemoService.h

```
#include <doro_peis/demo_service.h>
#include <std_msgs/Float64.h>
#include <std_msgs/String.h>
#include <sstream>

static int ID = 777;


DemoService::DemoService( int argc, char ** argv ) :
  TupleHandler::TupleHandler( argc, argv )
{
  robotName = "doro";
  ROS_INFO( "DemoService: pattern is '%s'", getPattern().c_str() );
}


DemoService::~DemoService() {
  // empty
}


std::string DemoService::getPattern() {
  return robot_name + ".demo_service.*.COMMAND";
}
```

Figure 45: DemoService.cc (1/2)

```
bool DemoService::processTuple( PeisTuple* t ) {
  const std::string fullyQualifiedTupleKey = getTupleKey( t );
  const std::string payload = t->data;

  ROS_INFO( "processTuple: <%s,%s>",
            fullyQualifiedTupleKey.c_str(), payload.c_str() );

  if (payload == "ON") { // create a new tuple with inremented ID
    std::stringstream ss;
    ss << "doro.demo_service." << (++ID) << ".COMMAND";
    publishTupleRemote( 995, ss.str(), "OFF" );
  }
  return true;
}


int main(int argc, char **argv)
{
    // ros init, PEIS-ROS init, register tuple callback
    ros::init(argc, argv, "peis_ros_demo_service");
    DemoService tupleHandlerDemo( argc, argv );
    tupleHandlerDemo.init();

    ros::Rate loop_rate(10); // ros::Time::init();
    while( ros::ok() ) {
       ros::spinOnce();
       loop_rate.sleep();
    }
}
```

Figure 46: DemoService.cc (2/2)

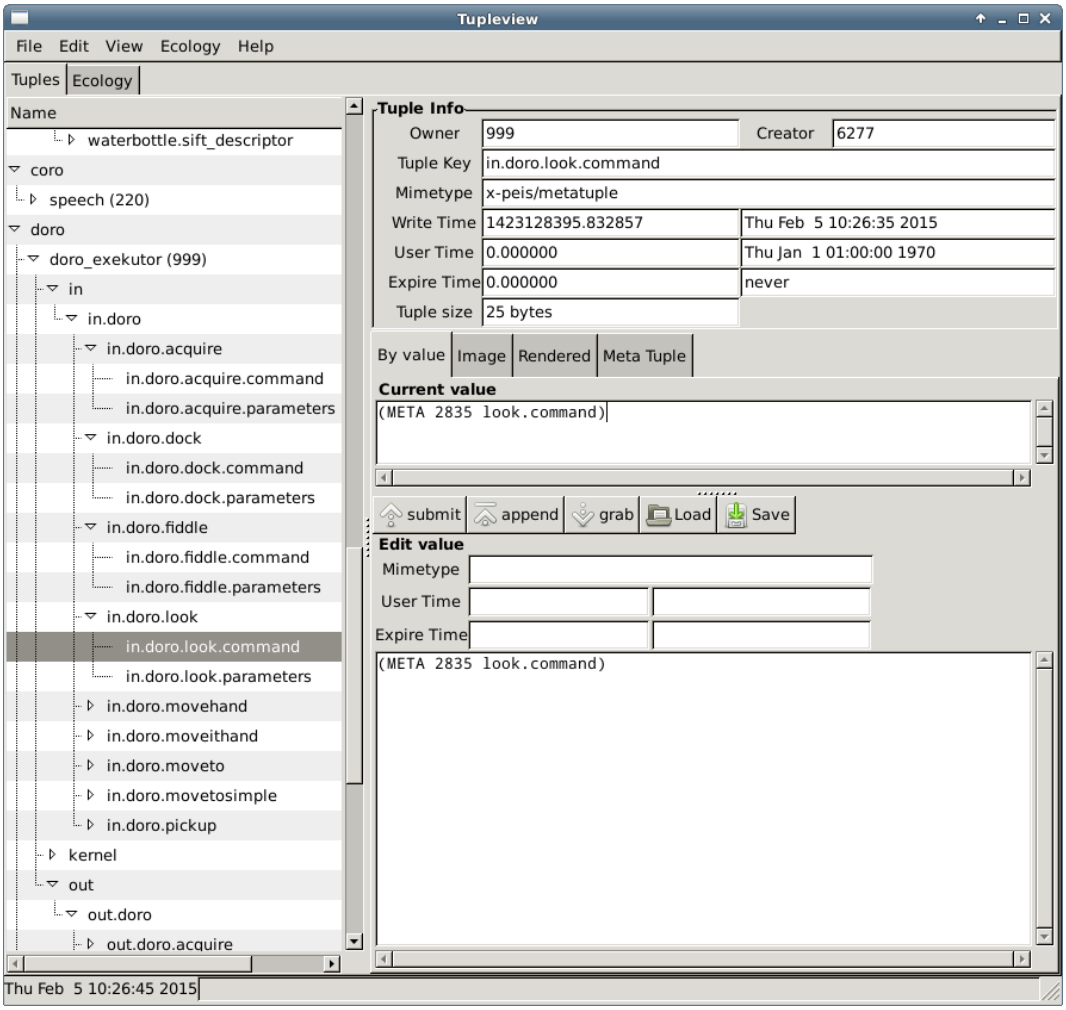


Figure 47: Screenshot of the PEIS *tupleview* tool showing a set of active exekutor instances corresponding to different services of the Domestic Robot. Using the meta-tuples handler, the configuration planner can simply create and link to new tuples instead of modifying existing tuples with hard coded names.

4 Services

This chapter documents the abstract *services* provided by the Robot-Era Domestic Robot and specifies the PEIS tuples used to configure and start the corresponding service. Following the project decision to switch the PEIS planning interface to the meta-tuples architecture, most Domestic Robot services were updated correspondingly during the end of 2014. The re-implementation effort was acceptable, as most of the changes on the PEIS side are encapsulated in the *action_exekutor* class. Only small changes were required on our existing ROS robot code, because both PEIS-ROS and exekutor interface to ROS via actionlib.

Note that the service descriptions in this handbook are not auto-generated from the actual source-code, but are simplified a bit to improve readability. Before using any services on the real robot or in simulation, we recommend to browse and download the latest documentation available on the project website *www.robot-era.eu*, the ROS wiki at *www.ros.org/wiki/robot-era*, and the Robot-Era software repository.

The robot services are and grouped by function in this handbook. First, subsection 4.1 lists a set of low-level *basic skills* of the robot, giving access to sensor data or triggering simple motions. Typically, each PEIS service in this group corresponds to one specific ROS node and service, so that the skills can be triggered from either PEIS or ROS.

The next group of services, described in subsection 4.2 on page 115 lists the *intermediate skills* that can be considered useful building blocks for the construction of typical robot tasks, for example, detecting and grasping an object.

The third group of *high-level services* is sketched in subsection 4.3 on page 133. These are the services used in the user-driven scenario descriptions developed within WP2 of the Robot-Era project.

Please refer to the website and software repository for the full list of implemented services, including all parameters and state/feedback options.

4.1 Low-Level Services

The services in this group encapsulate the *basic skills* of the Doro robot, typically targeting only a single sensor or actuator of the robot. While the services provide the basic sensing and motion infrastructure of the robot, most of the Robot-Era scenarios will not use the low-level services directly. Instead, the scenarios are designed to rely on the intermediate- and high-level skills described in subsections 4.2 and 4.3 starting on pages 115 and 133 below.

Of course, the higher-level skills are implemented internally from a suitable set of basic skills triggered in the required sequence. Exposing the low-level services via their individual tuple-handlers provides the Robot-Era system with better flexibility, because the individual skills can also be called by the planner and combined in new ways not implemented by intermediate- or high-level skills. The skills also provide a very useful means for robot software debugging and error-recovery, for example by allowing an expert user to move the arm around obstacles not handled by the OMPL motion planners, or by retracting the mobile base from a collision.

Note that most of the services described in this section correspond directly to one specific ROS service or actionlib service. In addition to being callable via PEIS, all skills can also be triggered via ROS messages, and several skills are accessible for interactive use via the software in the doro_teleop package.

During robot startup, one PEIS-ROS or exekutor tuple-handler is created and started for every low-level service. Each tuple-handler connects to the ROS topics and services required for the particular skill, and then monitors the PEIS network for tuples matching its own pattern. Once a tuple with matching key is received, the corresponding data is extracted from the value field(s) of the tuples, and stored internally. The skill is triggered as soon as the command=ON tuple is received, and executes until completed or until an error condition is detected. All long-running skills are expected to also provide an actionlib cancel operation, in order to pre-empt the task execution whenever necessary.

For every service, a short overview description is given, followed by the name of the software modules (*tuple-handlers*) implementing the service and the specification of the PEIS tuples defined to trigger the service and to monitor its execution progress. Where applicable, the implementation status of the service is also described.

4.1.1 EmergencyStop

Synopsis This service requests an emergency-stop of the Doro robot. The SCITOS-G5 mobile platform is stopped immediately, while the laser-scanners and sonars (when enabled) are kept running for obstacle monitoring and localisation. The service is activated by publishing to the *request_emergency_stop* topic, or by pushing the emergency-stop switch on the robot.

Any ongoing movements of the PTU and manipulator are stopped and the brakes of the arm are activated. As the Kinova Jaco arm has no hardware brakes, any ongoing motion is instead cancelled and the current robot position is sent as the new set-point. The arm is kept powered, because a switch-off results in the arm falling down and potentially harming the user and equipment.

To restart the robot, a message must be published to the *reset_emergency_stop* topic.

Handler `doro_peis/emergency_stop_handler.cpp`

Tuples

`in.doro.emergency_stop.command=` `OFF || ON`
`in.doro.emergency_stop.parameters=` none
`out.doro.emergency_stop.state=` `IDLE | ACTIVE`
`out.doro.emergency_stop.result=` unused
`out.doro.emergency_stop.progress=` unused

Status Implemented and tested.

4.1.2 GetCameraImage

Synopsis Requests to return the current image from one of the cameras on the robot sensor-head. This service bypasses the PEIS layer, because PEIS lacks support for structured image data and the Robot-Era planner currently does not use or process image data at all.

Instead, a client can directly request either single images or a MJPEG-encoded image stream from the *mjpeg_server2* running on the robot using standard `http` requests. This way, the images can also be shown on the table graphical user interface, e.g. to display the robot camera image of a unknown person in the surveillance scenario.

The mjpeg-server node is started at robot start-up time and keeps running continuously. However, images are only processed and published when a client requests an image or image-stream.

Handler `mjpeg_server2`
The service uses http or https requests directly instead of PEIS encapsulation. The service can also be run on the Condominium Robot.

Tuples
`http://doro:8080/embeddedstream?topic=/left_camera/image_raw ?width=320?height=240?quality=40`
Quality can be chosen from 1 (lowest) to 100 (highest).

Example
http://doro:8080/embeddedstream?topic=/left_camera/image_raw?width=640

Status Implemented and tested.

4.1.3 GetKinectImage

Synopsis Requests to return the current RGB or depth image from the Kinect/XtionPro depth-camera on the robot head. This service bypasses the PEIS layer, because PEIS lacks support for structured image data and the Robot-Era planner currently does not use or process image data at all.

Instead, a client can directly request either single images or a MJPEG-encoded image stream from the *mjpeg_server2* running on the robot. The mjpeg-server node is started at robot start-up time and keeps running continuously. However, images are only processed and published when a client requests an image or image-stream.

The mjpeg_server2 node is started at robot start-up time and keeps running continuously. Images are only processed and published when a client requests an image or image-stream.

Note: When accessing non-RGB data, the server only transmits the raw stream of image data. The client is in charge of demultiplexing and decoding the image. Most nodes accessing depth image data should subscribe directly to the corresponding ROS topic.

Handler `mjpeg_server2`
The service uses http or https requests directly instead of PEIS encapsulation. The service can also be run on the Condominium Robot.

Tuples
`http://doro:8080/stream?topic=/xtion_camera/depth_registered/image_raw?width=320?height=240?quality=40`

Example
doro:8080/stream?topic=/xtion_camera/depth/image_raw

Status Implemented and tested.

4.1.4 GetLaserScan

Synopsis Requests to return the current scan data from either the front (Sick S300) or rear (Hokuyo-URG) laser-scanners. This returns a string (char-stream) representation of the ROS *sensor_msgs/LaserScan* message, including timestamp, reference frame, and an array of distances in meters. We also return the m distance to the nearest obstacle, which may be useful to the high-level planner for re-planning when the on-board navigation is stuck and might need a changed high-level plan.

Handler `get_laser_scan_exekutor.cpp`

Tuples

`in.doro.get_laser_scan.parameters` = base_scan || base_scan_rear
`in.doro.get_laser_scan.command`= OFF || ON
`out.doro.get_laser_scan.state`= IDLE || RUNNING || COMPLETED || FAILED
`out.doro.get_laser_scan.result`=
timestamp reference-frame number-of-distances [distance]* min_distance
`out.doro.get_laser_scan.progress`= completion-percentage || error-description

Status The service is currently not used by the Robot-Era planner or the HRI modules. All ROS-nodes on the robot should simply subscribe to the */base_scan* or */base_scan_rear* topics directly.

4.1.5 GetSonarScan

Synopsis A Requests to return the latest distance data from the ring of 24 sonar sensors on the SCITOS base. Returns an array with 24 float values, each one giving the minimum distance to an obstacle reported by the corresponding sonar sensor.

Handler `get_sonar_scan_exekutor.cpp`

Tuples

`in.doro.get_sonar_scan.parameters` = base_scan || base_scan_rear
`in.doro.get_sonar_scan.command`= OFF || ON
`out.doro.get_sonar_scan.state`= IDLE || RUNNING || COMPLETED || FAILED
`out.doro.get_sonar_scan.result`=
timestamp reference-frame number-of-distances [distance]*
`out.doro.get_sonar_scan.progress`= completion-percentage || error-description

Status Not implemented. The project decided to rely on the much more robust and precise laser-scanner data for navigation and obstacle detection. The sonar-sensors are currently disabled on the robot.

4.1.6 MoveTo

Synopsis Requests the robot to drive to a goal position. The position is specified as a 2D-pose (x, y, Φ) consisting of the x and y coordinates (in meters) and an optional yaw-angle (in radians). The orientation and origin of the coordinate system are based on the current `/map` system of the robot. Optionally, the xy- and yaw-goal-tolerances for the final robot position can be specified in the parameter tuple. Otherwise, the ROS param server is queried to specify the goal tolerances of the driving motion:

- `/move_base/TrajectoryPlannerROS/xy_goal_tolerance`
- `/move_base/TrajectoryPlannerROS/yaw_goal_tolerance`

Mira/Cognidrive is used for controlling the SCITOS-G5 platform [43], while the ROS-MIRA bridge software interfaces MIRA to ROS. Robot localisation, motion planning and trajectory replanning to avoid dynamic obstacles are available and are considered stable. See section 3.2.2 for an overview of the ROS-MIRA bridge.

As a motion command will typically take many seconds before the robot has reached the goal position, the service relies on a ROS actionlib interface. When necessary, an active MoveTo service can be cancelled. Feedback about the robot position is not published to PEIS, but is easily available via the ROS *tf* topic.

Handler `moveto_exekutor.cpp`

Handler `moveto_simple_exekutor.cpp`

Tuples
`in.doro.moveto.parameters` = x y yaw (xy-tolerance (yaw-tolerance))
`in.doro.moveto.command`= OFF || ON
`out.doro.moveto.state`= IDLE || RUNNING || COMPLETED || FAILED
`out.doro.moveto.result`= unused
`out.doro.moveto.progress`= unused

Status Implemented and tested.

4.1.7 Dock and Undock

Synopsis Requests a robot motion to dock to a given object, or to retract (undock) from the object. The reference pose (x, y, Θ) of the target object is retrieved from the AmI CAM module, where Θ specifies the preferred approach/retract direction to and from the object. Coordinates are interpreted relative to the `/map` system.

The default action is to *dock* to the object specified as the single parameter in the *parameter* tuple. If *undock* is specified, this automatically refers to the previous *dock*-operation and target-object, so that no object needs to be specified.

Driving the robot is performed using calls to the *move-to-simple* exekutor.

Handler `dock_exekutor.cpp`

Tuples
`in.doro.dock.parameters` = object-name || undock
`in.doro.dockmoveto.command`= OFF || ON
`out.doro.dockmoveto.state`= IDLE || RUNNING || COMPLETED || FAILED
`out.doro.dockmoveto.result`= unused
`out.doro.dockmoveto.progress`= unused

Status Implemented and tested.

4.1.8 MovePtu

Synopsis Requests the robot to move the pan-tilt unit and therefore the sensor-head to a given goal position specified by the pan- and tilt-angles (radians) with respect to either the robot base coordinate system or the world-coordinate system.

With the current mount position of the PTU46 pan-tilt unit and the *doro_ptu46* ROS node, the orientation of the camera head is as follows:

- pan,tilt zero: cameras point straight forward
- pan: positive values are left, negative values are to the right. For example, 0.78 means 45° to the left, −1.57 means 90° to the right.
- tilt: positive values are upwards, negative values downwards. For example, 0.5 is 30° upwards, 0 is level, −0.5 is 30° down, −0.8 is 46° down.

Handler `move_ptu_exekutor.cpp`

Tuples

`in.doro.move_ptu.parameters` = pan-angle tilt-angle [coordinate-system]
`in.doro..move_ptu.command`= OFF || ON
`out.doro.move_ptu.state`= IDLE || RUNNING || COMPLETED || ERROR
`out.doro.move_ptu.result`= pan-angle tilt-angle
`out.doro.move_ptu.progress`= completion-percentage || error-description

Status Implemented and tested.

4.1.9 RetractJacoArm

Synopsis Requests a joint-level motion of the Jaco arm back to its *retract* (home) position. This service enables the PEIS layer to request moving the arm back to its initial position. This is required because several Kinova API functions can only be called after the arm has been retracted to its home position. Without this service, the robot-planner would not be able to initialise (or re-initialise) the Jaco arm.

Handler `peis_ros/RetractJacoTupleHandler.cpp`

Tuples

Status Implemented. Replaced by *MoveJacoArm* with corresponding joint-angles.

4.1.10 ParkJacoArm

Synopsis Requests a joint-level motion of the Jaco arm back to its *park* (safe) position. The Jaco arm lacks brakes on its joints, and the planetary gears have little friction and are not self-locking. As such, the arm needs to be parked in specific positions in order to avoid the arm falling down under the effects of gravity when the arm is switched off. This service enables the PEIS layer to request moving the arm back to a safe parking position.

Handler `peis_ros/ParkJacoTupleHandler.cpp`

Tuples

Status Implemented. Replaced by *MoveJacoArm* with corresponding joint-angles.

4.1.11 MoveJacoArm

Synopsis Requests a cartesian or joint-level motion of the Jaco arm, using either absolute values or values relative to the current hand position.

For cartesian movements, the coordinates are specified as six double values interpreted as (x, y, z) position and (R, P, Y) Euler angles. If *absolute* is specified, values are interpreted in respect to the *jaco_link_base* frame of the arm (arm mount), while *relative* values are relative to the current hand position.

For joint-level movements, the target is also specified as six double values. If *absolute* is specified, values are interpreted as absolute joint-angles $(j_1, j_2, j_3, j_4, j_5, j_6)$, while *relative* specifies the per-joint offset to the current arm position.

Handler `move_hand_exekutor.cpp`

Tuples

`in.doro.movehand.parameters` =
cartesian || joints
absolute || relative
x y z R P Y || j1 j2 j3 j4 j5 j6
`in.doro.movehand.command`= OFF || ON
`out.doro.movehand.state`= IDLE || RUNNING || COMPLETED || FAILED
`out.doro.movehand.result`= unused `out.doro.movehand.progress`= unused

Example

`in.doro.movehand.parameters` = joints absolute -1.5 0.2 0.3 0.4 3.14 0.6

Status Implemented and tested.

4.1.12 MoveJacoCartesian

Synopsis Requests a joint-level motion of the Jaco arm to the given cartesian pose (x, y, z, R, P, Y). Note that the current implementation takes coordinates with reference to the Kinova Jaco coordinate system.

Handler `move_hand_exekutor.cpp`

Tuples See MoveJacoArm.

Status Implemented and tested.

4.1.13 MoveitJacoCartesian

Synopsis Requests a collision-aware joint-level motion of the Jaco arm to the given cartesian pose $(x, y, z, q_x, q_y, q_z, q_w)$ (position, quaternion) with respect to the `base_link` coordinate system of the robot. The motion trajectory is planned by MoveIt with default parameters for goal position and orientation constraints.

Handler `moveit_hand_exekutor.cpp`

Tuples

`in.doro.plan_and_move_arm.parameters =`
x y z qx qy qz qw
`in.doro.plan_and_move_arm.command=` OFF || ON
`out.doro.plan_and_move_arm.state=` IDLE || RUNNING || COMPLETED || FAILED
`out.doro.plan_and_move_arm.result=` unused
`out.doro.plan_and_move_arm.progress=` unused

Status Implemented and tested.

4.1.14 MoveJacoFingers

Synopsis Requests to move the fingers to the given joint-angles. While Jaco finger motions are fast and usually complete in less than one second, the current implementation still uses actionlib calls based on the *jaco/finger_action* interface of the JacoROS arm driver node.

The actual motion is selected by the *open*, *close*, or *absolute* command keyword in the PEIS parameter tuple, where *absolute* is followed by three numerical values for the target finger positions for fingers F1, F2, and F3. The useful range is $[-1, +1]$, where positive values indicate closed fingers, 0 is open, and -1 is required sometimes to reset the hand.

Handler `fiddle_exekutor`

Tuples

`in.doro.fiddle.command=` `OFF | ON`
`in.doro.fiddle.parameters=` open || close || absolute f1 f2 f3
`out.doro.fiddle.state=` `IDLE | ACTIVE`
`out.doro.fiddle.result=` `SUCCESS || FAILED`
`out.doro.fiddle.progress=` unused

Status Implemented and tested.

4.2 Intermediate Services

According to the terminology introduced above, the *Intermediate Services* collect multi-modal perception tasks and manipulation actions that form the building blocks to construct meaningful robot tasks and the actual Robot-Era services.

On the sensing level, the intermediate services encapsulate perception tasks that use information from more than one sensor or include significant pre-processing. Typical examples object detection using markers or the sensor-fusion from point-cloud and image-feature data.

For manipulation tasks, most grasping actions fall in this category, because coordinated hand and finger motions must be executed in combination with environment perception to avoid collisions.

4.2.1 DetectKnownObject

Synopsis Known objects can be detected using one of the following methods.

- SIFT-based object detection
- PCL-based object detection (including SIFT based classification)[1]
- AprilTag Markers

In this step only the appearance of the object is detected, not the exact grasp points.

Handler `get_detected_objects_exekutor`

Tuples
`in.doro.visionhub_detection.command=` `OFF | ON`
`in.doro.visionhub_detection.parameters=` unused
`out.doro.visionhub_detection.state=` `IDLE | ACTIVE`
`out.doro.visionhub_detection.result=` `SUCCESS || FAILED`
`out.doro.visionhub_detection.detections=` object_list, comma separated+

Status
Implemented, final tuple format definition may need to be modified.

[1] In this case SIFT is only used to classify the defected point cloud by finding the best match in the object database

4.2.2 DetectUnknownObject

Synopsis Unknown objects can be detected using the Xtion sensor and the tabletop-segmentation stack from ROS. Internally, the detection first calculates the position/orientation of a table-top surface and then searches for point-cloud clusters on top of this surface. For each cluster, the bounding-box, predominant colour, and centroid are calculated and used for object matching against known objects. Otherwise, the bounding-box can be used for approximate object grasping. Information about detected objects is forwarded to the AmI CAM-module.

Handler `acquire_exekutor.cpp`

Tuples
`in.doro.acquire_tabletop.command=` `OFF | ON`
`in.doro.acquire_tabletop.parameters=`
all tolerance || known tolerance || signature id tolerance

`CAM_peis_id` = 9898
`CAM_peis_id.object_name.pos.geo.update=` estimated-object-position x y z
`CAM_peis_id.object_name.color.rgb.update=` estimated-object-colour r g b
`CAM_peis_id.object_name.boundingbox.update=` estimated-object-colour r g b

Example
`in.doro.acquire_tabletop.parameters=` all 0.8
`in.doro.acquire_tabletop.parameters=` signature nespresso-1 0.7

Status
Implemented and tested.

4.2.3 GraspAndLiftKnownObject

Synopsis This service implements grasping and lifting a known object, referring to an object whose properties are known to the system. Object geometry will be based on a small set of known basic shapes (sphere, cylinder, box) or the full 3D-mesh of the object.

Once triggered, the service will try to move to the given location, try to detect the object via image- and depth-image processing, and estimate the object pose. The database is then queried for the set of possible grasps, and the constraints- and collision-aware motion planners will try to find a suitable arm trajectory. The trajectory is then executed to grasp the object, with visual servoing as possible and force-feedback from the Jaco arm to check the grasp result.

Handler `grasp_exekutor`

Tuples
`in.doro.grasp_exekutor.command=` `OFF | ON`
`in.doro.grasp_exekutor.parameters=` `OBJECT_NAME`, optional comma separated list of arguments: top, side
`out.doro.grasp_exekutor.state=` `IDLE | ACTIVE`
`out.doro.grasp_exekutor.result=` `IDLE` || `RUNNING` || `SUCCESS` || `FAILED`
`out.doro.grasp_exekutor.progress=` unused

Status
Implemented for box-shaped objects, arbitrary shapes are supported when manually defining the grasppoints. Database structure and setup to be decided: reuse household-objects from ROS, or start with something in PEIS?

4.2.4 SideGraspAndLiftObject

Synopsis The side grasp service previously defined as a standalone-service is now included in the grasp_exekutor, where the calling client can restrict the grasps to side-grasps.

Handler `grasp_exekutor`

Tuples
`in.doro.grasp_exekutor.command=` `OFF | ON`
`in.doro.grasp_exekutor.parameters=` `OBJECT_NAME`, optional comma separated list of arguments: top, side
`out.doro.grasp_exekutor.state=` `IDLE | ACTIVE`
`out.doro.grasp_exekutor.result=` `IDLE` || `RUNNING` || `SUCCESS` || `FAILED`
`out.doro.grasp_exekutor.progress=` unused

Status
Implemented.

4.2.5 TopGraspAndLiftObject

Synopsis The top grasp service previously defined as a standalone-service is now included in the grasp_exekutor, where the calling client can restrict the grasps to top-grasps.

Handler `grasp_exekutor`

Tuples
`in.doro.grasp_exekutor.command=` `OFF | ON`
`in.doro.grasp_exekutor.parameters=` `OBJECT_NAME`, optional list of arguments: top, side
`out.doro.grasp_exekutor.state=` `IDLE | ACTIVE`
`out.doro.grasp_exekutor.result=` `IDLE` || `RUNNING` || `SUCCESS` || `FAILED`
`out.doro.grasp_exekutor.progress=` unused

Status Implemented.

4.2.6 PlaceObjectOnTray

Synopsis After an object has been grasped, place it on the tray of the robot. The caller can either use one of three predefined positions (leftcenterright) or specify the exact pose of the object base. The transformation endeffector to object base will be calculated automatically.

Handler `place_exekutor`

Tuples

`in.doro.place_exekutor.command=` `OFF` || `ON`
`in.doro.place_exekutor.parameters=`
tray,
cartesian-place-pose x y z R P Y ||
position leftcenterright
`out.doro.place_exekutor.state=` `IDLE` || `RUNNING` || `SUCCESS` || `FAILED`
`out.doro.place_exekutor.result=` `SUCCESS` || `FAILED`
`out.doro.place_exekutor.progress=` unused

Status
Implemented, theoretically working, final tests necessary before beginning of second experimenatation loop.

4.2.7 PlaceObject

Synopsis After an object has been grasped, place it on either another object or a cartesian position. The caller can specify the exact pose of the object base with respect to the position of another object. The transformation endeffector to object base will be calculated automatically.

Handler `place_exekutor`

Tuples

`in.doro.place_exekutor.command=` `OFF` || `ON`
`in.doro.place_exekutor.parameters=`
object to place currently grasped object on,
cartesian-place-pose x y z R P Y
`out.doro.place_exekutor.state=` `IDLE` || `RUNNING` || `SUCCESS` || `FAILED`
`out.doro.place_exekutor.result=` `SUCCESS` || `FAILED`
`out.doro.place_exekutor.progress=` unused

Status
Implemented, theoretically working, final tests necessary before beginning of second experimenatation loop.

4.2.8 DropObject

Synopsis After an object has been grasped, drop it into another object or a at a cartesian position. The caller can specify the exact pose of the object base with respect to the position of another object. The transformation endeffector to object base will be calculated automatically.

Handler `drop_exekutor`

Tuples

`in.doro.drop_exekutor.command=` `OFF` || `ON`
`in.doro.drop_exekutor.parameters=`
object to drop currently grasped object into,
cartesian-place-pose x y z R P Y
`out.doro.drop_exekutor.state=` `IDLE` || `RUNNING` || `SUCCESS` || `FAILED`
`out.doro.drop_exekutor.result=` `SUCCESS` || `FAILED`
`out.doro.drop_exekutor.progress=` unused

Status
Implemented, theoretically working, final tests necessary before beginning of second experimentation loop.

4.2.9 GraspObjectFromTray

Now implemented in the grasp_exekutor, that can store the exact position where the object has been placed on the tray by interacting with the Visionhub node. Works similar to the GraspAndLiftKnownObject service, except that the release-position is used to re-grasp the object, as currently the tray is not visible with the cameras.

Synopsis

Handler `grasp_exekutor`

Tuples
`in.doro.grasp_exekutor.command=` `OFF | ON`
`in.doro.grasp_exekutor.parameters=` `OBJECT_NAME`, optional comma separated list of arguments: top, side
`out.doro.grasp_exekutor.state=` `IDLE | ACTIVE`
`out.doro.grasp_exekutor.result=` `IDLE || RUNNING || SUCCESS || FAILED`
`out.doro.grasp_exekutor.progress=` unused

Status Needs extensive testing.

4.2.10 HandoverObjectToUser

Synopsis This service executes an object handover from the Domestic Robot to the user, where the actual handover is triggered by the forces applied by the human when taking the object.

Prerequisite: The robot has grasped an object and is close to the human. Where necessary, the *find-user* service followed by a *move-to* or *move-to-simple* should be called by the planner first. Additionally, the planner should ensure that the grasp pose of the Jaco hand is suitable for the given size, shape, and weight of the object and any additional task constraints. For example, a side grasp might be required for successful handover of a cup filled with liquid. On the other hand, a grasp pose with two fingers below the object increase success for a heavy object.

The robot now triggers a Jaco arm motion towards the user and stops close to the presumed user position. The robot than waits until the specified force-threshold is detected on the Jaco arm, which triggers the *open-fingers* motion to release the object. Optionally, the handover pose can be specified using either cartesian coordinates (with respect to robot /base_link) or using joint-angles. The force-thresholds to trigger the release motion can be specified individually for each joint of the Jaco arm. See section 3.8.4 and references [8] and [9] for a description of the state-machine and the algorithms used.

Handler `handover_exekutor`

Tuples

`in.doro.handover.command=` `OFF` || `ON`
`in.doro.handover.parameters=`
empty ||
cartesian-handover-pose x y z R P Y ||
joints-handover-pose j1 j2 j3 j4 j5 j6
(force-threshold j1 j2 j3 j4 j5 j6)
`out.doro.handover.state=` `IDLE` || `RUNNING` || `SUCCESS` || `FAILED`
`out.doro.handover.result=` `SUCCESS` || `FAILED`
`out.doro.handover.progress=` unused

Status Implemented and tested.

4.2.11 HandoverObjectFromUser

Synopsis This service executes an object handover from the user to the Domestic Robot, where the grasping of the object is triggered by the forces applied by the human.

Prerequisite: The robot robot is close to the human and the object presented by the human is within reach of the Jaco hand. Where necessary, the *find-user* service followed by a *move-to* or *move-to-simple* should be called by the planner first. Additionally, the planner should ensure that the grasp pose of the Jaco hand is suitable for the given size, shape, and weight of the object and any additional task constraints.

The robot now triggers a Jaco arm motion towards the user and stops close to the presumed handover position. The robot than waits until the specified force-threshold is detected on the Jaco arm, which triggers the *close-fingers* motion to grasp the object. The handover pose can be specified using either cartesian coordinates (with respect to robot /base_link) or using joint-angles. The force-thresholds to trigger the release motion can be specified individually for each joint of the Jaco arm.

Handler `handover_from_user_exekutor`

Tuples

`in.doro.handover_from_user.command=` `OFF` || `ON`
`in.doro.handover_from_user.parameters=`
empty ||
cartesian-handover-pose x y z R P Y ||
joints-handover-pose j1 j2 j3 j4 j5 j6
(force-threshold j1 j2 j3 j4 j5 j6)
`out.doro.handover_from_user.state=` `IDLE` || `RUNNING` || `SUCCESS` || `FAILED`
`out.doro.handover_from_user.result=` `SUCCESS` || `FAILED`
`out.doro.handover_from_user.progress=` unused

Status Implemented.

4.2.12 PourLiquidMotion

Synopsis This service was documented in the previous version of the robot handbook [58], because a corresponding function is included in the Kinova Jaco API and driver. However, the function relies on user-teleoperation and would be very hard to implement successful using autonomous control. Also, drinking support is not used in any current Robot-Era scenario.

Handler `canceled`

Tuples

Status Not used in any Robot-Era scenario for the second experimental loop. Not implemented.

4.2.13 MoveHingedDoor

Synopsis

Handler `peis_ros/MoveDoorTupleHandler.cpp`

Tuples

Status Not implemented. Where necessary, automated doors will be used in the second experimental loop.

Due to the inadequate support of force- and tactile-sensor readout in the Jaco driver, precise impedance-controlled motions are not possible, and the risk of damage to the Jaco arm or objects would be very high in position-controlled motions. Implementation will be reconsidered for robot arms with adequate sensors and software.

4.2.14 LookAt

Synopsis This service moves the pan-tilt unit so that the cameras looks at an object known to the PEIS CAM-module. The CAM in turn returns the corresponding position of the object as a point (x, y, z) in world coordinates, or optionally any coordinate system known to the *tf*-transformation library. Unlike the low-level *MovePtu*-service, this service allows the user or planner to request images (or point-clouds) from a given target location without having to worry about the current position and orientation of the robot.

Handler `look_exekutor.cpp`

Tuples
`in.doro.look.command=` `OFF` || `ON`
`in.doro.look.parameters=` object-name (on CAM module)
`out.doro.look.state=` `IDLE` || `RUNNING` || `SUCCESS` || `FAILED`
`out.doro.look.result=` `SUCCESS` || `FAILED`
`out.doro.look.progress=` unused

Status
Implemented and tested.

4.2.15 DetectPerson

Synopsis This service uses the robot sensors to detect persons near to the robot. Several sensor modalities and algorithms may be combined, but the actual implementation is not exported to the PEIS layer. When the *move-head* flag parameter is included, the robot may use PTU motions to also detect people not directly in front of the current head position of the robot. No platform motions are attempted; but the PEIS planner can initiate robot motions to look for people in different rooms.

The current implementation is based on a pre-trained SVM classifier working on point-cloud data from the Xtion camera. In the first step, a ground-plane estimation step is used to find clusters in the point cloud, which are then classified according to their size and shape. The ground-plane estimation is initialised from the robot PTU pan- and tilt-angles, and confidence values are calculated for each cluster. Positions of detected persons are calculated with respect to the given reference frame (usually /map for absolute positions, or /base_link for positions relative to the robot base).

The *result* tuple contains the number of detected persons in view of the robot $(0 \ldots n)$ and one tuple consisting of the estimated (x, y, z) position of one person together with a confidence level (float) for that person. Work is underway to also estimate the relative motion of the person (direction and velocity) with respect to the robot. Progress info is not published, because the service completes when a person is detected of after 10 seconds runtime.

Handler `find_user_exekutor.cpp`

Tuples

`in.doro.find_user.command`= OFF || ON
`in.doro.find_user.parameters` = (move-head)
svm || openni (|| face-detector|| leg-detector) x y z
`out.doro.find_user.result`= n (x y z)*n
`out.doro.find_user.state`= IDLE || RUNNING || COMPLETED || FAILED
`out.doro.find_user.progress`= unused

Example

`in.doro.find_user.parameters`= svm 3 1 1.8
`out.doro.find_user.result`= 0
`out.doro.find_user.result`= 1 3.14 0.71 1.83 0.84
`out.doro.find_user.result`= 2 3.14 0.70 1.79 0.66 2.80 -1.6 1.59 0.33

Status
Implemented and tested using the SVM clustering classifier.

4.2.16 TrackPerson

Synopsis This service uses the robot sensors to detect and track persons near to the robot. Several sensor modalities and algorithms may be combined, but the actual implementation is not exported to the PEIS layer. The input (x, y, z) parameters of the service specify which one of multiple users to track.

Using PTU motion commands, the head of the robot rotates so as to keep the robot looking straight at the user. If no user is detected or the user is lost, the service returns *failed*. Otherwise the service continues until canceled by the planner. The current runtime is sent back to the planner using the *progress* tuple.

Handler `track_user_exekutor.cpp`

Tuples
`in.doro.track_user.command=` OFF || ON
`in.doro.track_user.parameters` =
svm || openni || face-detector|| leg-detector x y z
`out.doro.track_user.result=` none
`out.doro.track_user.state=` IDLE || RUNNING || COMPLETED || FAILED
`out.doro.track_user.progress=` runtime-in-seconds

Example
`in.doro.track_user.parameters=` svm 3 1 1.8

Status
Prototype implemented (NiTE2 tracker). Needs to be updated for the SVM-cluster detector.

4.2.17 RecognizePerson

Synopsis Use the robot sensors to recognize a person near to the robot. The current implementation is based on the Eigenfaces algorithm [23], and requires the corresponding set of pre-processed images for each of the known persons. The recognition result is either the name/index of a known person, together with the estimate of the recognition probability, NOT-RECOGNIZED to indicate that an unknown person was detected, or NO-PERSON to indicate that no person could be detected in the current robot sensor data.

Handler `peis_ros/RecognizePersonTupleHandler.cpp`

Tuples

`in.doro.recognize_person.command=` OFF || ON
`in.doro.recognize_person.parameters` =
    empty || camera-name region-of-interest: xl yt xr yb
`doro.recognize_person.state=` IDLE || RUNNING || COMPLETED || FAILED
`doro.recognize_person.result=`
    known-person-name|| unknown-person|| no-person-detected
`doro.recognize_person.progress=` unused

Status
Prototype implemented (Eigenfaces algorithm). Not yet integrated or tested, as automatic detection of persons has legal consequences that have to be clarified before use during the experimental loop.

4.3 High-level Services

While the low-level and intermediate robot skills described in the previous sections are valuable building blocks for the robot programmer, those skills are certainly not useful for the typical end-user. Instead, the scenario storyboards developed by Robot-Era from the user studies and questionnaires refer to much more complex actions like *clean the dinner table*. This also includes the benchmark task of all service robots, *bring me a cup of coffee*, where the robot has to identify the user, find the users' preferred cup, prepare coffee, carry the full cup without spilling the liquid, and finally hand-over the cup to the user.

The high-level robot services listed in this section directly correspond to the activities requested by the users and documented in the project storyboards. All of the services require *significant robot autonomy* and include complex action sequences, including detection and identification of objects in cluttered environments, skilled manipulation and transport of objects, and interaction with the user(s). Some of the requested tasks are clearly beyond the current state of the art, and only experience will tell whether the Domestic Robot with the Jaco arm is capable of the tasks at all. For example, the *clean the window* service involves the handling and manipulation of tools, possibly even the grasping of a wet soft spoon, and very difficult perception tasks involving transparent and mirroring objects. Due to the high-level nature of the services, the command interface from PEIS is very simple. The AmI just prepares the required parameters ("bathroom window") and triggers the corresponding robot service.

In order to keep a clean implementation structure and to be able to switch between tasks in a reliable way, many of the high level services are composed by the planner by calling several middle level services consecutively. This way it is more convenient to implement error handling or cancellation of tasks.

However, given the large number of intermediate and low-level skills required to execute the service, a lot of things can go wrong and *execution-monitoring* and *error recovery* become very important aspects of the implementation. Whenever the robot cannot execute the active part of the overall action plan, the service will be canceled and the error reported to PEIS. The AmI planner is then responsible to handle the error, and to attempt a recovery, for example by requesting the user to point-and-click on the target object in a camera image of the robot when the image processing algorithms fail to identify the object.

4.3.1 WalkingSupport

Synopsis This service requests walking support for the user. The robot first moves to the given initial position, tries to detect and recognize the user, and rotates so that the user can easily reach grasp the handle. The robot then slowly moves to the target position, with careful execution monitoring to ensure that the users keeps up with the robot and maintains a stable grasp on the handle.

The service has been split into an autonomous platform motion towards the user, followed by interactive drive control of the robot by the user using the switches in the tilting handle.

Handler `moveto_exekutor`

Tuples None.

Status To provide a better user experience, this service has been replaced by the direct tele-operation of the Domestic Robot. Using the switches in the tilting handle, the user can directly control and drive the robot. Note that the tele-operation software monitors the laser-scanners to detect and prevent imminent collisions with obstacles in front of the robot.

4.3.2 SwipeSurfaceService

Synopsis This service provides a simplified prototype version of *indoor cleaning* tasks that is matched to the hardware capabilities of the Kinova Jaco arm and the Moveit motion planning.

The tasks uses a set of pre-defined tools (sponges and brushes) that have been selected or have been modified to be graspable by the Jaco hand. Due to lack of force-control on the arm, tool compliance is used to protect the arm and objects during position-controlled motions. Also, only selected objects will be available for cleaning (e.g. rectangular regions on tables, windows, and kitchen surfaces), so that location and surface properties can be modelled in the AmI layer (CAM module).

The actual cleaning tasks will consist of task-aware swiping motions which demonstrate realistic and coordinated arm and tool motions. No water or cleaning liquids will be used, to reduce the risk of accidents with damage to the robot or furniture. This obviously limits the efficiency and results of the cleaning, but it is expected that the users will understand the reasons and rate user acceptance as if the task were executed with liquids.

The elasticity properties of the tools and suitable motion plans (swiping left-to-right and right-to-left, swiping alternating with free-space return motions, circular motions, handling of corners) will be encapsulated in the swiping service and are not exposed or exported to the AmI/CAM or planning layer. Where necessary, AprilTag markers will be placed on or close to the target surface, in order to guarantee the position accuracy required by successful task execution.

For the full cleaning scenario, the following action sequence is executed by the planner, AmI/CAM, and Domestic Robot:

1. user requests the *cleaning tasks* via speech or GUI,
2. user also selects the room+object to be cleaned,
3. the planner queries the CAM for the appropriate tool and its current location,
4. the planner generates the *moveto*, *look* and *acquire*, and *pick-object* commands for Doro to locate and grasp the tool,
5. the planner generates the *moveto* command to drive the robot to a position in front of the target surface,
6. the planner generates the parameters for the *swipe-surface* command, including the *tool-name*, *surface-name*, the four *corners* or the *region-of-interest* to be cleaned, and optionally the *exit-position* where the swiping motions should converge to,
7. Doro generates motion plans for the given tasks and executes them autonomously. Progress feedback is periodically sent back to the planner and user interface,
8. Doro retracts the arm into the home position and indicates *SUCCESS* or *FAILED*,
9. the planner generates commands to place the tool and informs the CAM about the new location.

Handler `swipe_surface_exekutor.cpp`

Tuples

`in.doro.swipe_surface.command=` OFF || ON
`in.doro.swipe_surface.parameters` =
tool-name surface-name
corner1: (x,y,z) corner2: (x,y,z) corner3: (x,y,z) corner4: (x,y,z)
exit-position: (x,y,z)
`out.doro.swipe_surface.result=` unused
`out.doro.swipe_surface.state=` IDLE || RUNNING || COMPLETED || FAILED
`out.doro.swipe_surface.progress=` percentage-of-completion || -1 (FAILED)

Example

`in.doro.swipe_surface.parameters` =
brush-1 kitchen-table 2 2 0.72 2.5 2 0.72 2.5 2.3 0.72 2 2.3 0.72
`in.doro.swipe_surface.parameters` =
sponge-3 floor 0.1 0.1 0 0.5 0.1 0.01 0.1 0.5 0.01 0.5 0.5 0.02 0.1 0.1 0

Status Prototype implemented; needs testing. The service will be tested during the second experimental loop.

4.3.3 CleanFloorService

Synopsis Based on analysis of the user studies, the *clean floor service* consists of two different tasks, namely picking up objects lying around on the floor, but also sweeping the floor. This service was included as a placeholder in the first version of the Domestic Robot handbook, D4.2 [58].

For the second experimental loop, the service will be performed in two steps by calling the corresponding individual services. The *grasp-object* service will be used to for searching and the picking-up of objects on the floor, while the *swipe-surface* service is used for floor sweeping. The following limitations apply:

- realistic use of a full-size broom is not possible due to torque limits on the hand and fingers of the Jaco arm. Smaller brushes or a broom with shorter handle will be used, limiting the sweeping workspace to a region on the right side of the Domestic Robot.
- due to occlusion, the areas directly in front and to the right of the robot cannot be observed by the XtionPro sensor and the cameras on the sensor head. Therefore, the side camera will be used for object/obstacle detection and grasping/sweeping will be performed on the right of the robot.
- Alternatively, perception of objects on the floor must be performed while the robot is still about 1 m away from the objects, followed by a platform movements, followed by (blind) grasping.

Handler
`grasp_exekutor`, `swipe_surface_exekutor`

Tuples
Please refer to the corresponding services.

Status
The necessary services are implemented and tested.

4.3.4 CleanWindowService

Synopsis Requests the robot to clean the specified window. As originally specified, the robot is expected to remove any obstacles in front of the window, to prepare a bucket with water and detergent, put the bucket onto the tray, grasp a sponge, move to the window, perform cleaning motions, bring the dirty water to the kitchen, clean bucket and sponge, and finally restore any removed objects back to their original places.

Unfortunately, the complexity remains beyond the state-of-the-art of service robotics in general and the capabilities of the Domestic Robot hardware in particular. Therefore, only a simplified version will be demonstrated during the second experimental loop, where the robot performs cleaning motions using a sponge. This can be performed by a corresponding parametrisation of the *swipe-surface service* with the sponge specified as the tool object and the coordinates of the window selected by the planner.

Handler `swipe_surface_exekutor`

Example

```
in.doro.swipe_surface.parameters =
    sponge-2  window-1  2.0 3.0 1.0  2.0 3.5 1.0  2.0 3.5 1.5  2.0 3.0 1.5  2.0 3.25 1.0
```

Status Prototype implemented; needs testing.

4.3.5 CleanTableService

Synopsis As originally specified, the robot moves to the given table, looks for known and unknown objects, grasps the objects and places them on the transportation tray, drives to the kitchen, and puts the objects from the tray to the kitchen sink or the dishwasher. The robot then returns to the table with a wet sponge or cloth, and swipes the table clean. Optionally, the table is laid again.

The project decided that this service cannot be executed atomically by the robot alone. Instead, the driving motions, object detection, pick and place, and the cleaning are commanded by the planner in combination with the Ami CAM-module. The integrated service is thereby replaced by the corresponding lower-level services and using known objects.

Handler
`moveto_exekutor`, `dock_exekutor`,
`look_exekutor`, `acquire_exekutor`,
`moveit_hand_exekutor`, `grasp_exekutor`, `place_exekutor`,
`swipe_surface_exekutor`.

Tuples
See the corresponding individual services.

Status This service is beyond the state-of-the-art of service robotics (considering unknown objects are placed on arbitrary positions), and will be very difficult to execute with the current Doro hardware setup. A simplified version of the service using known objects is expected to be tested as part of the second experimental loop.

4.3.6 BringFoodService

Synopsis The BringFoodService is expected to be tested in the second experimental loop. A special box featuring a handle and an optical marker has been designed, that can be handed over from the tray of the Condominium Robot to the Domestic Robot, that will fetch this box from the apartment door and place it on the table. This service calls the docking executor, the grasp_exekutor and the place executor. In a planned extension of this scenario, the robot detects (find_user_exekutor) that the user is in front of the robot, and instead of placing the object on the table it directly starts the handover procedure (handover_exekutor).

Handler
`grasp_exekutor`, `place_exekutor`, `docking_exekutor`, `find_user_exekutor`, `handover_exekutor`

Tuples
Please refer to the corresponding low- and intermediate-level services.

Status
The necessary low and intermediate level services are implemented and tested. Object handover has been tested during the M38 integration week and works reliably.

4.3.7 CarryOutGarbage

Synopsis The CarryOutGarbage is expected to be tested in the second experimental loop. A special bucket featuring a handle and an optical marker has been designed, that can be grasped by the Domestic Robot and handed over to the Condominium Robot. The garbage bin is placed on a defined location in the room. The planner drives the robot to a location, where the marker of the bucket can be detected, then the *grasp*-service will be called which calculates a grasp that can be executed. When the bin is grasped, the Domestic Robot docks to the Condominium Robot and calls the *place_exekutor* in order to put the bin on the tray of Coro.

Handler
`grasp_exekutor`, `place_exekutor`, `docking_exekutor`

Tuples
Please refer to the corresponding low- and intermediate-level services.

Status
The necessary low and intermediate level services are implemented and tested.

4.3.8 LaundryService

Synopsis The LaundryService is expected to be tested in the second experimental loop. A special box featuring a handle and an optical marker has been designed, that can be grasped by the Domestic Dobot and handed over to the Condominium Robot. The procedure and the handlers correspond directly to that of the *CarryOutGarbage*-scenario.

Handler
`grasp_exekutor`, `place_exekutor`, `docking_exekutor`

Tuples
Please refer to the corresponding low- and intermediate-level services.

Status
The necessary low and intermediate level services are implemented and tested.

5 Software installation and setup

This chapter summarises installation, setup, and configuration instructions for the different main software packages required for the Domestic Robot.

5.1 Ubuntu 12.04 and ROS Hydro

While ROS can be compiled on many Linux platforms, the availability of pre-built packages with regular updates is best for Ubuntu-based platforms. Therefore, Ubuntu 12.04-LTS was chosen as the operating system of the Domestic Robot. It needs to be installed on all on-board computers (the main Scitos-G5 computer and the two additional Intel NUC D5250WYK modules).

The original software documented in D4.2 [58] was based on ROS version Fuerte, but three new releases (Groovy, Hydro, Indigo) have been introduced since then. In particular, a robust version of the MoveIt! framework for collision-aware manipulation is only available for ROS version Hydro and later. Other improvements relevant for the Domestic Robot include the new *catkin* build system, a cleaner specification of the URDF robot model, and the interface to the recent versions of the Gazebo robot simulator. While ROS versions can be mixed to some extent, it is usually easiest to stay with a single version. As ROS Hydro is still actively developed and includes most of the recent improvements, but can be used on Ubuntu 12.04, the decision was taken to switch the Domestic Robot software to the Hydro version of ROS.

Either version can be started by setting the corresponding binary and library search paths and the `ROS_PACKAGE_PATH` environment variables. This is usually done as part of the users' `setup.bash` shell setup files.

PCAN kernel module Note that the PCAN kernel module required by MIRA is not part of the standard Ubuntu Linux kernels. After updating the Linux kernel, you will have to recompile the PCAN kernel module and generate the `/dev/pcan32` device file. This is documented in the MIRA installation guide.

5.2 Software Installation Paths

The different software packages are installed according to usual Linux (Debian/Fedora/Ubuntu) practice, where large non-standard software packages like ROS and MIRA are installed into the `/opt` path of the filesystem.

Again according to current practice, the user-developed ROS stacks and packages are installed into a local *ROS workspace* managed by the *rosws* and *rosinstall* tools, below the users' home directory. So far, most of the software development is carried out using the default *demo* user account. The default home directory in turn is `/home/demo`, but this is only used for the Kinova-specific post-installation stuff, namely the license files created by the Windows-installer from Kinova. The actual ROS workspace files including the Domestic Robot stack is installed into `/localhome/demo/ros_workspace`.

`/home/demo/Kinova/`	Kinova license stuff
`/localhome/demo/`	actual *demo* user home
`/localhome/demo/ros_workspace`	Robot-Era ROS software
`/localhome/demo/ros_workspace/domestic_robot`	Doro-Software
`/localhome/demo/ros_workspace/robot_common`	PEIS,MIRA bridges
`/opt/MIRA/`	MIRA framework software
`/opt/MIRA-commercial`	CogniDrive
`/opt/MIRA-licenses`	MIRA license files
`/opt/ros`	ROS software root
`/opt/ros/hydro`	Hydro installation
`/usr/local/*/`	PEIS installation

Figure 48: Software installation paths

5.3 MIRA and CogniDrive

The Domestic Robot comes with a pre-installed version of MIRA and CogniDrive, including the required license and configuration files. For localisation, it will be necessary to create and provide a map of the robot environment. To re-install or upgrade the MIRA and CogniDrive components, please follow the instructions from the MIRA homepage at *www.mira-project.org/MIRA-doc-devel/index.html*.

PCAN kernel module Note that the PCAN kernel module required by MIRA is not part of the standard Ubuntu Linux kernels. After updating the Linux kernel, you will have to recompile the PCAN kernel module and generate the `/dev/pcan32` device file. This is documented in the MIRA installation guide.

2D Nav Goal in rviz To set the doro navigation goal via *rviz*, you may have to change the default topic used for the command. Open the *too properties* window (typically on the top right panel in rviz), then select *2D Nav Goal*, and enter the topic name */move_base_simple/goal*.

5.4 PEIS

The robot comes with a pre-installed version of PEIS, using the default configuration with installs the files into the */usr/local* tree. To upgrade and re-install, follow the instruction from the PEIS homepage at *http://aass.oru.se/~peis/*. Note that building version G6 on a multi-user system can be a bit annoying, as the makefiles fail to set all file permissions. You may have to set file permissions from the shell in all affected subdirectories, e.g. *chmod -R go+rX /usr/local/include/peiskernel*. For *tupleview*, you may need to install the *libglade2-dev* libraries.

Once building is complete, simply run *tupleview* in a terminal to check that the system works, and to watch the current state of the PEIS tuple-space.

For performance reasons, it is recommended to use known PEIS owner-IDs whenever possible. The default ID of the configuration planner is 995. Further details about PEIS and the installation of the configuration planner can be found in deliverable D3.2 [54].

5.5 Kinova Jaco Software

The Kinova Jaco software is pre-installed on the Domestic Robot. When necessary, re-install from the USB-stick supplied by Kinova. Change to the sub-directory with the Ubuntu software, and follow the installation instructions from the *readme.txt* file. For example,

```
cd /media/kinova_usbstick/Release_2012-02-15/4 - API [4.0.5.7]/Ubuntu
```

If a first-time installation fails, the license files for the robot may not have been created. A workaround for this case is to install the software on an Windows PC, and later copy the created license files to the robot.

When compiling the software, you will need the *mono-devel* and *mono-gmcs* packages. Also install the latest version of *libusb-devel*.

Note that the Kinova stack is a required dependency for building the Domestic Robot ROS software. However, when just compiling the software for us in Gazebo simulation, the actual hardware interfaces are not required. Therefore, it is possible to just put *ROS_NOBUILD* files in the *jaco_node*, *jaco_test_node* and *jaco_api* nodes, and then running *rosmake* in the Domestic Robot stacks.

The two reference positions for the arm are:

- *retract* : -1.5717 -2.0940 1.0982 -1.5329 3.0482 2.7943
- *home* : -1.7891 -2.0163 0.7994 -0.8739 1.6888 -3.1031

5.6 ROS Hydro

On Ubuntu 12.04 LTS, simply install the required pre-built packages for *ros-hydro-xxx* via *apt-get* or a GUI-tool like *synaptic*. This installs all files below the *opt/ros/hydro* directory tree.

ROS Hydro is the currently supported version. The different sensor and actuator systems of the Domestic Robot, as well as the MoveIt-based motion planning are implemented and tested using ROS Hydro.

5.7 GStreamer Libraries

The necessary GStreamer libraries are installed during the catkin_make procedure, if necessary the catkin_make asks the user to install necessary dependencies.

The "'gstreamer_pipelines"' package includes the GStreamer-ROS adapter as well as the plugins for object detection and pose estimation.

The full list of dependencies can be installed with:

```
sudo apt-get install libtool automake cvs gettext \
bison flex libglib2.0-dev libxml2-dev liboil0.3-dev \
intltool libgtk2.0-dev libglade2-dev libgoocanvas-dev \
libx11-dev libxv-dev gtk-doc-tools libgstreamer0.10-dev \
libcv-dev libhighgui-dev libcvaux-dev libgsl0-dev \
libgstreamer-plugins-base0.10-dev yasm libgtk-3-dev \
liborc-0.4-dev gstreamer-tools mplayer \
gstreamer0.10-ffmpeg gstreamer0.10-plugins-bad \
gstreamer0.10-plugins-bad-multiverse \
gstreamer0.10-plugins-good gstreamer0.10-plugins-ugly libopencv-dev
```

While in the first version the Gstreamer libraries had to be compiled seperately from the ROS workspace, they are now included in the *catkin* workspace, are build within the *catkin_make* procedure and do not need to be installed in the systems' lib directories.

If plugins are not found, it may be the case that they are blacklisted, as dependencies have not been found once. In that case remove the .gstreamer0.10 folder in the home directory.

5.8 Robot-Era ROS stack

All software components for the Domestic Robot ROS software are developed and maintained in the Robot-ERA SVN repository. The default installation path for the user *demo* on the robot PC is */localhome/demo/ros_workspace*. Use *svn status* to check whether the current local copy is up-to-date, or use *svn update* to upgraed to the head revision of the repository.

Creating the runtime ROS node graph

```
rxgraph -o doro.dot
dot -T png -o doro-rxgraph.png doro.dot
dot -T pdf -o doro-rxgraph.pdf doro.dot
```

Dependencies For command-line parsing, the Cognidrive-ROS bridge module requires the *libtclap-dev* package.

Building for Gazebo without Kinova software See the section about the Kinova software above for instructions on how to setup the Domestic Robot software when just building for Gazebo simulation without the actual Kinova DLLs.

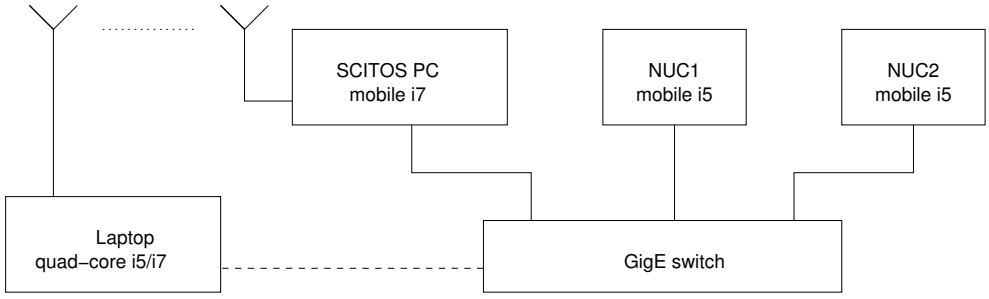


Figure 49: Block diagram of the Domestic Robot on-board computer network. Currently, the main Scitos-PC and two Intel NUC modules are installed. Developer PCs and Laptops can be connected via cable or via WLAN.

5.9 Network Setup and Wired/Wifi Bridge

Due to the installation of the additional Intel NUC computers, the network setup for the Domestic Robot needs to be configured. A software Ethernet bridge is used to connect the wireless network with the wired Ethernet network between the three on-board computers. See figure 49 for a block-diagram of the current design, which consists of the Scitos-G5 main computer, two Intel NUC D5250WYK computers, and a Gigabit Ethernet switch. External computers or laptops can connect to the on-board network either via cable or a WiFi connection to the Scitos-G5 computer.

The planned allocation of tasks to the different computers is as follows. The main Scitos-G5 controls the motors and laser-scanners, runs MIRA/Cognidrive and the PEIS AmI interface, and runs the roscore process and a few other ROS nodes required for the robot navigation. Other ROS nodes, in particular the Kinect point-cloud processing, the SIFT-based object detection and matching, and the MoveIt! manipulation planner are started on the Intel NUC computers. Additional ROS nodes can be launched on external PCs or laptops that connect to the robot either wirelessly via WiFi or via cabled connection.

The following description assumes that the Ubuntu 12.04 has been installed on all machines, that the required ROS Hydro stacks are installed, and the user-accounts have been set-up. If necessary, download and install the Linux Ethernet bridge software and the *iftop* utility to trace network load:

```
apt-get install bridge-utils
apt-get install iftop
```

The WLAN card on the ScitosPC needs to be configured for the Wifi network as usual, including address, netmask, gateway, and dns servers. Next, the cabled ethernet interfaces on the scitosPC and the NUCs must be configured. Using fixed IP addresses is probably the easiest solution. See figure 50 for an example `/etc/network/interfaces` file.

After changing the settings in `/etc/network/interfaces`, restart the Linux network system:

```
# /etc/network/interfaces example
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
  address 192.168.1.10
  netmask 255.255.255.0
  network 192.168.1.0
  gateway 192.168.1.1
  dns-nameservers 192.168.1.5
  dns-search example.com

auto wlan0
  ...

# Bridge between eth0 and wlan0
auto br0
iface br0 inet static
# optionally, set address netmask network gateway for the bridge
# address 192.168.1.11
# netmask 255.255.255.0
# network 192.168.1.0
# ...
  pre-up ip link set eth0 down
  pre-up ip link set wlan0 down
  pre-up brctl addbr br0
  pre-up brctl addif br0 eth0 wlan0
  pre-up ip addr flush dev eth0
  pre-up ip addr flush dev wlan0
  post-down ip link set eth0 down
  post-down ip link set wlan0 down
  post-down ip link set br0 down
  post-down brctl delif br0 eth0 wlan0
  post-down brctl delbr br0
```

Figure 50: Example network setup file with WiFi/wired Ethernet bridge.

```
sudo /etc/init.d/networking restart
```

For testing, the bridge settings can also be configured on the command-line. For example:

```
brctl addbr br0
brctl addif br0 eth0
brctl addif br0 wlan0
ifconfig br0 up
```

The network setup must ensure that all machines are known to each other, so that ROS nodes can connect to the main *roscore* process and exchange messages. Therefore, add the hostnames/IPs of the three on-board computers as well as all external computers that should connect via ROS to the `/etc/hosts` files on all computers. Please check that the `ROS_MASTER_URI` environment variable is set correctly to point to the main Scitos-G5 control computer:

```
export ROS_MASTER_URI=http://scitosPC:11311
```

5.10 Remote Roslaunch Setup

In order to use the ROS *roslaunch machine tags* for automatic remote execution of nodes, additional network configuration is needed. Because the default Ubuntu client installation only includes the ssh-client software, the first step is to install the openssh-server package on all machines that are supposed to run ROS nodes.

```
apt-get install openssh-server
```

When using a *machine tag* inside a *roslaunch* file, the ROS system creates a ssh-connection to the specified machine, initialises the path and configuration settings according to the specified *environment shell file*, and then tries to execute the specified ROS node with any given parameters. This in turn requires that the user that started the *roslaunch* file can log-in to the given machine and executes programs.

There are three different options, namely host-based authorisation, password-based authorisation, and public-key authorisation. All mechanisms are documented in detail on the *roslaunch* Wiki pages. The password-based mechanism is easy to set-up, but requires to keep both a valid username and the corresponding password in the plain text launch files. This can be acceptable on the robot itself if special user accounts are used, but is not recommended for normal user-accounts.

The better and secure way is to use the public-key authorisation together with an empty password. The required key files are created by the user, the private keys are kept private, and only the public keys are copied to the configuration files of the computers that should be accessed. A good tutorial (in German) is http://www.schlittermann.de/doc/ssh.

The first step is to read the required host-keys for all computers that should be able to connect via ssh. This includes the on-board control PC, the NUC computers, and any developer laptops or desktop machines. Enter the IP addresses for all machines to the `/etc/hosts` file, and enter the host keys to the `/etc/ssh/ssh_known_hosts` file:

```
/usr/bin/ssh-keyscan -t rsa scitosPC
/usr/bin/ssh-keyscan -t rsa nuc1
/usr/bin/ssh-keyscan -t rsa nuc2
/usr/bin/ssh-keyscan -t rsa laptop1
...
gedit /etc/hosts
gedit /etc/ssh/ssh_known_hosts
gedit /etc/ssh/sshd_config
```

Note that roslaunch uses paramiko to create ssh connections, and paramiko still does not support the ecdsa algorithm. Any existing ecdsa entries must be removed from the global and users `known_hosts` files, or remote roslaunch will fail.

Next, create the RSA private/public key pair for the user. Note that the users' home-directory must not writable by other users, and access to the users' `.ssh` directory must be restricted:

```
chmod go-w /home/user
chmod 700 /home/user/.ssh
ssh-keygen -t rsa
... empty passphrase
ssh-copy-id -i ~/.ssh/id_rsa.pub user@scitosPC
ssh-copy-id -i ~/.ssh/id_rsa.pub user@nuc1
ssh-copy-id -i ~/.ssh/id_rsa.pub user@nuc2
...
ssh user@remote-system
ssh -v user@remote-system    (for debugging)
```

Instead of using the *ssh-copy-id* tool, the public key can also be added directly to the users' `authorized_keys` file. Note that you may have to restart the *ssh* service on all machines in order to enable the changes:

```
slogin -l root@remote-system
service ssh restart
```

Next, the *env-loader* shell-script must be adapted to the ROS configuration to be used. The default script is `/opt/ros/hydro/env.sh`, but changes are needed to configure the Robot-Era workspace. See figure 51 for a typical example *env.sh* environment file.

Once the setup is configured, *roslaunch* is able to start ROS nodes on remote machines:

For host-based authentication, the following ssh entries might be useful:

```
/etc/ssh/sshd_conf
  IgnoreRhosts no
  IgnoreUserKnownHosts yes
  RhostsAuthentication no
```

```
#!/usr/bin/env sh
# example remote roslaunch environment loader file.
source /home/demo/catkin_ws/devel/setup.bash
export ROS_PACKAGE_PATH=/localhome/demo/hydro:${ROS_PACKAGE_PATH}
export ROS_PACKAGE_PATH=/localhome/demo/mypackage:${ROS_PACKAGE_PATH}
# Gazebo-1.9 + Hydro
export GAZEBO_MODEL_PATH=/home/demo/.gazebo/models
export ROSLAUNCH_SSH_UNKNOWN=1
export ROS_MASTER_URI=http://scitosPC:11311
exec "$@"
```

Figure 51: Example *env.sh* environment file for remote roslaunch.

```
# export ROSLAUNCH_SSH_UNKNOWN=1
# export ROS_MASTER_URI=http://scitosPC:11311

<!-- password-based authentication, default environment loader  -->
<machine name="nuc1"
     address="134.100.13.147"
     env-loader="/opt/ros/hydro/env.sh"
     default="true" user="hendrich" password="secret" >
</machine>

<!-- RSA public-key authentication, custom environment loader
     for global ROS Hydro and user-defined packages -->
<machine name="nuc2"
     address="134.100.13.148"
     env-loader="/home/demo/bin/catkin.env.sh"
     default="true" user="demo"  >
</machine>
...
node pkg="move_base" type="move_base" respawn="false"
        ns="coro"
        machine="nuc1"
        name="move_base" output="screen">
...
```

Figure 52: Example roslaunch *machine* definitions and usage.

```
  RhostsRSAAuthentication yes

/etc/ssh/ssh_config
  RhostsAuthentication no
  RhostsRSAAuthentication yes
```

5.11 Robot calibration

See section 3.1.8 for details.

6 Summary

This reports describes the hardware and software setup of the Robot-Era *Domestic Robot* (Doro) in preparation for the second experimental loop of the project.

Notable updates following analysis of the first experimental phase include a complete redesign of the outer appearance of the robot to improve user acceptance. The hardware itself was only slightly changed, with a new mount position of the Jaco arm and the main pillar resulting in a significantly larger object transportation tray and better observability. A gyroscope sensor improves robot localisation, especially on slippery or wet floor. Two compact on-board computers were added to increase the available processing power.

The overall software architecture developed during 2012 and 2013 was largely unchanged, but almost all software modules were updated to new versions and compatibility with ROS Hydro. The PEIS-ROS interface was redesigned completely and is now using PEIS metatuples for easier reconfiguration of the services. Several robot skills were added for the second experimental loop, and most skills were tuned and updated for better robustness.

The last chapter of this handbook list the main software modules required for robot operation and includes installation instructions and tips for troubleshooting.

References

[1] E. Olson, *AprilTag: A robust and flexible visual fiducial system*, Proc. ICRA 2011, 3400–3407, 2011.

[2] A. Richardson, J. Strom, E. Olson, *AprilCal: Assisted and repeatable camera calibration*, Proc. IROS 2013.

[3] Groothuis, Stramigioli, Carloni: *Lending a helping hand: toward novel assistive robotic arms*, Robotics and Automation Magazine, March 2013,

[4] Xu and Cakmak, *Enhanced Robotic Cleaning with a low-cost Tool Attachment*, Proc. IROS 2014.

[5] Reflexxes Motion Library, http://www.reflexxes.com/

[6] T. Kroeger, *On-Line Trajectory Generation in Robotic Systems*, Springer Tracts in Advanced Robotics, Vol. 58, Springer Verlag, Berlin/Heidelberg, Germany, January 2010.

[7] N. Hendrich, H. Bistry, B. Adler, J. Zhang, *User-driven software design for an elderly care service robot*, Proc. 8th Int. Conference on Pervasive Computing Technologies for Healthcare, Pervasive Health 2014, Oldenburg, Germany

[8] N. Hendrich, H. Bistry, J. Liebrecht, J. Zhang, *Multi-modal clues for efficient human-robot object handover: a case study with elderly users*, Workshop on New frontiers of service robotics for the elderly, RO-MAN 2014, Edinburgh, UK

[9] N. Hendrich, H. Bistry, J. Liebrecht, J. Zhang, *Natural Robot-Human Handover Combining Force and Tactile Sensors*, Workshop on Assistance and Service Robotics in a Human Environment, IROS 2014, Chicago, USA

[10] N. Hendrich, H. Bistry, J. Zhang, *PEIS, MIRA, and ROS: Three frameworks, one service robot - a tale of integration*, ROBIO 2014, Bali, Indonesia, accepted

[11] H. Bistry, *Intelligente Kamerasysteme im Anwendungsfeld mobiler Service-Roboter*, (Smart Camera Systems in the Application Field of Mobile Service Robots), PhD-thesis, University of Hamburg, 2014

[12] S. Starke, *Fast and Robust People Detection on RGB-D Data*, MSc-project, University of Hamburg, 2014

[13] C. Ferrari and J. Canny; *Planning Optimal Grasps* In Proceedings of the IEEE Int. Conference on Robotics and Automation, pages 2290–2295, Nice, France, 1992.

[14] K.B. Shimoga, *Robot grasp synthesis algorithms: a survey*, International Journal of Robotics Research, vol.15, 230–266, 1996

[15] A. Bicchi, V. Kumar, *Robotic grasping and contact: A review*, IEEE International Conference of Robotics and Automation, 348–353, 2000

[16] M.R. Cutkosky, *On grasp choice, grasp models, and the design of hands for manufacturing tasks*, IEEE Transactions on Robotics and Automation, vol.5, 269–279, 1989

[17] H. Kjellström, J. Romero, D. Kragic, *Visual Recognition of Grasps for Human-to-Robot Mapping*, Proc. 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, 3192–3197, 2008

[18] J. Aleotti, S. Caselli, *Grasp Recognition in Virtual Reality for Robot Pregrasp Planning by Demonstration*, IEEE Intl. Conference on Robotics and Automation, 2801-2806, 2006

[19] R.S. Dahiya, G. Metta, M. Valle, G. Sandini, *Tactile Sensing—From Humans to Humanoids*, IEEE Transactions on Robotics, vol. 26, no.1, 1–20, 2010

[20] Ch. Borst, M. Fischer and G. Hirzinger; *Grasp Planning: How to Choose a Suitable Task Wrench Space*. Proceedings of the IEEE Intl. Conference on Robotics and Automation (ICRA), New Orleans, USA, 2004.

[21] A.T. Miller, S. Knoop, H.I. Christensen, P.K. Allen, *Automatic grasp planning using shape primitives*, IEEE International Conference on Robotics and Automation, 1824–1829, 2003

[22] J. Kim, J. Park, Y. Hwang, M. Lee; *Advanced Grasp Planning for Handover Operation Between Human and Robot: Three Handover Methods in Esteem Etiquettes Using Dual Arms and Hands of Home Service Robot*, 2nd Intl. Conference on Autonomous Robots and Agents, 2004

[23] M. Turk and A. Pentland, *Eigenfaces for recognition*. Journal of Cognitive Neuroscience 3 (1): 71–86, 1991.

[24] N. Koenig, and A. Howard, *Design and use paradigms for gazebo, an open-source multi-robot simulator*, Proc. IROS 2004, 2149–2154

[25] Bullet Physics Library, http://bulletphysics.org/, 2006

[26] Hao Dang, Jonathan Weisz, and Peter K. Allen, *Blind Grasping: Stable Robotic Grasping Using Tactile Feedback and Hand Kinematics*, Proc. ICRA-2011, 2011

[27] Rosen Diankov, *Openrave: A planning architecture for autonomous robotics*, Robotics Institute, Pittsburgh, PA, Tech. Rep., (July), 2008.

[28] A. Morales, T. Asfour, P. Azad, S. Knoop, and R. Dillmann. Integrated grasp planning and visual object localisation for a humanoid robot with five-fingered hands. In *Proc. of IROS 2006*, 2006.

[29] Z. Li R. Murray and S. Sastry. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, Boca Raton, FL, 1994.

[30] Marc Peter Deisenroth, Carl Edward Rasmussen, and Dieter Fox. Learning to control a low-cost manipulator using data-efficient reinforcement learning. In *Robotics: Science and Systems*, 2011.

[31] Abhinav Gupta, Aniruddha Kembhavi, and Larry Davis. Observing human-object interactions: Using spatial and funcitional compatibility for recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(10):1775–1789, 2009.

[32] Anis Sahbani, Jean-Philippe Saut, and Veronique Perdereau. An efficient algorithm for dexterous manipulation planning. In *IEEE International Multi-Conference on Systems, Signals and Devices*, 2007.

[33] D. Fox, W. Burgard, and S. Thrun, *The Dynamic Window Approach to Collision Avoidance*, 1997

[34] O. Brock and O. Khatib, *High-speed Navigation using the Global Dynamic Window Approach*, Proc. ICRA-99, 341–346, 1999.

[35] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berge, R. Wheeler, and A. Ng, *ROS: an open-source Robot Operating System*, ICRA Workshop on Open Source Software, 2009
pub1.willowgarage.com/~konolige/cs225B/docs/quigley-icra2009-ros.pdf

[36] K. Hsiao, S. Chitta, M. ciocarlie, E. G. Jones, *Contact-Reactive Grasping of Objects with Partial Shape Information*,
www.willowgarage.com/sites/default/files/contactreactive.pdf

[37] N. Siegwart, I.R. Nourbakhsh, *Introduction to Autonomous Mobile Robots*, MIT Press, 2004

[38] J. W. Capille, *Kinematic and Experimental Evaluation of Commercial Wheelchair-Mounted Robotic Arms*, Thesis, University of South Florida, 2010

[39] Metralabs Gmbh, SCITOS-G5 Service Robot, www.metralabs.com

[40] Kinova Inc., Jaco robot arm, www.kinova.ca

[41] Kinova Inc., Inverse kinematics for the Jaco arm, private communication.

[42] Shadow Robot Dexterous Hand, www.shadowrobot.com

[43] Metralabs Gmbh, *Cognidrive*, xxx, 2012.

[44] Taymans, W. and Baker, S. and Wingo, A. and Bultje, R. and Kost, S. *GStreamer Application Development Manual (0.10.21.3)*, October 2008,
gstreamer.freedesktop.org

[45] Russell Smith, *Open Dynamics Engine*, 2005, http://www.ode.org

[46] AstroMobile project, www.echord.info/wikis/website/astromobile

[47] Robot-Era project, deliverable D2.1, *First report on the robotic services analysis with respect to elderly users*, www.robot-era.eu, 2012.

[48] Robot-Era project, deliverable D2.3, *First report on the usability and acceptability requirements*, www.robot-era.eu, 2012.

[49] Robot-Era project, deliverable D2.4, *First report on the S/T requirements of the Robot-Era components*, www.robot-era.eu, 2012.

[50] Robot-Era project, deliverable D2.6, *Second report on the structures of Robot-Era services*, www.robot-era.eu, 2014.

[51] Robot-Era project, deliverable D2.7, *Second report on the usability and acceptability requirements*, www.robot-era.eu, 2014.

[52] Robot-Era project, deliverable D2.8, *Second report on the S/T requirements of the Robot-Era components*, www.robot-era.eu, 2014.

[53] Robot-Era project, deliverable D3.1, *Report on the implementation of the AmI infrastructure modules*, www.robot-era.eu, 2012

[54] Robot-Era project, deliverable D3.2, *AmI infrastructure and reasoning services, first version*, www.robot-era.eu, 2013

[55] Robot-Era project, deliverable D3.3, *Lessons learnt from the integration of service robots with the AmI infrastructure*, www.robot-era.eu, 2014

[56] Robot-Era project, deliverable D3.4, *AmI infrastructure and reasoning services, second version*, www.robot-era.eu, 2014

[57] Robot-Era project, deliverable D4.1, *Domestic Robot Specification*, www.robot-era.eu, 2012.

[58] Robot-Era project, deliverable D4.2, *Domestic Robot Handbook*, First Domestic robotic platform prototype for the first experimental loop, www.robot-era.eu, 2013.

[59] Robot-Era project, deliverable D5.2, *First Condominium robotic platform prototype for the first experimental loop*, www.robot-era.eu, 2013.

[60] Robot-Era project, deliverable D5.3, *Final Condominium robotic platform prototype for the second experimental loop*, www.robot-era.eu, 2014.

[61] Robot-Era project, deliverable D6.3, *Integration and implementation of the outdoor robotic platform prototype for the second experimental loop*, www.robot-era.eu, 2014.

[62] Robot-Era project, deliverable D8.2, *Report on results obtained in the first cycle of the Robot-Era experimentation*, www.robot-era.eu, 2014.

[63] Robot-Era project, deliverable D8.3, *Experimental protocol for the second cycle of experiments of Robot-Era services*, www.robot-era.eu, 2014.