CaterROS Documentation

Release 0.1

SUTURO

Sep 30, 2017

Components of the CaterROS system

Contents:

CHAPTER 1

Overview

Vision

The basis for our SUTURO project is the vision of the CaterROS cafe:

"As Thomas enters the CaterROS cafe, he is greeted by Pepper, the smart robot. During the course of the dialogue with Pepper, Thomas gets assigned a seat. Furthermore, he has the possibility to order one or more pieces of cake. Pepper knows the kitchen roboter Raphael, a PR2, can perform this task. The order will be relayed via RPC, then Raphael can start with its task.

The uncut cake is positioned in front of the PR2. The robot takes a knife with a specifically designed handle from a holder. It then proceeds to cut out a piece of the cake and puts it on a plate.

In the meantime, Thomas can talk to Pepper, he can specify his request (order more cake, have the cake delivered to him, etc.). Peppers function will be that of a waiter.

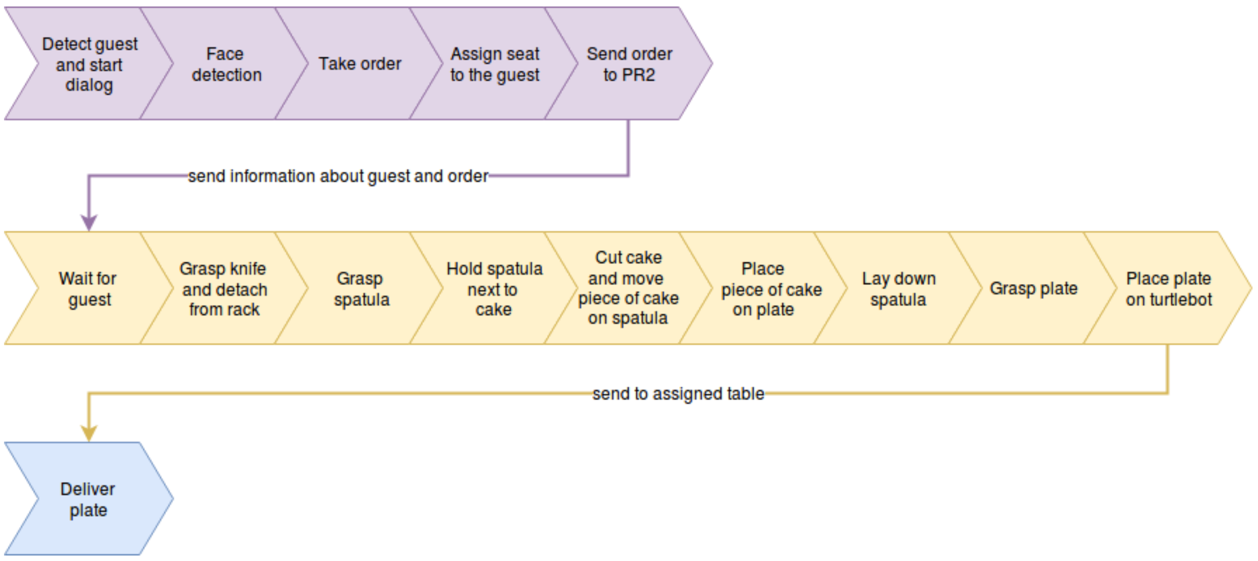
When the PR2 has cut the cake and placed it on its plate, it will send that information to Pepper. Pepper will then tell Thomas that his order is ready. Depending on Thomas' order, the plate will be delivered to him by a turtlebot, or he will be asked to get the plate himself."

Scenario

Taking our vision as a guideline, we developed all the features needed to realize the CaterROS cafe. You can see a step-by-step visualization of the resulting scenario in the following picture:

The purple steps are related to Pepper, the yellow ones to the PR2 and the blue step is related to the Turtlebot. The arrows show the flow of information between the robots.

Whenever a guest approaches, Pepper will detect them and start a dialog. Additionally, Pepper has a face detection included which enables it to recognize people it knows already. New people can tell it their name or decide to stay unknown. Guests that order an amount of pieces of cake get registered and the corresponding information (their guest id and the amount of pieces of cake they ordered) is send to the PR2. Pepper then assigns a table to the guest. The PR2 is waiting for guests to arrive and whenever it gets to know about guests that still need pieces of cake, it starts to prepare the order. Therefore, it grasps a knife from the rack and detaches it. Next, it grasps a spatula and holds it next

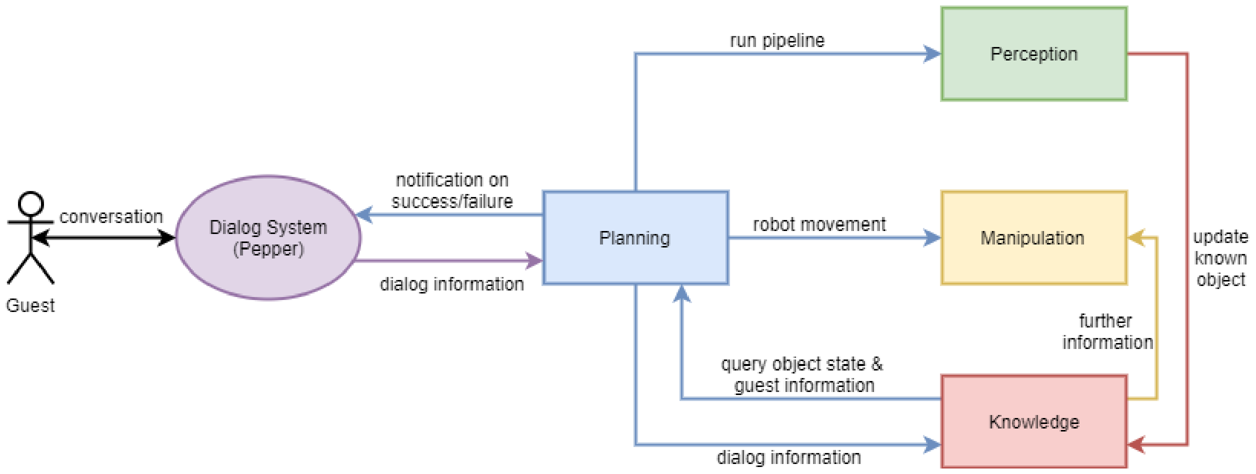

Detect guest and start dialog
Face detection
Take order
Assign seat to the guest
Send order to PR2
send information about guest and order
Wait for guest
Grasp knife and detach from rack
Grasp spatula
Hold spatula next to cake
Cut cake and move piece of cake on spatula
Place piece of cake on plate
Lay down spatula
Grasp plate
Place plate on turtlebot
send to assigned table
Deliver plate

to the cake. It then cuts a piece of cake from the cake in front of it and pushes it on the spatula using the knife. Then, it can lift the spatula, place it over the plate that is laying nearby and flips the spatula so that the piece of cake falls on the plate. The process of holding the spatula next to the cake, cutting a piece of cake and placing it on the table is repeated as many times as are needed for the given order. Afterwards, it lays down the spatula to be able to grasp the plate. After it placed the plate on the turtlebot, it sends the turtlebot the command to bring it to the table where the guest is waiting. The turtlebot drives to the given location and the guest can get the plate from it.

The only part of the scenario that is still not working properly is the preparing of orders with more than one piece of cake. The feature is already implemented, the only thing missing to make it work is the repetition of the perception process after the cutting of a piece of cake. After a piece of cake was cut, the cake gets smaller. Without knowing that, the PR2 is not able to cut any more pieces of cake properly.

Architecture

Here you can see the overall architecture of the CaterROS system:



The round figure is the module concerning Pepper, the rectangular figures are the modules concerning the PR2. The Planning module is the only one also concerning the turtlebot. The arrow direction describes the direction of information flow and the corresponding color the module actively requesting or sending information. So for example the blue arrow from Knowledge to Planning means that Planning asks Knowledge about object or guest information and receives them from Knowledge.

The Dialog System running on Pepper is the user interface of the CaterROS system. This module enables the robot to recognize faces and to understand spoken words in realtime. Whenever an order is made, the Dialog System sends the corresponding information to the Planning module. Planning is responsible for the management within the CaterROS system. It organizes the tasks needed to be executed by the PR2 and the turtlebot to fullfill a particular goal: serving a given amount of pieces of cake to a specific table. Therefore it uses the interfaces of the other groups. The information concerning the order is sent from Planning to Knowledge, the module that contains and manages all information extracted from the robots' environment, including data about perceived objects as well as guest and order data. Objects within the reach of the PR2's camera are recognized by the pipeline of the Perception module. The pipeline can be started by the planning module whenever an update of the perceived environment is needed. Knowledge saves the corresponding data and makes it accessible for Planning. To make the PR2 act, Planning uses the controllers provides by the Manipulation module. Using the information received from Knowledge, Planning can call the controllers for grasping objects, cutting cake, etc. The Manipulation's controllers use additional information from the Knowledge module. The navigation of the turtlebot is, on the other hand, realized by Planning itself.

CHAPTER 2

Installation: Setting up basic components

This article provides a step-by-step guide to the basic setup needed to install and run the Caterros project components.

General installation & setup

Install ROS

Install and setup ROS indigo on Ubuntu 14.04. Therefore, follow the instructions from http://wiki.ros.org/indigo/Installation/Ubuntu. Recommended is to install the package ros-indigo-desktop-full.

Install additional packages

1. Install needed tools:

```
sudo apt-get install ros-indigo-catkin ros-indigo-rospack python-wstool
```

2. Install rosjava:

```
sudo apt-get install ros-indigo-rosjava ros-indigo-rosjava-*
```

3. Install the PR2 navigation:

```
sudo apt-get install ros-indigo-pr2-navigation
```

Setup your Workspace

1. Prepare a catkin workspace for the project (see http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment for further information):

```
mkdir -p ~/suturo_ws/src
cd ~/suturo_ws/
catkin_make
source devel/setup.bash
```

2. Prepare another catkin workspace for the project dependencies:

```
mkdir -p ~/suturo_dep/src
cd ~/suturo_dep/
catkin_make
source devel/setup.bash
```

Clone repositories

Clone the git repositories of all the modules you need on your machine into the src folder of your project workspace. To run the whole project, each module is needed but they don't have to run on the same machine. The corresponding git command is:

```
git clone <URL>
```

You can find all the URLs in the following table:

Name	SSH	HTTPS
planning	git@github.com:suturo16/planning.git	https://github.com/suturo16/planning.git
perception	git@github.com:suturo16/perception.git	https://github.com/suturo16/perception.git
knowledge	git@github.com:suturo16/knowledge.git	https://github.com/suturo16/knowledge.git
manipulation	git@github.com:suturo16/manipulation.git	https://github.com/suturo16/manipulation.git
suturo_msgs	git@github.com:suturo16/suturo_msgs.git	https://github.com/suturo16/suturo_msgs.git

After you have set up the corresponding dependencies, you can build your workspace again:

```
cd ~/suturo_ws/src
catkin_make
```

The description of the installation of each module can be found in the corresponding chapter.

suturo_msgs

This module is needed by all the other modules, so clone it first into the src folder of your project workspace.

CHAPTER 3

Perception

The purpose of perception in this project is to detect various objects and facts in or about the kitchen.

Tools for serving cake, the cake itself and the position of the robot in the kitchen are detected.

For data acquisition a kinect on the robots head and a laserscanner at the base of the robot are used.

Installation

The framework used for detection is RoboSherlock, for which installation instructions can be found here: http://robosherlock.org/install.html

(Also needed is PCL 1.8, for which installation instructions can be found here: https://github.com/hsean/Capstone-44-Object-Segmentation/wiki/PCL-1.8:-Ubuntu-14.04-Installation-Guide)

To get the pipeline itself, one needs to clone the perception repo of the Caterros project: https://github.com/suturo16/perception

The normal pipeline can be launched with *roslaunch percepteros cateros.launch*.

The plate pipeline needs a RoboSherlock annotator that is currently only existing in a pull request here: https://github.com/RoboSherlock/robosherlock/pull/86

CaterrosRun

Implemented classes: CaterrosRun, CaterrosControlledAnalysisEngine, CaterrosPipelineManager

How to execute: *rosrun percepteros caterrosRun caterros*

This is a customized version of the robosherlock run functionality. It starts RoboSherlock while offering the possibility to change the executed Pipeline during runtime. A service is offered at percepteros/set_pipeline, which takes a list of object names for which it will then start a customized pipeline. Currently only one object is supported, this can however easily be expanded by adding appropriate pipelines.

Pipelines are defined as yaml files in the config folder. In order to let the system know a new pipeline exists, the name of the pipeline has to be added to the setPipelineCallback function in CaterrosPipelineManager, where object names are mapped to config files.

Initially, all Annotators have to be known to RoboSherlock. They can be defined in an analysis engine xml. The xml used in the SUTURO Project is caterros.xml.

Annotators

BoardAnnotator

Uses results from the following annotators: CakeAnnotator.

Results in: Recognition annotation and pose annotation for board.

How to use: Is present in the pipeline for cake detection, which can be started by *rosrun percepteros caterrosRun cake*

Changeable parameters (in perception/percepteros/descriptors/annotators/BoardAnnotator.xml):

- None

The BoardAnnotator uses the pose of the cake to determine the possible places for the board (must be directly the cake) and then searches for circles in the vincinity.

ColorClusterer

Uses results from the following annotators: PointCloudClusterExtractor.

Results in: Rack annotation for board, ToolAnnotations for ColorClusters found on the board.

How to use: Is present in the pipeline for knife detection, which can be started with *rosrun percepteros caterrosRun knife*

Changeable parameters (in perception/percepteros/descriptors/annotators/ColorClusterer.xml):

- minHue(Standard: 150) - Minimal hue value points must have to pass as belonging to the rack.
- maxHue(Standard: 360) - Maximal hue value points can have to pass as belonging to the rack.
- diffHue(Standard: 7) - Difference of hue values used in color clustering on the rack.
- diffVal(Standard: 10) - Difference of value values used in color clustering on the rack.
- diffDist(Standard: 0.01) - Cutoff value of point distance used in color clustering on the rack.
- minPoints(Standard: 3000) - Minimal number of points of correct color to make a cluster a rack.
- minCluster(Standard: 1000) - Minimal number of points for clusters found in color clustering on the rack.

The ColorClusterer checks all clusters if they have enough points of the rack color, and thus finds the rack.

The rack is enriched with a rack annotation for average surface normal.

The rack is then again clustered by color, to detect the tool clusters. Theses clusters are new and are thus added to the scene, whereby they get an tool annotation with average hue and value.

KnifeAnnotator

Uses results from the following annotators: ColorClusterer.

Results in: Recognition annotation and pose annotation for knife cluster.

How to use: Is present in the pipeline for knife detection, which can be started with *rosrun percepteros caterrosRun knife*

Changeable parameters (in perception/percepteros/descriptors/annotators/KnifeAnnotator.xml):

- minHue(Standard: 40) - Minimal hue value a tool cluster must have in order to be considered a knife.
- maxHue(Standard: 70) - Maximal hue value a tool cluster can have in order to be considered a knife.

The KnifeAnnotator checks all tool clusters for the correct color of the knife (yellow) and calculates the right pose for the knife.

PlateAnnotator

Uses results from the following annotators: PointCloudColorSegmentation, PrimitiveShapeAnnotation.

Results in: Recognition annotation and pose annotation for plates.

How to use: Is present in the pipeline for plate detection, which can be started with *rosrun percepteros caterrosRun plate*

Changeable parameters (in perception/percepteros/descriptors/annotators/PlateAnnotator.xml):

- minHue(Standard: 100) - Minimal hue value a cluster must have in order to be considered a plate.
- maxHue(Standard: 360) - Maximal hue value a cluster can have in order to be considered a plate.

The PlateAnnotator checks all color clusters for detected circles, and tries to fit a second circle into the cluster.

If both circles are found and fulfill some criteria the cluster is assumed to be a plate.

CakeAnnotator

Implemented classes: CakeAnnotator

Uses results from the following annotators: PointCloudClusterExtractor, ClusterColorHistogramCalculator

Requirement: Localized robot

Results in: Recognition annotation and pose annotation for boxes.

The core functionality of this module is to detect boxes of the color which is specified in the Annotator xml file. In order to classify an object as a box, there need to be 3 visible planes which satisfy a number of constraints. The biggest plane is found first, the 2 subsequent planes are each smaller than it's predecessor. Both of the smaller planes need to be perpendicular to the biggest plane, and the smallest plane also needs to be perpendicular to the second biggest plane.

TODO Bild

Cakes are always assumed to be standing on a table which results in their z-axis pointing in the same direction as the z-axis of the map frame. Therefore the z-axis of the first plane is restricted to be the z-axis of the map.

SpatulaRecognition

Implemented classes: SpatulaRecognition

Uses results from the following annotators: PointCloudClusterExtractor

Requirement: None

Results in: Recognition annotation and pose annotation for the spatula.

The core functionality of this module is to detect the spatula according to a parameter set. The Annotator works the following way: it analyses all the clusters and identifies the first cluster with sufficently close parameters as a spatula. The parameters are the eigenvalues from a 3D principal componant analysis and the hue value. The euclidean distance serves as a distance measure. As the only the handle of the spatula is recognized as a cluster, the first eigenvector and the up-vector of the scene serve as a basis for the axis computation.

ObjectRegionFilter

Implemented classes: ObjectRegionFilter

Uses results from the following annotators: none

Requirement: located Robot

Results in: Filtered point cloud around specified object location.

This Annotator was designed to stabilize the perception of objects whose approximate location is known beforhand. It operates on the following parameters given in a specified yaml file:

regionID: this specifies for every pipeline initiation which region parameters should be used

viewsToProcess: the name of the cas view to process although in its current state only point clouds with a pcl::PointXYZRGBA type are valid

region center: [x, y, z] coordinates of the region relative to the head_mount_kinect_rgb_optical_frame frame

range: the width of the region for each axis

ROSPublisher

Message type:

Topic name:

The ROSPublisher advertises a topic on the ROS Network. On this topic it publishes all objects with a recognition annotation. Contained in the published message are the object pose as well as the name, type and dimensions as needed.

ChangeDetector

The change detector is an experimental feature which calculates the changed clusters between two pointclouds. There are two different methods for this implemented in SzeneRecorder.cpp. The first Method uses Octrees and the pcl function OctreeChangeDetection to find voxels in the second Octree which were not present in the first Octree. The distance of the new Voxels is then checked against the distance in the first image in order to decide, whether a voxel is new because it was occluded or because it was added to the scene. The second approach uses the depth images and applies a threshold to the difference between the two images. The contours of the resulting binary image are then used to calculate the clusters, which are again annotated to be newly added or formerly occluded.

CHAPTER 4

Dialog System

This software allows the Pepper robot to hold a human-like conversation in the SUTURO16 Project from the Institute of Artificial Intelligence -University of Bremen. The task of the robot in the project consists in taking care of the clients in a Cafe. The robot welcomes the clients, informs them about the Cafe services, takes their orders, forwards them to the robot baker and informs the clients on the evolution of their requests.

Architecture

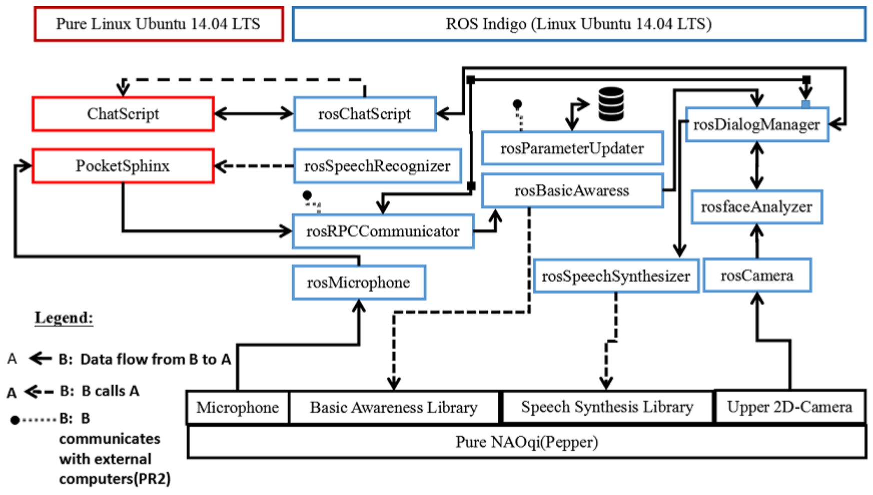


Fig. 4.1: *Fig1. Dialog System's Architecture*

As we can see from the above architecture, the Dialog System is highly heterogeneous from the os perspective.

- **NAOqi OS**: operating system of the target robot(Pepper). Pure libraries were needed for robot control access.
- **ROS Indigo**: needed for a more efficient management of the Dialog System's components and their intercommunication. Moreover, this packaging of components into ros nodes allows the Dialog System to interact with

ROS environments. Just to recall that ROS Indigo is a virtual OS running on top of Linux Ubuntu.

- **Linux Ubuntu 14.04 LTS**: needed to run critical components which could neither be completely built from scratch, neither be appropriately adapted according to a chosen OS in the deadlines of the project. They could only be appropriately adapted to the project without changing the target OS to run them.

Consequently, the simplest way to launch the Dialog system consists in installing and running it remotely on any Linux Ubuntu 14.04 LTS platform.

Ros-based Image Streaming

This module is represented by the component **rosCamera** in the architecture and accessible at rosCamera.py. It samples images from the upper 2D-camera of Pepper, converts them into ROS images and publishes them over the ROS Image topic. The ROS parameters of this nodes are accessible at dialog.launch and follow:

- **VIDEOMODE**: indicates the position of the target camera. Value is **local** for a pc-webcam or **remote** for a robot camera
- **PEPPERIP**: indicates the Ip address of Pepper
- **PEPPERPORT**: indicates the port, Pepper should be accessed through

Face Recognition

This module is represented by the component **rosfaceAnalyzer** in the architecture and accessible at faceAnalyzer.py. It subscribes to ROS Image topic described above, detects faces based on Haar-like features described in haarcascadeFrontalfaceDefault.xml, introduces the detected faces into the dataset at faces and the detections into the folder detection. Then, it trains the classifier for face recognition from time to time with the data from faces. This module communicates with the *Dialog Manager* through suitably defined ROS messages, servers and actions accessible at dialogsystemMsgs. The ROS parameters of this nodes are accessible at dialog.launch and follow:

- **CVRESET**: indicates whether the module should be reinitialized at start or not. Value is **off** for volatile mode(all data lost on stop) and **on** for permanent mode(the model is preserved even on stop)
- **PATH_TO_DATASET**: absolute path to dataset folder. Any image must be saved with the format *personName_id_instance.jpg*. Example: franklin_1_3.jpg is the third image of the person having Franklin as name and 1 as Identification number.
- **PATH_TO_DETECTOR**: absolute path to detection folder
- **CVTHRESHOLD**: under this threshold, the recognized face is accepted
- **CVRTHRESHOLD**: under this threshold, the recognized face is considered as resemblance. Over this threshold, the recognized face is ignored
- **CVFACEWINDOW**: the minimal size of the side of the square, a detected face can fit in
- **CVDIMENSION**: length of the image after pca-based compression(number of dimensions retained)
- **CVNEIGHBOR**: minimal number of meaningfull objects that should be detected around a presumed detected face before the latter is accepted
- **CVSCALE**: for scaling images before applying the detector, because The detector was trained on fixed-size images
- **CVSCANFREQUENCY**: for consistent visualization, a moving average with a window of size CVSCANFREQUENCY over the image stream takes place
- **CVINSTANCEFREQUENCY**: maximal number of images to save per face when detected

- **CVIDIMENSIONDEFAULT**: default size of the side of the square, each detected face should fit in
- **CVIDIMENSION**: size of the side of the square, each detected face should fit in. Same as **CVIDIMENSION-DEFAULT**, but dynamic because inferred from the available sizes of faces in the dataset.

Speech Recognition

This module is represented by the component **rosSpeechRecognizer** in the architecture and accessible at sphinx-Asr.py. It sets the parameters of the pure c++ module **PocketSphinx** and starts it. **PocketSphinx** receives Speech from a Gstreamer-TCP-server, recognizes it and then publishes the result for further processing. It is accessible at continuous.cpp and was derived from CMUSphinx. The ROS parameters of this nodes are accessible at dialog.launch and follow:

- **ASRCWD**: path prefix to access **PocketSphinx**
- **MLLR**: base path to access the speaker adapter of the speech recognizer. Allows online adaptation to speaker
- **HMM**: base path to access the acoustic model of the speech recognizer
- **ASRPATH**: base path to access the speech recognizer's object file
- **TRESHOLD**: the decoded speech is only considered under this threshold
- **DATAPATH**: base path to access the dictionary and language models of the speech recognizer
- **NBTHREADS**: the number of instances of speech recognizer to execute simultaneously and then combine their results into a more accurate one. It allows an ensemble learning-based recognition
- **BEAMSIZE**: only the **BEAMSIZE** best results from the **NBTHREADS** available must be combined to get the final result
- **INDEX**: this parameter is a positive integer and is used for naming of dictionary and language models. Example: **NBTHREADS** = 2 and **INDEX** = 33, then the folder **DATAPATH** will contain the files pepper33.dic(dictionary model of first thread/instance), pepper33.lm, pepper34.dic, pepper34.lm(language model of second thread)
- **HOST**: IP address of the underlying computer
- **PORT**: port of the Gstreamer-TCP-server
- **RPCPORT**: port of the RPC server, the decoded speech will be sent to
- **ORDER**: used to synchronize starts of Gstreamer-TCP-client and Gstreamer-TCP-server. while value is 0, the Gstreamer-TCP-client must wait for Gstreamer-TCP-server to start

Gstreamer-based Audio Streaming

This module is represented by the component **rosMicrophone** in the architecture and accessible at gstreamer-Sphinx.py. It configures and starts a Gstreamer-TCP-client on Pepper, which receives audio samples from the microphone of Pepper and sends them regularly to the Gstreamer-TCP-server described above for decoding into text. The ROS parameters of this nodes are accessible at dialog.launch and follow:

- **RHOST**: indicates the IP address of the host, which the Gstreamer-TCP-client runs on. Pepper's IP by default
- **RPORT**: indicates the port, which the SSH service for launching the Gstreamer-TCP-client can be accessed through
- **RUSERNAME**: indicates the username of the user accessing the ssh service on the robot
- **PASSWORD**: indicates the password of the user accessing the ssh service on the robot

- **HOST**: indicates the IP address of the host, which the Gstreamer-TCP-client is running on
- **PORT**: indicates the port, which the Gstreamer-TCP-server is listening to
- **ORDER**: used to synchronize starts of Gstreamer-TCP-client and Gstreamer-TCP-server. while value is 0, the Gstreamer-TCP-client must wait for Gstreamer-TCP-server to start

Basic Awareness

This module is represented by the component **rosBasicAwareness** in the architecture and accessible at speechRecognizer.py. It starts a pure NAOqi empty behavior as proxy on Pepper to get a total robot control, launches some services from the robot libraries to guarantee the basic awareness(stimuli tracking, Human detection, breathing), receives decoded speech from the RPC-server and forwards it to the dialog manager for further processing. The ROS parameters of this nodes are accessible at dialog.launch and follow:

- **PEPPERIP**: indicates the IP address of the robot Pepper
- **PEPPERPORT**: indicates the port, which Pepper is accessed through
- **NAOQIPACKAGEUUID**: indicates the identification number of the empty behavior on the robot
- **PATHTOBEHAVIOR**: indicates the path to the empty behavior given **NAOQIPACKAGEUUID** on the robot
- **busy**: used to clearly distinguish the speaking phases from the hearing phases of the robot. If value is 1(*robot is speaking*), then **rosBasicAwareness** ignores results from speech recognizer. reset to 0 after speaking

ChatScript

This module is represented by the component **rosChatScript** in the architecture and accessible at dialogCoreServerManager.py. It starts and interacts with the **ChatScript** platform accessible at ChatScript. The **ChatScript** platform is a text-based natural language processing toolkit. It provides us with a language to completely specify the core of the Dialog System(understanding, dialog flow control, answer generation) and a server-like interpreter of those specifications. In the Dialog System's pipeline, **rosChatScript** acts like a bridge between the speech recognition and the speech synthesis through the dialog manager. The ROS parameters of this nodes are accessible at dialog.launch and follow:

- **CORESERVERIP**: indicates the IP address of the host, which the server-like interpreter of **ChatScript** runs on
- **CORESERVERPORT**: indicates the port, which the server-like interpreter is accessed through
- **CORESERVERCWD**: indicates the absolute path to ChatScript's folder
- **CORESERVERPATH**: indicates the relative path from **CORESERVERCWD** to ChatScript's object file
- **PATH_TO_USERDIALOGDATA**: indicates relative path to dialog-related user data's folder

Dialog Management

This module is represented by the component rosDialogManager in the architecture and accessible at dialogManager.py. It serves as router to all other components for their connections and communications. The module has currently no ROS parameters at dialog.launch.

Speech Synthesis

This module is represented by the component **rosSpeechSynthesis** in the architecture and accessible at naoSpeech.py. It receives textual outputs from the **rosDialogManager** module and then calls pure NAOqi libraries to synthesize speech from the input text. The ROS parameters are accessible at dialog.launch and follow:

- **PEPPERIP**: indicates the IP address of the robot Pepper
- **PEPPERPORT**: indicates the port, which Pepper is accessed through
- **busy**: used to clearly distinguish the speaking phases from the hearing phases of the robot. If value is 1(*robot is speaking*), then **rosBasicAwareness** ignores results from speech recognizer. reset to 0 after speaking

RPC-Client

This module is the client part of the component **rosRPCCommunicator** in the architecture and accessible at rpc-Client.py. It directly receives from a topic requests published by the dialog manager and forwards them through RPC calls to the robot PR2. The ROS parameters are accessible at dialog.launch and follow:

- **PR2IP**: indicates the IP address of the robot PR2
- **PR2PORT**: indicates the port, which PR2 is accessed through

RPC-Server

This module is the server part of the component **rosRPCCommunicator** in the architecture and accessible at rpc-Server.py. On the one hand, It directly receives textual outputs from the speech recognition, then retrieves from the received text structured information thank to the Dialog System's **utility** component and forwards the structured information to the **rosBasicAwareness** module. On the other hand, it receives feedbacks from the robot PR2 and forwards them to the dialog manager. The ROS parameters are accessible at dialog.launch and follow:

- **RPCSERVERIP**: indicates the IP address of the host, which this server runs on
- **RPCSERVERIPPORT**: indicates the port, which this server is accessed through
- **FOLDER**: indicates the absolute path to the dataset (set of sentences, expressions and words) for speech recognition

Network Parameter Update

This module is represented by the component **rosParameterUpdater** in the architecture and accessible at netparamupdater.py. It sleeps and wakes up at regular intervals of time to silently update network parameters(changing permanently) such as IP addresses and ports of hosts and programs taking place in the whole project *SUTURO* from the inside environment(Dialog System, Pepper) as well as from the outside environment(Perception, Planning, Knowledge, Manipulation, PR2). It presents RPC-server-like and RPC-client-like functionalities to send and receive updates. The updates do not require any restart of the programs or computers. The ROS parameters are accessible at dialog.launch and follow:

- **PR2IP**: indicates the IP address of the robot PR2
- **PR2PORT**: indicates the port, which PR2 is accessed through
- **PEPPERIP**: indicates the IP address of the robot Pepper

- **PEPPERPORT**: indicates the port, which Pepper is accessed through
- **RPCSERVERIP**: indicates the IP address of the host, which this module runs on
- **RPCSERVERIPPORT**: indicates the port, which this module is accessed through

Utility

This module acts as proper library of the Dialog System and is accessible at utility.py. It provides the above described components with a set of mathematical functionalities. It generates several datasets(set of sentences, expressions and words) from a single dataset in order to implement an Ensemble-Learning technique for speech recognition. Moreover, it implements a vector space classifier for information retrieval during the speech recognition and a couple of encoding-decoding algorithms for a bijection NxN to N. The module has currently no ROS parameters at dialog.launch.

Prerequisites, Installation and Start

As prerequisites,

- Linux Ubuntu 14.04 LTS 64bits
- Python 2.7 64bits
- ROS Indigo

To install,

- Create a general workspace folder and name it as you want. Let say **Dialog**
- Clone the pepper-dialog's git repository in the general workspace folder **Dialog**
- Copy the installation file installer.sh of the pepper-dialog's repository to the general workspace folder **Dialog**
- Download the package pynaoqi SDK version 2.5.5.5 from the Aldebaran-Softbank Robotics's website. You may need to create a user account before downloading the tar.gz package. The download folder must neither be **pepper-dialog** nor inside it.
- Open the file *installer.sh* in **Dialog** and set the environment variable **PYTHON_NAOQI_TAR_GZ_PATH** to the above downloaded package's absolute file path
- Download and install Choregraphe version 2.5.5.5 from the Aldebaran-Softbank Robotics's website.
- Install the pure NAOqi package suturo16-0.0.0.pkg on Pepper robot using Choregraphe 2.5.*
- Run the installer: **./installer.sh**

To start,

- Make sure the parameters are correctly set at dialog.launch. The Ip addresses and ports should be imperatively adapted.
- run the launcher located in folder **Dialog/pepperdialog**: **./launcher.sh**

CHAPTER 5

Manipulation

Welcome to the documentation page of the manipulation components used in the CaterROS project! On this page we provide installation instructions for the system, an overview over our components and instructions on how to use them.

Installation

To install the manipulation system to you workspace, you need clone the repository to the source folder of your workspace. After that you can use *wstool* to install the needed dependencies.

Listing 5.1: HTTPS

```
cd ~/path_to_my_ws/src
git clone https://github.com/suturo16/manipulation.git
wstool merge manipulation/dependencies.rosinstall
wstool update
```

If you have an ssh key setup for the use with GitHub, you can also use the ssh to clone the repository and its dependencies.

Listing 5.2: SSH

```
cd ~/path_to_my_ws/src
git clone git@github.com:suturo16/manipulation.git
wstool merge manipulation/ssh.rosinstall
wstool update
```

All that's left to do now, is build your workspace once using catkin.

System Overview and Usage

Overall the system provides the ability to use the constraint based motion frame work *Giskard* to move the PR2 robot, as well as a variety of controllers needed for the CaterROS scenario. The heart of the system is our custom action server. It requires a controller and assignments for the conroller's parameters as goals and uses these to generate commands to move the robot. During the execution, the server outputs feedback and debug information. The feedback includes a value calculated within the controller that rates how well the overall goal of the controller is satisfied. Additionally, the repository provides a couple of tools for manually testing controllers.

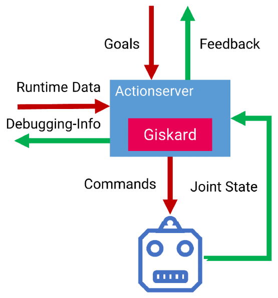


Fig. 5.1: An overview of the structure and communication of the suturo action server.

Using the Action Server

The action server executable is provided by the `suturo_action_server` package and is called `suturo_action_server`. On startup it loads its configuration and a robot description from the parameter server. The configuration must be provided by the parameter `config` within the node's namespace and is parsed as a yaml configuration. The configuration can declare exceptions for the handling of specific joints.

Listing 5.3: Example configuration for the PR2

```
position_controllers:
    head_tilt_joint: /head_tilt_position_controller/command
    head_pan_joint: /head_pan_position_controller/command
gripper_controllers:
    r_gripper_joint: /r_gripper_controller/command
    l_gripper_joint: /l_gripper_controller/command
joint_state_command_topic: /js_command
visualization_target_frame: base_link
cpqs_target_frame: base_link
```

In some cases it might be neccessary to control certain joints using position commands instead of velocitiy commands. These joints can be specified using `position_controllers` map of the configuration. In this map the names of the names of joints are associated with the names of the topics on which the position commands should be published.

Another exception that can be managed by the action server are grippers. These are assumed to be controlled using the *control_msgs/GripperCommand* message. The joints that are to be treated as grippers, can be specified in the `gripper_controllers` map.

The parameter `joint_state_command_topic` specifies a topic on which the commands are published as *sensor_msgs/JointState*. This is mainly a feature to make the server compatible with the current version of the iai_naive_kinematics_sim.

The parameter `visualization_target_frame` determines the name of the frame that all threedimensional visualization will be published in. By default this value is set to `base_link`.

The parameter `cpqs_target_frame` determines the name of the frame that the in-scene points calculated by the closest point query system should be relative to. By default this value is set to `base_link`.

Executing a Controller

The action server requires a goal of type *suturo_manipulation_msgs/MoveRobotGoal*.

Listing 5.4: suturo_manipulation_msgs/MoveRobotGoal

```
string[] controlled_joints  #All the joints to use for this action
string controller_yaml      #The complete content of the generated controller yaml
string feedbackValue
suturo_manipulation_msgs/TypedParam[] params
```

The argument `controlled_yaml` contains the controller string. `feedbackValue` specifies the name of a scalar value within the controller that is included in the feedback message. The parameter assignment for the controller is defined in the `params` list. The argument `controlled_joints` can be ignored, as it only exists for legacy reasons.

Parameter Assignment

Parameter assignments are transmitted to the action server in the form of a *suturo_manipulation_msgs/TypedParam*.

Listing 5.5: suturo_manipulation_msgs/TypedParam

```
uint8 DOUBLE=0       # Scalar data
uint8 TRANSFORM=1    # Transformation data
uint8 ELAPSEDTIME=2  # Time since the start of controller's execution in seconds
uint8 VECTOR=3       # Vector data
uint8 VISUALIZE=4    # Parameter contains a visualization rule

bool isConst     # Is the value constant
uint8 type       # Type of the parameter
string name      # Name of input in controller
string value     # Value of this assignment
```

Parameters have a name, a type, a flag marking them as constant and a value. If the parameter is supposed to be assigned to an input of a controller, it's name has to match the input's name. In this case the type argument also to match the type of the input. Out of Giskard's four elementary datatypes, three are supported at the moment: Scalars, vectors and transformations. Each parameter can be marked as either constant or dynamic. Constant parameters are only assigned once, during the start of the controller, and will not be updated again. Dynamic parameters are updated during every controller update cycle. Currently, only dynamic updates for transformations are supported.

Important: Joints are handled automatically! They are updated each control cycle even when they are not being controlled.

The way the action server interprets the `value` string of the message depends on the parameter's type and its `isConst` flag. The formatting for constant values is the following:

- Scalar: `VALUE`, e.g. `1.0`
- Vector: `X Y Z`, e.g. `1 2.5 -3.9`
- Transformation: `X Y Z AX AY AZ ANGLE`, e.g. `1 2.5 -3.9 1 0 0 3.1415`

Dynamic transformation parameters cause the server to do a TF-lookup each update cycle. The frame's names are encoded as `TARGET_FRAME SOURCE_FRAME`.

A special type of parameter is the clock parameter. The clock assigns the elapsed time since the start of the current controller to a scalar input in the controller. This value can be used to generate motions based on the passing of time.

An example could be a nodding or waving motion. It should be noted that this parameter will only be interpreted when it's `isConst` flag is set to `false`. The `value` attribute of the parameter is not used.

Additionally to the parameters assigning values to controller inputs, there are parameters that can be used to configure the action server.

The first of these special parameters can be used to configure the effort that should be used to command a gripper. To set this effort, the name of the parameter has to match the name of a controlled gripper. The parameter's value will then be decoded as scalar and used to fill the `effort` field of the *control_msgs/GripperCommand* message.

The second type of special parameters is marked by the type value `VISUALIZE`. These parameters define visualization rules for values from the controller's scope. The visualization is refreshed during each controller update cycle. The visualization system supports scalars, vectors and transformations. Vectors and transformations are visualized using *visualization_msgs/MarkerArray* messages which are published on the topic `visualization` within the server's namespace. Scalars are published in the form of *suturo_manipulation_msgs/Float64Map* on the topic `debug_scalar` within the server's namespace.

The `value` field of the parameter specifies which value should be visualized. Frames are always visualized relative to the frame named by `visualization_target_frame`. Vectors can be visualized either as points, also relative to `visualization_target_frame`, or as vectors. To do so, the name of another vector must be provided that will serve as base point for the direction vector. This done by seperating the names of the vectors in the `value` string with a space, e.g. `DIRECTION BASE`. The vector will also be visualized relative to `visualization_target_frame`.

Automated Behavior

The action server has some automated behavior that is triggered by the successful construction of a new controller.

The first part of this behavior aims to aid developers with the debugging of controllers by allowing them to specify values to visualize directly within the controller's code. This way, the visualization is always active without the need to always transmit the visualization parameters for each controller. Because Giskard does currently not support the attachment of custom data to the controllers, the action server recognizes the values to visualize by their names. When a value's name begins with the prefix `VIS__` the action server will mark it for visualization. In the visualized data, the value's name will lose the prefix. The name of the base point for the visualization of vectors is also specified in the value's name and is separated by a double underscore (__).

So to visualize the *Z* unit vector with the name *up*, you'd put the line `VIS__up__ZERO: unitZ` in the scope of your yaml controller file. Prerequisites for this to work are of course that the vectors `ZERO` and `unitZ` are defined within your controller.

The second automated behavior of the server tries to ease the usage of collision avoidance in controllers. A subsystem of the server provides a very simple query system for finding points in the environment that are closest to the robot. After finding these points, the algorithm provides the point closest to a link and the corresponding point on the link's surface. These two points can then be used inside of controllers for very simple collision avoidance.

The action server uses the names of a controller's inputs to automatically determine which link's of the robot should be queried for by the algorithm. If an input is meant to be used for collision avoidance, it's name must follow this naming convention `COLL:(L|W):[LINK NAME]`. The `L` and `W` define whether this input is supplying the point on the robot's link or in the world. The `LINK NAME` needs to be the name of a link defined by the robot's URDF.

Closest Point Query System

The closest point query system finds the closest points between a robot link and the environment. The environment is represented as point clouds, which are converted to octrees to accelerate query process. The query process is executed once every controller update cycle and its results are input into the current controller through designated inputs. The points can be used for a very simple collision avoidance mechanism, that enforces a safety margin between the points.

Testing and Tools

The *manipulation* repository contains a few tools for easier development and testing of controllers. This section will introduce them.

Client for Easy Controller Testing

In addition to the action server itself, the *suturo_action_server* package also provides a simple test client that loads parameter assignments from yaml files. This way users don't have to build the goal messages for the action server by hand for every test.

Listing 5.6: Call Pattern for the Test Client

```
rosrun suturo_action_server client_test <Controller File> <Parameters.yaml> <Feedback␣
↪Value>
```

The parameters' file must contain a list of yaml dictionaries, which have to match the following keys:

Listing 5.7: Parameter Dictionary in yaml

```
name: <String>
type: double | transform | elapsedtime | vector | visualize
const: true | false
value: <String>
```

The argument `Feedback Value` provides the name of the value that is logged to the action server's feedback topic.

During our controller development, we defined aliases wihtin our bash environment for common client runs. This way we could test the manipulation aspect of our system without the need for higher level components.

Mass Checking of Controllers

The *suturo_action_tester* package provides the *controller_checker* executable that recursively searches a folder for controller files and uses them to generate controllers. If problems arise during this process, the error messages are displayed. At the end of the execution, the number of all found controller files and the number of successfully generated controllers are displayed. All of the successfully generated controller's inputs are logged to a file called *controller_interfaces*, which is created in the directory, the checker is executed in. The tool was initially developed to easily find controllers which were faulty because of a giskard language update. The checker currently uses the yaml language, as well as the two custom languages developed for the CaterROS project.

Listing 5.8: Call Pattern for the Controller Checker

```
rosrun suturo_action_tester controller_checker <PATH 1> <PATH 2> ....
```

Mockup Environment

Aside from the controller checker, the *suturo_action_tester* package also provides a Python node called *ObstacleScene* and an RVIZ panel called *Suturo Simulation Panel*. Together, these two provide a very simple test environment that allows users to build and save scenarios using interactive markers in RVIZ. The data of these markers is published to TF and can be used by the action server.

Note: These two systems are deprecated and only being documented here for the sake of completeness. The giskard_ide package should be used for this purpose now.

Controllers

a list including every controller with documentation can be found here

Languages

The *giskard_suturo_parser* package contains two parsers for custom controller languages that were developed during the CaterROS project.

The goal of the first language was to have a language that was more readable than the yaml language provided by the giskard library. The language is designed to work well with Python syntax highlighting in regular text editors. For brevities sake, we will refer to this language as GLang. Systems loading controllers from disk associate the suffix *.giskard* with GLang. The structure of a controller file for this language is very simple.

Listing 5.9: Structure of a GLang File

```
scope = {
    value1 = someExpression;
    ...
    valueN = someOtherExpression
}

controllableConstraints = {
    controllableConstraint(lowerLimit, upperLimit, weight, "Name1");
        controllableConstraint(lowerLimit2, upperLimit2, weight2, "Name2")
}

softConstraints = {
    softConstraint(lowerLimit, upperLimit, weight, expression, "Some name");
    softConstraint(lowerLimit2, upperLimit2, weight2, expression2, "Some other name")
}

hardConstraints = {
    hardConstraint(lowerLimit - someExpression, upperLimit - someExpression,␣
↪someExpression);
    hardConstraint(lowerLimit2 - someOtherExpression, upperLimit2 -␣
↪someOtherExpression, someOtherExpression)
}
```

The overall form is pretty simple. However note that the last entry of any structure is not followed by a semicolon. A complete list of commands and attributes supported by the language can be found here.

Warning: The GLang parser has an inconsistent evaluation order. The same operators are evaluated from left to right. Otherwise equations are evaluated from right to left. Example: `10 - 5 - 2` is evaluated to `3`; `10 + 5 - 2 + 3` is evaluated to `10` because it is bound as `10 + (5 - (2 + 3))`. A fix for this bug is contained in this commit. However it is currently not included in the master, as it will break some of our controllers.

The goal of the second language was to support the modularization of controllers and a definition of custom functions. Together these features are supposed to enable developers to build libraries for giskard controllers and ultimately use these to generate controllers automatically. We'll call it GLang++ from now on (because uncreative naming patterns are a proud tradition). GLang++ has not actually been used in controllers used by the CaterROS system. A first controller using that language has been built for the *fetch* robot. It is part of the fetch_giskard package and is named pos_controller.gpp. The file illustrates how GLang++ controllers are structured. A full list of built-in functions of the language can be found here.

CHAPTER 6

Knowledge

The knowledge modules are basically about storing information and making it accessable in an easy way. Part of these information surely are about the world and its objects. For robots it is crucial to use these information, especially about objects, in order to interact with them. Another part of the information stored is information emerging from a spoken dialog. Intentions about a dialog for example, are stored in the knowledge base.

Installation

The knowledge system depends on KnowRob and SWI-Prolog. First install SWI-Prolog

Listing 6.1: SWI-Prolog

```
sudo apt-get install swi-prolog swi-prolog-*
```

Now you need to install KnowRob. For this we use a fork of an older KnowRob Version.

Listing 6.2: Knowrob

```
rosdep update
cd ~/catkin_ws/src
wstool merge https://raw.github.com/suturo16/knowrob/master/rosinstall/knowrob-
↪base.rosinstall
wstool update
rosdep install --ignore-src --from-paths .
cd ~/catkin_ws
catkin_make
```

To install the knowledge system to your workspace, you need clone the repository to the src folder of your workspace. The easiest way to do this, is to copy the following lines into your terminal and replace the path with your local path. Make sure that you have created a workspace before executing the following commands.

Listing 6.3: setup

```
cd ~/suturo16/suturo_ws/src/
git clone git@github.com:suturo16/knowledge.git
cd ../
catkin_make
```

If you want to run all knowledge modules of the CaterROS Package, you need to execute the following command:

Listing 6.4: launch

```
roslaunch knowledge_launch suturo_knowledge_full.launch
```

System Overview and Usage

World state

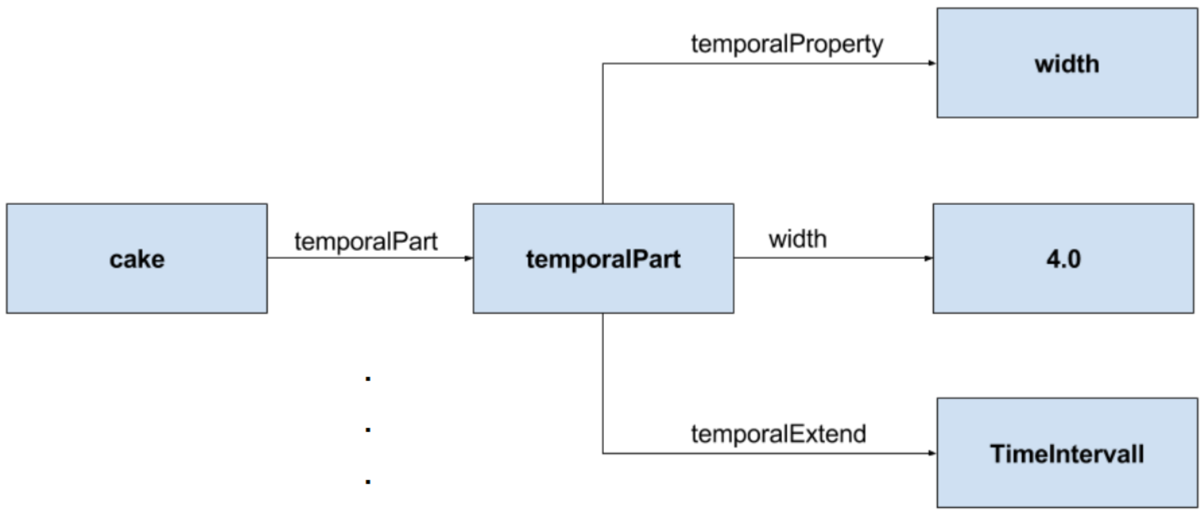
The world state comprises of the important concepts. To run the world state node call:

Listing 6.5: launchws

```
roslaunch object_state object_state.launch
```

Temporal Parts of Objects

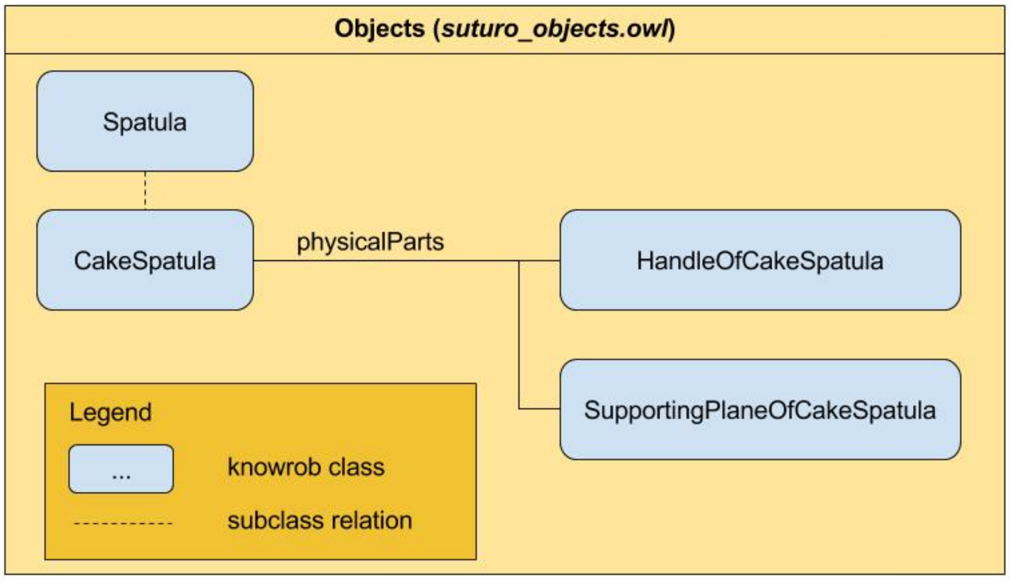
When storing spatial information about objects, it is crucial to connect these information to a certain point in time because it is the only way to enable the knowledge base to reason over new and old data respectively. In the picture below you can see how temporal parts are structured in KnowRob.



Due to this concept it is possible for objects in KnowRob to have their attributes connected to a specific point in time.

Physical Parts of Objects

To improve on the modelling concept of objects in KnowRob, physical parts were introduced. Objects now consist of subobjects. For example a cake spatula consists of two subobjects for its handle and for its supporting plane. This is useful because sometimes you want to grasp specific parts of an object. In case of the cake spatula you probably wanna grasp it at its handle, therefore it is easier to just lookup the pose of the handle instead of the object itself, which is defined as the center of the object. Despite the fact, that this way of representing objects is more plausible from a modelling side of view, it also makes it easier to store constants for specific offset values that belong to a physical part of an object. The illustration below provides an example of a cake spatula object in KnowRob.



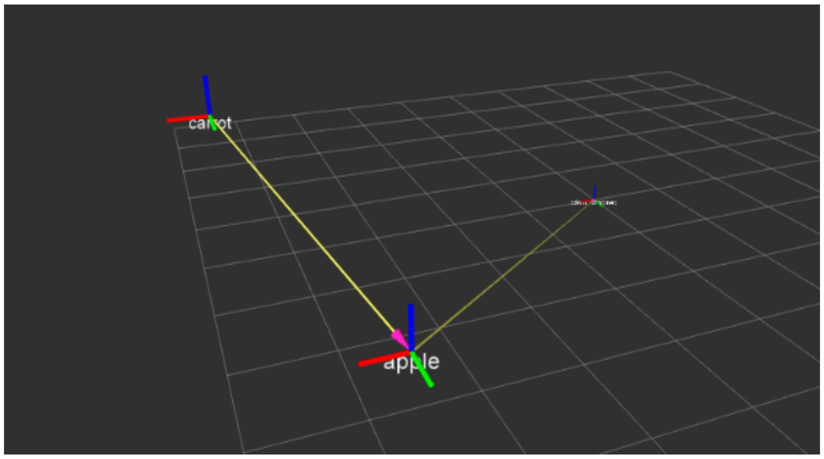
Physical parts save spatial information about themselves, like the pose for example, relative to the position of their parents. The spatial information are automatically handled by the publishing module, which will be described later in this documentation.

Connecting Frames

Since it is very difficult to recognize objects that are currently in the gripper of the robot, we came up with this solution. Once an object is successfully grasped by the PR2, we execute the `connect_frames function`. This way the knowledge base has the information that the object is gripped and more importantly, new perceptions of the grasped object type are ignored. The position of the grasped object is now published relative to the gripper position. This way the knowledge base always knows where the grasped object is. When the object is dropped, the `disconnect_frames` function allows new perceptions of the object again, so that the position of the object will automatically update itself again.

Receiving of Perceptions

The perception group publishes their perceptions to this topic: `percepteros/object\_detection`. The knowledge group implemented a subscriber for this topic to process the published data internally. The subscriber can be found in the knowledge package named `object_state/scrips/subscriber.py`. Following the flow of the subscriber, the acquired information are then internally handled by the object_state.pl module, where the KnowRob objects are created and published. The publishing of spatial information about objects is described in the next section.



Publishing of Perceptions

After creating the KnowRob representation of the perceived data, spatial information about the objects are published to the tf topic.

The python script `object_state/scripts/fluents_tf_publisher.py` creates a ROS node to transform the poses of objects. In a first step the required information for the transformation such as Name, FrameID, Pose and Orientation are queried, using the prolog function `get_tf_infos()`. The result of the transformation is forwarded to the tf topic by the publisher, making it accessable for manipulation.

CaterROS Cafeteria Modelling

Furthermore we needed an ordering system for the cafe CaterROS. For this we created a customer model in the file `suturo_customer_model.owl`. We modelled the customer who has the temporal property visit to Visit instances. Visit instances have `hasOrder` properties for each order and an property to store the table of the customer. An order stores the ordered product, the corresponding ordered amount and the coresponding delivered amount.

We extended this by a model of the CaterROS Cafe in the file `suturo_cafetaria.owl`. This contains e.g. the position of the customer tables.

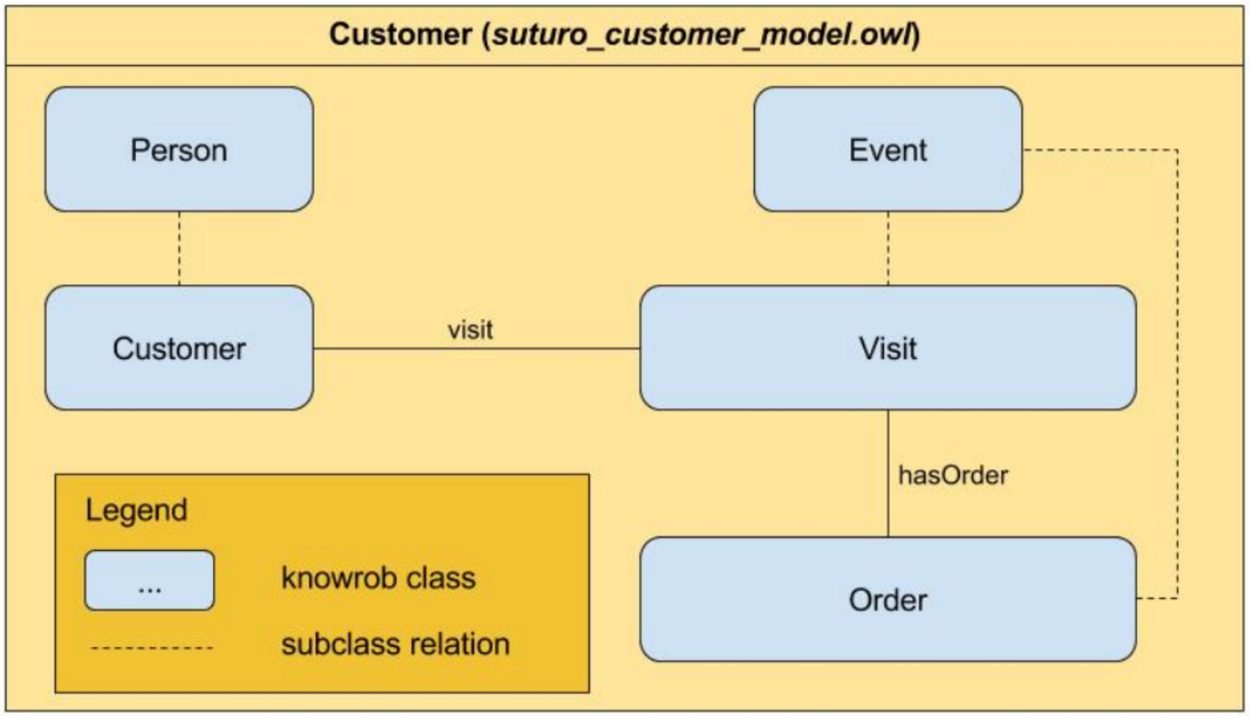
Interpretation of DialogIntention

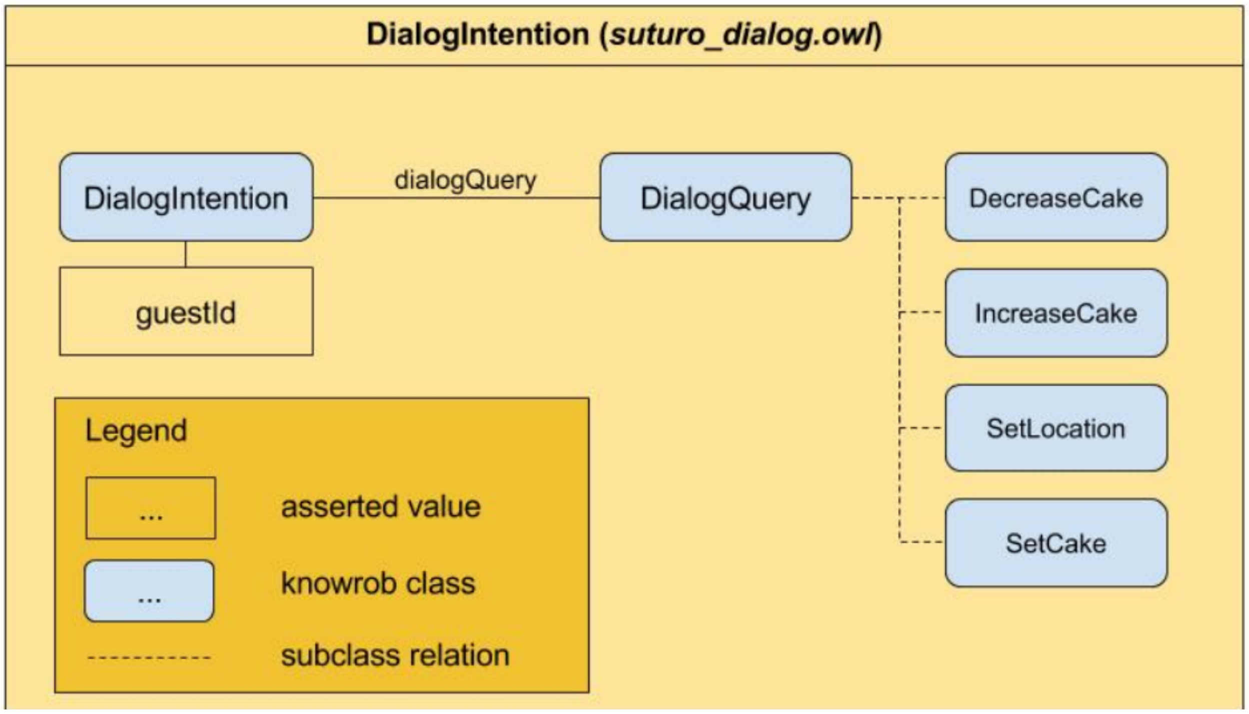
Additionally a system to align the knowledge base to the results from the dialog system was implemented. The dialog system creates a JSON-String, that contains the intention derived from the dialog of the customer and the system. This intention is then filtered by the Planning System and then parsed to create a DialogIntention Object.

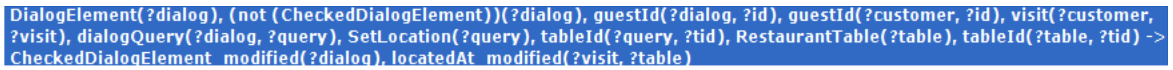
A DialogIntention objects saves the guestID and a DialogQuery. A dialog query object can be of different types, that denotes the semantic of the intention. For example the type IncreaseCake denotes the intention of the customer to order more cake.

To align the knowledge base to the DialogIntention Object we use SWRL Rules. This allows us to create dialog intentions and their interpretation solely on the modelling level, without adding new code. Here an example rule:

We only consider DialogIntentions, that weren't considered before by marking considered intentions with the Type `CheckedDialogIntention`. The dialog intention contains a query, that detemines the semantic for the interpretation. In this case the Type is SetLocation. We use the information from intention and query to gain further information of the guest and the table. We then use these Informations to project the effect into the knowledgebase: During his visit, the customer will be assigned to the defined table.






```
DialogElement(?dialog), (not (CheckedDialogElement))(?dialog), guestId(?dialog, ?id), guestId(?customer, ?id), visit(?customer,
?visit), dialogQuery(?dialog, ?query), SetLocation(?query), tableId(?query, ?tid), RestaurantTable(?table), tableId(?table, ?tid) ->
CheckedDialogElement_modified(?dialog), locatedAt_modified(?visit, ?table)
```

Developed Tools / Libraries

There are a lot of different packages for calling Java code from Prolog, but we wanted to take advantage of the scipting functionalities of Python inside our code. With that in mind, a library for the purpose of calling Python from Prolog was programmed internally. At first the library started an interpreter each time a call was made, that was clearly ineficient and after some strenuous coding and optimizing the library it got almost 30 times faster than its first version. It was now feasible and even fast making calls, thanks to Prython.

You can find the library and further documentation at https://github.com/sasjonge/Prython

For working with models and its capabilities Protégé, from the Stanford University, proved to be an excelent tool. With it it is easy to edit and view the OWL ontology files. The program was clearly not made with ROS environments in mind though, it can't handle any dynamic for example. As their code is completely open source we forked the repository and modified some parts of it to handle our dynamic paths, automatically recognizing the package path and selecting the appropriate files. That way we could spend less time looking for the right folders and more time adding new functionality.

You can find the modified Protégé and Instructions, how to install it at https://github.com/raeglan/protege

CHAPTER 7

Planning

The planning system of the CaterROS project controls all the other systems of the project and let's them work together. It uses the planning framework CRAM to control the PR2 and TurtleBot via flexible plans. The system communicates with the Pepper robot's systems to gather knowledge about the dialog with guests.

Installation

To install the planning system to your workspace, you need clone the repository to the source folder of your workspace. After that you can use wstool and rosdep to install the needed dependencies:

```
cd ~/path_to_my_ws/src
git clone https://github.com/suturo16/planning.git
wstool merge planning/dependencies.rosinstall
wstool update
rosdep install -riy --from-paths .
```

Now you can build your workspace using catkin.

Optional: Plan Generator

If you want to use the plan generator you have to install the fast downward planer from http://www.fast-downward.org/ in addtion. This package is not needed for building the planning module. You can find a detailled description of how to setup and use the fast downwards planner at http://www.fast-downward.org/ObtainingAndRunningFastDownward.

1. Create a new folder within your dependency workspace, e.g. "planner".
2. Within this folder, create a new file named "setup.py" with the following structure:

```
#!/usr/bin/env python

from distutils.core import setup

setup(name='planner',
    version='1.0',
    description='pddl planning system',
    author='someone',
    author_email='someone@stuff.net',
    url='https://www.python.org/sigs/distutils-sig/',
    packages=['downward'],
)
# You can choose arbitrary values for the given fields.
```

3. To ensure that all necessary dependencies are installed, execute:

```
sudo apt-get install cmake g++ g++-multilib mercurial make python python-pip
```

4. Then, you can clone the planer to the folder that you created in step 1:

```
cd planner
hg clone http://hg.fast-downward.org downward
```

5. Build the planner:

```
cd downward
./build.py
```

6. Create an empty file named "__init__.py" within the "downward"-folder.
7. Go to the subfolder "driver" and within the file "main.py" uncomment the line "sys.exit(exitcode)":

```
# sys.exit(exitcode)
```

   This is needed because otherwise the plan generator's server won't be able to give a return value when being called.

8. Now, you can finally install the planner as a python module. This is necessary so that the plan generator can get access to it. Go to the folder you created in step 1 and execute:

```
sudo pip install -e .
```

System Overview

The main focus of the system is of course the execution of plans. There are two ways supported by the system: Direct and Guest-centered Execution. Direct means a user of the system executes a plan by calling the appropiate function and the system halts after the completion. The guest-centered on the other hand can run permanently until explicitly stoppped. It keeps track of the current state of the PR2 and decides what to do based on the guest's orders. This enables the system to dynamically react to changes in the guest's orders. Additionally to the predefined plans, the system also can generate plans based on the orders by guests.

Architecture

Let's have a look at our architecture. The following picture depicts our modules and how they use each other.

Modules

planning_common

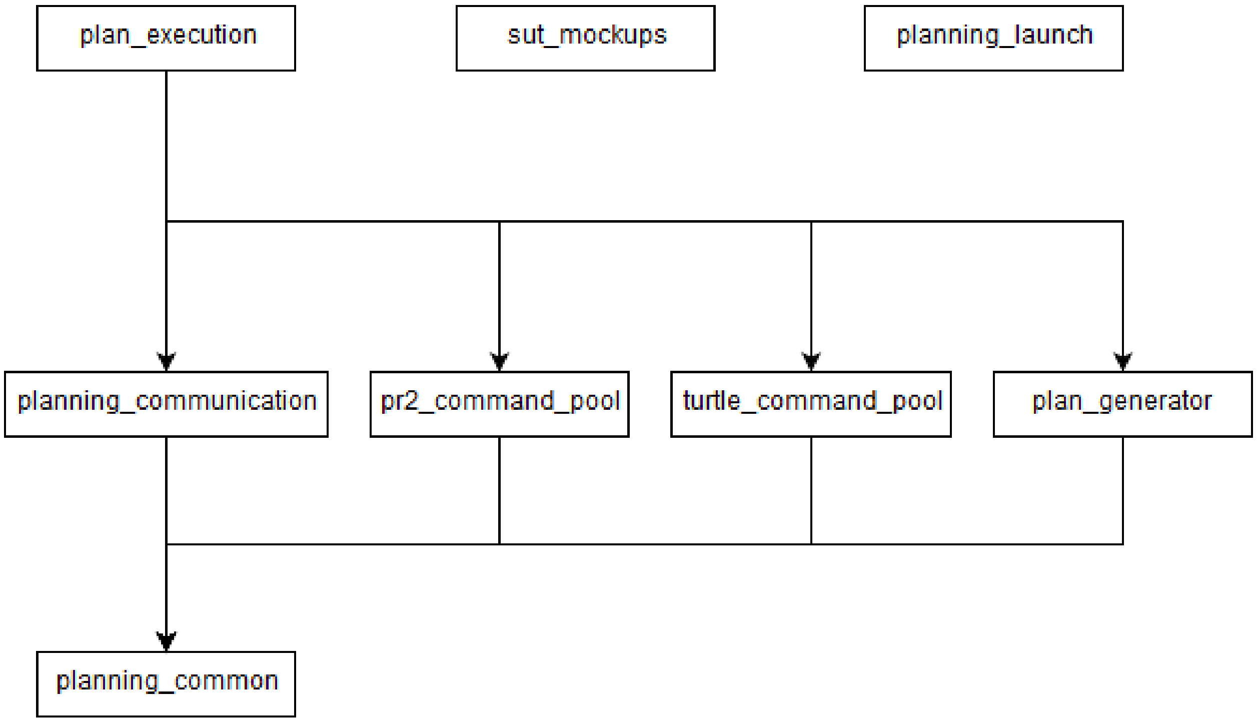


Fig. 7.1: An overview of the architecture of the CaterROS planning system.

The common module of our planning system. It holds functionality which is or was considered to be universally useful for our system. This ranges from simple utility functions and object definitions to the interface for calling prolog function of the knowledge base and robot-agnostic service calls and such. In here you can also find everything that didn't belong in the other packages but didn't warrant it's own package.

Robot-specific command pools

The robot-specific modules hold functionality especially necessary for that robot.

The pr2_command_pool package mostly holds calls to the action_move_robot function from planning_common to move the PR2. Some utility for these calls can also be found here.

The *turtle_command_pool* package holds everything which is needed to manipulate the turtle from the high-level, which means cram action an location designator definitions, the process module of the turtle and the action client.

planning_communication

The planning_communication module contains a RPC server implementation to enable the planning system to communicate with Pepper. It also contains a RPC client for the system to call functions of Pepper's RPC server. Additionally the parser for Pepper's JSON-data-structures can be found here.

plan_generator

This module consists of an Python service to call the plan generator and the Lisp interface to generate problems for the generator and a parser for the generated plans.

plan_execution

The top-level module of our system. If you want to execute our plans, you have to load the system in this package. It contains all of our plans and the utilities to execute them. This includes our CRAM process modules and designator referencing.

The loop-function for the guest-centered plan execution can also be found here.

sut_mockups

There are mockups for all major nodes of the other systems. Those can be found in this package.

planning_launch

This package only holds two launch files for launching our mockups.

Plan Architecure

As mentioned above we use the CRAM framework in our system. More specifically we use process modules and action designators to make our plans dynamic. For a more detailed look at CRAM itself check the website

In the planning_execution package are the files `toplevel`, `process-modules`, `selecting-process-modules` and `action-designators`. Those contain everything that uses CRAM in our code. In `toplevel` we have the `execute` function which takes a string and executes the corresponding task. We define a task as a set of actions, or rather action designators. Each of the action designators gets referenced by using prolog to query the knowledge base. After this they get executed by the right process module. This happens automatically, as we have defined rules in `selecting-process-modules` to match process modules to designators. The process modules then directly call the plans in `plans.lisp` with the referenced information in the designator. The plans can also call sub-plans, but they don't use any more designators. All the information a single plan needs is obtained when an action designator is first referenced.

For the guest-centered execution we have the `manager`. The name is a bit misleading, as this module doesn't actually manage the plan state or anything. It provides a function `start-caterros` which starts a loop that can be started anytime and then waits for guests to arrive. This is where the communication module comes into play. Through the RPC communication with Pepper the knowledge base gets updated parallel to the running loop. The loop can query the knowledge base through prolog.

Communication

There are three robots whose actions and knowledge are to combine. The planning_communication package prvides the communication between the Peppers dialogsystem and the ROS network, where the PR2 and Tortugabots are monitored. To include Pepper we implemented an RPC server on the Planning side, whose functions can be called from everywhere within the network, while concentrating on Peppers information. On the other hand the Planning side can feed Pepper with data and notify her about important changes in the world. To enhance the monitoring aspect of the Planning system, we save and update connection credentials of every system communicating with Plannings server.

Setup Pepper Communication

A setup file launches the RPC server and registers Peppers IP and Post to the list of available clients. Also it sends the IP and Port of the current machine to Pepper. This seput is called by the plan_execution init function, but if we want to look deeper into the planning_communication package, let's make the setup by ourselves. The final call, updating Peppers information about this machine, would fail anyway, if Peppers server isn't currently running.

RPC-Server

To initialize the RPC server, first load the planning_communication system in your REPL:

```
, r-l-s RET
planning_communication RET
RET
,!p pcomm RET
```

Now that we work in the pcomm package, run the init-function of the server:

```
(init-rpc-server)
```

This function will simply start up a new ROSnode in the REPL and register all the functions provided by our RPC interface. The core functions used by Pepper are updateObserverClient, asserDialogElement, getGuestInfo and getAllGuestInfo.

updateObserverClient takes the ID of the robot (0 for Pepper), its ip as a string and its port as a number. **assertDialogElement** takes a JSON string, that will be translated and forwarded to te knowledgebase. The whole variety of JSON queries is explained later. An example JSON string to order two pieces of cake looks like this:

```
{
guestId:1,
query:  {
        type:setCake,
        amount:2,
        guestName:Arthur
        }
}
```

The function will always answer the request with a JSON as well, telling if the request was processed successfully. This is the answer to the order sent previously:

```
{
guestId:1,
return: {
        type:setCake,
        success:1,
        tableId:table1
        }
}
```

Only upon the request of a new order (type: setCake) the response contains the tableId of the guest, every other response lacks this information.

getGuestInfo needs a guest-id and returns all information about the order identified by this specific guest-id. A common response for the guest-id 1, considering we transmitted the order above, looks like this:

```
{
guestId : 1,
return: {
        type: getGuestInfo,
        name: Arthur,
        location: table1,
        total: 2,
        delivered: 0
        }
}
```

getAllGuestInfos returns a list, containing all orders in the same format as a request for a specific guest (see **getGuestInfo**). It is called with any arbitrary parameter (there is a conflict when calling RPC function from Python to LISP, when the LISP function has no parameters).

RPC-Client

The core functionality of the RPC client is to send RPC to Pepper. Mainly we use update-connection-credentials and fire-rpc-to-client. To make those calls more developer/user friendly, we have a list of clients, that use the Planning RPC server. We can take those connection credentials to fire a call to clients, using only their keynames.

update-connection-credentials will send the IP and port of the current machine (where the Planning server is running) to a remote client identified by its keyname, or to a yet unknown client using its IP and Port. The client must have an *updateObserverClient* function implemented on their side. After this call, the remote client will have information about our server. Here is an example usage:

```
(update-connection-credentials :client :pepper)
```

fire-rpc-to-client calls a function at a remote client. It uses the clients keyname, the function name and arguments needed in the function:

```
(fire-rpc-to-client :pepper "notify")
```

Plans

We defined different plans to realize our scenario:

grasp

There is one plan that enables the PR2 to grasp various objects. How to grasp the actual object is decided on the basis of the given object type. Possible objects that can be grasped are: a knife, a plate, a spatula and a cylinder.

place-object

The plan place-object can be used hold a given object to a given location. Optionally, the object can be released so that this plan can also be used to drop objects to a given location. The given object has to be grasped already.

detach-object-from-rack

This plan is used to detach objects that should be taken from the rack. It assumes that the given object was grasped already. In our scenario, this is only used for getting the knife.

cut-object:

The plan cut-object is used to cut a given object (in our case: a cake) with a given knife. It assumes that the knife is grasped already. Additionally, a target can be defined optionally. If a target is given, the slice that was cut is moved there. In our scenario, we pass the spatula as a target so that the PR2 pushs the piece of cake onto it after it was cut.

move-n-flip:

Move-n-flip is used to move a given tool to a given location and then flip it. In our scenario, we use it to drop the piece of cake on the plate after it was pushed on the spatula.

Plan Generation

The plan_generator module allows to dynamically generate a sequence of actions that are needed to fulfill a given goal (in our case: serve a given amount of pieces of cake). Therefore, it provides access to the classical planning system Fast Downward from http://www.fast-downward.org/ using a ROS service in python. The service returns the resulting actions in a JSON-format that can easily be transformed to the action designators that are needed by our system.

The Fast Downward planning system needs two inputs: a domain definition and a task definition written in the Planning Domain Definition Language (PDDL). You can find a good introduction on PDDL at: http://www.cs.toronto.edu/~sheila/2542/s14/A1/introtopddl2.pdf. The domain definition describes the given environment which is mainly about the kind of objects that can be found there, the properties that they can have and the actions that can be used to change their properties. In our case, all the tasks that can be occur are placed in the same domain, so we defined the CaterROS domain. A task defines an initial state and a goal state. The planning system then shall find a sequence of actions (as defined in the given domain) to get from the initial state to the goal state. Therefore, the task definition also contains the concrete objects that are given (for example knife0 as a knife). The plan_generator module provides methods to generate a task file automatically depending on the amount of pieces of cake that should be served. The other objects and properties of the objects are fixed since they cannot (and don't have to) change within our scenario.

To use the plan generator for the CaterROS scenario, just start the python service:

```
rosrun plan_generator generate_plan.py
```

Now the demonstration can be run using the plan generator.

Executing Plans

There are two ways to execute the plans. Either by calling the `execute` function directly or by having guests in the knowledge base and let the system decide what to do on it's own.

Setup

To call the plans you need to load the `plan-execution-system` in the `plan_execution_system`. So open up the roslisp REPL by opening a terminal and typing:

```
roslisp_repl
```

In the REPL type:

```
CL-USER> (ros-load:load-system "plan_execution_system" :plan-execution-system)
```

And go into the package:

```
CL-USER> (in-package :pexecution)
```

Direct

Now you just have to call:

```
PEXECUTION> (execute "demo")
```

To start the demo task. The task gets evaluated to designators and those get referenced to real plans. In `toplevel.lisp` is a function `task->designators` in which all the tasks and theirs corresponding designators are defined. The most important ones are the "steps", which can be executed in order to execute the whole scenario of the CaterROS project. The `prep`, `cut` and `deliver` ones are also important as they are the ones called by the guest-centered method, but htey can also be executed directly.

Guest-centered

Now you call:

```
PEXECUTION> (start-caterros)
```

This starts the guest-centered plan execution loop (or GCPEL, as I certainly will never call it). As long as there is no guest present in the knowledge base the loop prints a message that it's waiting for a guest. When a guest arrives and makes an order, the loop will start executing the plans. First it will execute the `prep` task, to grasp the tools. Then it will `cut` as often as the guest ordered pieces of cake. And lastly it will `deliver` the plate with the cake onto the TurtleBot, which will then bring it to the table.

If you want to test this without using Pepper's Dialog system, you can call the `test-guest` function. It will generate a dummy guest in the knowledge base.

Mockups

The mockups package provides mockups scripts for all major components of the CaterROS project (excluding Knowledge) written in Python.

Usage

To start the mockups there are two launch files in the `planning_launch` package. You can start the mockups themselves with:

```
roslaunch planning_launch mockups.launch
```

If you want to use the knowledge base, use:

```
roslaunch planning_launch mockups_w_knowledge.launch
```

It can happen that the `tf_subscriber` node fails to launch properly when launching latter the first time. If this happens, just relaunch it and it should be fine.

You can only run plans if you launch with knowledge, because every plan needs to query the knowledge base. The first launch file is only for testing purposes when implementing service or action calls for example. But with the knowledge base launched you can run any plan and check if the plans themselves can be run without errors.

Most of the mockups have some support for the ROS parameter server. The graspkard mockup can either always instantly return an error value of 0 or simulate a optimization process over a few seconds. And the perception publisher's objects can be altered as well. For more detailed information on the how just look at the code. It's pretty simple.

Robot-specific Commands

The CaterROS planning system provides modules for the PR2 and TurtleBot robots. These contain mostly functions to control these robots.

PR2

To control the PR2 the planning system uses an action client for the action server provided by Manipulation. The pr2_command_pool package holds functions which call the action server with different controllers to move the PR2. The planning system continuously handles the feedback given by the server. Based on a break condition defined on a per-action level the feedback is evaluated. If the condition is given, the server is preempted and the action is considered successful. The break condition can consider the raw error value, the alterration rate of this value or both to evaluate whether an action is completed.

Turtlebot

The turtle_command_pool holds the designator definitions and an action client which will forward the pose to the action server of the tortugabot.

The chain is basically the following: plan_execution has an action designator, which tells the tortugabot to go to the location of an location designator. The pose which is transmitted to the location designator, is the pose of the table the current customer is at - we receive that information from knowledge. The location designator is being resolved with the help of multiple costmaps. First a circle is created with a certain radius around the position point of the table (or the tf-frame of the table). Then another circle is created, smaller in radius, around the same point. This one gets substracted from the first, so that one receives an donut shape. After this, another costmap is overlayed, which substracts all kinds of obsticles which are within the donut, from the donut. This prevents point creation within walls, since these would be unreachable. Then we receive a pose from that costmap, which through the action designator, is build into a cl-tf:pose-stamped. This gets forwarded to the action server of the turtlebot, and the turtle will then try and find a path to that point.

CHAPTER 8

PR2

bashrc setup

Everybody who wants to connect to the PR2 needs to add the folloging lines to his/her bashrc.

```
alias pr2a=192.168.102.60
alias pr2b=10.68.0.2
export ROS_MASTER_URI=http://pr2a.ai.loc:11311
export ROS_IP=<your-IP>
export ROS_HOSTNAME=<your-IP>
```

PR2 start

First connect to the PR2 via **ssh**.

```
ssh caterros@pr2a
```

With **robot claim** you get control over the robot. If anybody else has already claimed control over PR2, ask this person for permission first.

```
robot claim
```

To provide an environment, where anybody can access the processes running on the PR2, we use **byobu**. In byobu you can create a new window with **F2**, switch between windows with **F3** and **F4** and close a window with **Ctrl-d**. *Please do* **NOT** *call byobu inside of a byobu session!*

```
byobu
```

Now start the main launchfiles of the PR2. Use **F2** to create a new window for each process.

```
roslaunch /etc/ros/indigo/robot.launch
roslaunch pr2_manipulation.launch
roslaunch iai_maps iai_maps.launch
```

Make sure to disable the motors of PR2 before you launch the graspkard node.

```
roslaunch graspkard pr2.launch
```

To use the kinect for perception we need the **openni** node. We start that on the second computer, **PR2b**. For that we can ssh to pr2b from within byobu. This is ok, but a second byobu session isn't!

```
ssh pr2b
roslaunch /etc/ros/indigo/openni_head.launch
```

PR2 shutdown

When you are ready, kill all the processes in byobu and close each window with **Ctrl-d**. When shutting down the last window you should be in a common ssh connection with pr2a. Now you can stop the remaining/hidden/background processes and release you claim on the PR2.

```
robot stop
robot release
```

CHAPTER 9

Turtlebot

The following will include how to setup the Turtlebot itself in order for it to be used with the PR2 as a ROSmaster, how to maintain the Tortugabot and also what the high-level interface includes.

Tortugabot handling & startup

1. Turn on the Laptop on top of the Robot and log in. This might take a while to boot. Generally keep it plugged in whenever you are not driving the robot, so that it is always charged when you actually want to drive the robot around.
2. Connect it to the correct wifi, which is the PR2WLAN24.

   TROUBLESHOOTING: If you don't see the wifi icon, try to plug out and back in the USB cable, which connects the laser scanner to the laptop. If it doesn't reappear, reboot the laptop. If it is still gone, reboot again, but this time during the loading screen of ubuntu, when it says "configuring Network", plug out and in the USB connector again. After that, it should have wifi.

3. We assume all the necessary packages are already set up (which would be tortugabot_brngup, amcl, move_base, navigate_map...) and that you have pulled the sut_tortugabot package. If you aren't sure, just go to a terminal and see if you can roscd into them.
4. Plug in the battery.

   Note: Tortugabot1 has one battery while Tortugabot2 has two connected batteries, acting as one, since it has bigger motors.

5. open a terminal on the robot or connect to it via ssh. Type:

   pr2master

Which is an alias to set the PR2 as master. Then open byobu.

6. execute:

```
roslaunch sut_tortugabot t1_complete.launch
```

This will launch internally (in summary):

```
t1_minimal_bringup.launch
t1_amcl_laser_without_map.launch
t1_move_base_laser.launch
```

The first one brings up the motordriver (roboclaw), the laser (hokyo), joystic teleop, and a few other things. (a little chain of launch files.) The second is amcl aka. localization. Without the map server since the map is published by the PR2. The last brings up move_base so that the robot can drive around.

Done!

Turtlebot batteries

The thing with the batteries.... Keep in mind how much you use the robot, since there is no way of checking how full the battery is. At least, not without disconnecting it from the robot. So if you drive around a lot, you should probably charge it after 1-2 hours. If the robot is mostly standing still, you might charge it after 3-4 hours. It is important that the battery never runs low completly, since it might explode when completly depleated. To charge the battery, connect it to the charger (ask Alexis or Gaya which one that is). Make sure that the settings are: LiPo, charge, 2 Amp, V auto. If these are selected, hold the green button and the charger will beep and start chareging. For the Turtlebot1 battery the values the charger shows during chareging should be 3 cells, and around 12 V (+ - 1 or 2 Volts). For the Turtlebot2 battery, it's 6 cells, 24 V (+ - 1 or 2 Volts). If the values are off, fetch the supervisors.

Don't just unplug the batteries during the chareging process. If you need to interrupt it, hols the stop button. The charger will beep and let you know that is has stopped.

When the chareging process is completed, the charger will beep a few times. Wait till it stops beeping. It will show that the batteries are Full. Press the stop button, and disconnect the batteries. If you don't use the battery anymore that day, please put it back into it's lipo saver bag. Just in case.

More Troubleshooting:

If it cannot connect to Hokuyo or Roboclaw: try relaunching the launch file a few times, plugging in the battery in and out a few times. If that doesn't help, reboot, repeat untill it works. If not, call for help.

Note: In the package of sut_tortugabot, are many more launch files. Basically you can call amcl and move_base separately for debugging purposes, some files are being called by other files, so please, just keep them there.

The sut_tortugabot package

This package basically holds everything that needs to be executed localy on the Turtlebot itself. Which means, it holds a slightly adjusted roboclaw_node, the necessary parameters for move_base, the current map of the lab which we used and very many launch files, which are all prefixed.

What does prefixing mean?

It means that if we want to use multiple Robots in the same network, with one common master, they will likely have several topics in common. And if we then go on and publish something on these topics, it will affect all robots at the same time. For example, if we publish something on the topic cmd_vel, all robots will start moving and it's likely that one of them will crash into something. We don't want that to happen, therefore we need to keep all the topics and nodes separated. Also, it makes identifying which robot runs what, much easier. So now, instead of just having multiple topics called /cmd_vel, we have something like: /tortugabot1/cmd_vel and /tortugabot2/cmd_vel. The non prefixed topics are the ones of the PR2. But even though they are not prefixed, we can now tell them appart from the others.

Changes to these launch files:

These launch files are basically replicas of the basic tortugabot launch files, just that here, all the nodes and topics are prefixed and remapped so that everything the tortugabot does, has basically a prefix of tortugabot1 to it. This way, it won't clash. Also the Tortugabot has his own tf tree, which gets published to the the master on a regular basis, but slow frequency. Also we included some topic_tools nodes into some of these files, in order to remap and trottle down some of the publish rates of some of the turtle nodes. Otherwise it would flood the network with huge tf and laser scanner data, and loose localization in the process. Some of the values of amcl were also adjusted to the situation.

some interesting amcl parameters odom_alpha1-4: these basically describe how much you trust your laser scanner data over odom. The higher this value, the more you prefer to trust your laser scanner and not odom. It might be viable to set this parameter high when your odom is unstable. Current value is 20.0. You can see which odom value describes what here: http://wiki.ros.org/amcl

min_particles / max_particles: These particles describe the amount of assumed positions. (The little cloud of arrows which one sees in Rviz from the pose array.) Rule of thumb: if they are all over the place: it's not good. if they are all in one spot that it looks like there is just one or two arrows: that's bad as well. There should be a little cloud of them so to speak. In our case 100-200 were good values. You can also check the behaviour of the PR2 and it's pose array for reference.

use_map_topic is a boolean. When set to true, amcl will listen to the /map topic instead of getting the map via service call. Set to false in our case.

first_map_only also a boolean. Uses the first map it receives and that's it.

All other parameters should be documented in amcl.

Changes to roboclaw:

The changes here are minor. Basically the topics are hardcoded and do not accept the parameters for frames like odom and base_footprint, so we had to prefix them manually here. Also, we commented out one diagnostic updates line, since it was making the robot lag terribly.

Changes to the costmap parameters for move_base:

There is an own folder which holds all the parameters .yamls for move_base. Some of these values got adjusted as well. Generally, a good reference for calibrating is this http://wiki.ros.org/navigation/Tutorials/Navigation%20Tuning%20Guide One important point to mention is the **sim_time** within the dwa_local_planner.yaml parameters. Setting this wrong can result in the robot spinning rather then moving towards it's goal (quote from that tutorial, actually, so check it out for further reference.) A good value is usually 1-2 seconds.

CHAPTER 10

Launching CaterROS

Prerequisites

- You can launch the nodes on any machine you want as long as all of them share the same master (most likely the PR2).
- Of course to launch a specific system, you have to have that system installed on the machine. Check the system's documentation for installation instructions.
- Have the *PR2* and *Turtlebot* up and running.

Launch the CaterROS nodes

Note: If you want to start some or all of the system on a single machine (e.g. an server) you can use byobu to have simple control of the machine when connecting via ssh.

Launch Manipulation action server.

```
roslaunch suturo_action_server pr2_action_server.launch
```

Launch Knowledge.

```
roslaunch knowledge_launch suturo_knowledge_full.launch
```

Launch Perception.

```
rosrun percepteros caterrosRun cateros.xml
```

Launch peppers dialog system.

```
~/pepperdialog/pepperdialog/launcher.sh
```

Launch plan generator.

```
rosrun plan_generator generate_plan_server.py
```

Then start a REPL and start planning:

```
roslisp_repl
```

In the REPL type:

```
,
r-l-s TAB RET
plan_execution RET
TAB RET
,
i-p TAB RET
pexecution RET
```

Now you are in the plan_execution package and can start executing plans.

PR2 localization

If you want to (re)localize the PR2, you have to set your Rviz straight. In ''Global Options" the ''Fixed Frame" has to be set to ''map'', or else this won't work. After this you can use the button "2D Pose Estimate" at the top of the Rviz panel and set the position of the PR2 by clicking on the desired position and dragging the spawning arrow into the right direction. For further aid you can set the PoseArray topic to ''/partickecloud'', to see, where the PR2 thinks he is localized. The vizualization of the pose arrays seems like a huge mess at first. Just take the controller and drive the PR2 around a bit, this should refine the estimated pose.

Nice to know

If you want to run Rviz all the time but your machine is not the newest, you can put most of the workload for running rviz on the pr2b. For that you need to install (vglconnect.).

Then you can run:

```
vglconnect caterros@pr2b
vglrun rosrun rviz rviz
```

Done. You might need to reconfigure Rviz a bit, though. But your machine will thank you!

CHAPTER 11

Schnittstellen

Im Folgenden finden sich die Definitionen der Schnittstellen der einzelnen Gruppen sowie übergreifender Module.

Perception

Services

- **/percepteros/set_pipeline(List<ObjectNames>) -> Liste mit Objekten für die keine Pipeline gestartet werden konnte**
  Führt dazu, dass in Perception eine Pipeline generiert wird, die die spezifizierten Objekte sucht.

Topics

- **percepteros/object_detection -> suturo_perception_msgs/ObjectDetection** Gefundene Objekte werden auf diesem Topic gepublished.

Knowledge

Prolog

- **set_info(+Object, +[Info])**

  Dieses Prädikat ermöglicht das Ablegen und Ändern von Informationen zu einem Objekt oder als Parametersatz.
  Ein erfolgreiches ablegen der Informationen wird mit True bestätigt.

  Object: Das Objekt oder die Zuordnung der Information. Dies kann durch folgende Angaben erfolgen:
  ObjectInstance: Knowrob Objektinstanz in der Form 'Knife_xYz' ObjectName: rdf Parameter mit der Kante knowrob:nameOfObject und dem Objektnamen als String aus create_object_info() wäre dies der Name 'Knife1', Knife2' etc. ObjectType: Knowrobtyp des Objektes welches erzeugt werden soll um die Daten abzulegen. Bsp. 'Knife'

Info: Die Informationen welche abgelegt werden sollen als [Bezeichnung, Wert] Listenelemente. Dies führt zu einen Aufruf des Prädikates ähnlich zu set_info('Knife42', [[xCoord, 1.0],[yCoord,2.0],[zCoord,5.0],[isDirty, true],...])

Ein Info-Wertepaar der Form [nameOfObject, Name] wird ebenfalls als Identifier benutzt und würde versuchen ein entsprechendes Objekt in der KB zu finden. Ist dieses noch nicht vorhanden, wird es angelegt.

- **get_info(+Variables, -Returns)** Fragt beliebige Informationen ab die den in Variables gegeben Konditionen entsprechen. Bsp.aufruf wäre get_info([xCoord, [nameOfObject, 'Knife42'], isDirty], Returns) –> Antwort: -[[isDirty, true],[xCoord,1.0]]. Variables: Liste mit Konditionen als [Bezeichnung, Wert] e.g. [nameOfObject, 'Knife42'] und Abfragewerten wie xCoord, y Coord, typeOfObject, etc.
- **seen_since(+Name, +FrameID, +Timestamp) -> True/False** Wurde das Objekt mit Namen "Name" und der Frame-ID "FrameID" seit dem Timestamp "Timestamp" wieder gesehen?
- **disconnect_frames(+ParentFrameID, +ChildFrameID)** Trennt zwei Objekte mit den gegebenen Frames, so dass die zuvor konstante Transformation genutzt wird, um die neue "absolute" Position des Objektes zu berechnen und zu publishen.
- **cap_available_on_robot(Capability, Robot)** Kann genutzt werden um einen Roboter zu identifizieren mit der bestimmten Fähigkeit oder um die Fähigkeiten eines bestimten Roboters zu erfragen.

  Capability in der Form: srdl2cap:'AcousticPerceptionCapability' srdl2cap:'PerceptionCapability' srdl2cap:'ObjectRecognitionCapability' srdl2cap:'VisualPerceptionCapability' ...

  Robot in der Form: pepper:'JulietteY20MP_robot1' oder pr2:'PR2Robot1'

Service

- **connect_frames_service(String ParentFrameID, String ChildFrameID)** Typ: suturo_knowledgE_msgs/srv/ConnectFrames.srv Verbindet zwei Objekte mit den gegebenen Frames, so dass in TF eine konstante Transformation vom Parent zum Child gepublisht wird.

Manipulation

Der Actionserver zur Bewegung des Roboters bekommt ein Ziel in Form einer Nachricht vom Typ *suturo_manipulation_msgs/MoveRobotActionGoal*. Diese setzt sich zusammen, aus einer Liste von Gelenken, die vom Controller benutzt werden sollen, einer Beschreibung des Controllers, dem Namen des Feedback-Wertes und eine Liste von Parametern.

```
string[] controlled_joints
string controller_yaml
string feedbackValue
suturo_manipulation_msgs/TypedParam[] params
```

Die Liste der Gelenke, der Name des Feedbacks und die Liste der Parameter sind abhängig vom gewählten controller.

Parameter

Parameter werden in Form von *suturo_manipulation_msgs/TypedParam* übergeben.

```
uint8 DOUBLE=0
uint8 TRANSFORM=1
uint8 ELAPSEDTIME=2
```

```
bool isConst
uint8 type
string name
string value
```

Da es nicht möglich ist, generische Nachrichtentypen zu bauen, bzw. solche schwierig zu debuggen wären, werden alle Daten als Strings kodiert und ihr Typ mittels enumerierten Werten im Attribut **type** der Nachricht festgehalten. Der Name des Parameters, welcher im Attribut **name** vermerkt wird, dient eigentlich nur dem Debugging. Die einzige Ausnahme stellen folgende Namen dar:

- **r_gripper_effort**: Setzt immer die Griffstärke des rechten Greifers
- **l_gripper_effort**: Setzt immer die Griffstärke des linken Greifers

Bei den Parametern wird zwischen konstanten und dynamischen Parametern unterschieden. Dynamische Parameter werden während der Ausführung des Controllers fortlaufend aktualisiert. Ob ein Parameter konstant oder dynamisch ist, wird über das Attribut **isConst** festgehalten.

Unterstützte Typen:

- **double**
  - *konstant*: Eine Zahl mit oder ohne ”.”
- **transform**
  - *konstant*: Sieben durch Leerzeichen getrennte **double**, wobei die ersten drei die Position, die nächsten drei eine Rotationsachse und die letzte eine Rotation um diese Achse in Radianten beschreiben. Beispiel: “0 0 0 1 0 0 0” für die Identitätstransformation.
  - *dynamisch*: Zwei Namen von Frames die im TF-Baum existieren. Der erste Name ist dabei der des gesuchten Frames, der zweite der des Frames, zu dem der erste relativ bestimmt werden soll. Beispiel: “glass table” um den Frame *glass* relativ zu *table* bestimmen zu lassen.
- **elapsed time**
  - Gibt die Zeit seit Start des Controllers in Sekunden an. Alle Felder dieses Parameters werden ignoriert.

Die Reihenfolge der Parameter ist wichtig, da diese der in den Controllern entsprechen muss. Eine Ausnahme stellen dabei die fest benannten Parameter, welche gesondert behandelt werden und die Reihenfolge nicht beeinflussen.

Controller

graspkard/gripper_control.yaml

Description Changes the gripper opening width to the desired value.

Joint list

- *graspkard/config/pr2_right_gripper.yaml*: Right gripper
- *graspkard/config/pr2_left_gripper.yaml*: Left gripper

Parameter

- **double**: Desired gripper opening width *m*

Feedback *feedback* The closer to 0 the better.

Example parameters

- *graspkard/test_params/grasp_l_50.yaml* Grasping with the left gripper with 50% strength

- *graspkard/test_params/grasp_l_50.yaml* Grasping with the right gripper with 50% strength
- *graspkard/test_params/release_l_50.yaml* Releasing the left gripper with 50% strength
- *graspkard/test_params/release_l_50.yaml* Releaseing the right gripper with 50% strength

graspkard/pr2_grasp_control_r.yaml

Description Grasps a cylinder with the right arm. Opens the gripper in the process.

Joint list

- *graspkard/config/pr2_upper_body_right_arm.yaml*: Torso, right arm and gripper

Parameter

- **transform**: Frame of the cylinder in the reference frame of the Robot. This is *base_link* for the PR2.
- **double**: Diameter of the cylinder in *m*
- **double**: Hight of the cylinder in *m*

Feedback *feedback* the closer to 0 the better.

Example parameters

- *graspkard/test_params/approach_cylinder_r.yaml*: Moves the right arm to a cylinder with the name *cylinder* with the dimensions 5x14 *cm*.

graspkard/pr2_grasp_control_l.yaml

Description Grasps a cylinder with the left arm. Opens the gripper in the process.

Joint list

- *graspkard/config/pr2_upper_body_left_arm.yaml*: Torso, right arm and gripper

Parameter

- **transform**: Frame of the cylinder in the reference frame of the Robot. This is *base_link* for the PR2.
- **double**: Diameter of the cylinder in *m*
- **double**:Hight of the cylinder in *m*

Feedback *feedback* the closer to 0 the better.

Example parameter

- *graspkard/test_params/approach_cylinder_l.yaml*: Moves the right arm to a cylinder with the name *cylinder* with the dimensions 5x14 *cm*.

graspkard/pr2_upper_body_joint_control.yaml

Description Moves the joints of the torso into a goal position

Joint list

- *graspkard/config/pr2_upper_body.yaml*: Torso, both arms, no grippers

Parameter

- **double** Position des Gelenks *torso_lift_joint* in *m*
- **double** Position des Gelenks *l_shoulder_pan_joint* in *rad*
- **double** Position des Gelenks *l_shoulder_lift_joint* in *rad*
- **double** Position des Gelenks *l_upper_arm_roll_joint* in *rad*
- **double** Position des Gelenks *l_elbow_flex_joint* in *rad*
- **double** Position des Gelenks *l_forearm_roll_joint* in *rad*
- **double** Position des Gelenks *l_wrist_flex_joint* in *rad*
- **double** Position des Gelenks *l_wrist_roll_joint* in *rad*
- **double** Position des Gelenks *r_shoulder_pan_joint* in *rad*
- **double** Position des Gelenks *r_shoulder_lift_joint* in *rad*
- **double** Position des Gelenks *r_upper_arm_roll_joint* in *rad*
- **double** Position des Gelenks *r_elbow_flex_joint* in *rad*
- **double** Position des Gelenks *r_forearm_roll_joint* in *rad*
- **double** Position des Gelenks *r_wrist_flex_joint* in *rad*
- **double** Position des Gelenks *r_wrist_roll_joint* in *rad*

Feedback *feedback* the closer to 0 the better.

Example parameter

- *graspkard/test_params/upper_body_praying_mantis.yaml*: The *Praying Mantis* pose

graspkard/pr2_right_arm_joint_control.yaml

Description Moves the joints of the right arm into a goal position. Even though the gripper is included in the joint list and parameters, it is not comtrolled.

Joint list

- *graspkard/config/pr2_right_arm.yaml*: Right arm and gripper

Parameter

- **double** Position des Gelenks *r_shoulder_pan_joint* in *rad*
- **double** Position des Gelenks *r_shoulder_lift_joint* in *rad*
- **double** Position des Gelenks *r_upper_arm_roll_joint* in *rad*
- **double** Position des Gelenks *r_elbow_flex_joint* in *rad*
- **double** Position des Gelenks *r_forearm_roll_joint* in *rad*
- **double** Position des Gelenks *r_wrist_flex_joint* in *rad*
- **double** Position des Gelenks *r_wrist_roll_joint* in *rad*
- **double** Position des Greifers in *m* - wird ignoriert

Feedback *feedback* the closer to 0 the better.

Example parameter

- *graspkard/test_params/r_arm_praying_mantis.yaml*: The *Praying Mantis* pose of the right arm

graspkard/pr2_left_arm_joint_control.yaml

Description Moves the joints of the left arm into a goal position. Even though the gripper is included in the joint list and parameters, it is not comtrolled.

Joint list

- *graspkard/config/pr2_left_arm.yaml*: Left arm and gripper

Parameter

- **double** Position des Gelenks *l_shoulder_pan_joint* in *rad*
- **double** Position des Gelenks *l_shoulder_lift_joint* in *rad*
- **double** Position des Gelenks *l_upper_arm_roll_joint* in *rad*
- **double** Position des Gelenks *l_elbow_flex_joint* in *rad*
- **double** Position des Gelenks *l_forearm_roll_joint* in *rad*
- **double** Position des Gelenks *l_wrist_flex_joint* in *rad*
- **double** Position des Gelenks *l_wrist_roll_joint* in *rad*
- **double** Position des Greifers in *m* - wird ignoriert

Feedback *feedback* the closer to 0 the better.

Example parameter

- *graspkard/test_params/l_arm_praying_mantis.yaml*: The *Praying Mantis* pose of the left arm

graspkard/pr2_place_control_r.yaml

Description Places a cylinder in a traget area with the right arm.

Joint list

- *graspkard/config/pr2_upper_body_right_arm.yaml*: Torso, right arm and gripper

Parameter

- **transform** Frame of the target area *base_link*.
- **transform** Frame of the cylinder in *r_wrist_roll_link*.
- **double** Diameter of the cylinder
- **double** Height of the Cylinder

Feedback *feedback* the closer to 0 the better.

Example parameter

- *graspkard/test_params/place_cylinder_r.yaml*: Places a cylinder called *cylinder* in a target area called *goal_area*

graspkard/pr2_place_control_l.yaml

Beschreibung Places a cylinder in a traget area with the left arm.

Joint list

- *graspkard/config/pr2_upper_body_left_arm.yaml*: Torso, left arm and gripper

Parameter

- **transform** Frame of the target area *base_link*.
- **transform** Frame of the cylinder in *r_wrist_roll_link*.
- **double** Diameter of the cylinder
- **double** Height of the Cylinder

Feedback *feedback* the closer to 0 the better.

Example parameter

- *graspkard/test_params/place_cylinder_l.yaml*: Places a cylinder called *cylinder* in a target area called *goal_area*

graspkard/knife_grasp.yaml - Messer greifen

Description Grasps a knife with the right arm.

Joint list

- *graspkard/config/pr2_upper_body_right_arm.yaml*: Torso, left arm and gripper

Parameter

- **transform** Frame of the knife in *base_link*
- **double** Height of the knife in *m*
- **double** Length of the knife hanfle in *m*

Feedback *feedback* the closer to 0 the better.

Example parameter TODO

graspkard/TODO - Messer umgreifen

Description Messer sitzt beim ersten Greifen ungeeignet für das Schneiden im Greifer und wird mit Hilfe dieses Controllers in eine geeignete Position gebracht.

Joint list

- *graspkard/config/pr2_upper_body_grippers.yaml*: Torso, rechter Arm, linker Arm, rechter Greifer, linker Greifer

Parameter

- **transform** Frame des Messers in *base_link*
- **double** Länge des Messers in *m*
- **double** Länge des Griffes in *m*
- **double** Höhe des Griffes in *m*

Feedback *feedback* the closer to 0 the better.

Example parameter TODO

graspkard/pr2_cut_r.yaml

Beschreibung Cuts a cake parallel to its YZ-plane with the right arm.

Gelenklisten

- *graspkard/config/pr2_upper_body_right_arm.yaml*: Torso, left arm and gripper

Parameter

- **transform** Frame of the cake in *base_link*
- **double** Length of the cake (X-dimension)
- **double** Width of the cake (Y-dimension)
- **double** Depth of the cake (Z-dimension)
- **transform** Frame of the cake in *r_wrist_roll_link*
- **double** Height of the knife
- **double** Length of the knife handle
- **double** Width of the piece of cake

Feedback *feedback* the closer to 0 the better.

Beispiel-Parameter *graspkard/test_params/cut.yaml*: Cuts a 1,5cm wide piece of cake *cake* with a knife *knife*.

graspkard/pr2_cut_position_r.yaml

Beschreibung Moves the right arm into position for cutting the cake

Gelenklisten

- *graspkard/config/pr2_upper_body_right_arm.yaml*: Torso, right arm and gripper

Parameter

- **transform** Frame of the cake in *base_link*
- **double** Length of the cake (X-dimension)
- **double** Width of the cake (Y-dimension)
- **double** Depth of the cake (Z-dimension)
- **transform** Frame of the knife in *r_wrist_roll_link*
- **double** Height of the knife
- **double** Länge des Messergriffs
- **double** Width of the piece of cake

Feedback *feedback* the closer to 0 the better.

Beispiel-Parameter *graspkard/test_params/cut_pos.yaml*: Moves the right arm into position for cutting a 1,5cm wide piece of cake from the cake *cake* with a knife *knife*.

graspkard/pr2_detatch_knife_r.yaml

Beschreibung Detaches an object from the rack with the right arm. The Y-axis must point towards the rack. That way it is possible to use the last pose of the knife as rack pose.

Gelenklisten

- *graspkard/config/pr2_upper_body_right_arm.yaml*: Torso, right arm and gripper

Parameter

- **transform** Frame of the knife io *r_wrist_roll_link*.
- **transform** Frame of the rack in *base_link*.

Feedback *feedback* the closer to 0 the better.

Beispiel-Parameter

- TODO

graspkard/pr2_look_at.giskard

Beschreibung Points the RGB-Sensor of the Kinect at the center of a frame.

Gelenklisten

- *graspkard/config/pr2_lookAt_joints.yaml*: Torso, Neigungs- und Drehgelenk

Parameter

- **transform** Frame to look at in *base_link*.

Feedback *feedback* the closer to 0 the better.

Beispiel-Parameter

- *graspkard/test_params/poi_test.yaml*

graspkard/pr2_grasp_plate_r.giskard

Beschreibung Uses the right arm to grasp a circular edge. The center of the edge is passed as frame. The Z-axis is the rotational axis. The angle between the outer edge and the rotational axis is also passed. SO it it usually greater than 90°.

Gelenklisten

- *graspkard/config/pr2_upper_body_right_arm.yaml*: Torso, right arm and gripper

Parameter

- **transform** Frame of the center of the edge in *base_link*.
- **double** Radius of the edge in *m*.
- **double** Upper Z-Coordinate of the edge in the center frame.
- **double** Width of the dge in *m*.
- **double** Angle between the edge and the rotational axis in *rad*.

Feedback *feedback* the closer to 0 the better.

Beispiel-Parameter

- *graspkard/test_params/pr2_grasp_plate_r.yaml*

graspkard/pr2_release_r.giskard

Beschreibung Releases an object that was grasped with the right gripper. The gripper is moved backwards 12cm along the X-axis. The rotation does not change in the meantime. In order for this to work the starting position of the gripper must be passed as constant frame. Aditionally the opening width for the gripper must be passed.

Gelenklisten

- *graspkard/config/pr2_upper_body_right_arm.yaml*: Torso, right arm and gripper

Parameter

- **transform** Starting frame of the gripper in *base_link*.
- **double** Opening width of the gripper in *m*.

Feedback *feedback* the closer to 0 the better.

Beispiel-Parameter

- Derzeit keine

graspkard/pr2_move_and_flip_r.giskard

Beschreibung It is used to drop an object in a target area from a plate or spatula. Required are the transformation of the spatula relative to the right gripper, the frame of the target area, the width of the spatula and the radius of the target area.

Gelenklisten

- *graspkard/config/pr2_upper_body_right_arm.yaml*: Torso, right arm and gripper

Parameter

- **transform** Spatula relative to *r_wrist_roll_link*.
- **transform** target area in *base_link*.
- **double** Width of the spatula in *m*.
- **double** Radius of the target area in *m*.

Feedback *feedback* the closer to 0 the better.

Beispiel-Parameter

- *graspkard/test_params/move_and_flip.yaml*

Planning

Auch, wenn Funktionen wie *cutCake()* intern keine Parameter benötigen, muss für die Kommunikation von Python zu Lisp mindestens ein Parameter in der Signatur angefragt werden. Das Aufrufen von Funktionen ohne Parameter ist von Python zum Lisp-RPC-Server nicht möglich.

```
- RPC-Server
      - updateObserverClient(clientID, host, port)
          Der RPC-Server verwaltet eine Map von Clients und deren IPs/Ports. Bekommt␣
↪er diese Anfrage updatet er die Infos des entsprechenden Clients oder legt ihn neu␣
↪an.

      - cutCake(status)
          Um den Plan zum Kuchen schneiden anzustoßen. Soll sofort zurückgeben, wie␣
↪lange das etwa dauern wird (also z.B. wie viele Aufträge vorher noch ausgeführt␣
↪werden müssen). Return -1 bei serverseitigem Fehler.

      - stressLevel(status)
          Gibt die Auslastung des Servers als numerischen Wert zurück. Entspricht der␣
↪Anzahl der Aufgaben, die noch durchzuführen sind.

      - do(task)
         Führt die gegebene Aufgabe **task** aus.

      - assertDialogElement(json-string)
         Sendet das JSON an die Knowledgebase. Das Format ist hier https://docs.
↪google.com/document/d/1wCUxW6c1LhdxML294Lvj3MJEqbX7I0oGpTdR5ZNIo_w definiert.

      - getCustomerInfo(customer-id)
         Liefert die Info zum Customer mit gegebener ID als JSON.

      - getAllCustomerInfos(status)
         Liefert Liste aller Customer Infos zurück.
```

Pepper

```
- RPC-Server
      - updateObserverClient(clientID, host, port)
          Der RPC-Server verwaltet eine Map von Clients und deren IPs/Ports. Bekommt␣
↪er diese Anfrage updatet er die Infos des entsprechenden Clients oder legt ihn neu␣
↪an.

      - notify()
          Benachrichtigung, dass der Kuchen geschnitten ist.

      - new_notify(json-string)
          Benrachrichtigung wenn eine customer order fertig ist.
```

CHAPTER 12

Indices and tables

- genindex
- modindex
- search