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

D4.2

Domestic Robot Handbook

Due date of deliverable: 31/03/2013
Actual submission date: [31/03/2013]

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

Organisation name of lead contractor for this deliverable: UHAM
Deliverable author: N. Hendrich, F. Cavallo, M. Di-Rocco, C. Martin

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	Version	Change	Author
2013.03.19	0.1	created	N. Hendrich
2013.03.20	0.11	ROS/Gazebo/services	N. Hendrich
2013.03.22	0.12	software overview	N. Hendrich
2013.04.08	0.2	manipulation, robot-bringup	N. Hendrich
2013.04.09	0.x	ROS-Peis	N. Hendrich
2013.04.xx	1.0	revision and final version	xxx

Future Updates

This handbook reports the current state of the Domestic Robot hardware, software, and service architecture. It is expected that the software will be undergo significant developments and that many bug-fixes and improvements will result based on experience gained during the upcoming first experiment phase of the project.

Updated releases of the handbook will be uploaded to the project repository as the software and services evolve. An updated version of this handbook may be submitted after the experimental loop has completed and the lessons learned have been integrated into the software.

Contents

List of Figures

1 Overview

This handbook summarizes the hardware, software, and service architecture of the *Domestic Robot*, short *doro*, of project Robot-Era. See Fig. 1 on page 3 for a photo of the current prototype. The hardware design of the robot has been completed and the overall software architecture has been decided on. Therefore, most of the ongoing project work concentrates on the implementation and developement of the robot software, and in particular on the end-user driven *services* to be provided by the robot. As such, the intended audience of the handbook are the project partners and the software developers for 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 efficency 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 implemention of the robot services, the project also monitors the feasibility, scientific/technical effectiveness and the social and legal plausibiliy 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* center 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.

While a separate report will be provided by the project to describe its *condominium robot* [40], it should be noted that the hardware platforms of the condominium and domestic robots are very similar, and both robots will be equipped with the same sensor head including cameras and depth-camera. Therefore, large parts of the software architecture can and will be shared among the two types of robots, and the corresponding sections of this handbook also apply to the condominium robot. This includes the software overview, the navigation and localization algorithms, the object and environment perception, and all services that are not related to manipulation.

As explained 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* (*AmiI*) 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 [37] and D3.2 [38].

Outline

This report is part of the public project deliverable D4.2, 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 summarizes 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 2.2 describes the SCITOS-G5 mobile differential-drive platform selected as the mobile base of the *domestic robot*, while section 2.3 summarizes key data of the Kinova Jaco 6-DOF arm and integrated gripper selected as the manipulator on the robot. Section 2.4 describes the moveable (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 2.5 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 ongoing integration into the ROS manipulation stack and MoveIt! motion planning framework is decribed in 3.6. The complete ROS/Gazebo simulation model of the robot is explained in 3.8. Finally, section 3.9 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 summarizes 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 setup of the major software components for the domestic robot.

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

Figure 1: The Domestic dobot (front view) combines the SCITOS-G5 differential-drive platform with the Kinova Jaco 6-DOF arm and 3-DOF gripper and a moveable sensor-head. Sensors include one front and rear laser-scanner plus a ring of 24 sonar sensors for navigation and obstacle avoidance, the Asus XtionPro RGB-D camera and two high-res cameras equipped with different lenses. Voice input is possible via the microphones built into the XtionPro device. The handle on the right carries the iPad tablet-computer that provides a touch-screen interface and additional sensors. The colored area on the robot base-plate serves as the tray for carrying objects. On this picture, the robot is still shown without the cover.

2 Hardware

This chapter documents the hardware components of the domestic robot. See Fig. 1 on page 3 for a photo of the completed robot (but still without cover). Please refer to project report D4.1 [39] 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 localization 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 lifes, 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 characterize the properties required for the robot [36]. Fig 3 highlights some of the aspects 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 *astehtics* and *friendliness*. In short, the robust must be capable of the tasks expected by the users, but must be non-obstrusive and integrate into the living environments. Please refer to project report D2.4 [36] for the in-depth analysis.

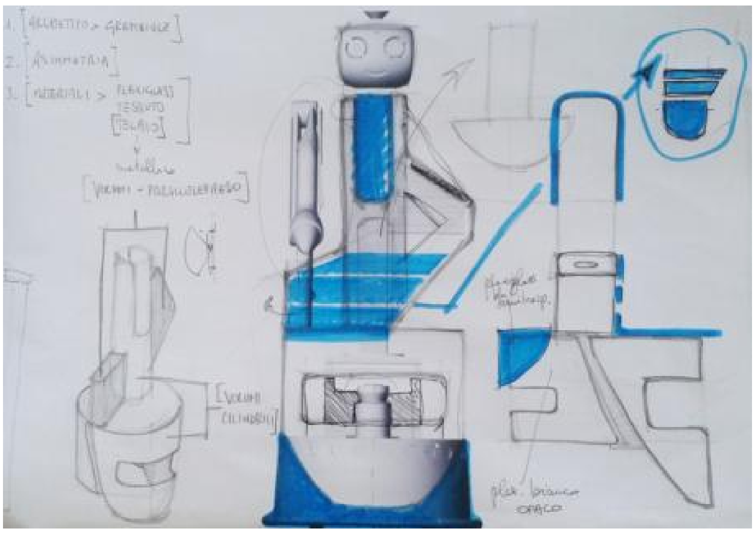


Figure 2: Designer concept of the domestic robot.

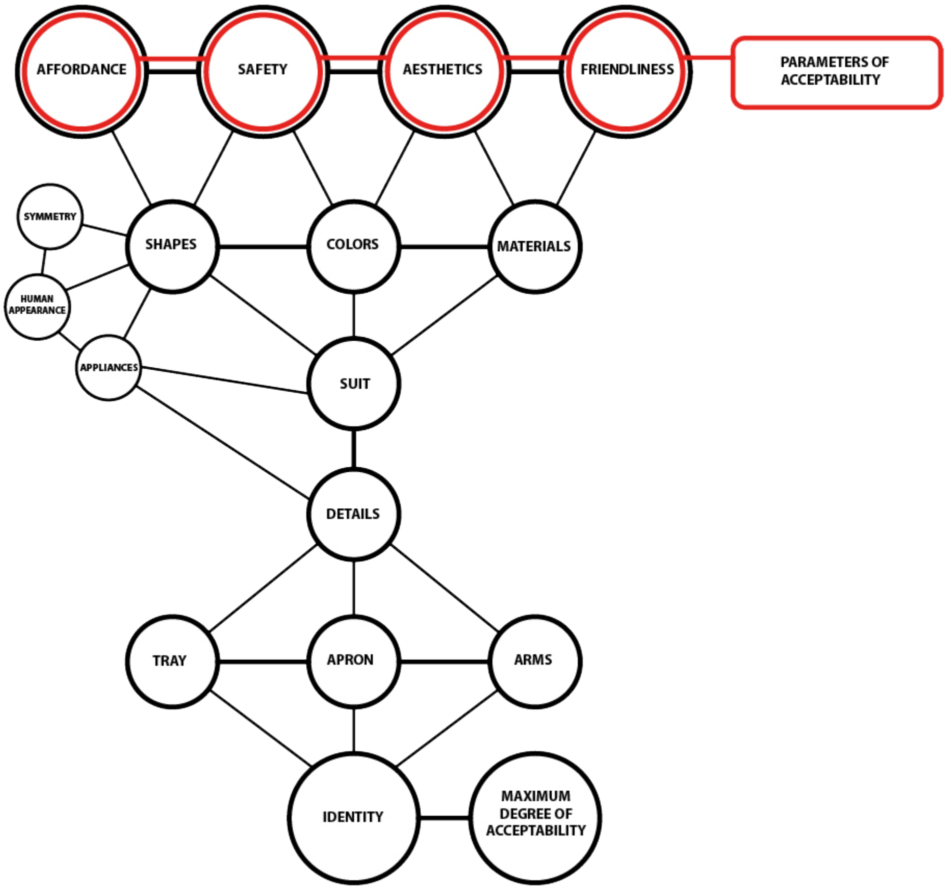


Figure 3: Aspects identified as important for the acceptability and user-friendliness of a service robot.

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 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 current design of the robot easily allows us to install a second arm should this be required. 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.

See Fig. 4 and 5 for another add-on of the domestic robot that was identified during the user-studies. In the final version, the robot will be equipped with a fixed horizontal handle. This provides additional support for the elderly user when trying to get up from a chair or the bed, and also for full walking support. Experiments with a similar robot in the ECHORD Astromobile project [33] have proven that the SCITOS-G5 platform is both robust and stable enough for this task.

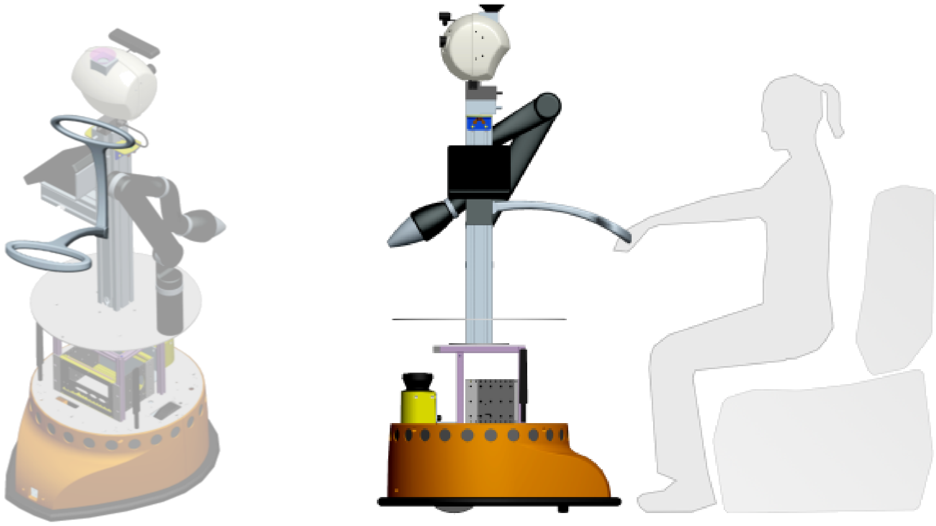
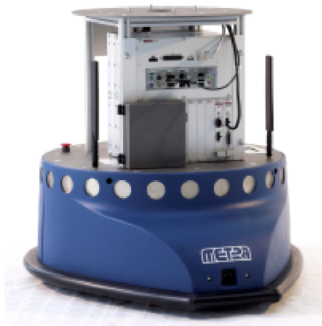


Figure 4: Sketches of the domestic robot with a fixed handle. With this add-on, the robot provides indoor walking-support to users with limited mobility. Due to the weight of the batteries and the low center of gravity, the SCITOS-G5 robot is surprisingly stable despite its narrow wheel-base, resulting in reliable support for the user.

Figure 5: The domestic robot is equipped with an horizontal handle, located on the back of the robot at a fixed height. Two buttons are located on the left and right side of the handle so pushing them the user can drive the robot forward, left and right. In this way the domestic robot provides indoor walking-support to users with limited mobility.

2.2 SCITOS G5 platform

2.2.1 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
- bumper ring with mechanical emergency stop
- 126x64 pixel graphic display with selection knob
- 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

Computing:

- Industrial embedded PC, Intel QM57 Express chipset
- 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
- Linux Fedora 14 (pre-installed and configured)
- MIRA and CogniDrive for navigation and localization

2.2.2 Sick S300 safety laser-scanner

- scanning range 270 deg
- angular resolution 0.5 deg
- distance measuring range up to 30 m
- support for user-defined safety zones
- Linux driver

2.2.3 Hokuyo URG-04LX laser-scanner

- scanning range 270 deg
- angular resolution 0.35 deg
- distance measuring range from 0.2 m to 6 m
- USB connection, Linux driver

2.3 Kinova Jaco manipulator

Figure 6: 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

2.3.1 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

2.4 Sensor head

2.4.1 Asus XtionPro

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

2.4.2 Camera DFK 31BF03

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

2.4.3 Camera DFK 21BF04

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

2.4.4 Directed Perception D46 pan-tilt unit

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

2.5 Human-Robot interface

2.5.1 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

2.5.2 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

2.5.3 Handle for Walking Support

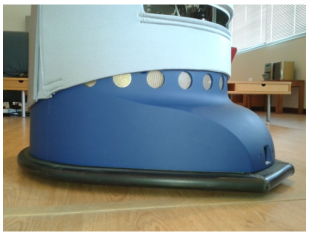
The horizontal handle is located on the back of the robot at a fixed height. The handle is equipped with two buttons on the left and right side of the handle so pushing them the user can drive the robot forward, left and right.

2.6 Safety Features

2.6.1 Emergency-stop switches

A red switches button is located on right side of the domestic robot, under the tablet location. The switches currently only stop the SCITOS platform, not the PTU nor the Jaco.

2.6.2 Bumper ring



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

2.6.3 Safety laser scanner

The SICK S300 Safety Laser Range Finder (described in 2.2.2) also provides safety features.

3 Software

This chapter summarizes 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* [22] 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 summarizes the URDF robot model created for the domestic robot.

Next, section 3.2 explains the main software components for localization 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 ROS manipulation-stack and the upcoming improved MoveIt! framework is sketched in sections 3.6 and 3.7. The next section 3.8 describes the simulation model of the domestic robot created for the Gazebo [11] simulator.

Last but not least, section 3.9 explains the interface layer between the PEIS ambient intelligence network and the domestic robot software. The interface is based on dedicated *tuplehandler* ROS nodes 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* [39], 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 localization. See Fig. 7 for a block diagram that highlights the main components of the software.

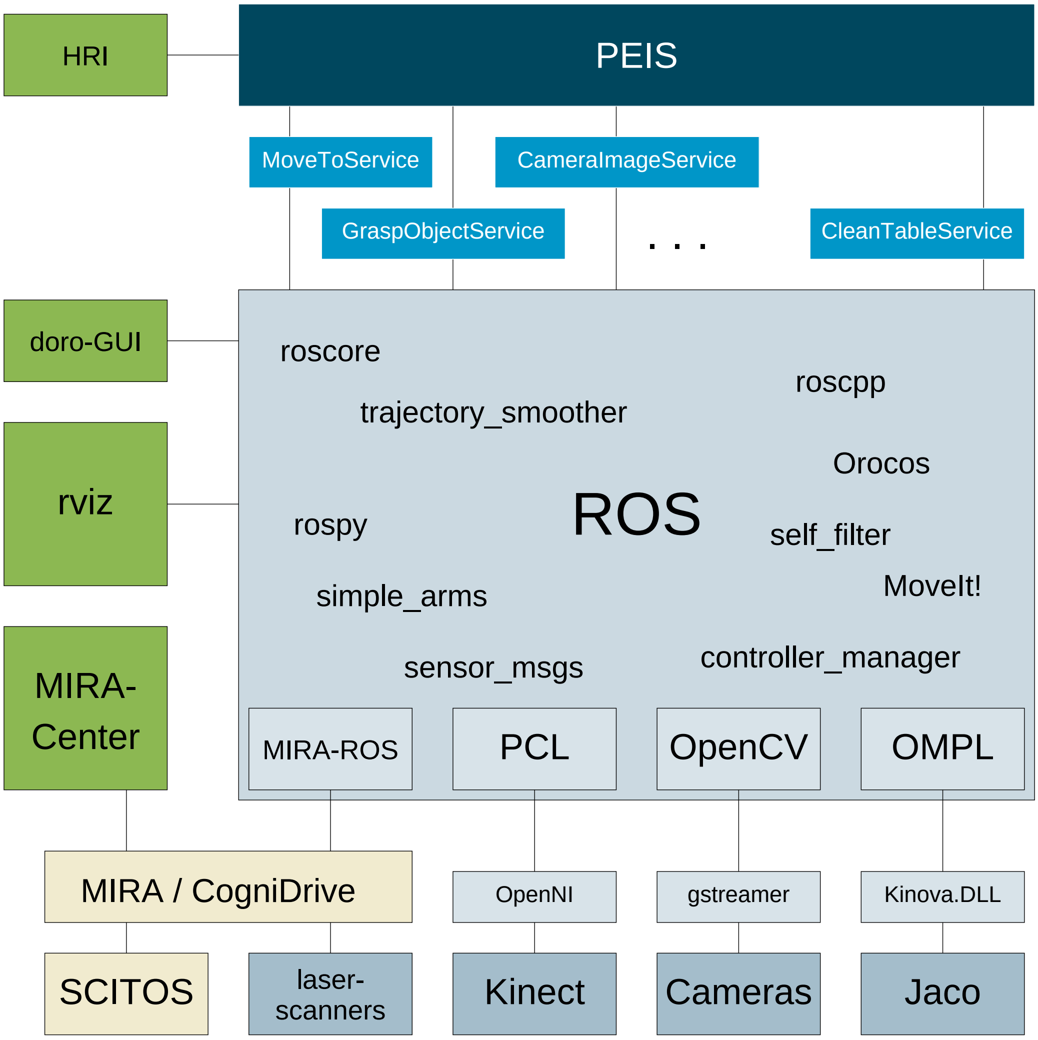


Figure 7: Software Architecture of the domestic robot with the five main layers and key modules. The topmost layer consists of the user-interface and the PEIS infrastructure, which provides access to the ambient sensor network and other robots. It also includes the multi-robot planner.
A collection of *services* implemented as PEIS-ROS *TupleHandlers* forms the software interface between PEIS and the domestic robot; see chapter 4 for a list of the planned 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, a *TupleHandler* 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 runtime is used to wrap the Windows DLLs required for the Kinova Jaco. The MIRA/Cognidrive [30] software manages navigation and localiziation 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 laserscanners, 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 labeled *HRI* (human-robot-interface) summarizes 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 will be described in detail in a separate technical report [38].

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 favorite middleware solutions for robot integration [22]. 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 manipulation stack integrates a large set of open-source software for constraints- and collision-aware motion-planning and grasping, with optional support for 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 capabable 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 centralized single control level or monotlithic 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 RosJava interface.

Roscore and parameter server In a typical ROS system, there are only two centralized 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 lookup-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* loookup 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 canceled by the client at any time. Also, the server can provide a periodic *feedback* to indiciate progress to the client before the original service goal has been reached.

Stacks and packages Apart from the core runtime functionality, ROS also suggests a specific file-sytem structure for its components, organized 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. 8. 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.

`robot-era`	root of the project software 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`	documentation
`    doro_msgs`	robot specific messages
`    doro_teleop`	joystick/tele-operation tools
`    doro_peis`	PEIS services for the domestic robot
`    ...`	
`  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`	PEIS services for navigation
`    ...`	
`  kinova_jaco`	Kinova Jaco arm ROS stack
`    jaco_api`	Kinova API and CSharp-wrappe
`    jaco_description`	Jaco arm model and launch files
`    jaco_node`	Jaco joint-level control node
`    ...`	
`  peis`	PEIS stack

Figure 8: Robot-Era software repository structure with ROS stacks and packages.

Build system To manage the compilation of hundreds of software packages, ROS provides its own build system, with the *rosmake* tool and its command-line parameters like *–pre-clean* or *ROS_NOBUILD* hints as the main entry point. 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, 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. At the moment, the Robot-Era domestic robot software has been tested with versions *Fuerte* and *Groovy* of ROS.

Real-time robot control Based on the control architecture designed for the PR2 service-robot, ROS also includes support for real-time robot control. In particular, 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.

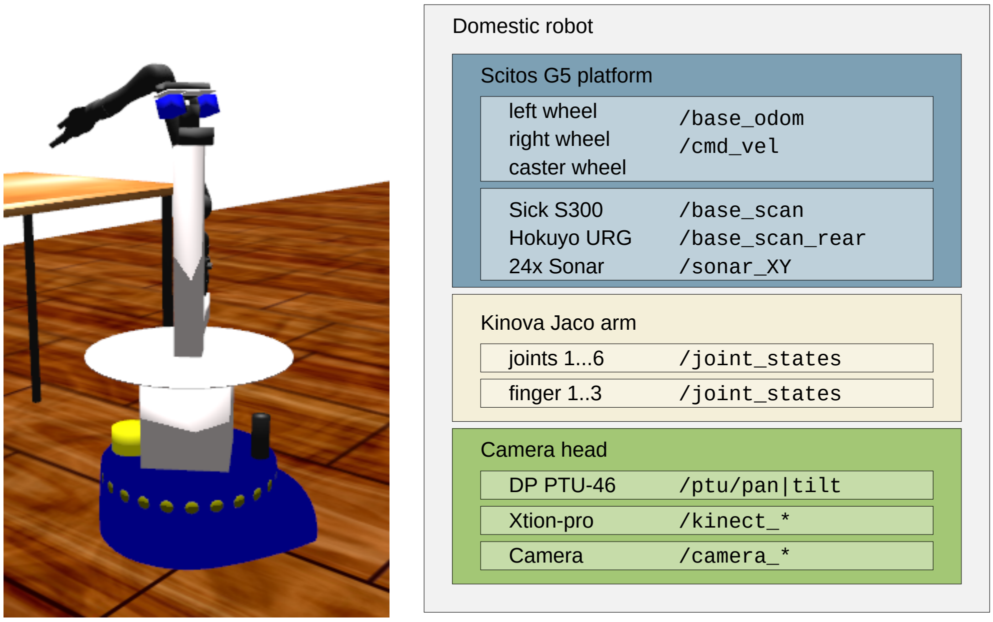


Figure 9: The basic structure of the URDF robot description for the domestic robot, consisting of the SCITOS-G5 mobile platform with the wheels and navigation sensors, the Kinova Jaco arm, and the camera head.

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 9. 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; this will be replaced with actual data for the specific cameras following the whole-robot calibration.

Note that neither the tray nor the cover nor the iPad holder are yet included in the model shown in the figure, as the final geometry is still not finished. However, the missing components are static and can be added to the robot description very easily; also, they don't impact the use or simulation of the domestic robot.

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 halfways 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. 10 for a screenshot of the robot in the *rviz* visualization tool, with the tf coordinate-system markers enabled and overlayed 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. 12 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 visualized 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*.

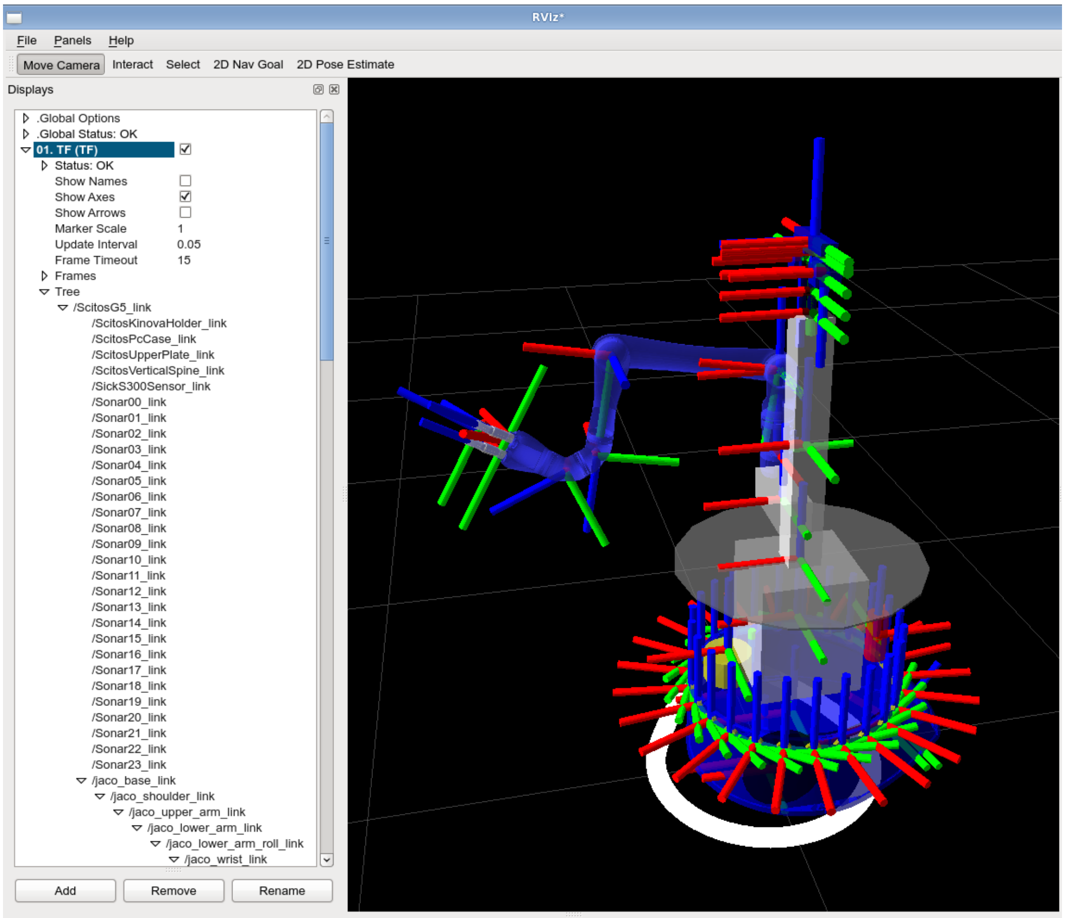


Figure 10: Coordinate frames of the domestic robot. The figure shows the rviz visualization of all tf coordinate frames used in the domestic robot URDF model (red: x, green: y, blue: z). The main coordinate system of the platform has positive x towards the front, y to the left, and z upwards. In the 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.

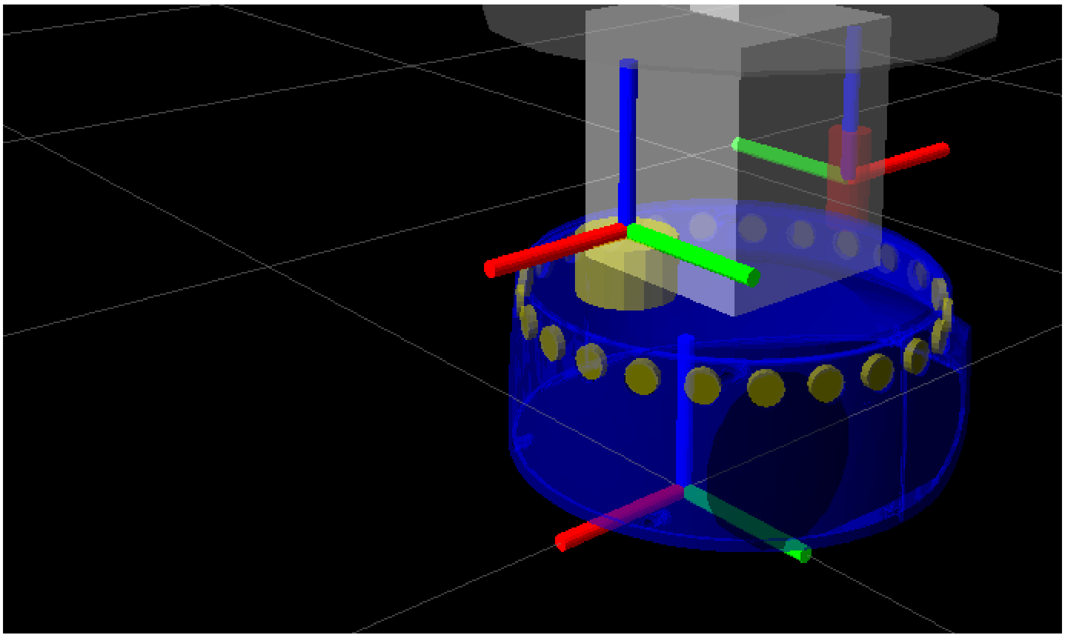


Figure 11: Base coordinate frames of the domestic robot. The figure shows the base coordinate system *base_link*, halfways between the driving wheels, and the two coordinate systems for the front and rear laserscanners.

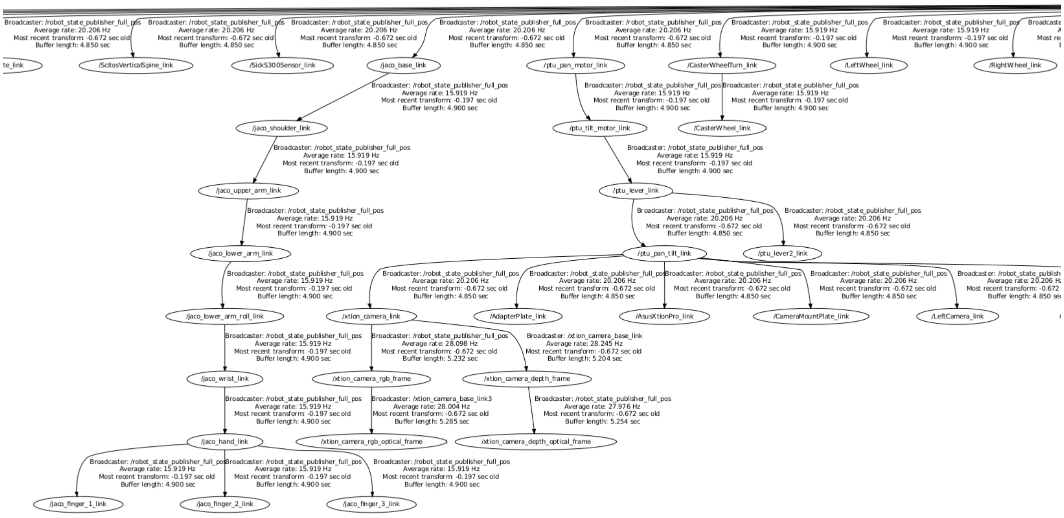


Figure 12: Coordinate frame graph of the domestic robot. The figure shows a zoomed-in region of the total graph with the most relevant tf coordinate frames of the robot, including the wheels, the Jaco arm with hand and fingers, and the sensor head with pan-tilt unit and the cameras.

3.1.4 Launching the domestic robot

To simplify operation of the domestic robot, the startup of the different required device drivers and ROS nodes is managed by a set of ROS launch files. At the moment, the startup is not (yet) fully automatic, but consists of several steps under control of the user. 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_description domestic_manipulation.launch
shell-4> 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 localization 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 Xtion-Pro RGB-D camera and the firewire cameras.
- starts several utility nodes, including the *doro_joint_state_merger* 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 47 for details) before running the launch script. If initialization 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 initialization sequence. In particular, the PTU node is known to crash sometimes, when the PTU initialization 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 visualize 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
/leftcamera/image_proc
/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 `/cognidrive_ros` provides the platform control and navigation, `/jaco_node` provides joint-level control of the arm, and `/ptu` controls the PTU. The laser-scanner data and localization is published by `/cognidrive_ros`, while `/leftcamera` and `/xtion_camera/*` are the controller nodes for the Firewire RGB- and XtionPro 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
```

```
/leftcamera/camera_info
/leftcamera/image_color
...
/leftcamera/image_mono
/leftcamera/image_raw
/leftcamera/image_rect_color
/leftcamera/image_rect_color/compressed
/leftcamera/image_rect_color/compressed/parameter_descriptions
...
/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
...
```

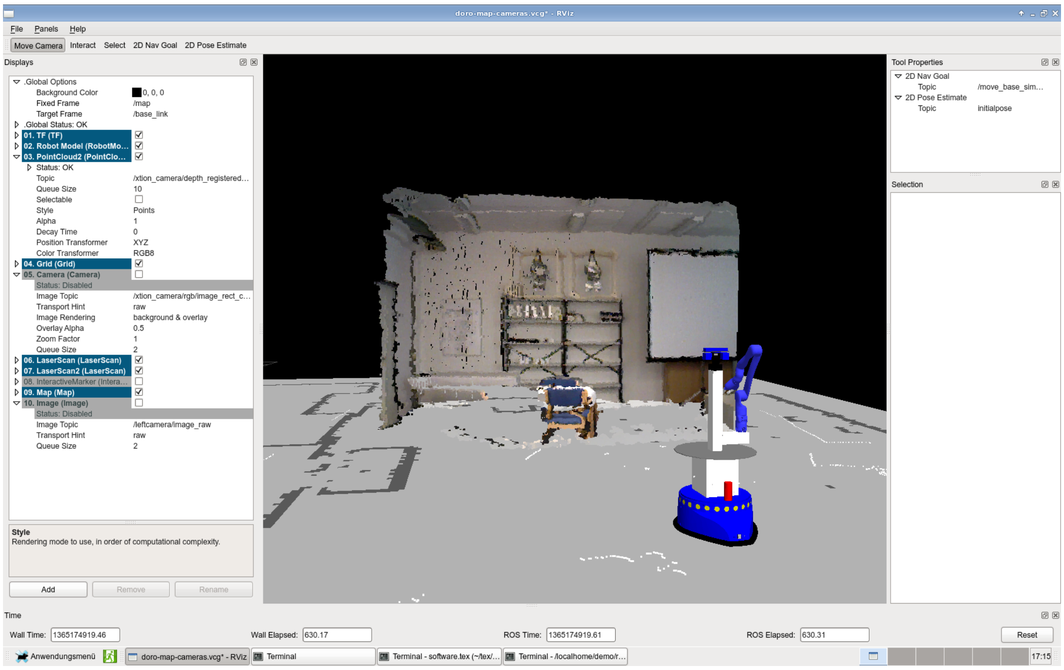


Figure 13: An example rviz configuration for control of the domestic robot. The *robot_description* and *tf* topics should be selected, with *map* as the fixed-frame and *base_link* as the root link of the robot. The 3D view in the center shows the robot localized on the map, with overlayed laser-scans and colored point-cloud from the XtionPro on *xtion_camerea/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* visualization 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 visualization of the incoming sensor-data overlayed 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. 13 for a screenshot of rviz using this configuration.

Alternatively, the MIRA-Center software provides an easy-to-use interface for the navigation and localization tasks on both the Condominium and the domestic robot (see Fig. 15 on page 35 for a screenshot).

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 *realfull.launch* file (currently necessary on an external system with a screen, as an RViz instance is launched),

```
roslaunch doro_moveit_new realfull.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 stack 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 *fulldemo_peis.launch* file. This assumes that the robot bringup and manipulation launch files have been started and all services are running.

```
roslaunch fulldemo_peis fulldemo_peis.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
192.168.0.37 numbercruncher
192.168.0.44 laptop
```

Once the network and hostnames have been set-up on all participating computers, the *roscore* process is started on one of the machines. 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 ...
rosrun ...
```

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.

The usage of the Sixaxis joystick is illustrated in Fig 14. 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.

Figure 14: 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.

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

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 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 localization 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 modeling 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 capabilites under autonomous control. Experience gained throughout the project experimental phases will show whether any additional modeling 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.

Full robot calibration including cameras and arm and pan-tilt-unit is a future issue.

3.2 Robot localization and navigation

This section summarizes the localization, 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 localization and path-planning methods and a static map of the environment can be used. See [24] for a review of the relevant basic algorithms.

As explained in the earlier project report D4.1 *Domestic robot platform specification* [39], 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. 7 on page 13 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 laser-scanners. Robust localization, 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 [39] (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 visualization 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 visualization, 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 [39], 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 localization capabilities of both the domestic and condominium robots. See chapter 5 of the previous report D4.1 [39] 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 *subtasks*, 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 specialized for certain tasks like following a person, driving at a certain speed or direction, etc. The objectives are realized 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's motor controllers. Some objectives require additional information from other navigational modules such as localization and mapping algorithms or modules for user interaction like person trackers.

Each sub-task can be parametrized 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 robot's 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* $\pm 0.5\,m$ *and turn to the orientation of 70° with an accuracy of ±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 [20] 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 robot's 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 [21], 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 forbidded 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 parameterized to follow a person while taking privacy into account by keeping a given minimum distance between the robot and the useers. 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 16. 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 localization.

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 localization 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.

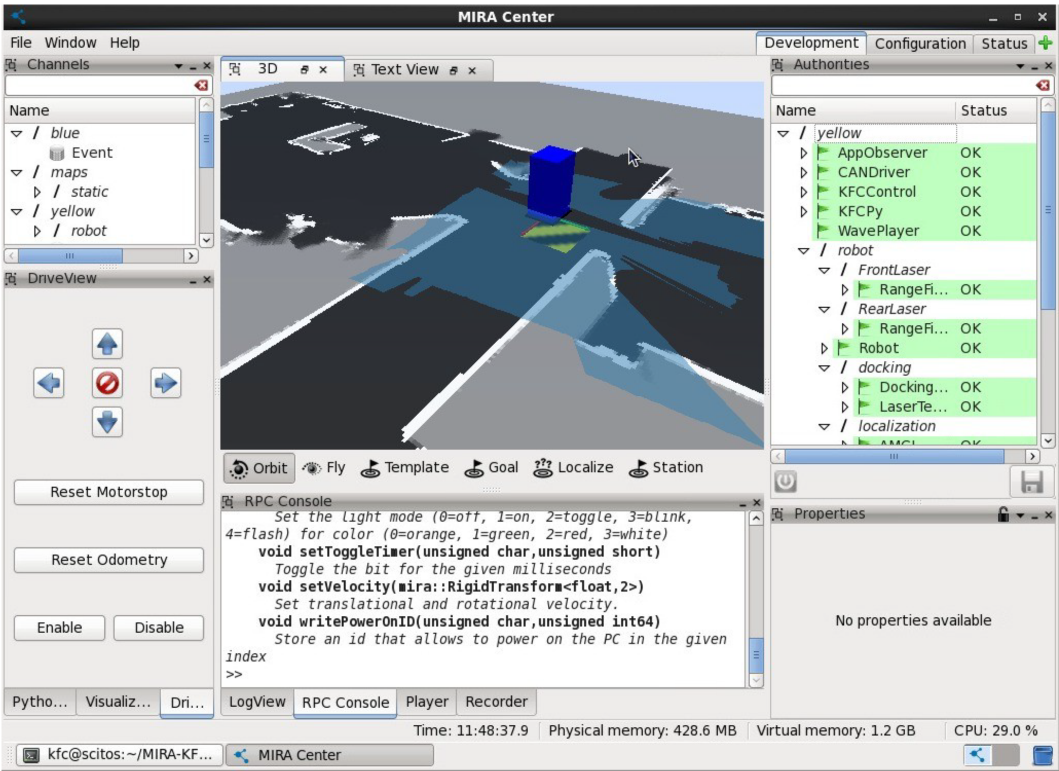


Figure 15: The user-interface of the MIRA-Center software with visualization of the robot and laser-scanner data inside the map.

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 data format used by MIRA, and Cognidrive is then able to calculate the robot localization base on the simulated laserscans.

To summarize, 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

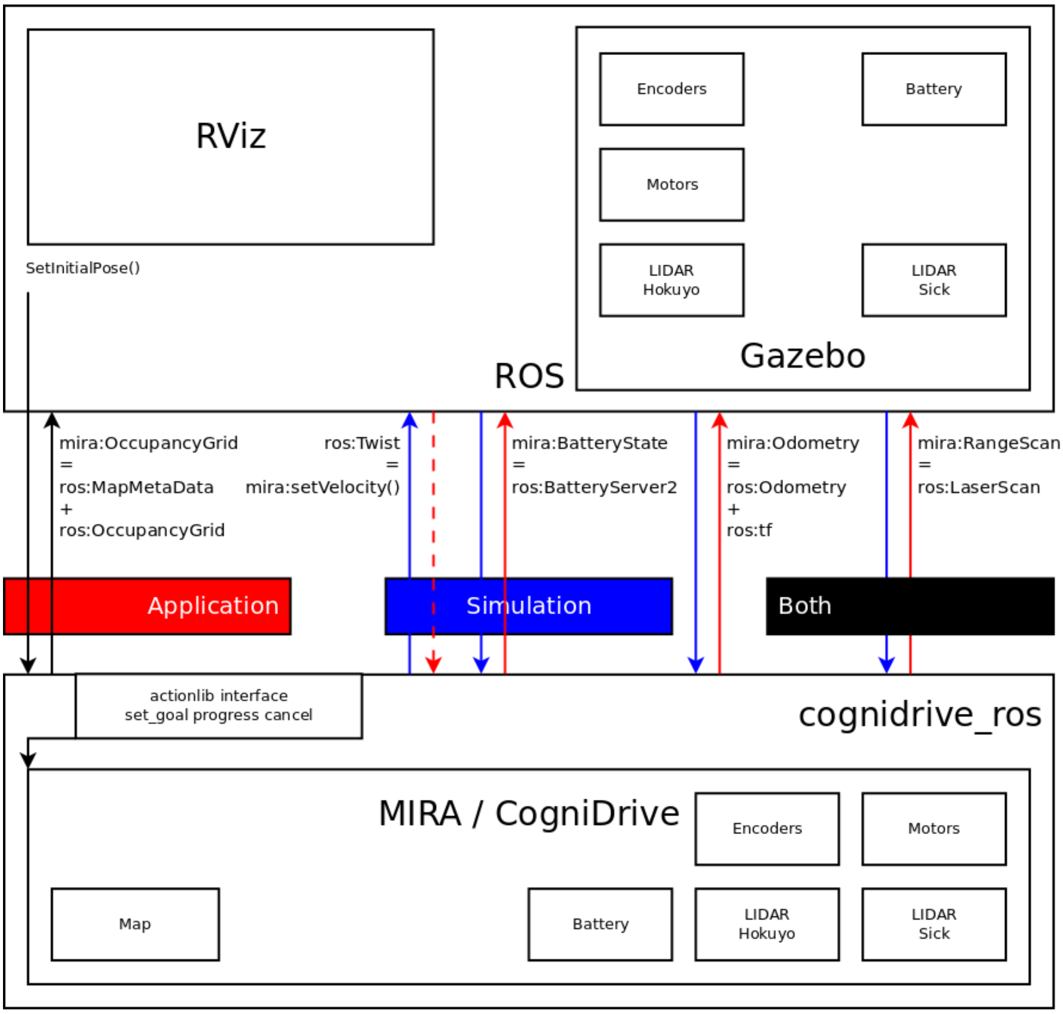


Figure 16: Architecture of the ROS-Cognidrive bridge. See the text for details.

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 MetrLabs Scitos G5 robot running Ubuntu 12.04 LTS 32bit and ROS fuerte.

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 Gazebo, so that MIRA is getting data from the simulator.
- start `rosrun rviz rviz` to visualize laserscans and transforms, set pose estimates or 2D navigation goals.
- start `miracenter` and connect to the running 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 localized correctly. While the AMCL algorithm is capable of localizing 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 localization 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.

- 2 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 [31] 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:

- 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 analyzed 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 synchronize 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 adress

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 See section 5.9.

3.3.4 Usage

The pipeline is started in the startcamera.sh script, located in the doro_description directory (roscd doro_description). It contains the command:

```
LD_PRELOAD=/opt/ros/groovy/lib/libimage_transport.so
  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=leftcamera frame_id=LeftCamera_link
      camerafile=/etc/leftcamera.txt sync=0 peeralloc=1
```

It is possible to adapt the framerate 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/leftcamera.txt*. Also, it is configured that the images will be published on the topic *leftcamera* and the frame_id is LeftCamera_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 filename 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 /leftcamera/image_raw

You can check this by running:

rosrun image_view image_view image:=/leftcamera/image_raw

With the calibration pattern found in Orebro living lab: rosrun camera_calibration cameracalibrator.py –size 8x6 –square 0.02986 image:=/leftcamera/image_raw transport:=compressed camera:=/leftcamera

With the PR2 calibration pattern pattern: rosrun camera_calibration cameracalibrator.py –size 5x4 –square 0.0245 image:=/leftcamera/image_raw camera:=/leftcamera

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 imagesize 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/leftcamera.txt*)
- **or** commit: advises the camera node to store the value internally - overwrites the camera config file */etc/leftcamera.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 opecv 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 startet in the cameras.launch file with the call:

ROS_NAMESPACE="/leftcamera" rosrun image_proc image_proc

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

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

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

3.3.5 Kinect RGB-D Camera and Point-Clouds

If roslaunch openni_launch openni.launch can not be executed, we have to reinstall it seperately. sudo apt-get install ros-hydro-openni-launch

Starting the tabletop segmentation:

The tabletop_object_detector package mainly provides two nodes, the basic one is the

roslaunch tabletop_object_detector tabletop_segmentation.launch tabletop_segmentation_points_in:=/xtion_camera/depth/points

Check whether the object detoctor 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

3.3.6 MJPEG-Server

ROS provides the possibility to stream arbitrary video topics directly to a browser (including mobile devices).

It is automatically started within the cameras.launch file on port 8081.

Manual start:

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

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

You can connect to the video streams using your webbrowser and opening the adress:

http://$ROBOTNAME:8080/stream?topic=/leftcamera/image_raw

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

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

http://$ROBOTNAME:8080/stream?topic=/leftcamera/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

3.3.7 Object Detection and Pose Estimation

This function will detect objects and publish visualization markers and stamped poses containing detectded 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 object SIFT-based object detection works the following way: All the images of the objects are placed in folder. This folder is a quick and dirty replacement of a database, that will be integrated later. The filename needs to contain the width of the object in millimeter. For example, box100.jpg is a box with a width of 100 mm. 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. It may be necessary to manually convert the ost.txt file from the ROS calibration routine explained above to the Distortions.xml and Intrinsics.xml file. The calibration is already ready and does not need to be repeated.

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

"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/tfMessage> 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.

With Calibration data, this command starts a GStreamer Pipeline that itself starts the node:

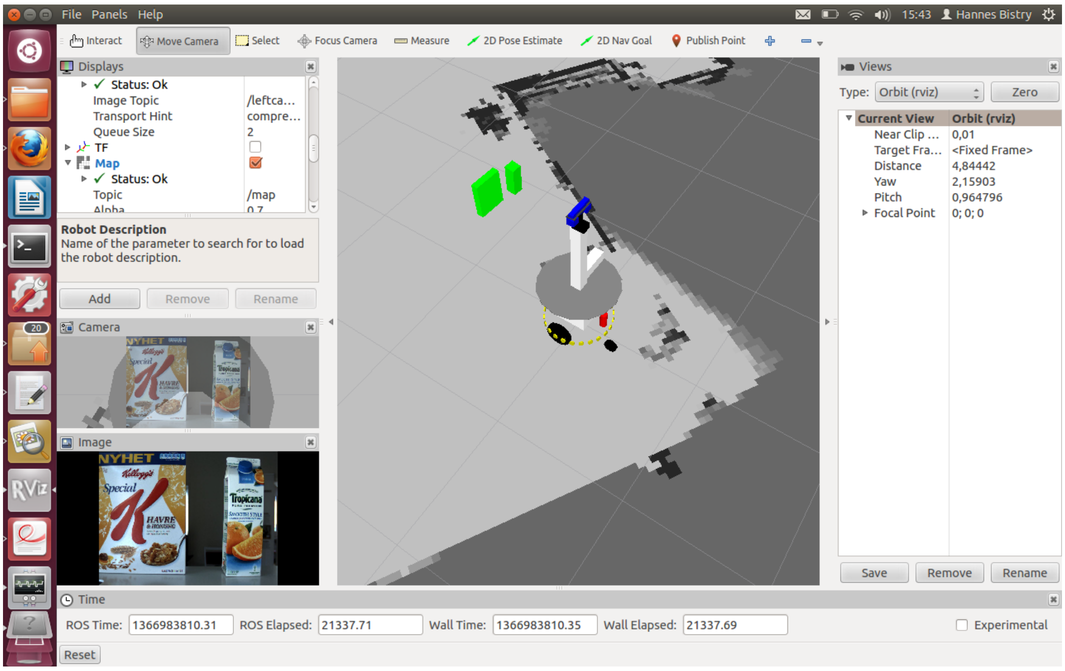
```
LD_PRELOAD=/opt/ros/groovy/lib/libimage_transport.so
  gst-launch rossrc topic=leftcamera/image_raw
    ! queue leaky=2 max-size-buffers=1 ! ffmpegcolorspace
    ! siftextractor
    ! rossiftfolder
        directory=/localhome/demo/camera_calib_data/Models_new
        caminfotopic=leftcamera/camera_info
        frame-id=LeftCamera_link
    ! fakesink
```

This line can be configured in the following way:

directory=/localhome/demo/camera_calib_data/Models ->where the images of the objects are stored.

caminfotopic=leftcamera/camera_info (retrieve the camera info from ROS)

(as an alternative, you can set
unddirectory=/localhome/demo/camera_calib_data/cal_April2013 ->where the Distorsions.xml and Intrinsics.xml are stored, deprecated)



3.3.8 Human Detection and Recognition

Human Detection and Recognition is currently under development and will complement the functionality of the Ambient Intelligence.

3.4 Manipulation

Object manipulation with robots is standard industry practice by today, but the typical factory solution is characterized 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 hardcoded. 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 apparantly simple tasks like picking up an object, several complex subproblems 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 stabilize 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 realized.

The next section 3.5 first summarizes the key features of the Kinova Jaco robot and the software architecture to control the arm and gripper. Next, section 3.6 describes the complete ROS manipulation stack, which integrates perception and motion-planning. It builds an environment model from the sensor data, matches recognized objects agains an object-database, and then performs collision-aware motions to grasp and move objects. Finally, section 3.7 sketches the recent MoveIt! framework, which extends manipulation stack with a more user-friendly interface and improved algorithms. 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.

3.5 Kinova Jaco API and ROS-node

As explained in the previous project report D4.1 [39], 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 stabilizes 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 8 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

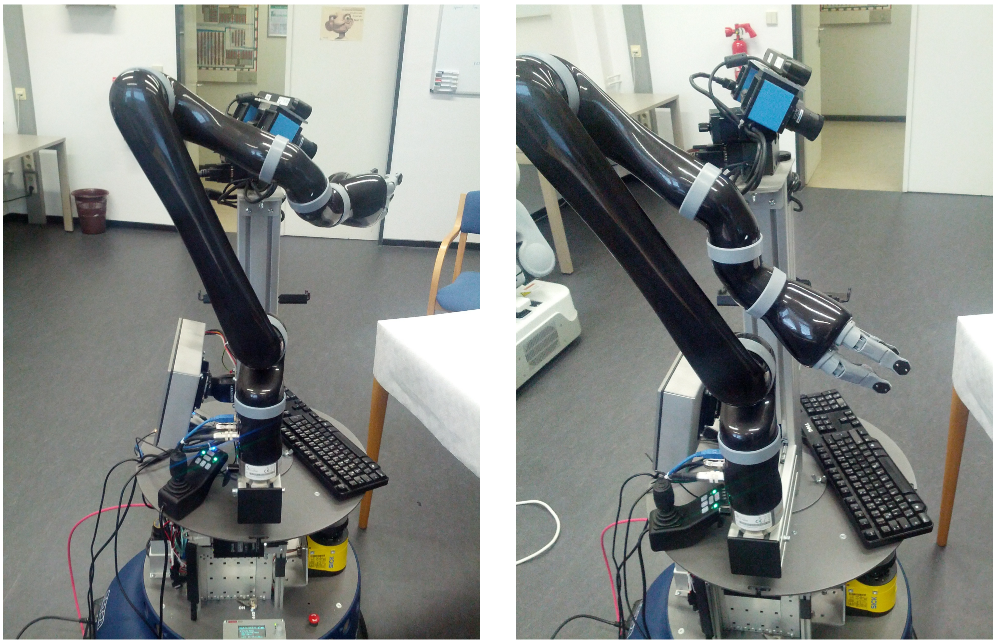


Figure 17: The *home* (left) and *retract* (right) positions of the Jaco arm. These are the two reference positions of the arm which can be reached by pressing (and holding) the yellow button on the Kinova joystick. Note that the *jaco_node* ROS node can only be used when the arm initializes into the Home position.

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 reference positions, namely the *retract* and *home* postions. 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 tele-operated 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 lastest 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 initalized via a call to the arm constructor, which expects the license key provided by Kinova as the password. Once initialized, 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 initialized 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 initializes the following subscribers and publishers.

Subscribers:

- jaco_node/cur_goal (geometry_msgs/PoseStamped)
- hand_goal (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) on 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 globel */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 0.45 -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.

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 underactual 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 modeled 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. 18, 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

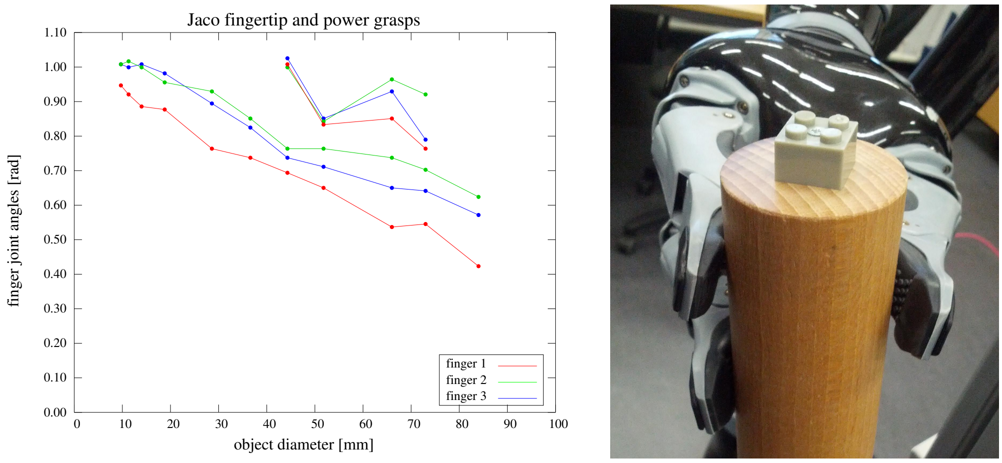


Figure 18: Jaco finger positions as a function of object size. The plot shows the joint-angle in radians for the fingers of the Jaco hand when grasping objects of known diameter using either the *power grasp* (solid lines) between the proximal links of the fingers, or the *fingertip grasp* (dash-dotted 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.

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 [28].

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 (see section 3.6 below) 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. Unimate 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 [14] 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 initialization may fail. Also, during initialization, the node toggles between the Jaco arm rest position and the home position. Manipulation is only possible if the arm initializes into the home position. If necessary, kill and restart the node until the arm initializes 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 initialization 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.

3.6 ROS Manipulation-stack

This section provides an overview of the ROS *manipulation stack* or *object manipulation stack*. See *www.ros.org/wiki/object_manipulation* for the full documentation including dependencies, installation instructions, and tutorials. The ROS stack provides functions for object pickup and placing, while avoiding collisions with the environment and all detected objects. The stack is designed to be robot independent as far as possible. Of course, sensor data is used to build a model of the environment and objects, and the corresponding sensors need to be configured into the software. Regarding manipulation, the robot arm geometry is taken from the URDF model of the robot, but the gripper needs to be configured.

It should be noted that the current version of the manipulation stack is targeted towards simple parallel grippers, while the advanced capabilities of multi-fingered robot hands are not supported. Therefore, in the initial software version, the Kinova hand is used only like a parallel gripper, with the first and third fingers moving in parallel and in opposition to the thumb (second finger). Support for additional grasp types will be considered at a later stage of the project.

On the other hand, the manipulation stack includes several advanced grasping strategies that exploit the tactile sensors mounted onto the fingertips of the PR2 gripper. These include the so-called reactive grasping and reactive placing, where tactile-sensor data is used to detect gripper-object and object-table contacts. Experience gained throughout the first experimental loop of project Robot-Era will show whether the reactive grasping can be implemented on the Kinova hand. See section 3.6 for documentation and the implementation of those parts for the domestic robot.

3.6.1 Overview

The manipulation stack defines a large set of custom ROS messages and services, and also includes a SQL database for models and properties of a set of known objects. The *object perception* pipeline from the manipulation stack tries to identify objects in a scene, working on point-cloud data from the Kinect/Xtion cameras or a stereo camera system. The basic assumptions are that the objects rest on a table or a similar flat surface, which forms the dominant plane surface in the scene, and that the objects are separated by a certain minimum distance. If the dominant plane is a wall of the floor, the algorithm will look for objects on that surface. If parts of the robot are in view during the segmentation, those parts may be returned as target objects despite the application of robot self-filter pre-processing.

For object recognition, the computed point-cloud clusters are then compared against all objects in a database of known-objects. When a match is found, the databased ID of the object is returned together with the point-cloud cluster, and provides additional object information including known grasps and approach directions. Note that the original manipulation stack is limited to rotational symmetric objects like bottles, cans, and glasses, because this simplifies the matching of point-cloud data against the known objects stored in the object database. Within Robot-Era, we also support additional object shapes.

However, even if the database match fails, it may still be possible to grasp and pick the object, using heuristics to calculate grasp approach direction and finger poses from the point-cloud cluster. See *www.ros.org/wiki/tabletop_object_detector* for documentation including the explanation of all software parameters.

1. detect the table
2. segment objects on the table
3. recognize objects on the table
4. build an environment collision map
5. retrieve parameters (e.g. good grasps) for known objects, or use heuristics (based on point-cloud) for unknown objects
6. plan arm trajectories to approach the target object
7. reach and grasp the object (with tactile feedback)
8. lift the object
9. plan arm trajectories to place the object
10. place the object (with tactile feedback)

3.6.2 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 minimize 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 multipe 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 visualized in rviz.

3.6.3 Object recognition

The next step of the manipulation pipeline tries to recognize objects on the table, matching the detected point-cloud clusters against 3D-models of known objects.

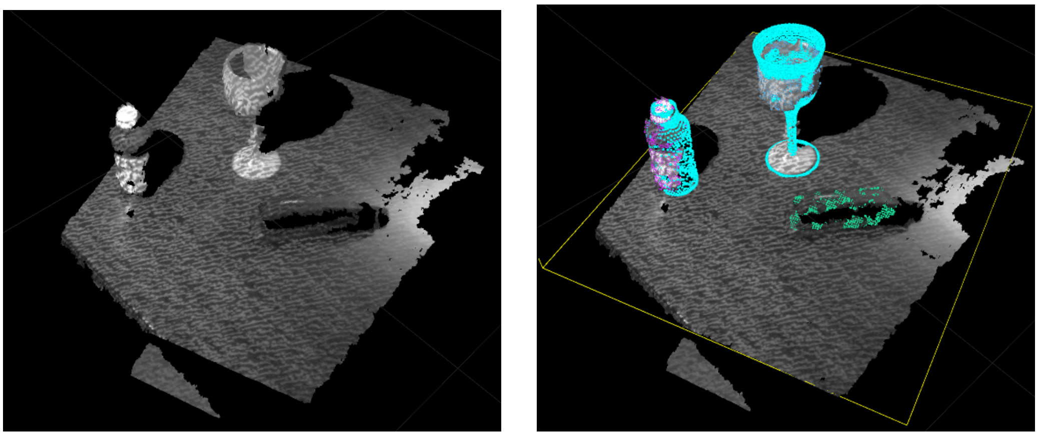


Figure 19: Tabletop object detection example. Left: point-cloud of a table and three objects. Right: detected table plane (yellow contour) and segmented clusters. The bottle and glass have been recognized, and the known 3D-mesh of the object is superimposed on the point-cloud data. (Taken from *www.ros.org/wiki/tabletop_object_detector*).

The *TabletopObjectRecognition* service takes as input the result of the segmentation service, connects to a database of known objects, and then attempts a 2D (x, y) matching of the cluster against the object mesh. That is, the other 4 dimensions are fixed: z is assumed to be the table height, and the (Φ, Ψ, Θ) angles are ignored, because the object is assumed to be upright and rotational-symmetric. For Robot-Era, a more advanced point-cloud matching service is under development, which also includes visual features (SIFT, SURF) and 6D-pose as part of the object matching. Several parameters are provided for the service, please consult the package documentation for details. For convenience, the *TabletopDetection* service combines both the object segmentation and detection steps in one ROS service.

3.6.4 Household objects database

The *household_objects_database* package includes the class definitions and a set of convenience functions for interfacing ROS with a specific SQL database. The stack also provides examples for using the C++ SQL-database interface from ROS. See *www.ros.org/wiki/household_objects_database* for the package documentation. Its main role is to provide the C++ data-types contained in the database and a ROS node wrapper for the most commonly used data-queries, so that the database is accessible via ROS topcs and services.

The default database contains 3D models of selected common household objects. A schema of the database is shown in Fig. 20. The ROS node wrapper provides some of the most common database queries as ROS services. It requires the following node parameters on startup to establish a connection with the database

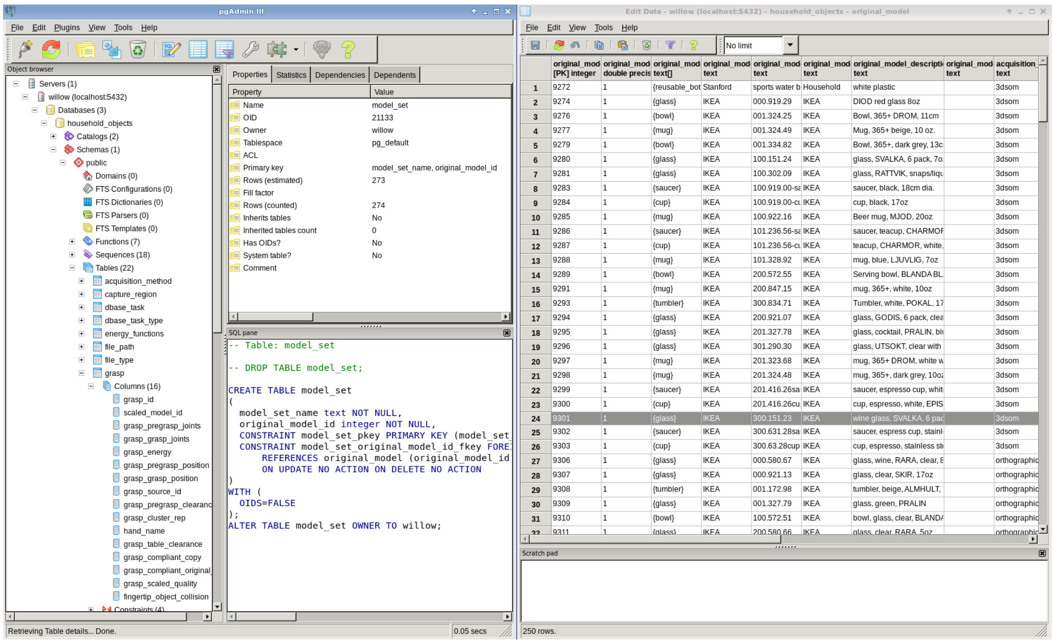


Figure 20: Household objects database. Left: SQL schema and grasps table in the database. Right: Household objects table. The screenshosts were taken using the *pgadmin3* tool.

server. Typically, the topic names are prefixed with the name of the database, E.g. `/household_objects_database`:

- *database_host*: the IP address of the database server
- *database_port*: the port to establish the connection on
- *database_user*: the database username ("willow")
- *database_password*: the password for the given username ("willow")
- *database_name*: the name of the database to connect to

Once started, the wrapper node provides the following ROS services:

- *GetModelList*: gets a list of models available in the database
- *GetModelMesh*: retrieves the 3D mesh for a given model
- *GetModelDescription*: gets a set of descriptive meta-data for a given model
- *GraspPlanning*: retrieves a list of database grasps for the given model.

See Fig. 20 for an overview of the tables required for the grasp recall. Note that the database node will also look on the parameter server for a description of the robot arm being used. See the *pr2_object_manipulation_launch* package for an example of this description. In general, information about database objects recognized in the robot's environment are passed around the system in the form of *DatabaseModelPose* messages.

In theory, it is possible to use the manipulation stack without the database, which effectively means that all objects are grasped just based on the perceived point-clouds without any pre-defined grasps and approach vectors from the database. However, several ROS nodes and services started by the default launch files wait on the services provided by the database, and need to be modified to bypass those.

3.6.5 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 *tabletop_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*.

3.6.6 Object pickup

3.6.7 Object place

just like in the Pickup action, we supply the collision names of both the grasped object and the support surface. They serve similar roles as in the pickup action. mirroring the pickup action, we have an "approach direction", which will be used by the gripper holding the object to approach the drop location, and a "retreat" which will be performed by the gripper after releasing the object. We specify the approach direction in the place request, but the retreat direction is a characteristic of the gripper. We only specify how much we want the gripper to retreat along this direction after releasing the object. just like for pickup, we specify desired approach and retreat distances, but those might not be feasible due to obstacles in the environment or kinematic constraints.

in this example, we approach the table by going "down", along negative z in the base_link frame. We ask for the approach motion to be 10cm, but are willing to accept the place location even if only 5cm of approach are possible we also ask for the gripper to retreat for 10cm after releasing the object, but are willing to accept the place location if at least 5cm are possible.

the additional parameter place_padding is used to decide if the requested place location brings the object in collision with the environment. The object's collision model is padded by this amount when the check is performed; use a smaller padding to fit objects into smaller spaces, or a larger padding to be safer when avoiding collisions.

3.6.8 Starting the manipulation pipeline

Installation and documenation:

```
sudo apt-get install ros-fuerte-pr2-interactive-manipulation
firefox http://www.ros.org/wiki/object_manipulation
firefox http://www.ros.org/wiki/pr2_object_manipulation
firefox http://www.ros.org/wiki/pr2_tabletop_manipulation_apps/
  Tutorials/Starting%20the%20Manipulation%20Pipeline
```

Launching manipulation stack on the PR2 (Doro):

```
robot start
export ROBOT=pr2
roslaunch pr2_tabletop_manipulation_launch
  pr2_tabletop_manipulation.launch
 [stereo:=true] [use_slip_controllers:=true] ...
```

Launching in the Gazebo simulation.

```
roslaunch manipulation_worlds pr2_table_object.launch
export ROBOT=sim
roslaunch pr2_tabletop_manipulation_launch
  pr2_tabletop_manipulation.launch
 [stereo:=true] [use_slip_controllers:=true] ...
```

Connection to the objects model database:

```
 <!-- database server running on a local machine -->
  <rosparam command="load" file=
     "$(find  household_objects_database)/config/wgs36.yaml"/>

  <node pkg="household_objects_database"
        name="objects_database_node"
        type="objects_database_node"
        respawn="true" output="screen"/>
```

Manipulation prerequisites:

```
<!-- manipulation prerequisites -->
<include file="$(find pr2_object_manipulation_launch)/
                    launch/pr2_manipulation_prerequisites.launch"/>
```

Manipulation pipeline (on the PR2):

```
<!-- manipulation -->
<include file="$(find pr2_object_manipulation_launch)/
                   launch/pr2_manipulation.launch">
  <arg name="use_slip_controllers" value="$(arg use_slip_controllers)"/>
  <arg name="use_left_arm" value="$(arg use_left_arm)"/>
  <arg name="use_right_arm" value="$(arg use_right_arm)"/>
  <arg name="use_task_cartesian" value="$(arg use_task_cartesian)"/>
  <arg name="sim" value="$(arg sim)"/>
</include>
```

Sensor processing for manipulation:

```
<!-- tabletop collision map processing -->
<node pkg="tabletop_collision_map_processing"
      name="tabletop_collision_map_processing"
      type="tabletop_collision_map_processing_node"
      respawn="false" output="screen"/>
  <param name="tabletop_collision_map_processing/get_model_mesh_srv"
         value="/objects_database_node/get_model_mesh" />
  <param name="tabletop_collision_map_processing/static_map_cloud_name"
         value="full_cloud_filtered" />

<!-- tabletop segmentation and object recognition -->
<include file=
    "$(find tabletop_object_detector)/launch/tabletop_complete.launch">
  <arg unless="$(arg stereo)" name="tabletop_segmentation_points_input"
       value="$(arg kinect_camera_name)/depth_registered/points"/>
  <arg name="flatten_table" value="$(arg flatten_table)"/>
</include>
```

3.6.9 Simple pick and place example using Pick And Place Manager

See Fig. 21 and 22 for an example Python class that calls *PickAndPlaceManager* to detect an object near a given position, then to pick the object up, and to place it into a given target area. The example code is taken from *www.ros.org/wiki/ pr2_pick_and_place_demos/Tutorials/ A Simple Pick And Place Example Using The Pick And Place Manager*.

3.6.10 Pick and place demo and keyboard interface

```
rosrun pr2_pick_and_place_demos pick_and_place_demo.py
```

There is also an interactive mode with keyboard input for the default manipulation stack pick and place demo. See *www.ros.org/wiki/pr2_pick_and_place_demos/ Tutorials/Pick%20and%20Place%20Keyboard%20Interface*.

The demo executive provides both continuous and step-by-step operation. In the continuous mode, the robot will move objects from one side of the table to another.

```
import roslib
roslib.load_manifest('pr2_pick_and_place_demos')
import rospy
from pr2_pick_and_place_demos.pick_and_place_manager import *
from object_manipulator.convert_functions import *

class SimplePickAndPlaceExample():

    # PickAndPlaceManager constructuor
    #
    def __init__(self):
        rospy.loginfo("initializing pick and place manager")
        self.papm = PickAndPlaceManager()
        rospy.loginfo("finished initializing pick and place manager")


    #pick up the nearest object to PointStamped target_point
    #
    def pick_up_object_near_point(self, target_point):
        rospy.loginfo("moving the arm to the side")
        self.papm.move_arm_to_side(0)  # 0=right arm

        rospy.loginfo("pointing the head at the target point")
        self.papm.point_head(get_xyz(target_point.point),
                             target_point.header.frame_id)

        rospy.loginfo("detecting the table and objects")
        self.papm.call_tabletop_detection(update_table = 1,
                             update_place_rectangle = 1,
                             clear_attached_objects = 1)

        rospy.loginfo("picking up the nearest object to the target")
        success = self.papm.pick_up_object_near_point(target_point, 0)

        if success:
            rospy.loginfo("pick-up successful! Moving arm to side")
            self.papm.move_arm_to_side(0)
        else:
            rospy.loginfo("pick-up failed.")

        return success
```

Figure 21: Pick and place example code

```
    # place the object held in the right arm (0=right) down
    # in the place rectangle defined by place_rect_dims (x,y)
    # and place_rect_center (PoseStamped)
    #
    def place_object(self, place_rect_dims, place_rect_center):

        self.papm.set_place_area(place_rect_center, place_rect_dims)
        rospy.loginfo("putting down the object in the gripper" );
        success = self.papm.put_down_object( 0,
                      max_place_tries = 25,
                      use_place_override = 1)
        if success:
            rospy.loginfo("place returned success")
        else:
            rospy.loginfo("place returned failure")
        return success

if __name__ == "__main__":
    rospy.init_node('simple_pick_and_place_example')
    sppe = SimplePickAndPlaceExample()

    # adjust for your table
    table_height = .72

    # .5 m in front of robot, centered
    target_point_xyz = [.5, 0, table_height-.05]
    target_point = create_point_stamped(target_point_xyz, 'base_link')
    success = sppe.pick_up_object_near_point(target_point, 0)

    if success:
        # square of size 30 cm by 30 cm
        place_rect_dims = [.3, .3]

        # .5 m in front of robot, to the right
        center_xyz = [.5, -.15, table_height-.05]

        #aligned with axes of frame_id
        center_quat = [0,0,0,1]
        place_rect_center = create_pose_stamped(
                         center_xyz+center_quat, 'base_link')

        sppe.place_object(0, place_rect_dims, place_rect_center)
```

Figure 22: Pick and place example code (cont'd)

In step-by-step mode, you can use keyboard commands to perform individual actions. To launch,

```
roslaunch pr2_pick_and_place_demos
                     pick_and_place_keyboard_interface.launch
```

Here is the root menu of the demo executive. Type:

`start`	to start the autonomous demo
`s`	to switch pick-up and put-down sides
`hs`	to point the head at either side
`r`	to control the right arm, l to control the left arm
`d`	to detect objects,
`dc`	to detect and take a new static collision map,
`dca`	to detect, take a new collision map, and clear attached objects
`p`	to pick up an object
`w`	to place the object in the place rectangle,
`wo`	to place the object where it came from
`h`	to point the head at the current place rectangle and draw it
`s`	to switch pick-up and put-down sides
`t`	to find the table
`rm`	to reset the collision map,
`tm`	to take a new collision map
`det`	to detach the object in the gripper
`q`	to quit
	press enter to continue

3.6.11 Useful topics and rviz markers

Here is a list of useful rviz markers that can be used to monitor the manipulation pipeline:

- */object_manipulator/grasp_execution_markers* shows the grasps being tested for execution by the object_manipulator
- */planning_scene_markers*: shows the last planning scene used for motion planning and collision avoidance
- */occupied_cells*: shows the current occupancy grid for the Octomap
- */point_cluster_grasp_planner_markers*: shows the behavior of the grasp planner for unrecognized point clouds
- */kinematics_collisions*: shows the collisions reported by IK queries; very useful for understanding why certain grasps or place locations have been rejected by the manipulation pipeline
- */tabletop_segmentation_markers* and

- */tabletop_object_recognition_markers*: shows the markers for the recognized objects from the *tabletop_object_detector*
- */attached_objects*: shows objects that have been attached to the grippers for collision avoidance purposes (Electric only)

3.6.12 Reactive grasping and gripper sensor messages

The manipulation stack pipeline also supports an advanced mode for the reach and grasp motions, called *reactive grasping*, where tactile feedback from the gripper is used to adjust and improve the grasp position. The gripper on the PR2 robot is equipped with tactile matrix sensors that cover the inner faces of the gripper as well as the extreme fingertips. When grasping an object off-center due to calibration errors or insufficient accuracy from the vision system, only parts of the tactile sensors are triggered, and algorithms are provided to adjust to arm until the gripper position is centered on the object. A description of the algorithms is presented in [23]. Additionally, there is support for object-slip detection based on the tactile sensors and the accelerometers.

While the Kinova Jaco gripper has no tactile sensors, measuring the motor currents may allow us to detect object contact and coarse tactile exploration. See *www.ros.org/wiki/pr2_gripper_sensor_msgs* for the (large) set of messages defined for the PR2, and Fig. 23 for a block diagram of the setup on the PR2 robot.

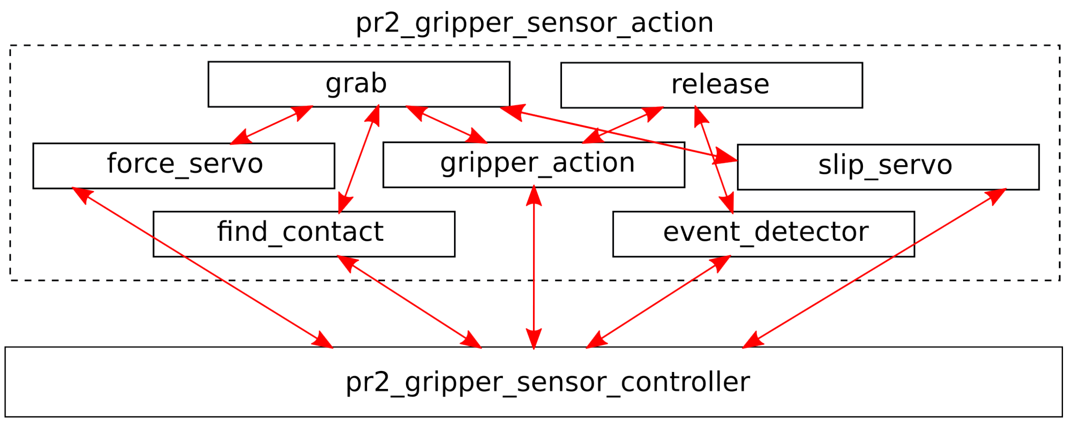


Figure 23: Overview diagram of the reactive grasping module within the ROS manipulation stack for the PR2 robot.

3.7 MoveIt! framework

The MoveIt! framework is set to replace the original manipulation stack. It is under consideration as the main motion-planning tool for the domestic robot. The software is under active development, please check the website and Wiki at *moveit.ros.org/wiki* for documentation and recent updates. The block-diagram in Fig. 24 presents an overview of the planning architecture and the interface to ROS. Please note that the MoveIt! toolchain is only supported on ROS version *Groovy* or higher. Visit *moveit.ros.org/wiki/index.php/Groovy/Installation* for details and installation instructions.

One of the key ideas 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 coordinate frames of the robot used for 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 Pecchioli/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

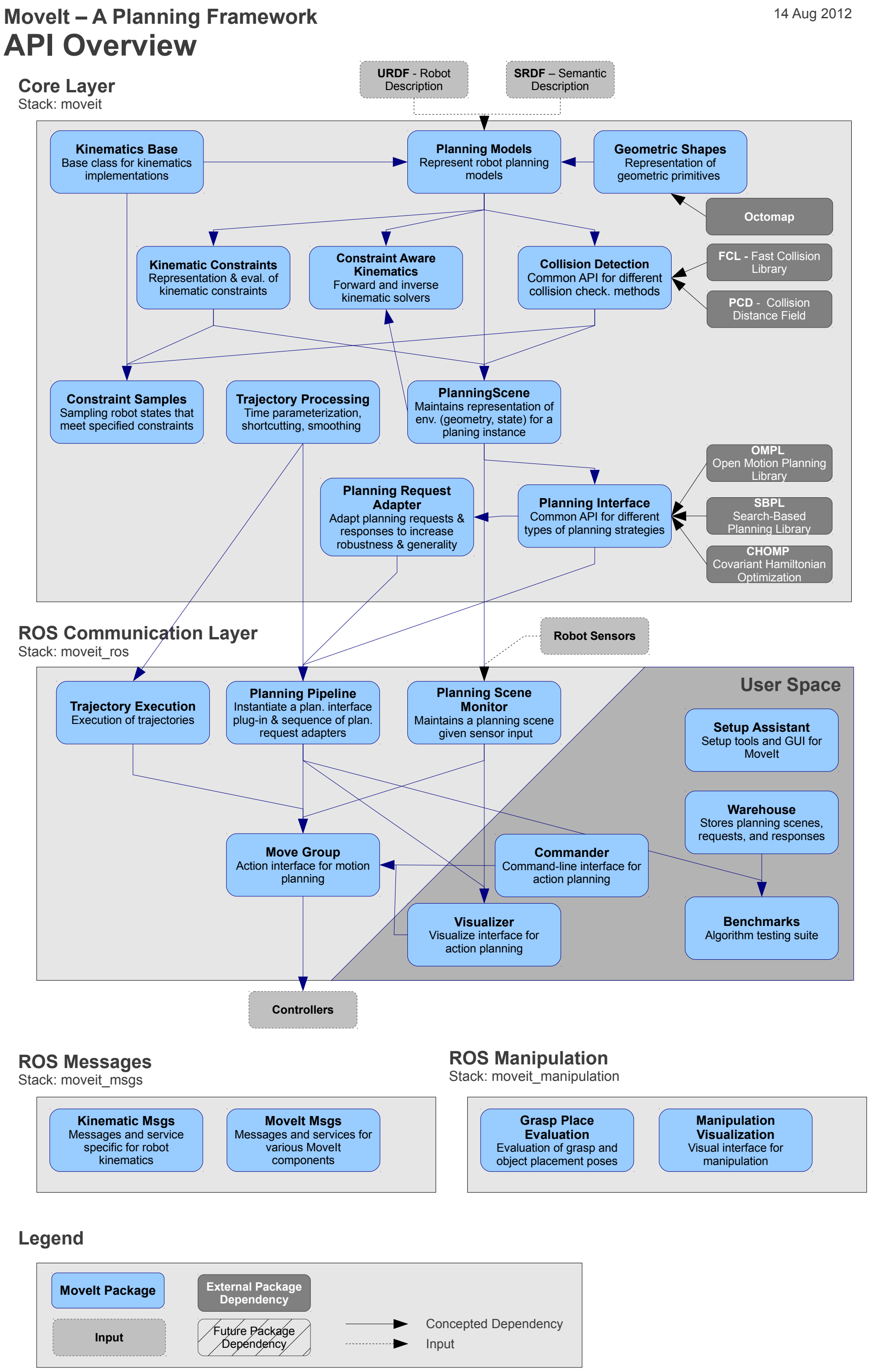


Figure 24: Block diagram of the main components in the MoveIt! framework. Copied from *http://moveit.ros.org/doxygen/pdfs/moveit_api_diagram.pdf*.

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.

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 *FllowJointTrajectoryActionGoal* object and publishes this on the Jaco *follow_joint_trajectory/goal* topic, which starts the corresponding Jaco joint-level trajectory:

```
// Johannes Liebrecht --- 8liebrec@informatik.uni-hamburg.de
#include <jaco_api/jaco_api.hpp>
#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>

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      = "testJohannes";

  // 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>
```

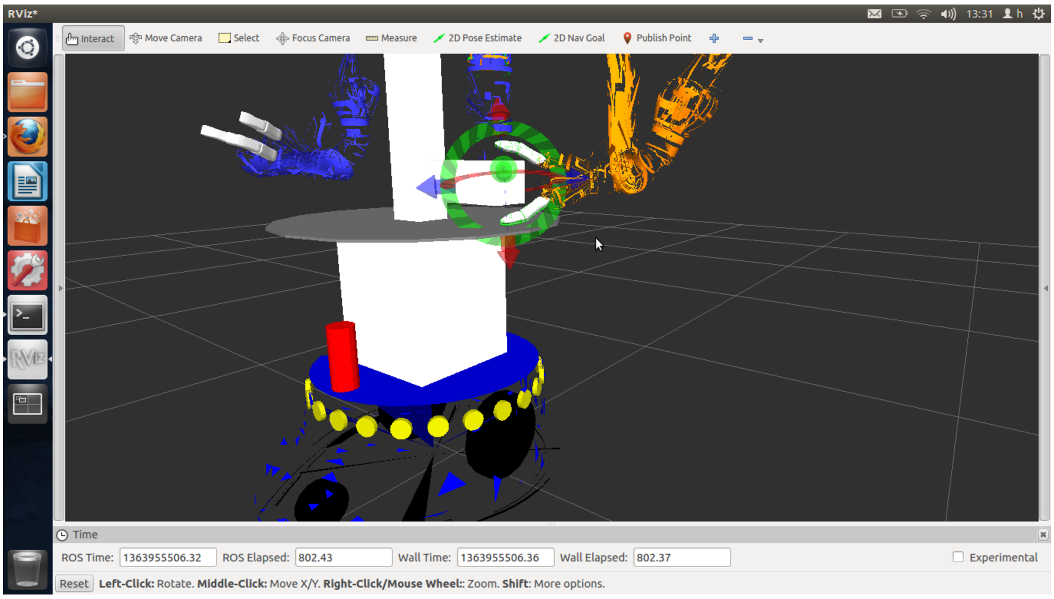


Figure 25: Screenshot of the MoveIt! planner for the domestic robot, showing the interactive marker for setting the goal position of the robot.

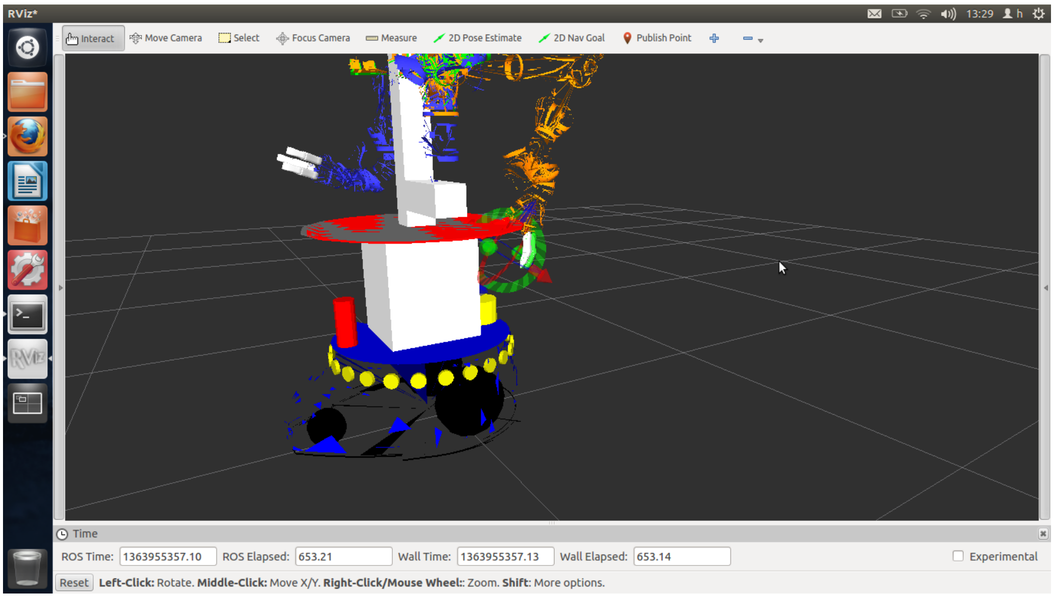


Figure 26: Screenshot of the MoveIt! planner for the domestic robot detecting a self-collision between the hand and the base of the robot.

3.8 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 [32] 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. 28 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.

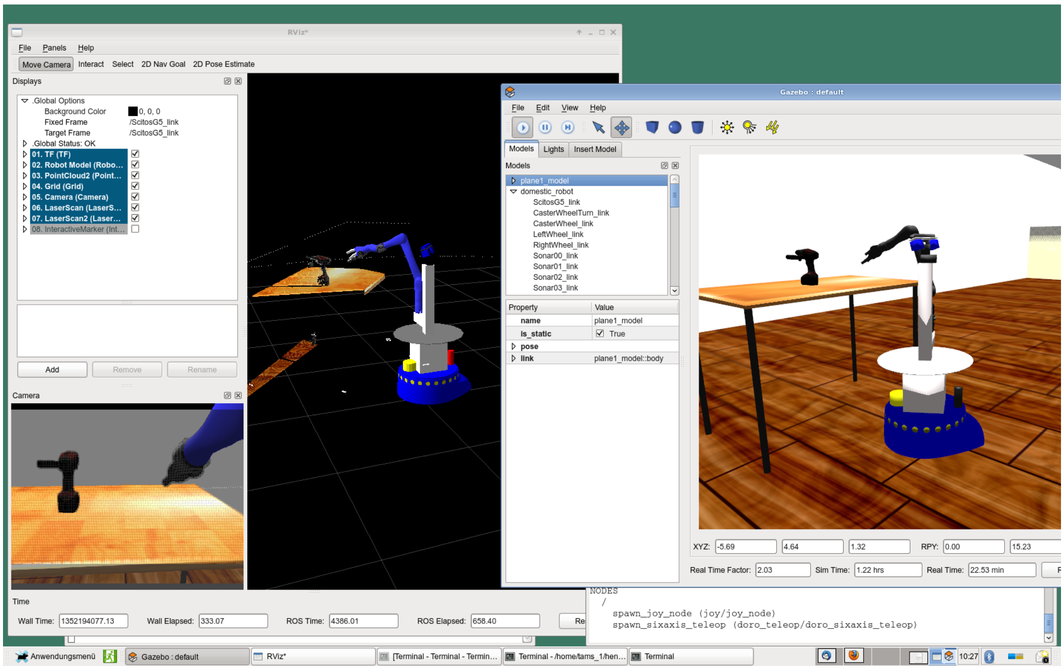


Figure 27: A screenshot of ROS rviz (left) and the Gazebo 3D viewer (right), showing the domestic robot in a simulated environment. The geometry of the robot and its actuators are taken from the ROS URDF model of the robot, and realistic sensor data (laser-scanners, cameras, sonar) is generated by the physics simulation engine.

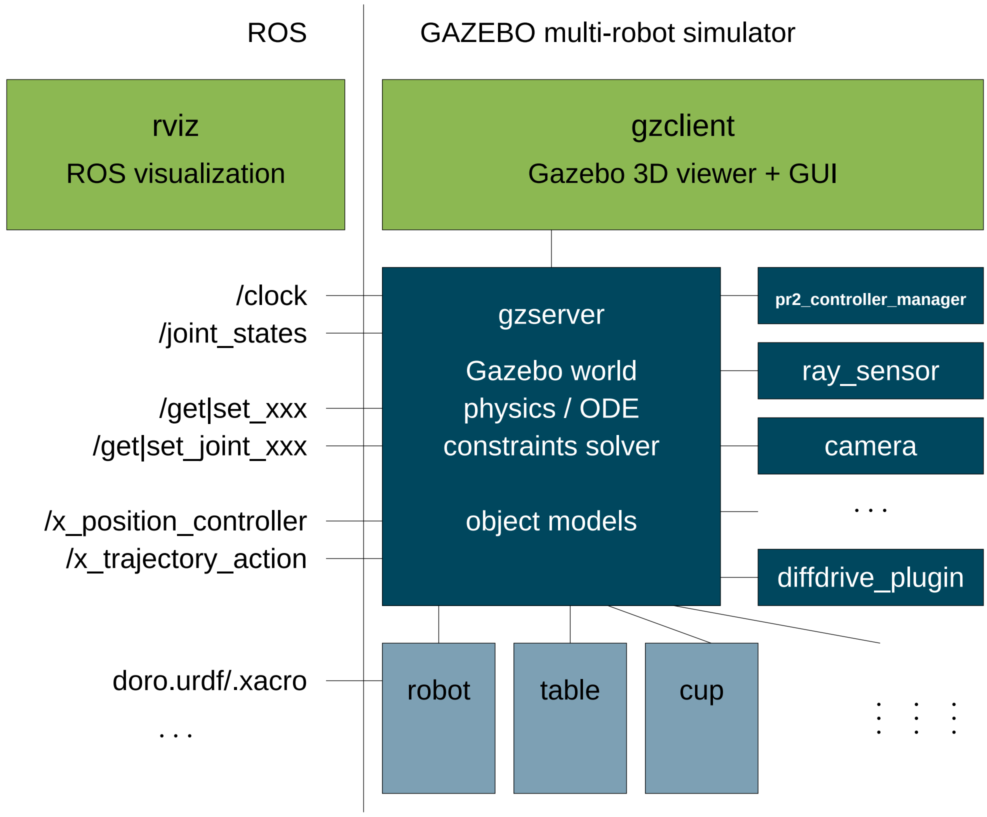


Figure 28: Block diagram of the main components of the Gazebo multi-robot simulator and the integration into the ROS framework. The *gzserver* process maintains the world model and calculates object motion based on the ODE physics engine. The server can load models described in its native SDF file format, 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 *pr2_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 programm calls that allow to query and modify the world state and to control the simulation engine.

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.

3.8.1 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.

In case of problems, please check the Gazebo documentation for a list of the required startup-files and any Gazebo/Ogre environment variables required to run you particular version of the simulator. Typically it is easiest to use the predefined launch files to start Gazebo and then launch the robot model. To keep track of the outputs of different processes, the use of diffrent shells/terminals is recommended:

```
shell-1> roslaunch gazebo_worlds empty_world.launch
shell-2> roslaunch doro_description domestic_robot.launch
shell-3> rosrun doro_teleop doro_keyboard_teleop
shell-4> roslaunch doro_teleop domestic_sixaxis.launch
shell-5> rosrun rviz rviz
shell-6> telnet localhost 7790
```

where steps 1 and 2 are required, while 3 and 4 start the basic keyboard-based or the joystick-based interactive tele-operation programs, and step 5 is recommended to watch and control ROS messages and the 3D *tf* transformation-tree of the robot.

The telnet session started in step 6 connects to a simple server process that is started as part of the *domestic_robot.launch* file. This allows you to query the state of the domestic robot and execute simple joint-level motions, one command per line. Note that the telnet-server expects joint-angles in degrees, for easier interpretation by humans:

```
telnet localhost 7790
telnet> help
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
```

3.8.2 Notes and version compatibility

The Gazebo simulator predates the development of ROS and can be installed as a standalone software package, However, development of the simulator has been coordinated by WillowGarage in the last few years, and the current versions of Gazebo are tightly coupled to several dependencies from ROS. This includes the controller-manager plugins which implement the real-time trajectory controllers, and the support for the URDF robot description file format in addition to the flat SDF simulation description file format used by Gazebo. As a result, only the versions of Gazebo shipped as parts of the ROS installation are known to work for complex robot models like the WillowGarage PR2 or the Robot-Era domestic robot. However, the Gazebo developers target to clean up those dependencies and interfaces for an upcoming new release of the simulator.

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 Gazebo developers concentrate more on new functionality and refactoring parts of the simulation engine than on compatibility and the interfaces to ROS. 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.

At the time of writing, Gazebo 1.02 is the official version for ROS-Fuerte, and this version has been used for the initial development of the domestic robot model. Note that self-collisions on the robot model are not working correctly, while collisions between the robot and other objects should work. Due to a race-condition in Gazebo 1.02 during simulator startup, it is not recommended to launch the simulator from the same ROS launch file that also spawns the robot model(s). Instead please first start the simulation server and the 3D viewer with the empty world, and only load the robots and objects once the simulation engine has finished its initialization.

```
shell-1> roslaunch gazebo_worlds empty_world.launch
shell-2> roslaunch doro_description domocasa_lab.launch
shell-3> roslaunch doro_description domestic_robot.launch
```

3.9 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. 7 on page 13), the interface layer consists of a set of largely independent modules, each of which provides one specific service from the Robot-Era storyboards.

3.9.1 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 analyzes 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* changle. 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.

3.9.2 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 canceled. See *www.ros.org/wiki/actionlib* for details and documentation.

Therefore, most PEIS-ROS TupleHandlers will support an *actionlib interface* to their services, including defintion 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,

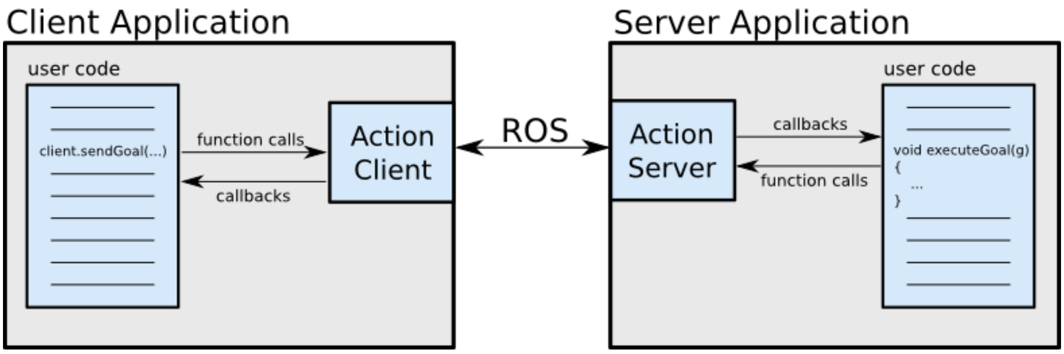


Figure 29: The interface between actionlib clients and servers. The PEIS-ROS TupleHandler classes are written as action clients, where incoming PEIS commands are converted to action goals. Corresponding PEIS tuples are created and/or updated when the action server sends new status and feedback data. The actual robot service is provided on the ROS server application side.

the TupleHandler 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. 30 for a state-machine diagram that shows the common states for the PEIS-ROS actionlib implementation. When starting the corresponding ROS node, the service initializes 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 *PREEMTPED*. Actual state and progress from the service is sent back to PEIS via the corresponding *STATUS*-tuples.

3.9.3 Synchronization

The PEIS configuration planner (CPM) is responsible for the scheduling of actions in the PEIS network, including the synchronization 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

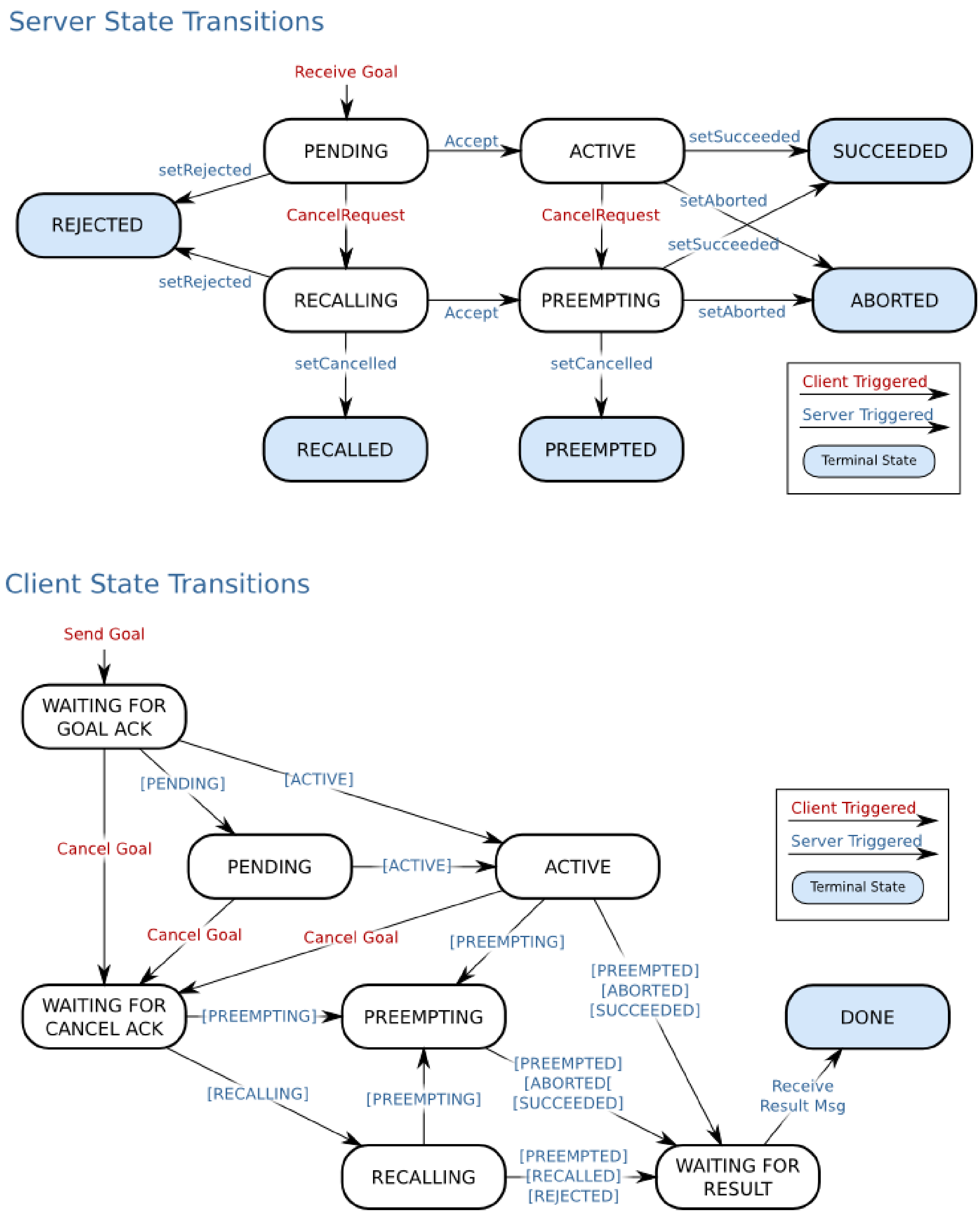


Figure 30: The server and client states for the ROS actionlib stack. Those states are also used in the PEIS-ROS bridge. See the text for an explanation. (Used with permission from www.ros.org/wiki).

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 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.

3.9.4 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/URIs (*unique resource identifiers*) as their return values. Strings are handled easily in PEIS and *tupleview*, and the client can then use a webbrowser or other tool to retrieve the information from the domestic robot. To tag ROS topics and services, the syntax *rosmsg://roscore:port/topicname* and *rossrv://roscore:port/servicename* will be used. See the service descriptions in chapter 4 below for the documentation of any URL/URI messages used by the domestic robot.

3.9.5 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. 31 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 initializiation 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 paramenters, and start execution of the corresponding activity.

A very simple example of a derived class is shown in Fig. 32 and 34. 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 initializes 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.

```
#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 ); // initializes 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.
  // When requested, we subscribe(), then getTuple(), then unsubscribe().
  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
};
```

Figure 31: 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 32: 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 33: 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 34: DemoService.cc (2/2)

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.

Please note that the descriptions in this version of the handbook are not yet auto-generated from the actual source-code, but correspond to a mixture between the design specifications of the services and the actual prototype implementation for the first experimental phase of project Robot-Era. 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.

Also note that the services are ordered by their complexity and grouped by function in this handbook, instead of being sorted alphabetically. 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, lists *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 115. 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 100 and 115 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 rectracting 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 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. 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 localization. This is done by publishing to the *request_emergency_stop* topic.

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 canceled and the current robot position is sent as the new setpoint. The arm is kept powered, because a switch-off results in the arm falling down and potentially harming the user and equipment.

Handler `doro_peis/emergency_stop_handler.cpp`

Tuples

`doro1.emergency_stop.id` = tupel-ID
`doro1.emergency_stop.parameters`= none
`doro1.emergency_stop.command`= `OFF | ON`
`doro1.emergency_stop.state`= `IDLE | ACTIVE`

Status To be designed and implemented soon.

4.1.2 OpenCloseTray

Synopsis A request to open and close the tray on the outdoor DustCart robot.

Handler `peis_ros/xxxTupleHandler.cpp`

Tuples

Status One of the prototype tuple-handlers acutally implemented so far, but on the outdoor DustCart robot. Doesn't refer to doro and should be removed here?

4.1.3 GetCameraImage

Synopsis Requests to return the current image from the high-res Firewire camera on the robot sensor-head. For example, the image can be shown to the user as an image-map, where the user can then select a particular object. Optionally, requests to get the camera image corresponding to the given timestamp.

Handler `peis_ros/GetCameraImageTupleHandler.cpp`

Tuples TBD.

Status Need to decide whether to return a PEIS-tuple with the image in some raw format, or returning a URI to the image, or both.

Example

4.1.4 GetKinectImage

Synopsis Requests to return the current RGB-D image from the Kinect/XtionPro depth-camera. For example, the image can be shown to the user as an image-map, where the user can then select a particular object.

Handler `peis_ros/GetKinectImageTupleHandler.cpp`

Tuples TBD.

Status Need to decide whether to return a PEIS-tuple with the image in some raw format, or returning a URI to the image, or both.

4.1.5 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 representation of the ROS *sensor_msgs/LaserScan* message, including timestamp and an array of distances in meters. We also return the minium 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 `peis_ros/GetLaserScanTupleHandler.cpp`

Tuples TBD.

Status Need to decide whether to return everything in one PEIS-tuple, whether to use binary or human-readable format, and whether to include "semantic" properties (e.g. obstacle in 0.3 m distance).

4.1.6 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 reportet by the corresponding sonar sensor.

Handler `peis_ros/GetSonarScanTupleHandler.cpp`

Tuples TBD.

Status Not implemented. Do we want to use the sonar sensors at all?

4.1.7 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 the yaw-angle (in radians). The orientation and origin of the coordinate system are based on the current map of the robot.

At the moment, Mira/Cognidrive is used for controlling the SCITOS-G5 platform [30], while the ROS-MIRA bridge software interfaces MIRA to ROS. Robot localization, 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 available on the ROS topics `/xxx/xxx` and the `/tf/` tree of transformation. To push the information into the PEIS ecology, periodic robot-pose information is published at about 10 Hz on the `xxx/xxx/xxx/` tupel.

Handler `peis_ros/MoveToTupleHandler.cpp`

Tuples

`doro1.emergency_stop.id` = tupel-ID

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 `peis_ros/MovePtuTupleHandler.cpp`

Tuples

`doro1.move_ptu.id` = tupel-ID
`doro1.move_ptu.parameters` = pan-angle, tilt-angle, [coordinate-system]
`doro1.move_ptu.command`= OFF || ON
`doro1.move_ptu.state`= IDLE || RUNNING || COMPLETED || ERROR
`doro1.move_ptu.log`= completion-percentage || error-description TBD.

Status Implemented.

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 initialize (or re-initialize) the Jaco arm.

Handler `peis_ros/RetractJacoTupleHandler.cpp`

Tuples

Status Implemented aware of self-collisions, obstacles and grasped objects. Tuple definition needs to be updated..

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 gravitiy 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, aware of self-collisions, obstacles and grasped objects. Tuple definition needs to be updated.

4.1.11 MoveJacoArm

Synopsis Requests a joint-level motion of the Jaco arm to the given joint-angles. Optionally finger-angles can be provided as well.

Handler `peis_ros/MoveJacoTupleHandler.cpp`

Tuples

Status Implemented, aware of self-collisions, obstacles and grasped objects. Tuple definition needs to be updated.

4.1.12 MoveJacoCartesian

Synopsis Requests a joint-elvel motion of the Jaco arm to the given cartesian pose. Note that the current implementation still takes coordinates with refernce to the Kinova Jaco coordinate system.

Handler `peis_ros/MoveJacoCartesianTupleHandler.cpp`

Tuples TBD.

Status Backend implemented, aware of self-collisions, obstacles and grasped objects. Tuple definition needs to be updated.

4.1.13 MoveJacoFingers

Synopsis Requests to move the fingers to the given joint-angles.

Handler `peis_ros/MoveJacoFingersTupleHandler.cpp`

Tuples

Status Implemented.

4.2 Intermediate Services

The intermediate services are services that are either a sequence of Low-Level-Services or incorporate data processing. They can be called either via PEIS-messages or directly from higher level services.

4.2.1 DetectKnownObject

Synopsis Known objects can be detected using the SIFT method from the camera images. As the size of the objects is known, the pose can be calculated.

Handler `peis_ros/ObjectDetectionTupleHandler.cpp`

Tuples

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

4.2.2 DetectUnknownObject

Synopsis Unknown objects can be detected using the Kinect and the tabletop-segmentation from ROS. This collection of nodes detect the surface of a table and clusters the objects on the table.

Handler `peis_ros/ObjectDetectionTupleHandler.cpp`

Tuples

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

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. Based on the household-objects database in ROS manipulation stack, required properties of the object are its geometry, reference coordinate system, a set of known stable grasps, pre-grasps and approach vectors for each of the known grasps, friction coefficients for the object surfaces, object weight and approximate inertial properties, and visual descriptors or point-cloud reference for detection and pose alignment. 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 `peis_ros/ManipulationTupleHandler.cpp`

Tuples

Status Implemented for box-shaped objects, other shapes to be implemented soon. Database structure and setup to be decided: reuse household-objects from ROS, or start with something in PEIS?

4.2.4 SideGraspAndLiftObject

Synopsis This service implements grasping and lifting an unknown object of given or estimated size, using a side-grasp aligned to a reconstructed point-cloud of the object. The main approach direction is typically horizontal but may also be vertical.

Handler `peis_ros/ManipulationTupleHandler.cpp`

Tuples

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

4.2.5 TopGraspAndLiftObject

This service implements grasping and lifting an unknown object of given or estimated size, using a top-grasp aligned to a reconstructed point-cloud of the object. The main approach direction of the hand is vertical.

Synopsis

Handler `peis_ros/xxxTupleHandler.cpp`

Tuples

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

4.2.6 PlaceObjectOnTray

Synopsis

Handler `peis_ros/xxxTupleHandler.cpp`

Tuples

Status Not directly supported yet, as there is no possibility to detect the objects placed on the tray.

4.2.7 GraspObjectFromTray

Synopsis

Handler `peis_ros/ManipulationTupleHandler.cpp`

Tuples

Status Not directly supported yet, as there is no possibility to detect the objects placed on the tray.

4.2.8 HandoverObjectToUser

Synopsis Prerequisite: The robot has performed a grasping operation and the grasp has been successful. The robot arm will be driven to a handover position. After this the planner needs to send a tuple to release the grasp.

Handler `peis_ros/ManipulationTupleHandler.cpp`

Tuples

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

4.2.9 HandoverObjectFromUser

Synopsis The robot arm will be driven to a handover position. After this the planner needs to send a tuple to close the gripper.

Handler `peis_ros/ManipulationTupleHandler.cpp`

Tuples

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

4.2.10 PourLiquidMotion

Synopsis

Handler `peis_ros/xxxTupleHandler.cpp`

Tuples

Status Not implemented/defined yet, will probably be removed depending on the scenario definitions.

4.2.11 MoveHingedDoor

Synopsis

Handler `peis_ros/xxxTupleHandler.cpp`

Tuples

Status Not implemented/defined yet, will probably be solved by automated doors.

4.2.12 MoveSlidingDrawer

Synopsis

Handler `peis_ros/xxxTupleHandler.cpp`

Tuples

Status Not implemented/defined yet, will probably be solved by automated doors.

4.2.13 RecognizeUserGesture

Synopsis

Handler `peis_ros/xxxTupleHandler.cpp`

Tuples

Status Not implemented/defined yet, will probably be removed.

4.2.14 LookAt

Synopsis This service moves the pan-tilt unit so that the cameras look at the given 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 `peis_ros/LootAtHandler.cpp`

Tuples

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

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. 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. For the first phase of the project, the approach taken is very simple: fail. 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 DetectPerson

Synopsis

Handler `peis_ros/xxxTupleHandler.cpp`

Tuples

Status To be defined/implemented by UOP (Human-Robot-Interaction).

4.3.2 RecognizePerson

Synopsis

Handler `peis_ros/xxxTupleHandler.cpp`

Tuples

Status To be defined/implemented by UOP (Human-Robot-Interaction).

4.3.3 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. Note that *Walking-Support* is substantially more difficult than the basic *MoveTo* service described in subsection 4.1.7 on page 93, because the robot must adjust speed and orientation to the speed and motions of the user, in order to guarantee safety and actual help the user.

Handler `doro_peis/WalkingSupportTupleHandler.cpp`

Tuples TBD.

Status Initial implementation might experiment with hardcoded linear and rotation speeds, matching recorded speeds for any given user. Final implementation might require force-torque sensors on the handle and advanced filtering techniques, allowing the user to guide the robot motion around the room.

4.3.4 CleanFloorService

Synopsis

Handler `doro_peis/clean_floor_handler.cpp`

Tuples

`doro1.clean_floor.id` = tupel-ID

4.3.5 CleanTableService

Synopsis

Handler `peis_ros/xxxTupleHandler.cpp`

Tuples

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 is expected to be available in time for 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 will be designed, that can be handed over from the condominium robot to the domestic robot, that will fetch this box from the apartment door

Handler `peis_ros/xxxTupleHandler.cpp`

Tuples

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

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 will be designed, that can be grasped by the domestic robot and handed over to the condominium robot.

Handler `peis_ros/xxxTupleHandler.cpp`

Tuples

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 will be designed, that can be grasped by the domestic robot and handed over to the condominium robot.

Handler `peis_ros/xxxTupleHandler.cpp`

Tuples

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

4.4 Implementation

Most of the services itself are currently implemented inside one single ROS-node. This node handles exclusive access to the hardware components and features a simple state machine, that limits the next possible actions.

Low level sensing service can directly fetch data from the corresponding ROS-node, as no exclusive access is necessary.

5 Software installation and setup

This chapter summarizes installation, setup, and configuration instructions for the different main software packages required for the domestic robot.

5.1 Ubuntu 12.04

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. At the time of writing, the Fuerte and Groovy variants of ROS are available as pre-built download packages. Original software development for the Domestic Robot started with ROS Fuerte, but some later improvements (notably for the Gazebo simulator and the MoveIt! frameworks) are only available as parts of ROS Groovy. 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`.

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 localization, 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*.

`/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/fuerte`	Fuerte installation
`/opt/ros/groovy`	Groovy installation
`/usr/local/*/`	PEIS installation

Figure 35: Software installation paths

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.

Details about PEIS and the installation of the configuration planner can be found in D3.2.

5.5 Kinova Jaco Software

The Kinova Jaco software is pre-installed on the domestic robot. When necessary, re-install from the USB-stick suppled by Kinova. Change to the subdirectory 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 building the *jaco_node*, *jaco_test_node*, and *jaco_api* stacks will look for headers and libraries at *mono/jit/jit.h* etc., while the *mono* installation puts the files at *mono-2.0/mono*. One quick workaround is to create a symbolic link,

```
ln -s /usr/include/mono-2.0/mono /usr/include/modo
```

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 Fuerte - Deprecated

Note: Most of the packages are currently porte to Hydro.

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

5.7 ROS Groovy - Deprecated

Note: Most of the packages are currently porte to Hydro.

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

5.8 ROS Hydro

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

ROS Hydro is the currently supported version. All different sensor and actuator systems, as well as the MoveIt-based motion planning is implemented and tested in ROS Hydro.

5.9 GStreamer Libraries

The vision Libraries include the GStreamer-ROS adapter as well as the plugins for object detection and pose estimation. These libraries are already installed on the robot.

On the robot system, only the following two asdditional dependancies are needed

```
sudo apt-get install libgsl0-dev libgtk-3-dev
```

Nevertheless, the full list of dependancies is

```
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
```

Checkout the robot-era repository and cd to

```
robot_common/visionlibs
```

As we have a 32-bit system, we need to

```
make distclean
./autogen.sh
./configure --libdir=/usr/lib/i386-linux-gnu
make
sudo make install
```

To check the installation, type

```
gst-inspect rossink
```

If it shows some configuration parameters of an element, everything is fine, if it throws some error messages, like library not found or unknown symbol, paste output to a file and send it to *bistry@informatik.uni-hamburg.de*

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

5.10 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 CognidriveROS 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.

5.11 Network setup

The network setup including cabled and wireless connections is pre-installed and configured. Where necessary, update the IP-configuration to match your network environment. Add the hostname or the domestic robot into the */etc/hosts* files for all external computers that should connect via ROS; similarly, update the */etc/hosts* on the robot to include all external computers that should be able to connect via ROS.

5.12 Testing the robot

TBD.

5.13 Robot calibration

See section 3.1.7 for details.

6 Summary

This reports describes the hardware and software setup of the Robot-Era *Domestic Robot* (Doro), including the complete list of robot skills implemented or planned for the robot.

Acknowledgements

To the whole Robot-Era consortium for the motivating and fruitful discussions.

References

[1] 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.

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

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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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.

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

[10] 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

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

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

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

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

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

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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

[23] 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

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

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

[26] Metralabs Gmbh, SCITOS-G5 Service Robot, www.metralabs.com

[27] Kinova Inc., Jaco robot arm, www.kinova.ca

[28] Kinova Inc., Inverse kinematics for the Jaco arm, private communication.

[29] Shadow Robot Dextrous Hand, www.shadowrobot.com

[30] Metralabs Gmbh, *Cognidrive*, xxx, 2012.

[31] 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

[32] Russell Smith, *Open Dynamics Engine*, 2005, http://www.ode.org

[33] AstroMobile project, www.echord.info/wikis/website/astromobile

[34] Robot-Era project, deliverable D2.1, *First report on the robotic services analysis with respect to elderly users*, www.robot-era.eu, 2012.

[35] Robot-Era project, deliverable D2.3, *First report on the usability and acceptability requirements*, www.robot-era.eu, 2012.

[36] Robot-Era project, deliverable D2.4, *First report on the S/T requirements of the Robot-Era components*, www.robot-era.eu, 2012.

[37] Robot-Era project, deliverable D3.1, *Report on the implementation of the AmI infrastructure modules*, www.robot-era.eu, 2012

[38] Robot-Era project, deliverable D3.2, *AmI infrastructure and reasoning services, first version*, www.robot-era.eu, 2013

[39] Robot-Era project, deliverable D4.1, *Domestic Robot Specification*, www.robot-era.eu, 2012.

[40] Robot-Era project, deliverable D5.2, *First condominium robotic platform prototype for the first experimental loop*, www.robot-era.eu, 2013