CogLaboration

Collaborative Project

FP7 – 287888

D4.10 Report on Design of hierarchical organization of cognitive systems

Lead Author: TECNALIA

Reviewer: RURobots

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	June 2012
Actual delivery date:	2012/27/06
Version:	1.0
Total number of pages:	26
Keywords:	Robotics, Architecture, software, human robot interactions

Abstract

This deliverable presents the software architecture that has been developed to implement the cognitive controller of the CogLaboration project. The architecture presented is focused on the control aspect, and thus does not concern the software architecture related with the perception. Supposing that the human and the object are correctly detected and tracked, the software described here is in charge of deducing the arm and hand control commands and sending them to the related hardware drivers. The document starts by describing the software layers that have been developed to interface with the robotic arm and the robotic hand. Then the cognitive controller is described, through its two layers. The low-level layers is related with the execution of a specific motion pattern, intended to be the best robot answer to the human motion taking place. The high-level layer is in charge of orchestrating the different control modes, maintaining an understanding of the advancement in the collaboration procedure, and synchronizing with the different perception layers. Most of the software here presented is developed within the ROS environment, taking advantage of its communication facilities.

Executive summary

This document describesthe software architecture that has been developed to implement the cognitive controller of the robotic system being designed in the CogLaboration project. The document focuses on the control aspect; the software architecture related to the perception layers (mainly related to the vision-based analysis of the scene) is not addressed in this document.

The software described here has been mainly developed within the ROS [1]framework, in which a complete and complex process flow is decomposed into processing units called nodes that communicate and synchronize in between themselves through built-in communication means.

The interface presented here covers the whole spectrum of the control system. First of all, the modules enabling the communication with the hardware componentsare described. The communication with the Kuka Light Weight Arm is performed through the deployment of an Orocos[2] component that exposes within the ROS environment the communication interface of the arm that is usually accessible through the Fast Research Interface. The robotic hand is made controllable through a specific ROS node that provides an abstraction of the low-level communication protocol provided with the hand. This node provides a better synchronization with the hand actions by monitoring the advancement of a despatched task, and informing the upper layer of the action status and completion.

The cognitive control system is then described in the document. To get a better understanding of the organization of the control system, the document gathers the different relations involved in the computation of the robot joint velocities at each iteration step. Such processing flow is based on the concept of Dynamic Movement Primitives[3, 4] which forms the core of the cognitive controller system. This dynamical system permits the deduction of the Cartesian linear velocity of the arm end-effector. A traditional Inverse Kinematics algorithm is then used to transform this information into joint velocity commands that can be sent to the robotic arm. During this process, the linear Cartesian velocity deduced from the motion pattern being reproduced is applied with priority onto the angular velocity that is used to correctly orientate the robotic hand to interact with the object. The joint velocity related to the angular motion is thus computed onto the null space of the linear velocity constraint to make sure this second task does not interfere with the first one.

The implementation of this mechanism is done through the definition of a specific controller plugin that, when executed, is in charge of computing the robotic arm joint velocity in a closed loop mode until the completion of the approach process. This plugin is loaded within a controller manager that enables the easy switching from that control mode to any other type of control mode. In particular, a classic point to point control mode is also implemented as a plugin and described in this document. The use of this controller manager permits to switch from a DMP-based control mode (during a control object exchange with the person) to a more classical arm motion mode to move to a fixed position in the 3D space (like for moving to a specific arm rest position).

The high-level controller is in charge of coordinating the actions of the different components of the complete system, maintaining a situation awareness during the complete collaboration process. The high-level system is developed as a hierarchical state machine in which a state corresponds to a specific phase of the collaboration and permits to synchronize accordingly the different components involved in its achievement. The highest level state machine is in charge of the initialization of the complete system, and then directs the computation flows to state machines dedicated to the acquisition or transfer of the object.

The interface being presented has been extensively tested, initially on a simple robot emulation, using the perception data acquired during the observation of the Human-human collaboration, as described within Deliverable D4.61. It has then been experimentally validated onto the real robot to get a system ready for performing the evaluation with human partner within the industrial setting. The interface presented here is likely to slightly evaluate with the delivery of the latest versions of the perception modules and particularly with the delivery of the sensorized robotic hand. The final version of the controller will thus be depicted within the deliverables D4.62 and D5.6.

Document Information

IST Project Number	FP7 - 287888	**Acronym**	CogLaboration
Full Title	Successful Real World Human-Robot Collaboration: From the Cognition ofHuman-HumanCollaboration to the Cognition of Fluent Human-Robot Collaboration		
Project URL	http://www.coglaboration.eu/		
Document URL			
EU Project Officer	JuhaHeikkilä		

Deliverable	**Number**	D4.10	**Title**	Report on Design of hierarchical organization of cognitive systems
Work Package	**Number**	WP4	**Title**	Design and implementation of a control architecture based on concepts from thecognitive neuroscience.

Date of Delivery	**Contractual**	M20	**Actual**	M20
Status	version 1.0		final x	
Nature	prototype □ report xdemonstrator □ other □			
Dissemination level	public xrestricted □			

Authors (Partner)	Tecnalia Research and Innovation			
Responsible Author	**Name**	Anthony Remazeilles	**E-mail**	Anthony.remazeilles@tecnalia.com
	Partner	Tecnalia	**Phone**	(+34) 946.430.850

Abstract (for dissemination)	This deliverable presents the software architecture that has been developed to implement the cognitive controller of the CogLaboration project. The architecture presented is focused on the control aspect, and thus does not concern the software architecture related with the perception. Supposing that the human and the object are correctly detected and tracked, the software described here is in charge of deducing the arm and hand control commands and sending them to the related hardware drivers. The document starts by describing the software layers that have been developed to interface with the robotic arm and the robotic hand. Then the cognitive controller is described, through its two layers. The low-level layer is related with the execution of a specific motion pattern, intended to be the best robot answer to the human motion taking place. The high-level layer is in charge of orchestrating the different control modes, maintaining an understanding of the advancement in the collaboration procedure, and synchronizing with the different perception layers. Most of the software here presented is developed within the ROS environment, taking advantage of its communication facilities.
Keywords	Robotics, Architecture, software, human robot interactions

Version Log			
Issue Date	**Rev. No.**	**Author**	**Change**
14/06/2013	0.1	Miguel Prada	First content completion
18/06/2013	0.2	Anthony Remazeilles	Use of the project template
20/06/2013	0.3	Anthony Remazeilles	First Complete version
24/06/2013	0.4	Anthony Remazeilles	Completion of the missing interfaces and figures. Version ready for internal review
24/06/2013	0.5	Mark Burgin	Internal review
27/06/2013	1.0	Anthony Remazeilles	Final version

Table of Contents

Abbreviations

DMP:	Dynamic Movement Primitives
DoW:	Description of Work Document
FRI:	Fast Research Interface
HAL:	Hardware Abstraction Layer
IK:	Inverse Kinematics
IPC:	Inter-Process Communication
LWR:	Lightweight Robot
ROS:	Robot Operating System
UDP:	User Datagram Protocol

1 Introduction

In this document, the software architecture, enabling the control of not only the robotic arm but also its hand, is described, providing information about the internal communication mechanisms, and specifying the expected input information as well as the resulting output control command.

This document summarizes the design of the hierarchical cognitive controller. To be able to better explain the controller architecture, a brief overview of the hardware abstraction layer is included in the first section. This hardware abstraction layer consists of two modules, one for the LWR manipulator robot and another one for the IH2 Azzurra hand.

The next section describes the organization of the cognitive controller, including information about the computations involved, as well as the interfaces of the software components involved. This section is divided in two sub-sections, one for the low-level cognitive controller and one for the high-level cognitive controller.

In the current iteration, the focus has been put on performing fluent, human-like, reaching motions towards the hand-over location, and not so much on the specifics of the hand-over. Once the sensorized version of the hand is developed, the attention will be shifted towards the integration of the tactile information in the control system and its use to perform a fluent hand-over.

As initially stated in the deliverable D5.11, the framework selected to develop the software is ROS (Robot Operating System). The architecture and the related interfaces being described in this document are thus strongly based on the concepts of that specific software environment. More information about ROS can be found in D5.11, but also within the ROS webpage[1].

[1]http://www.ros.org/wiki/

2 Overview of Hardware Abstraction Layers

This section describes the interface developed to communicate with the two main hardware components that are the KUKA LWR and the Azzurrahand provided by SSSA. These components described here are used by the cognitive controller to send them motion commands.

2.1 KUKA LWR Interface Layer

The interface to the Kuka Lightweight Robot (LWR) is done through the Fast Research Interface (FRI)[5], as already explained in D5.11 [6]. This interface allows communicating with the robot by exchanging messages through a pair of UDP sockets. The robot controller takes the role of the server, which initiates the communication by sending feedback messages at a periodic rate which is configured on initialization of the FRI. The PC controlling the robot through this interface must parse the feedback message from the robot controller and respond with a new command within the configured time window (i.e. before the controller sends the next feedback) to maintain the communication and connection with the robot.

Even though the major party of the development will be done within the ROS framework, we have decided to design an abstraction layer of this interface through the deployment of Orocos[2] components. As stated in [6], this framework, similar to ROS, ensures real-time performance which is not guaranteed by ROS. This aspect is fundamental for implementing the lowest level control loop that needs to run at high frequency and with very low latencies. The Orocos components expose some of their functionalities to the rest of the ROS architecture. Figure 1 illustrates this sub-system.

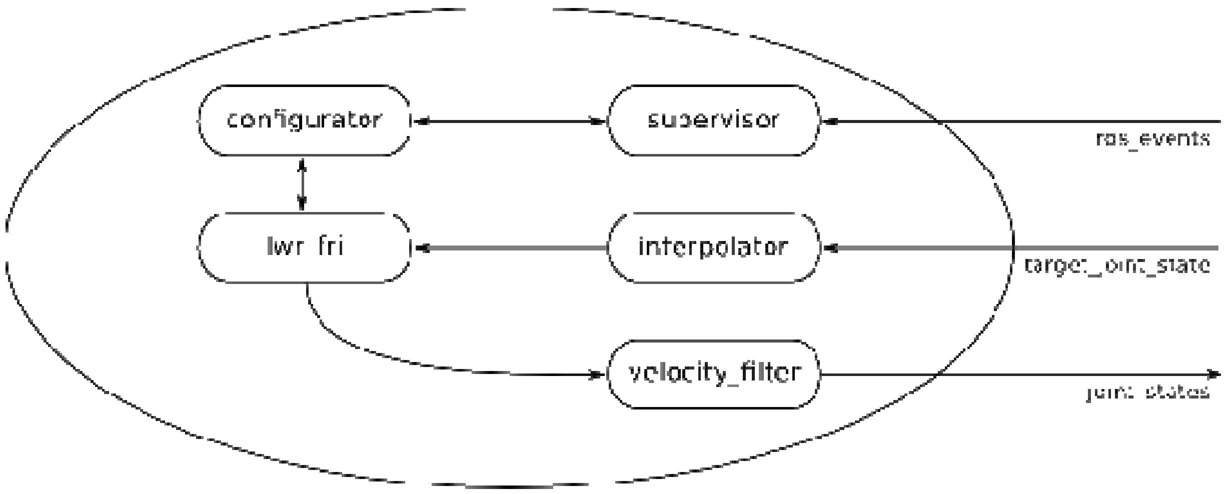

Figure 1: Kuka Lightweight arm access diagram

The components that form this layer can be divided between the ones that are involved in the actual control loop for the robot, and the ones that take care of housekeeping tasks. The components in the first category are:

- *lwr_fri*: this component is the UDP client connecting to the robot. It can be considered as a bridge component between the FRI and the native Orocos communication primitives.
- *interpolator*: this component up-samples and smooths the velocity commands received from the low-level cognitive controller (in ROS), while respecting pre-defined acceleration and jerk limits.

- *velocity_filter*: this component numerically filters the joint position data provided by the robot and outputs a velocity estimation together with the position data.

And the ones in the second category are:

- *supervisor*: this component runs a very simple state machine which coordinates the rest of the components when starting and stopping the robot.
- *configurator*: this is a helper component to the supervisor, which handles some long-running requests (like enabling or disabling the robot control mode) that would reduce the reactivity of the supervisor.

The interface of this abstraction layer to the rest of the ROS system is summarized in Table 1.

Table 1 : Interface of the Kuka Lightweight arm

Subscribed topics		
Name	**Description**	**Type**
ros_events	Commands can be sent to the supervisor component inside the sub-system via this topic. This is mainly used to ask to launch/stop and change the control mode.	string event
target_joint_states	This is the topic where the joint commands for the robot are received.	float[] velocity
Published topics		
Name	**Description**	**Type**
joint_states	Information about the robot joints is published to this topic. It contains the joint position and velocity, as well as the torque measured at the joint.	float[] position float[] velocity float[] effort

2.2 IH2 Azzurra interface layer

The IH2 hand provides a RS232-based interface, which exposes a series of commands to control either the complete hand, or the individual fingers. In order to integrate the hand into the cognitive control system, two layers of abstraction have been developed.

First, a pure python class has been developed, free of middleware dependencies, which exposes a functional interface for communicating with the hand. It internally takes care of packing the commands into byte-level messages to send to the hand, and of unpacking the feedback messages received, according to the communication protocol defined inD5.20[7]. This forms a starting point for the second abstraction layer.Table 2 lists the currently implemented functions enabling the access to the robotic hand.

Table 2 : Interface of the python hand driver. On the last column, first parameters are input parameters. When present, output parameters are specified after the "---" line separator

Provided Functions		
Name	**Description**	**Input/Output parameters**
handSerialDriver	Constructor, expecting as input parameter the name of the serial port to which the hand is connected.	string serial_port = None
calibration	Perform the calibration of the hand. Possible values are {"fast"|"complete"}	String data
move_motor	Move the motor in a given direction, until the motor reaches its limit or an external resistance	int motor int speed bool is_close_direction
set_finger_position	Move a motor to a given position	Int finger ([0;4]) int position ([0;254])
set_finger_force	Specify a tension to be apply onto a given finger	int motor ([0;4]) int force ([0;254])
stop_all	Stop all motions	
get_finger_position	Return the finger position, scaled to [0;254]	int finger ([0;4]) --- int position
get_finger_force	Return the finger force, scaled to [0;254]	int finger ([0;4]) --- int force
get_motor_current	Return the current being applied to a given motor	int motor ([0;4]) --- int current
get_finger_status	Return a structure containing the status of the given motor	int finger ([0;4]) ---- dict() status
get_all_finger_status	Compile in a list the status of all motors	--- [dict()] status
set_hand_posture	Position the motor to the provided values	int [ ] pos
set_grasp_mode grasp_mode_close	Configure the hand in a preshape configuration, according to the grasping mode given as parameter Close the hand, supposing the hand is already preshaped for a grasping	string mode string force_level bool is_current = True

The previous interface reproduces most of the functionalities provided by the hand communication protocol, as stated in D5.20[7]. The use of this module, presents a synchronicity limitation that is mainly due to the fact that the communication mode is notsynchronous: the operations required are sent to the hand controller, but no feedback is provided to know when a given operations succeeds. Furthermore, the status of the motor (that should be used to detect when an operation has been performed) is not automatically transmitted by the hand.

The second layer we propose addressesthese limitations. We proposed furthermore to implement it as a ROS component, so that the required functionalities are exposed using the common ROS communication primitives, using messages, services and actions.

Table 3 presents the interface of this component. In that table, the actions permit the higher level controller to receive a feedback message during the realization of the task, and a final message once the operation is finished. The information received in these three cases is depicted on the "Type" column. Once the hand node is launched, the status of all motors is periodically accessed and made accessible through the hand_status topic. When an action is activated, the motor status is checked at each update to monitor the progress of the task using the finger states. The action gets stopped and finished once the correct status is obtained, according to the specification of the motor status protocol.

Table 3: Interface of the ROS node dedicated to the hand control. The action type is divided in three blocks ("---" separator): the input parameters, the result sent once the operation finishes, and the feedback information transmitted during the execution of the task

Provided actions		
Name	**Description**	**Type**
hand_open	Open the hand at a given velocity, and informs once the operation is finished	#goal definition int32 speed --- #result definition bool isOk --- #feedback string status int32[] positions
hand_close	Close all the fingers at a given velocity, and informs once the operation is finished	int32 speed --- bool isOk --- string status int32[] positions
hand_preshape	Prepare an object grasping by preshaping the hand in a given mode. Informs once the operation is performed	string mode string level --- bool isOk --- string status int32[] positions
hand_preshape_close	Close the hand to grasp an object. We suppose that the hand has been previously been preshaped	--- bool isOk --- string status int32[] positions
Subscribed topics		
Name	**Description**	**Type**
hand_calibration	Launch the calibration process. This action, performed only once at the beginning	string

hand_status	Each action previously described dynamically subscribes to that topic to receive the status of each motor and, that way, monitor the advancement of the action being performed.	handStatus
Published topics		
Name	**Description**	**Type**
hand_status	The node periodically asks for the status of all motors and publishesit	handStatus

3 Cognitive Control System

The cognitive control system has been developed as a two-layered system, as described in the Description of Work (DoW) [8].

The upper layer is in charge of setting the parameters for the lower layer to control the robot at the beginning of the movement, and monitoring the execution of the selected movement in order to detect and follow the action transitions.

The lower layer receives a motion specification from the high-level controller and generates the motion according to it. The motion generation is performed by using motion primitives, which can adjust on-line to observed deviations on the users behavior.

In the current version, the hand is commanded directly from the high-level controller. It is done this way because the hand already has an embedded control system which allows it to perform most of the actions without the need of an external closed loop system. In the next iteration, the sensorized hand will allow us to consider the feedback from the tactile sensors in the arm control loop, and thus the hand will be interfaced from the low-level controller.

An overall picture of the system is shown in Figure 2 where the relations between the different components are shown.

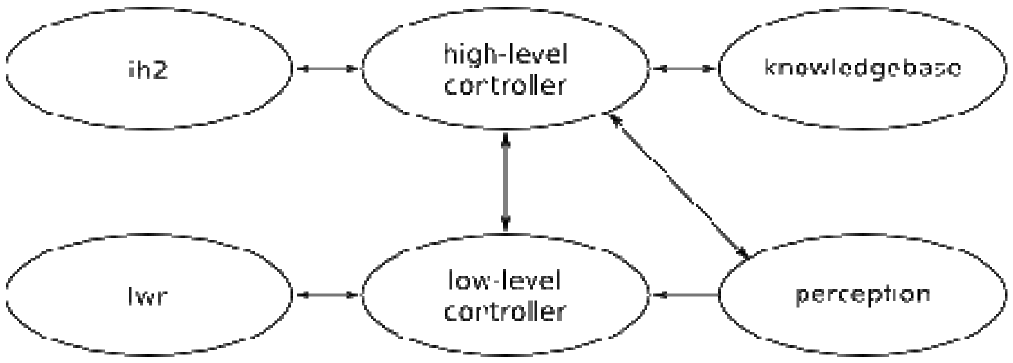

Figure 2: Cognitive control system organization

3.1 Low-level cognitive controller

3.1.1 Description

The low-level cognitive controller is responsible for closing the motion control loop of the LWR robot. It receives updates of the robot joint positions and velocities, gets the user's hand or the object's position from the perception components, and generates a new joint velocity command for the robot.

There are two main tasks involved here: the motion generation in the Cartesian space, and the generation of suitable joint velocity commands through Inverse Kinematics (IK). These two phases are now presented to get a better understanding of how the joint velocity commands get computed. The low-level controller interface will then be described.

3.1.2 Motion generation

The generation of new commands is mostly taken care through the DMP technique. This approach allowslearning arbitrary motions from a single demonstration, and is then able to reproduce it with the possibility to change the goal (final position) on-line.

This is achieved by using a dynamical system to generate the motion. This dynamical system is composed of a linear, second-order, part which attracts the system towards the goal position while providing a damping factor; and a non-linear part which drives the system away from its linear behavior to reproduce the learned trajectory. The specificequations of the system being used are:

$$\tau\dot{v} = \left(1 - w_g(s)\right)K(f_w + x_0 - x) + w_g(s)[K(g - x) + K_v\dot{g}] - Dv \quad (1)$$

$$\tau\dot{x} = v \quad (2)$$

$$\tau\dot{s} = -\alpha s \quad (3)$$

Equations (1) and (2) form what is known as the transformation system, where the evolution of the variable x dictates the output motion. The above formulation can be interpreted as three separate parts: a constant damping term Dv; an attractor towards the *goal*position $K(g - x)$, which is weighed by $w_g(s)$; and an attractor towards a moving point $K(f_w + x_0 - x)$, weighed by $(1 - w_g(s))$. Note, however, that since $w_g(s)$is a non-linear function, this arrangement of the equations does not allow easily separating the linear and the non-linear part. Nevertheless, rearranging equation (1) these parts can be separated as follows:

$$\tau\dot{v} = \left(1 - w_g(s)\right)K(f_w + x_0) + w_g(s)(Kg + K_v\dot{g}) - Kx - Dv = h(s, g, \dot{g}) - Kx - Dv \quad (4)$$

On this different arrangement, it is straightforward to see that the linear part of the system is stable as long as the K and D parameters are positive. This ensures that the output of the system will be bounded as long as the non-linear part, grouped in h is bounded.

The non-linear function f_w is formed by a sum of weighed exponential basis functions, which allows shaping this function arbitrarily by adjusting its weights.

$$f_w(s) = \frac{\sum_{i-1}^{N} \psi_i(s) w_i}{\sum_{i-1}^{N} \psi_i(s)} \quad (5)$$

Adjusting the weights is the core of the trajectory learning, and is very easily solvable problem once the centers and widths of the exponential are fixed, since f_w is linear on the w_i parameters. The function $f_w$is thus estimated during the reference motion pattern learning, and then used during the realization of the motion to deduce the next state of the system.

Another important fact in the formulation of the DMP is that the time-dependent terms of the equations are not explicitly dependent on time, but on the phase variable s instead. The evolution of this variable is dictated by the canonical system (3), which is independent of the transformation system and acts as a clock. The use of the additional variable and first-order linear system enables the speeding-up and slowing-down of the reproduction of the learned motion, even during the execution, by only modifying the time constant τ.

The previous model is used to estimate at each iteration the desired Cartesian position and velocity (on X,, Y and Z axis) of the robot end-effector. If each dimension is controlled with a specific DMP, the three dimensions are correctly synchronized by using the same phase variable.

3.1.3 Inverse Kinematics

After generating each new Cartesian motion set-point, the inverse kinematics algorithm proceeds to transform it into joint velocity commands for the robot. The inputs for this algorithm are the target Cartesian position and linear (or translational) velocity, the target orientation and rotational velocity, and the current robot joint position.

The targeted Cartesian position and translational velocity are directly provided by the approach described in the previous section, and the current joint position is provided by the robot controller itself. The target orientation and rotational velocity is related to the object being exchanged with the human partner. Each grasp mode of a given object is related with a grasping and delivery pose (position and orientation), corresponding to a desired position and orientation of the robot end-effector with respect to the object (or the human hand) frame. This orientation is considered as a secondary task to be achieved by the robotic system, while the Cartesian position from the DMP is considered as the main task to realize, by that means giving priority to the convergence towards the exchange site.

Therefore, instead of directly computing the instantaneous inverse kinematics (by inverting the complete Jacobian matrix) and getting the final joint velocity vector in one step, the computation is split in two parts. First the joint velocity resulting in the desired translational Cartesian velocity is calculated and afterwards the component coming from the desired rotational velocity is added, keeping its effect contained in the null space of the translational motion constraint. The advantages of adding this extra step are, on one hand, increased versatility (i.e. different constraints may be used if needed) and, on the other, increased robustness against singular or close-to-singular configurations, as is explained below.

The first step of the inverse kinematics algorithm consists in closing the position/orientation loop by adjusting the target velocities to use in the following steps of the algorithm. Since the robot is controlled in joint velocity space, it is important to close a position/orientation feedback loop at some point to avoid incremental drift. This is achieved by adjusting the target velocities with a feedback component based on the position/orientation error. This is quite straightforward to do in the position case, where the adjusted position can be computed as

$$v_{des} = v_{dmp} + K_{pos}\left(x_{dmp} - x_{rob}\right) \qquad (6)$$

wherev_{des} is the velocity for which the instantaneous inverse kinematics will be computed, x_{dmp} and v_{dmp} are the target Cartesian position and translational velocity generated by the DMP system, x_{rob} is the current Cartesian position of the robot, and K_{pos} is a gain which determines how fast the system should converge towards the desired position.

The rotational velocity component is computed in a similar manner:

$$\omega_{des} = \omega_{tgt} + K_{rot}.diff\left(R_{tgt}, R_{rob}\right) \qquad (7)$$

Where R_{tgt} and ω_{tgt} are respectively the current desired rotation and angular velocities that are interpolated along time to converge towards the desired final hand orientation at the exchange site. The feedback term can not be directly computed as the subtraction of the two orientations involved. In this case a special $diff$ function is used which computes the required rotational velocity to move from one orientation to the other in one time unit. This estimation is done by interpolating through Bezier curves the temporal variation of the orientation until convergence.

Once the two velocity components above are computed, the system performs the instantaneous inverse kinematics as follows.

Given the current robot configuration, the Jacobian matrix relating the joint velocities and the Cartesian twist (translational and rotational velocities combined), is computed, which results in a 6x7 matrix. This matrix can be decomposed in two 3x7 matrices, containing the three top rows and the three bottom rows. The top sub-matrix is the one which maps the joint velocities to translational Cartesian velocity of the robot's end effector, and the bottom one maps the joint velocities to its rotational Cartesian velocity.

$$J(q) = \begin{bmatrix} J_{\text{lin}}(q) \\ J_{rot}(q) \end{bmatrix};\ J(q) \in \mathbb{R}^{6\times 7}; J_{\text{lin}}(q), J_{rot}(q) \in \mathbb{R}^{3\times 7} \tag{8}$$

Considering this, the purely translational part of the inverse kinematics can be computed as

$$\dot{q}_{lin} = J_{lin}^{+} \cdot v_{des} \tag{9}$$

whereJ_{lin}^{+} is the Moore-Penrose pseudo-inverse[9]. Using the Moore-Penrose pseudo-inverse ensures that the resulting joint velocities are the minimum norm velocities that satisfy the translational velocity constraint.

The null space projection matrix of the J_{lin} Jacobian can be computed as

$$P_{lin} = I - J_{lin}^{+} \cdot J_{lin} \tag{10}$$

The rotational part of the inverse kinematics is then computed onto that null space:

$$\dot{q}_{rot} = (J_{rot} . P_{lin})^{+} (\omega_{des} - J_{rot} \dot{q}_{lin}) \tag{11}$$

The component $J_{rot}\dot{q}_{lin}$ is inserted to take into consideration the rotational velocity induced by the translational part.

The singularity robustness is added in this step. The pseudo-inverse of the $(J_{rot} . P_{lin})$ matrix is computed using its Singular Value Decomposition, which allows identifying singular and close-to-singular configurations very easily. In the current implementation, the system reacts to singular configurations by rejecting the rotational component altogether in the IK, while still being able to obtain a good solution for the translational velocity constraint.

If the rotation velocity constraint does not present singularity problems, the final joint velocity commanded to the robot is

$$\dot{q}_{rob} = \dot{q}_{lin} + \dot{q}_{rot} \tag{12}$$

Note that the translational velocity constraint can also present problems related with singularities in its associated Jacobian. However, given the nature of this Jacobian for the LWR, this can only happen in the robot's fully stretched configuration, a configuration that is dealt with separately by imposing workspace limits.

3.1.4 Implementation

Since the low-level cognitive controller is part of the robot's motion control loop, it has been implemented in a single ROS component on top of the robot HAL component, to avoid delays inherent to the IPC mechanisms.

A helper component has also been designed which allows transforming an arbitrary frame transform in *tf*[2]to a custom goal message for the DMP. This component is used to allow the perception module to broadcast the detected hand/object pose using the *tf* standard available in ROS, without needing to worry about providing the information to the cognitive controller in the appropriate reference frame. The cognitive controller directly accesses to the *tf* to get the current estimation of the goal, thereby avoiding the need for synchronization with the perception modules.

The operations mentioned in the two previous sections are performed within the cognitive controller which has an interface as described on Table 4. The service execute_dmp is used to start the execution of a DMP motion. As previously stated, the DMP motion mainly controls the Cartesian position of the robot end effector. The goal offset and the target orientations parameters are used to specialize the motion control to the considered object in order to bring the robotic hand to the correct position and orientation to grasp or deliver the object.

Table 4: Summary of the ROS interface of the cognitive controller plug-in. The signature column for the provided services contains the input arguments, a separator (---), and the return values. Some fields have been stripped out from the signatures when they were not relevant.

Provided Services		
Name	**Description**	**Signature**
load_dmps	Load the set of trajectories contained in a XML file. Return the number of loaded trajectories.	string filename --- int16 num_loaded
execute_dmp	Start executing a DMP motion. The *initial_state* can be used to indicate whether the motion generation should begin at the current position or use the initial position and velocity provided. This is used for on-line switching of primitives. The *dmp_ratio* parameter indicates the time constant used to generate the motion. The *goal_offset* and optional *target_orientation* are used to provide the relative position and orientation for grasping the object, retrieved.	string primitive_id intinitial_state geometry_msgs/Point initial_position geometry_msgs/Vector3 initial_velocity float dmp_ratio geometry_msgs/Point goal_offset bool orientation_control geometry_msgs/Quaternion target_orientation ---
switch_dmp	Substitute the currently running primitive for the requested one. The parameters for this service match the parameters in the *execute_dmp* service, except the ones related to the initial state.	string primitive_id float dmp_ratio geometry_msgs/Point goal_offset bool orientation_control geometry_msgs/Quaternion target_orientation ---

[2]tf (http://www.ros.org/wiki/tf) is a ROS package that permits the tracking of multiple coordinate frames along time. Typically, the perception modules broadcasts on that channel the estimated frame poses, and the controller listens to the transform it is depending on to update the control law. The tf mechanism handles internally the combination of the transforms (depending on their respective reference frames), and interpolates the transforms in between two broadcasts to handle the asynchronicity in between the data production and the data recuperation.

Subscribed topics		
Name	**Description**	**Signature**
dmp_goal	Current goal for the motion generation system.	geometry_msgs/Point goal
Configuration parameters		
Name	**Description**	**Signature**
pos_clik_gain	Gain for the closed loop component in the translational velocity part of the IK.	float
rot_clik_gain	Gain for the closed loop component in the rotational velocity part of the IK.	float
robot_base_frame_name	Base frame of the robot, used during the IK.	string
dmp_frame_name	Reference frame for the DMP.	string
enable_ik_debug	Whether to publish debug data.	bool

The cognitive controller previously presented is mainly used to handle motions while interacting directly with the human partner (human to robot or robot to human exchanges). We also implemented a point-to-point controller to manage arm motions, in which the human partner is not directly involved. This mode is used to move the robot towards a desired configuration in joint space (like a rest or home configuration for example). The point-to-point controller plugin has the interface described in Table 5.

Table 5: Summary of the ROS interface of the point-to-point controller plug-in. The signature column for the provided services contains the input arguments, a separator (---), and the return values. Some fields have been stripped out from the signatures when they were not relevant.

Provided Services		
Name	**Description**	**Signature**
add_ptp	Add a point-to-point motion command to the queue. Will start executing immediately if no other motion is ongoing. The returned status indicates whether the motion is started immediately, it is queued, or it is rejected. The returned id can be used to monitor the end of the motion.	float64[] position float64[] max_vel float64[] max_acc --- int32 status int32 ptp_id
Published topics		
Name	**Description**	**Signature**
finished_ptp	Indicates that the point-to-point motion with the provided id has been executed.	int32 ptp_id
Configuration parameters		
Name	**Description**	**Signature**
Kp	Gain for joint position loop closure.	float
max_queue	Queue size for point-to-point motions.	int

These controllers have been designed as plugins for a controller manager, designed as the central communication entry point for the higher level layers. The controller manager has the ability to load components (or controller plugins) dynamically from shared library objects, switch between them at runtime, and make sure the output of the running controller respects the workspace restrictions of the robot.

The controller manager can be configured to run periodically at any rate, and is currently being used at 200Hz. Even if the perception components are not able to provide new information at such high rate, since the controllers are responsible for closing the position control loop running them at a higher rate ensures smoother motion profiles.

In a typical exchange sequence, the high level controller will, first of all, request the controller manager to switch to the DMP-based controller. Once this controller is active and after a DMP motion has been requested using the execute_dmp service detailed inTable 4, the controller manager will be responsible for requesting the controller plug-in to update the motion command at the rate specified. The DMP controller will then perform the actions discussed in the previous section.

First, the last available goal position is grabbed, which is used to generate a new position and velocity setpoint for the robot by integrating the system in equations (1) to (3) for the desired cycle time. The state of the DMP system at the end of this integration is taken and used to compute the inverse kinematics as also explained in the previous section. Once the IK has been performed, the DMP controller plug-in hands it back to the controller manager, which in turn makes sure it respects the robot's joint limits and sends it to the robot component.

At any point during the execution of the DMP motion, a new DMP motion can be requested using the same execute_dmp service, which will start from the beginning. If it is desired to switch to a different primitive but keep the progress of the current motion, the switch_dmp service can be called. This system is the one which allows the high-level cognitive controller to change online the behavior of the low-level controller.

The interface of the controller manager component is summarized in Table 1, which includes the services it provides, the topics published and subscribed, and the configuration parameters it accepts.

Table 6: Summary of the ROS interface of the controller manager component. The type column for the provided services contains the input arguments, a separator (---), and the return values. Some fields have been stripped out from the signatures when they were not relevant

Provided Services		
Name	**Description**	**Type**
load_controller	Load a controller plug-in of a specific type and assign an id to it.	string controller_id string controller_type ---
list_controllers	List the loaded controllers, specifying their type and id.	--- string[] controller_ids string[] controller_types
switch_controller	Switch to the controller with the specified id. Returns the type of the controller loaded.	string controller_id --- string controller_type
get_active_controller	Get the id and type of the currently active controller.	--- string controller_id string controller_type
Subscribed topics		
Name	**Description**	**Type**
joint_states	Current state of the robot joints.	float[] position float[] velocity
Published topics		

Name	Description	Type
target_joint_states	Target joint velocity for the robot.	float[] velocity
Configuration parameters		
Name	**Description**	**Type**
cycle_time	The period of execution.	float
uppper_joint_pos_limit	Positive joint position limit. [Optional, default values provided.]	float[]
lower_joint_pos_limit	Negative joint position limit. [Optional, default values provided.]	float[]
joint_vel_limit	Joint velocity limit. [Optional, default values provided.]	float[]

3.2 High level cognitive controller

The high-level cognitive controller is in charge of coordinating the behavior of the different components in the system, including the arm and hand controllers, the perception components and the knowledgebase component.

One of the key tasks it performs is determining the type of motions to be carried out by the low-level cognitive controller, including the selection of a suitable primitive for motion generation and setting appropriate orientation constraints for each of the involved objects.

This layer of the controller is designed as a hierarchical finite state machine, and is implemented using the smach package from ROS[3]. The use of state machine permits theimplementation of quite complex behaviors by decomposing the actions into consecutive simpler tasks. This tends furthermore to stress again the importance of the modularity of the code being developed, to permits the reuse of the states implemented for reorganizing or extending the robotic behavior on demand.

Most of the states described in the following are mainly focused on explicitly calling the functionalities of the lower level modules described in the previous sections, while being connected to the experimenter through the Graphical Interface.

The highest level state machine, presented inFigure 3, is in charge of initializing all the different components, and then handles the general control loop which is organized around two hub states (wait without object and waitwith object) in which the robotic system is waiting for indication to launch the correct action. In the first version of the architecture that will be tested during the first evaluation, such information is provided through the Graphical Interface. The development performed within WP3 on the human motion recognition process will permit the automatic triggering ofthesetransitions.

The system initialization (coglab_init_system) is itself a state machine whosetwo main states are: coglab_init_vision and coglab_init_robot. They respectively perform in parallel the initialization of the vision modules, and the initialization of the robotic elements. The later component is another specific state machine, presented on Figure 4. As illustrated there, the system starts requesting the controller manager to load two controller plugins (the point to point and the DMP ones), and then configures the connection with the LWR. Once a trigger is received (provided through the Graphical interface), a calibration order is sent to the hand while the arm is being positioned to the “home position”. The “coglab_home_robot” (point to point arm motion) is an instance of state class that is frequently used within the other state machines. All the specific parameters (name of the DMP files, arm home position…) are provided to the system through configuration files that are easily accessedonline by the system. They are not displayed on the graphs for simplifying the visualization.

[3]http://www.ros.org/wiki/smach

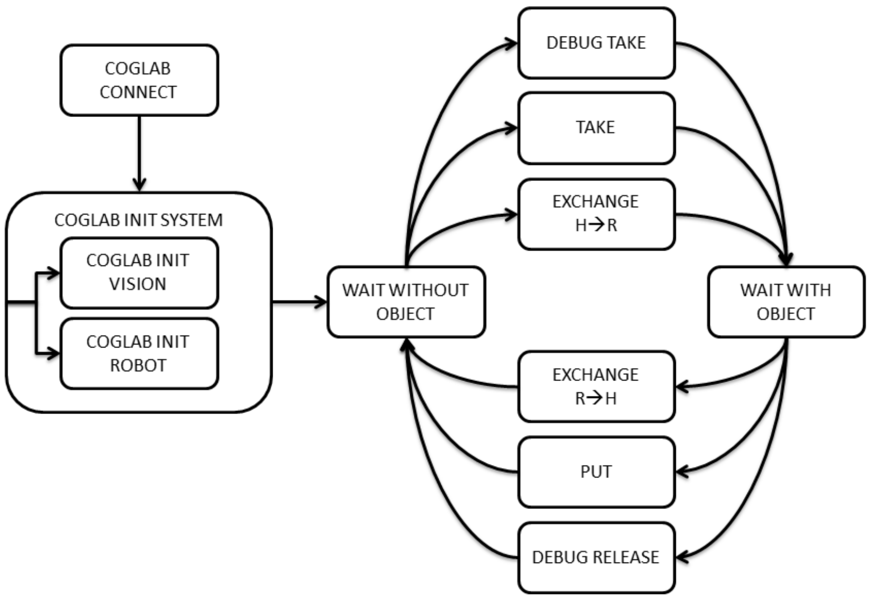

Figure 3: High-level controller: coglab_master state machine

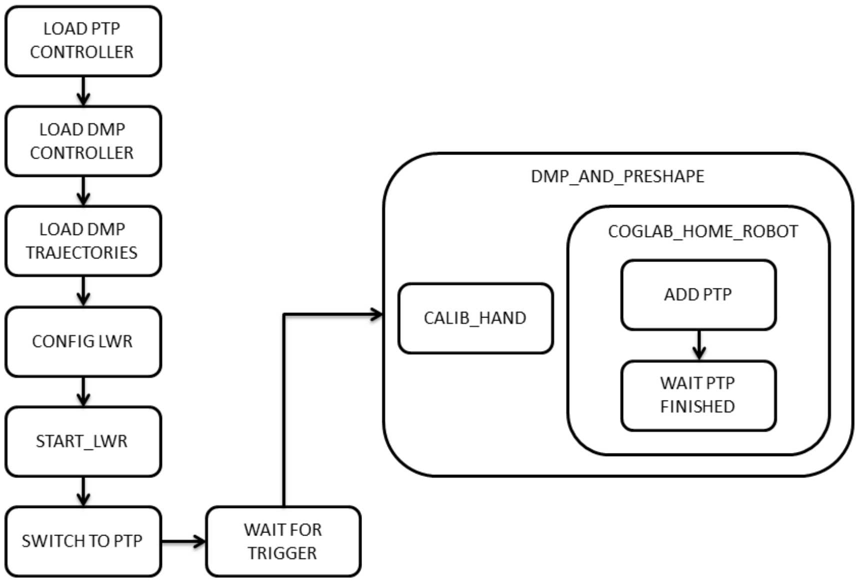

Figure 4: High-level controller: coglab_init_robot state machine

Figure 5presents the state machine being executed to perform an object interaction from the human to the robot. The operation starts with a query to the object database to get the needed information about the object we are willing to exchange with the person. The information obtained permits the specification of the nominal grasping mode of that object (that will be used to pre-shape the hand) and the desired pose of the arm with respect to the object to perform the grasping. This transform is used to adjust the online object or hand tracking results, so that the desired goal position of the arm, used in the DMP execution, corresponds to that grasping configuration. While the arm is moving, the hand gets pre-shaped.

The state execute_dmpis just a temporary state: it informs the controller manager to start the motion control. From that moment, the controller manager manages the close-loop arm control. The state machine directly transits to the next state, wait_stop. This state is subscribed to a contact trigger, which is currently used to inform that an object is within the robot hand. At that moment, the request to switch to a point to point control mode automatically stops the cognitive controller. During the execution of the motion control, the state monitor_dmp is used to verify that the current action (or DMP) is the most suitable answer to the motion being performed by the human partner. We are expecting to implement that component reusing the HMM developed in WP3 (task 3.5) to recognize different human reaching motion patterns, for each of which a specific DMP would be specified. When a switch to a different behavior is required, the switch_dmp service from the cognitive controller (see Table 4) manages to handle the transition from one to another.

Until the tactile sensors in the hand are available, a simple contact detector has been implemented by reading and filtering the torques estimatedat the robot joints. The robot controller filters the sensor readings to eliminate the robot dynamics (including the gripper, given that its inertial parameters are provided) and the friction in the transmissions, but the resulting value still contains a lot of noise when the robot is moving. This forces the use of the contact detection only when the robot is stopped (i.e. after the reaching motion towards the hand-over location is finished).

Even in this case, some residual torque readings are left in the sensors, due to inexact inertial parameter estimation. The solution proposed therefore uses the derivative of the torque signal to detect a contact, which is much better suited for our needs and limitations. The torque sensors used are not all the ones present on the robot, but only the ones more sensitive to forces in the plane perpendicular to the line going from the user to the robot, since it has been identified that these are the ones providing a clearer signal for the detection.

Once the robot correctly holds the object, the arm is then moved back to the home position, and the robotic system switches to the wait_with_object state.

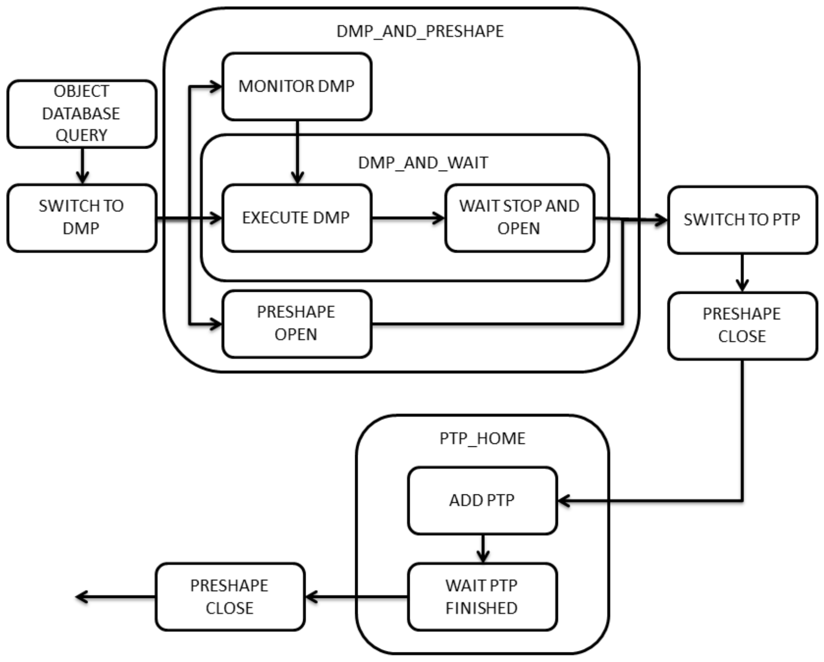

Figure 5: High-level controller: Exchange human → robot state machine

Figure 6presents the object exchange from the robot to the human. It is assumed here that, since the robot holds the object, the object database has already been accessed to get the exchange parameters (delivery mode, preferred pose with respect to the human hand to perform a correct delivery). These elements are considered as additional constraints fed in during the estimation of the robotic arm joint velocities, and combined with the desired Cartesian position as defined by the cognitive controller. The concrete values are transmitted from states to states to permit an access to thisinformation from any state of the state machine.

The mechanism put in place in that state machine is following the same principles as for the human to robot exchange.

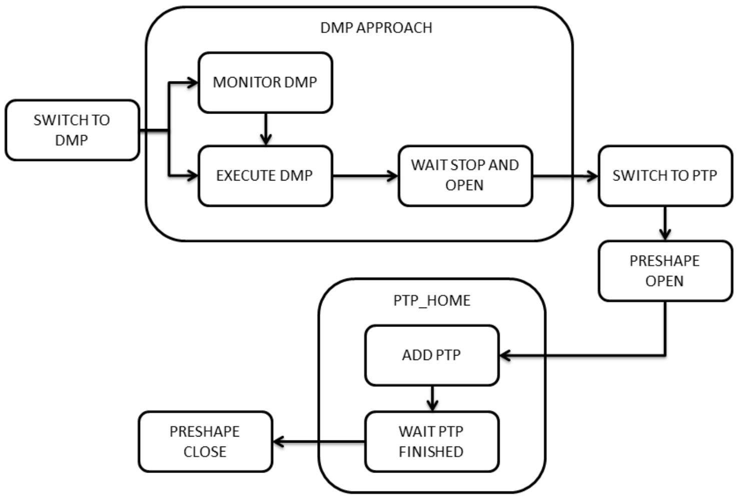

Figure 6: High-level controller: Exchange robot →Human state machine

The following two figures present the specific state machine developed for respectively acquiring (Figure 7) and delivering (Figure 8) an object from/to the object storage location. The take operation requires localizing the object onto the board, which is performed while the arm moves towards that panel. The detection of the object, and the grasping pose obtained from the database, permits the definition of the precise motion the robotic arm has to perform to bring the hand around the object of interest (In the present graph, the hand pre-shaping is done before the precise motion of the arm, but these two states could be launched concurrently to save time.

The release of the object on the board (see Figure 8: High-level controller: Put state machine) requires getting from the perception module an estimation of the free locations on the panel where the object could be delivered. Once obtained, the rest of the procedure is quite similar to the operations performed for taking the object from the board.

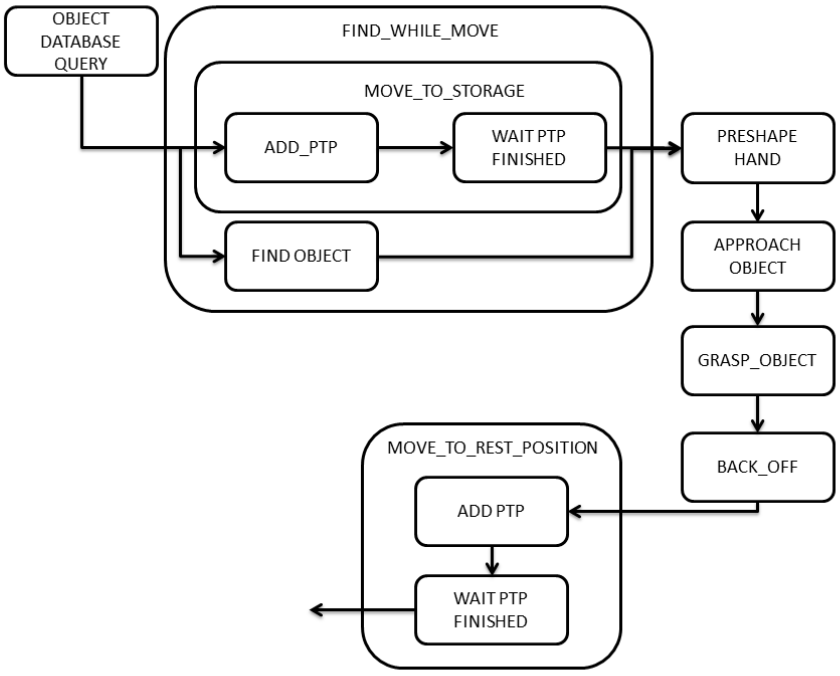

Figure 7: High-level controller: Take state machine.

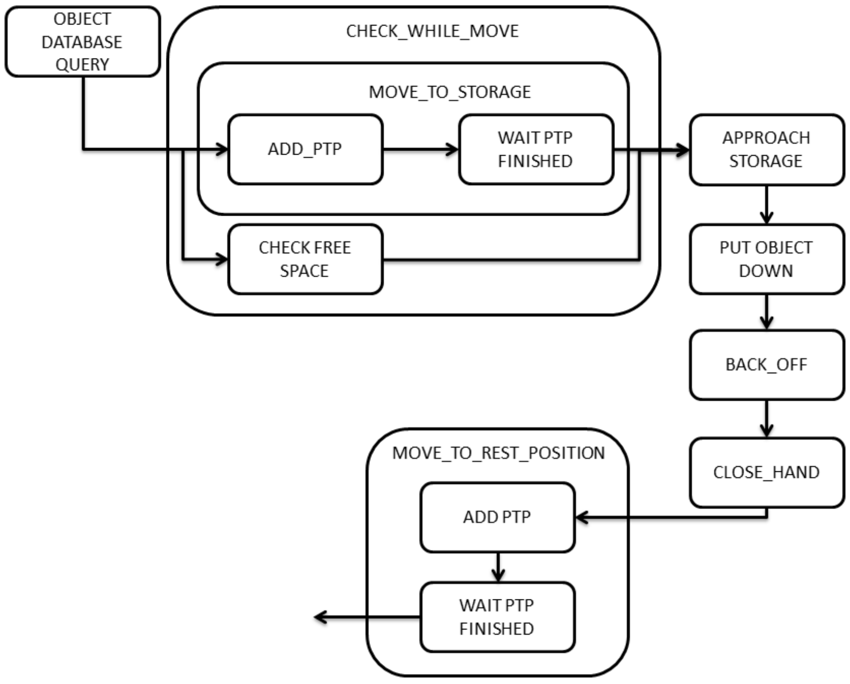

Figure 8: High-level controller: Put state machine

4 Conclusions

This document has described the architecture of the cognitive controller that has been developed to deduce the required motion of the robotic arm and to drive the activity of the robotic hand. Most of the developments have been performed in the ROS environment.

We have presented the interface we prepared for the Kuka Light Weight arm, which permits the deployment of the arm within the ROS framework, and easy access to the robot status and control of it through the regular communication means of ROS. The robotic hand has also been interfaced to ROS, wrapping the RS2323 communication protocol provided with the hand.

The core of the low level controller, based on the application of motion patterns has also been described in this document. We also presented how the Cartesian motions deduced from the motion primitive scheme are combined with the hand orientation constraints related to the object being exchanged. The components in charge of implementing and executing these approaches have been then presented in the document.

Finally, the higher-level controller has been introduced. It is implemented as a hierarchical finite state machine, and permits the handling of the successive operations of all the exchange cases of interest in our application. The modularity provided by the hierarchical organization of the controller will permit the easy adjustment of the behaviour of the robotic system if any evolution is required over time.

The architecture presented here has been tested experimentally and validated with simulation data, using the data acquired during the evaluation of the human behaviour in WP2. It has also been internally experimentally tested on the robotic system in the context of the integration work package 5. This architecture will soon be evaluated during the human robot exchange evaluations. These experiments will permit the assessment of the system behaviour and is likely to require some light changes of the robotic behaviours according to the expectations and feedback of the users.

This architecture may also evolve with the delivery of the software modules still in development in the perception work package WP3, even though we believe that our internal meetings and discussions has permitted the definition of an architecture that should fit with the developments envisioned in that work package. The delivery of the upgraded robotic hand with tactile sensors inserted will also have an impact on the presented architecture. We consider that the detection of the hand-over transitions will be much better with the information provided by that hand. Furthermore we expect to use the sensing capabilities to get a combined control of the hand and the arm during the hand-over to maximize the fluidity of that specific phase.

References

[1] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. R. Berger and A. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on Open Source Software*, 2009.

[2] orocos.org, "The Orocos Project: Smarter control in robotics and automation," [Online]. Available: http://www.orocos.org.

[3] A. Ijspeert, J. Nakanishi, T. Shibata and S. Schaal, "Nonlinear dynamical systems for imitation with humanoid robots," in *IEEE/RAS International Conference on Humanoids Robots*, Tokyo, Japan, 2001.

[4] M. Prada and A. Remazeilles, "Dynamic Movement Primitives for human robot interaction," in *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, IROS'12, workshop on Assistance and Service robotics in a human environment*, Algarve, Portugal, 2012.

[5] G. Schereiber, A. Stemme and R. Bischoff, "The Fast Research Interface for the Kuka Lightweight," in *IEEE ICRA 2010, Workshop on Innovative Robot Control Architectures for Demanding (research) applications*, Anchorage, Alaska, 2010.

[6] M. Burgin, "D5.11: Architecture specification, First version," CogLaboration, 2012.

[7] SSSA, "D5.20: First robot hand development," CogLaboration, 2012.

[8] C. consortium, "Coglaboration: Description of Work," CogLaboration, 2011.