Journal of Software Engineering for Robotics

5(1), May 2014, 36-47
ISSN: 2035-3928

A Case Study of ROS Software Re-usability for Dexterous In-Hand Manipulation

Guillaume WALCK[1] Ugo CUPCIC[2] Toni Oliver DURAN[2] Veronique PERDEREAU[1,*]

[1] University Pierre and Marie Curie, 75005 Paris, France
[2] Shadow Robot Company, London N1 1LX, UK

Abstract—Real world objects handling requires a complete robot and sensor system that must be managed by an adequate software architecture. The design of such a complex architecture is important not only to fulfill the hardware and software constraints but also to shorten the development and integration time. This paper presents the lessons learned in the integration of a software architecture based on ROS within the European project HANDLE. It focuses on the major concepts of component-based software engineering intrinsically available in the ROS framework and describes the best practices to take advantage of flexibility, interoperability and reconfigurability of packages offered by the ROS community for real world in-hand manipulation. The paper is divided in three parts. In a first place, the core concepts in the ROS framework are presented, detailing the multi-robot and distributed computing capabilities as well as existing software stacks for object manipulation applications. In a second place, a case study is done on the packages used or developed during HANDLE, alternatively describing the concepts related to component-based design patterns and their application. A last part covers integration choices relative to interfacing of the components, coordination and configuration of them in the final application.

Index Terms—System integration and implementation, software frameworks, reusable software, manipulation, dexterous hands.

1 INTRODUCTION

The case study presented in this paper was carried out within the European project HANDLE, which tackles the problem of in-hand manipulation with an anthropomorphic hand. The robot system involved in the project is composed of the Shadow [1] Dexterous robotic hand with 5 fingers for a total of 24 degrees of freedom (DoF) and the Shadow 4 DoF biomorphic arm coupled with several external sensors such as vision and tactile systems (Fig. 1).

The development of a software architecture capable of handling complex tasks and driving the arm and hand robot with all its sensors was required. Some specifications were particularly important in the design of the software architecture within the project:

Regular paper – *Manuscript received November 11, 2013; revised April 29, 2014.*

- *The research leading to these results has been supported by the HANDLE project (www.handle-project.eu), which has received funding from the European Community's Seventh Framework Program (FP7/2007-2013) under grant agreement: ICT 231640.*

- *Authors retain copyright to their papers and grant JOSER unlimited rights to publish the paper electronically and in hard copy. Use of the article is permitted as long as the author(s) and the journal are properly acknowledged.*

- The architecture should be based on existing components to accelerate the developments and permit to stay focused on innovative algorithms for in-hand manipulation,
- The chosen architecture should support distributed computing, with several computers involved in the control of the robot and the processing of the sensor data,
- The software layer should provide an easy solution to integrate all the components, in a high-level programming language,
- The architecture should use modular blocks to offer partial system testing possibilities and easy switch between simulated and real robot.

This paper presents how different components were integrated in order to fulfill the constraints of the specifications. It is oriented in a way to highlight underlying possibilities of existing software components, especially when they follow suggested design patterns for re-usability and reconfigurability [2]. It also shows how these components features were put into application during the design and integration phases of the HANDLE project. Although the architecture was implemented on robots built by the Shadow Robot Company, the solution is designed to be as generic as possible, that is, for any arm and hand robots.

The paper is organized as follows. Section 2 describes the

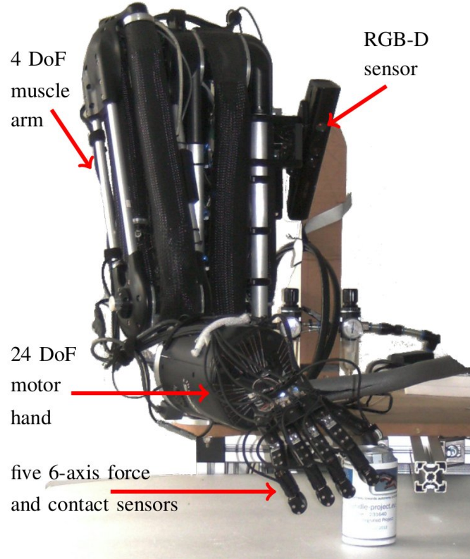


Fig. 1. HANDLE robotic platform: A biomorphic muscle-driven arm, a dexterous motor- and tendon-driven hand, a RGB-D camera and fingertip force sensors.

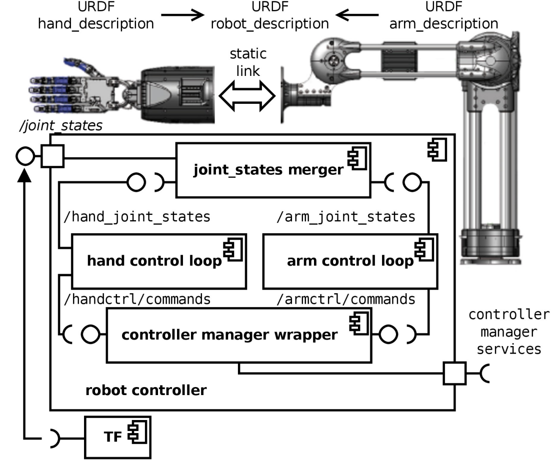


Fig. 2. Fusion of two robots requires encapsulation of low-level control interfaces to provide the same component interfaces as a single robot to the system.

main software elements of the chosen framework. In section 3 the main concepts of component-based software engineering are detailed in a case study of major components re-used or adapted for the in-hand object manipulation application, and in section 4 overall integration choices are presented together with useful hints to fully use the possibilities of the framework. A conclusion and future work in section 5 summarizes the contribution and presents work perspectives.

2 A VERSATILE FRAMEWORK

The ROS [3] (Electric) meta-operating system was chosen to develop the software architecture as it best fits the project requirements and had been used before for a basic object manipulation application on a different robot (PR2) [4]. This framework heavily relies on components in the form of nodes to handle different aspects of robotics such as control, planning, image processing, database storage, etc... The term component is admitted even if the ROS nodes do not usually offer an abstraction of the framework as expected in component-based software engineering [5]. However, re-usability, flexibility and reconfigurability are some aspects available in ROS as will be shown through the paper. The components are provided via packages developed by a large community. ROS supports distributed computing and multiple robot management. Applications can be developed in C++ or Python language.

In this section, key features of ROS will be detailed to highlight the component-based aspects of the framework used in this project and to point out important steps in the deployment of the architecture.

2.1 Multi-robot

The HANDLE project platform is composed of a multi-robot system. In ROS, components handling hardware interface and control of a robot necessarily run on the same machine in what is called the *realtime loop*. This loop ensures low-level state exchange (sensors / commands) between the host and the robot through an etherCAT driver and processes controllers update in a *controller manager* (CM). A special device access is also provided in the *pr2_ethercat* bus driver package to integrate non-etherCAT drivers in the loop. ROS is able to simultaneously handle several robots that can be linked together in a single robot description file using the Unified Robot Description Format (URDF). However, even if the hand and arm are statically linked to create a complete chain from arm base to hand fingertips, their drivers cannot co-exist on the same computer due to incompatible OS and communication buses (etherCAT and Controller Area Network buses). In order to provide a standard control interface to other components of the system, the two separate control loops were encapsulated into a component with similar ports (state and command) as for a single robot. For this purpose a merger utility was created to basically fuse the complete arm and hand joint states and a controller manager wrapper was added to access the controller managers of both robots as seen in Fig. 2.

Thanks to the standard control interface recreated in the encapsulating component, the frame transforms for each link of the robotic platform could be published in the usual manner with the *robot_state_publisher* node in package *TF*[1] by connecting directly to the merged joint_states topic.

1. TF handles coordinate frames relation in a tree structure

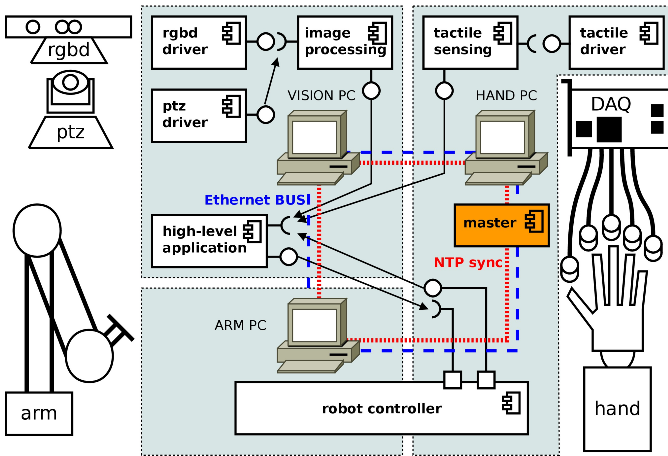


Fig. 3. Distributed computing over the network with all the components visible through a master. Precise time tracking with NTP is mandatory for event synchronization.

2.2 Distributed computing

As mentioned before, the fact that two robots are considered when working with the hand and the arm is a first reason why distributed computing features of ROS were used. Moreover, to deal with external sensors, new computers and drivers were added to the HANDLE platform. A vision system managing a RGB-D sensor and a Pan-Tilt-Zoom (PTZ) camera gives the robot platform some crucial information about the working environment. Due to the huge amount of data to process especially with point clouds, both devices are handled in a separate computer from the robot's computers.

ROS is designed to manage components processing data over several computers all connected on a network bus and visible through a master (Fig. 3). To correctly treat the events passed over from one component to another, the synchronization is operated with timestamps. Therefore, it is of particular importance to synchronize all the computer clocks. Consequences of a desynchronized multi-computer system are mainly seen on the robot chain when frame transforms are not coherent in time or on the fusion of data coming from internal and external perception systems. Hence, a system such as a NTP (Network Time Protocol) server is mandatory to *constantly* track and correct computer clock drifts at runtime.

2.3 Object manipulation libraries

One of the main advantage of ROS is the large amount of packages/libraries available thanks to a great community. A set of stacks forming an object manipulation pipeline [4], [6] is of particular interest as it deals with recognizing, reaching, picking up and placing objects in a complex environment. A clear diagram was created showing the stacks and regrouping their functionality in blocks (Fig. 4). Here is a short introduction to the libraries that were re-used or adapted as described in section 3:

- **Arm navigation**: In order to reach and grasp objects, the arm movements planning is done through the OMPL library [7] in the *arm_navigation* stack. Additionally, Orocos KDL [8] provides kinematics solving for any kinematic chain built in a robot description file (URDF).
- **Perception**: Processing vision data is simplified through the Point Cloud Library (PCL) [9] for RGB-D sensors and is highly integrated in ROS with means to cluster, detect and segment 3D objects. The OpenCV library [10] contains standard 2D image processing algorithms and is bridged to ROS via *cv_brige* and *image_transport* packages.
- **Household objects database**: To recognize the workspace, objects were 3D scanned and grasps were generated for most of them for a robot gripper. A postgresql database is delivered filled with models of daily life objects and grasps for the PR2 gripper [11].
- **Control and realtime loop**: Low-level controllers run in a realtime loop at 1kHz as plugins (*pluginlib*) and can be loaded/unloaded on-the-fly to implement any control solution commanding torques.
- **Pick and Place**: A node in *object_manipulator* coordinates the different steps of a pick and place scenario, calling the actions in the relevant nodes presented above.

3 Re-usability case study

In this section, we highlight the re-usability of ROS packages through well-known component-based concepts and show how they applied in the implementation and integration phases of the HANDLE project: i) *flexibility* applied to control design and database extension, ii) *reconfigurability* for enhanced perception, iii) *new use of packages* applied to planning beyond arm movements and iv) *configuration* for complex manipulation management.

3.1 Flexibility

Although most of the object manipulation stack packages are mentioned to be robot agnostic (can work with any robot), some modifications are sometimes needed. Different solutions were used to adapt, extend or derive packages and are described in this subsection. Adaptations were done respecting the license limits and were obviously possible thanks to the availability of the source code.

Principle of library extension: It sounds interesting to patch a library to add some functions. However, mid-level libraries such as kinematics libraries are linked to a lot of high-level packages (more than 80). Changing the mid-level may require to recompile the entire framework distribution and having a patch propagated upstream might take a while.

Fortunately, some libraries are flexible enough to be extended through a new package containing the required functionality. The best way is to use class inheritance and class

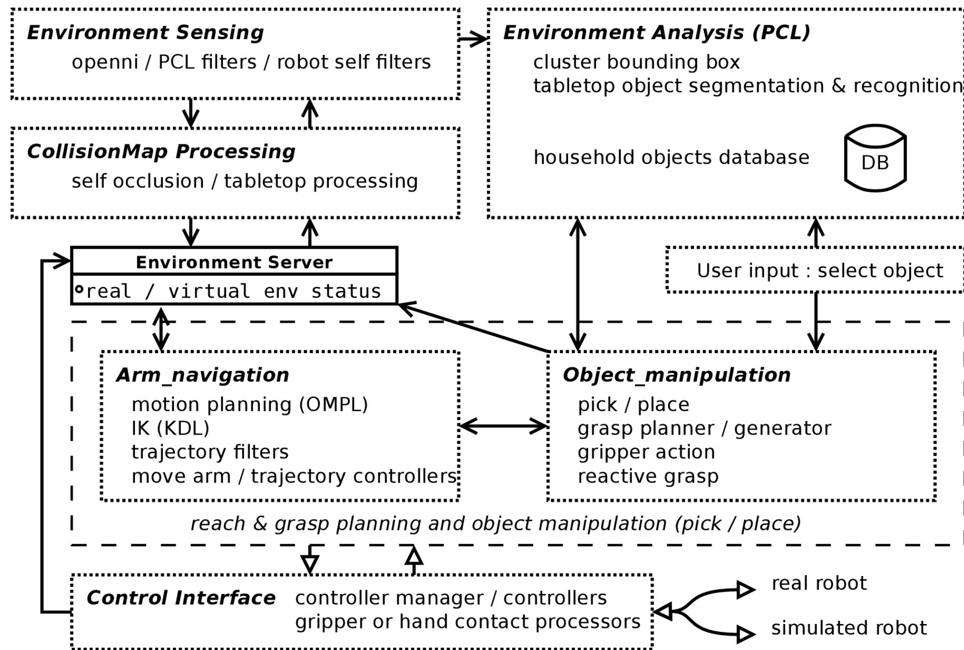


Fig. 4. Main components of the object manipulation pipeline re-arranged in functional blocks. Several stacks were re-used and adapted for in-hand manipulation such as arm navigation, control interface, household object database and object manipulation stacks.

composition to create a new specialized class adding members and methods to the parent class.

With this separation into main and extended packages, standard high-level software still link with the original library but new software using extended functionality link to both the extension and the original package.

Application of library extension: Orocos KDL package [8], on which *arm_navigation* package depends, provides a library for direct and inverse kinematics computation for a large number of kinematic chains. The standard library iteratively solves 6D pose inverse kinematics, which is fine for most manipulators. Indeed, the arm and two wrist joints of the HANDLE platform were linked to create a 6 DoF arm up to the hand palm and Orocos KDL nicely solved the kinematics.

Re-using the library to solve the hand kinematics was thought to be a good generic solution that could work for several hands rather than developing a specific solver for the Shadow robot hand. In fact, fingers can be seen as single chains but the initial version of the KDL library cannot handle coupled joints, which are present in a few robotic hands [1], [12], [13]. Moreover, fingers rarely contain more than 5 joints, rather 3, making the inverse kinematics impossible to solve with KDL 6D solvers. Therefore, an extension of Orocos KDL (*kdl_coupling*) was created with the technique above (Fig. 5). It handles joint coupling in a generic fashion involving a coupling matrix taken from a patch from [14]. The extension can also solve 3D pose inverse kinematics as well as provide a Jacobian for 3D control. The solution is based on a solver using WDLS (Weighted Damped Least-Squares) already available in KDL. Detailed explanations of the solver are beyond the scope of this document.

Principle of plugins and derived package creation: ROS implements a mechanism of plugins through the *pluginlib* that permits to dynamically load libraries with specified interface into a running node. The interface is described in a package and several plugins that implement this interface can be loaded at runtime. Nodes using the plugin principle do not need to compile with a plugin header, only with the defined interface header, thus enhancing the flexibility of the whole package.

When the plugin mechanism was not sufficient to adapt the functionality or when change in the data structure was required, new packages were derived from existing ones. The structure of the code was re-used as it implements the base functions properly but new methods or members were added to handle the robot specificity.

Application of plugins and derived package creation to controllers: Several joint-space controllers are provided in the *robot_mechanism_controllers* package and can be applied to any robot as long as the joints can be commanded through torques. Among the controllers proposed are not only velocity or position controllers for single joints but also joint trajectory controllers, tracking several single joints at the same time along a given trajectory parametrized with way points. However, two limitations can be seen. On one hand, not all

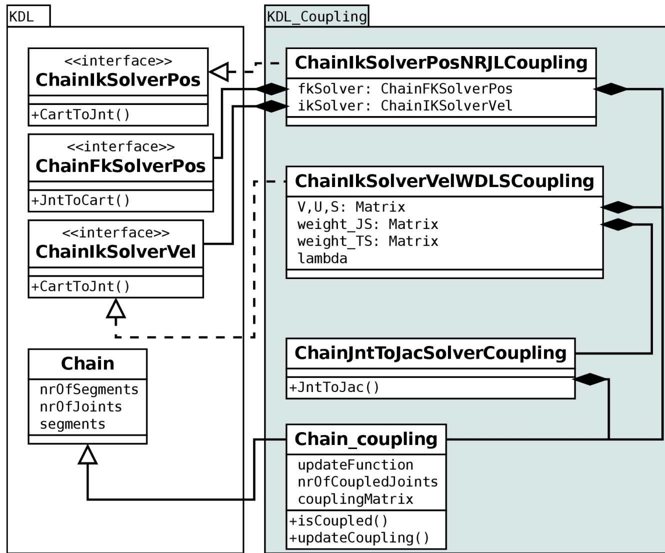


Fig. 5. Library extension through a package, involving inheritance and composition. The original package remains untouched, not modifying the framework distribution but new packages can take advantage of the extension.

robot systems are equipped with torque control at joint level, and therefore need a specific controller manager relying on other types of commands. On the other hand, all the proposed controllers are based on basic PID controllers, which might not be sufficient to overcome non-linear joint behavior such as those seen in tendon-driven or muscle-driven joints.

Mixed position and velocity controllers were created for the Shadow hand to command the tendon tension (similar to joint torque) and to compensate for various non-linear effects (friction). They were developed to implement the *pr2_controller_interface* and could then be loaded into the standard *pr2_controller_manager* node thanks to the mechanism of plugins (Fig. 6.a).

The Shadow biomorphic arm is composed of two antagonistic pneumatic muscles per joints that are only controllable in pressure at low-level. Due to the nature of the muscles, pressure commands are not equivalent to torque, for this reason the generic controller manager using the standard *robot_state* had to be replaced. A new *sr_arm_controller_manager* commanding pressures could be derived from the existing controller manager and loaded with non-linear position controllers through the powerful plugin mechanism (Fig. 6.b).

Principle of database completing: Even if a software component is robot agnostic, it may need to have new data fed into the system as a kind of basic knowledge on how the robot operates on the environment. This is particularly true for a database storing information on interacting elements (objects, obstacles). Two different cases were considered:

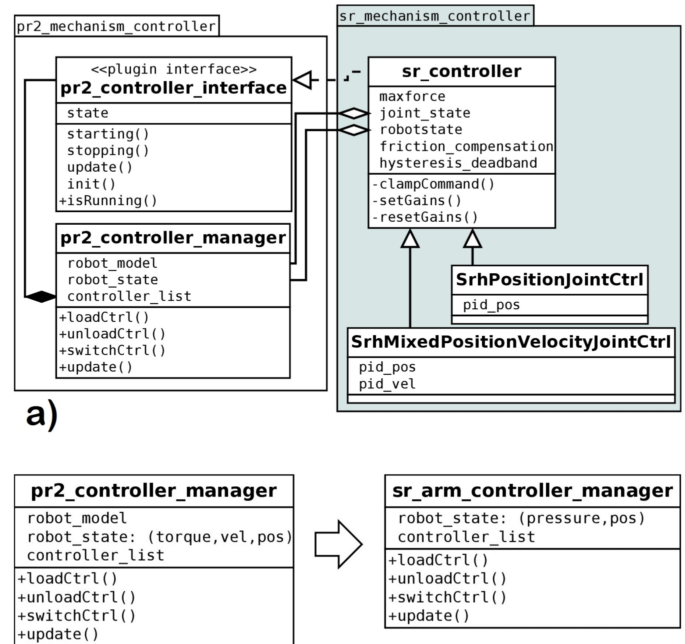


Fig. 6. a) Creating new controllers as plugins using a predefined interface permits to load them in an existing controller manager node handling such an interface. b) A new controller manager can be derived from an existing one with minor but necessary changes (torque to pressure change in the robot_state)

- Filling new data in existing database tables. Not modifying the tables permits to re-use all the nodes reading and filling these tables,
- Extending the tables with new kind of data. To maintain backward compatibility, the additional data are distributed between existing tables and new ones.

Application of database completing: The *household_objects_database* package relies on a database mainly containing object models and grasps to seize these objects with a 2-joint gripper. However, the database is flexible and one can easily change settings to a different gripper, namely the Shadow five-fingered hand, by detailing in the hand_description.yaml file that the device is composed of 20 joints. Adding a new entry in the grasp table is then immediate even for existing objects provided that a vector of size 20 is inserted for the grasp posture (Fig. 7).

The tools provided to add new objects or new grasps generated by GraspIt! [15] could have been used immediately to fill the unmodified tables but in the HANDLE project suitable grasps were extracted by learning from humans. Demonstration of a human grasping objects were recorded with datagloves [16] and parameters transposed to joint configurations of the robot hand and arm [17], [18]. The learnt grasps

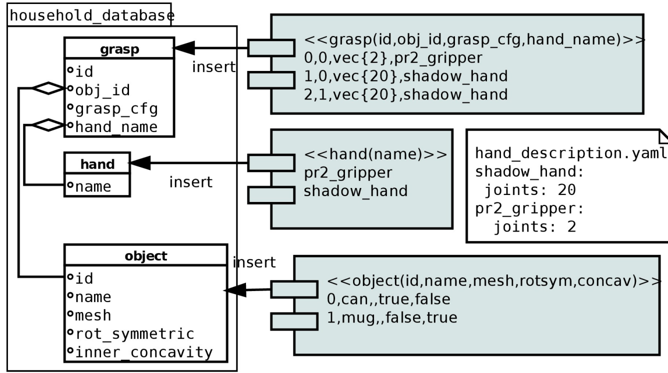


Fig. 7. The household database is flexible enough to handle new types of grippers described in a file. Shadow hand grasps with 20-joint configurations can be added in existing tables.

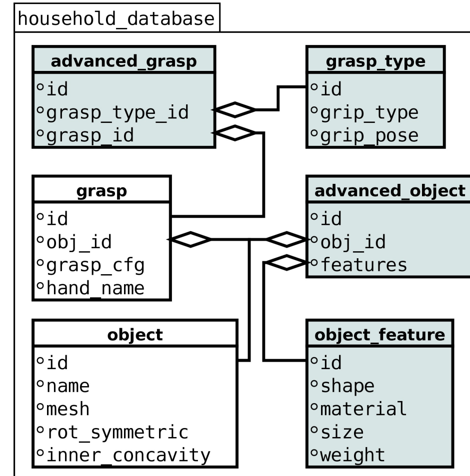


Fig. 8. New tables were added to the household database and combined with existing ones through wrapper tables to maintain backward compatibility.

with joint configurations adapted to the dexterous hand were filled in the grasp table thanks to a node based on a generic sql database interaction component (*database_interface* package). The same way, new objects models were recorded through a new 3D reconstruction method [19] and also inserted in the existing object tables.

Since the project also deals with in-hand manipulation, new kinds of data related to the grasp such as grip type (number of fingers and posture used) or grip pose (relative hand to object pose) had to be stored in the knowledge base. Instead of modifying the existing tables, new tables were added to store the advanced data, and wrapper tables would contain relation to backward compatible tables and to tables containing the new knowledge (Fig. 8). This way, existing components can make profit of new entries (without new type of knowledge) and new components can access entries with all their features.

3.2 Interoperability

It has been shown that the large panel of packages in ROS are flexible and can be re-used with another robot but the diversity of packages and the component-based framework also offers the possibility to combine them to create new advanced features. This is possible thanks to the interoperability of the components using common message types. This subsection presents two combinations of components at two levels:

- Combination of functions
- Combination of data

Both combination methods were employed to solve the problem of in-hand object tracking.

Combination of components for multi-marker tracking: Tracking objects is a complex task that is generally treated differently for specific cases. No off-the-shelf method could be found but a combination of several tools could do the job. Markers tracking was tested as a first step to track objects but two major issues arose:

1) objects are small to fit in-hand and so will be the markers. They need to be tracked by the vision system during the whole hand movement,
2) objects are grasped in the hand whose fingers might occlude some of the markers.

To solve the occlusion problem, multiple small markers were added on the same object, whose pose was then computed through a mean transform obtained from each markers filtered out pose. The marker poses are extracted by *ar_pose* [20] relying on the ARToolkit [21], which can process several elements in the same image. To solve the tracking of the markers in-hand, a pan-tilt-zoom (PTZ) camera was chosen to zoom on the hand, which pose was computed via forward kinematics. The hand pose permitted to set the direction of the optical axis and keep the markers in the field of view while the hand was moving. The configuration of the different components is shown in Figure 9.

Combination of data in sensor fusion: A second step towards advanced in-hand object tracking involved fusion of data. Basically the technique consists in using several sensors to get a more confident estimation of the object pose. A solution was developed based on visual and tactile information. Visual data are provided on one side by the scene camera recognizing objects through the *tabletop_detection* package (see Fig. 4), and on the other side by the multi-marker tracking pipeline previously mentioned. Tactile data are provided by the fingertip force sensors precisely locating the contact [22]. The fusion is represented in the lowest part of Figure 9. The challenges were to get all the data transformed in the same frame and to be sure the components would be able to get the correct type of information by the corresponding node.

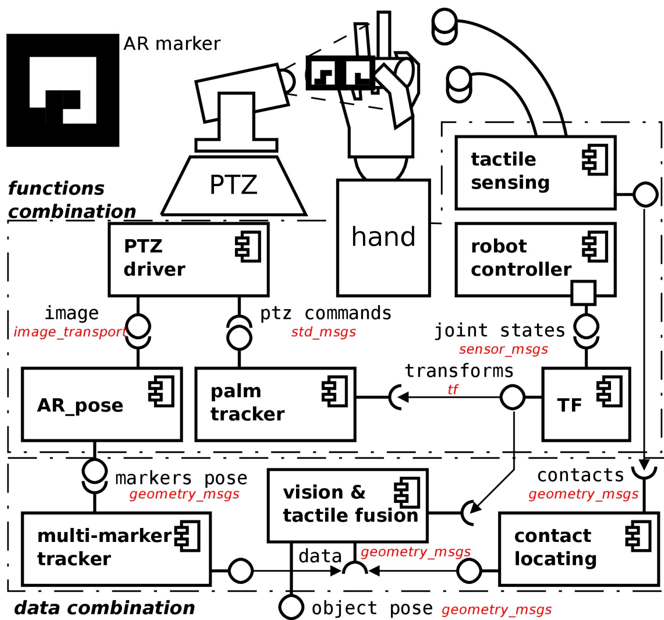


Fig. 9. Use of common messages types (std/image_transport/tf/sensor/geometry) implicitly offers interoperability that permits to combine functions (top part) or data (bottom part).

For the two types of combination, all the packages work as stand alone applications but could be easily connected together thanks to low-level packages implicitly offering interoperability. Most of the common message packages such as *geometry_msgs* or *sensor_msgs* were re-used as is, and two other packages were of particular interest: *image_transport* and *TF*. Indeed, a lot of image processing relies on the *image_transport* library to transfer image data. Camera configuration is also embedded in the transfer and was especially used to feedback the change of zoom, and hence of camera configuration of the PTZ camera to the *ar_pose* package. Regarding *TF*, the frame transform functions offered by the *tf_conversion* tools made it simple to transform the pose of the object, initially estimated by the scene camera in the world frame, to the palm frame. Then the pose was refined and tracked through the markers and finally optimized thanks to the fingertip contact poses changed to the palm frame.

This combination case highlights the fact that low-level tools are really important and can simplify integration. Developing new packages that rely on common messages and use of common tools is necessary to provide good interoperability and thus permit multiple combinations and extended re-use. A second point to mention is that ROS connects nodes via a communication protocol (based on protobuf), which permits to combine node programmed with different languages (or different licenses) by exchanging common messages.

3.3 New use

Some packages in the community are already created with the purpose of being generic and re-usable, and use low-level libraries to offer good connectivity. This is the case of packages that pack solutions for vast robotics problems such as planning. The *arm_navigation* stack is flexible and reconfigurable to permit easy integration of a robotic arm plus gripper into a manipulation process. It relies on OMPL [7] and was tested on different robot arm systems, even on industrial ones through ROS industrial project [23]. Thanks to the flexibility and good documentation, it is even possible to use such packages beyond their original design.

***arm_navigation* for each finger:** One concrete example is the finger motion planning for in-hand manipulation. A finger performing an action (press a button, lift a lever, turn a knob etc...) while other fingers maintain a stable grasp on the object is a common case. Planning the finger motion from grasp release position to action position without touching the object was a problem OMPL can solve. Reworking the integration of the library in a new package doing finger motion planning would have been cumbersome. Instead, the *arm_navigation* stack was completely re-used, for the full-arm plus 5 mini-manipulators, one for each finger (Fig. 10). Our extended KDL solvers (with coupling) were configured as IK solvers for the planner. With this planning environment set up, single finger motion could be planned out of the box. The only specificity lies in the fact that the obstacle is the object held by the hand, and is provided by higher level applications rather than autonomously by the *collision_map_processing* package.

***arm_navigation* for mixed chains:** A second example, pushing the limits of planning with the *arm_navigation* stack, was to create a new virtual kinematic chain composed of the arm and wrist to which virtual degrees of freedom are added. The virtual DoFs represent the possibility of the object to be moved by the fingers through in-hand motion. The complete chain has some redundancy and widens the possible reachable poses for the gripper around the object relative to the arm base. The planned trajectory on the virtual degrees of freedom can then be fed into an in-hand motion planner such as [24] to compute the adequate finger motions for in-hand manipulation.

In the end, the diversity of packages and the port to ROS of largely used libraries made it possible to try and apply existing solutions to new problems very rapidly.

3.4 Configuration

The last method showing re-usability is totally inherent to component-based programming. At a high level, creating a real robot application handling complex tasks requires to configure the components to correctly process the data flow, distribute the requests and collect the responses for each of the specialized components.

Pick and place: The *object_manipulor* package provides a configuration node responsible of connecting the components

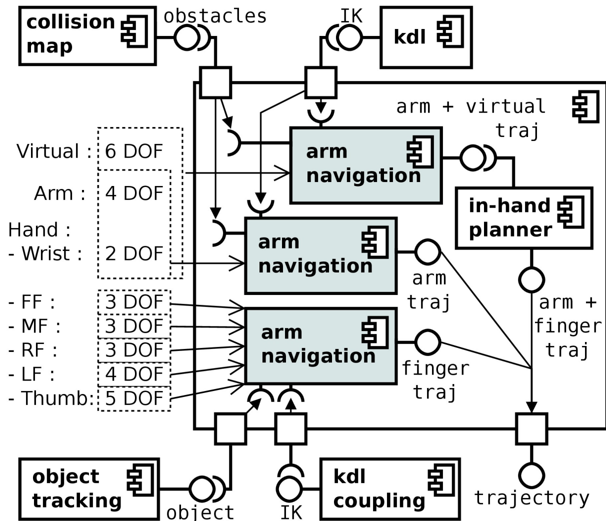


Fig. 10. The same component (*arm_navigation*) is used for arm planning, finger planning or advanced virtual robot planning.

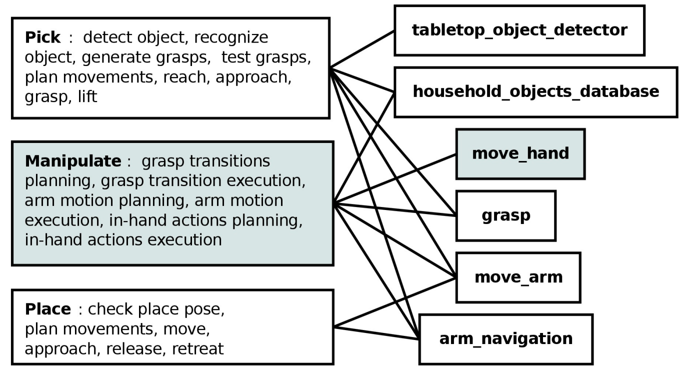


Fig. 11. Configuration nodes connect the components to fulfill a task. The connections can be reconfigured to execute a different task via a new configuration node.

for a pick and place application. Each of the interface of the specialized component is defined through a parameter that can be connected to other components at launch. It is straight forward to replace one of the component with a new one as long as it offers the same interfaces and services. However, when it comes to change the application, then a new or extended configuration node must be created.

Pick / manipulate / place: a pick and place application existed but adding in-hand manipulation in-between the pick and place tasks was done by rewriting a configuration node directly in Python using the same technique (Fig. 11). The individual components are connected and coordinated to follow a designed flow, similar to the original package. In-hand manipulation tasks make use of most of the existing components but also of new components to move objects in-hand.

4 Overall architecture

This section presents a discussion about choices made relative to interaction between nodes and describes several ideas implemented during the integration.

4.1 Actionlibs vs services

Before the integration process begun, components were developed and run in stand-alone applications to test the algorithms. The interfaces were used to interact with the user or with other components. At the time of integration, the implementation of the interfaces became a major concern to ensure all of the components were interoperable, and functions could be called properly.

ROS offers 3 types of interface in the components. Topics and services are in the core library whereas actionlibs were added as a package :

1) Topics [25] are event-based interfaces between the components/nodes. When a publisher node sends a message, every subscriber node is triggered via its callback function. A complete flow of data can be processed through topics only, having one node ”connected” to the next using the publisher/subscriber interaction. This connection is however not adapted when the flow is not fully linear and can hardly handle complex scenarios.
2) Services [26] are peer-to-peer interfaces between components/nodes. When a node calls a service proposed by a second component, the request/response communication permits to exchange data and/or execute tasks. To enhance the responsiveness, the communication link can remain open for successive calls but this solution has another major drawback: the caller node is blocked while waiting for the response.
3) Actionlibs [27] is a mechanism that partly combines the first two ideas and is one implementation of the task-state pattern [28]. A component proposing a service through an action server listens to a goal topic, and provides feedback and results to two other topics. The actionlib base class handles a state machine to easily interact with the server. A mean to pre-empt the running server is provided. This feature is particularly interesting to cancel or to update a goal. The caller node is not blocked, and can asynchronously check feedback and result.

In the HANDLE scenarios, event-based interfaces were excluded because the flow of data could become complex. However the topic-based interfaces were kept for low-level interaction (commands sent to controllers, raw data processed from sensors etc...). For high-level interaction, the choice remained between services and actionlibs. The following arguments convinced us to use actionslibs whenever possible:

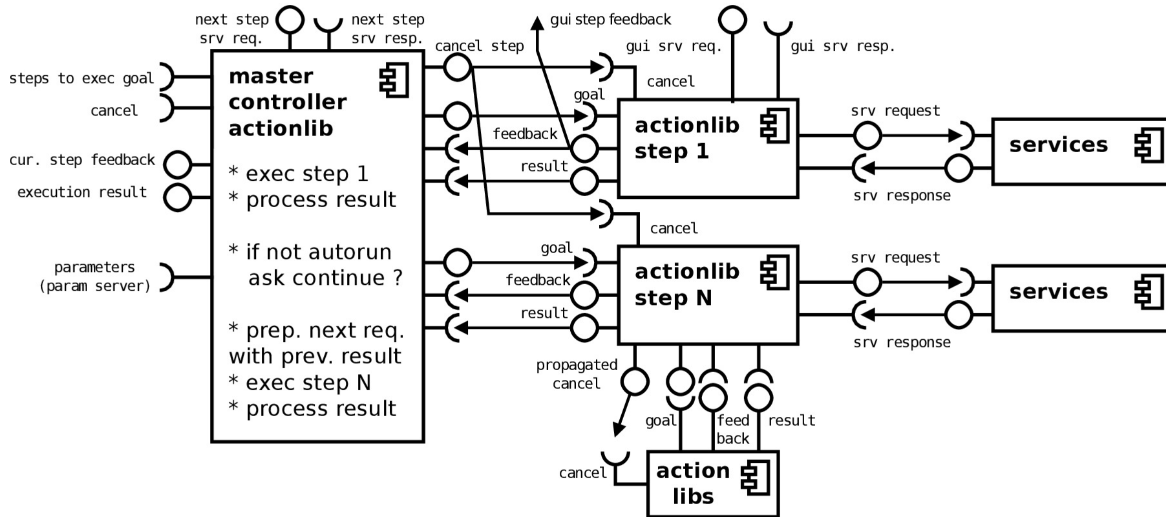


Fig. 12. A master controller actionlib interacts with the GUI to control and monitor the execution of the application, and interfaces with N steps actionlibs. Steps interface with other actionlibs and services. Each step can be pre-empted by a mechanism of cancel request propagation.

- Services are simple to implement but several nodes require interruption possibilities, especially execution nodes that move the hardware and must stop when something goes wrong. The original object manipulation stack [6] includes *reactive nodes* exactly doing that with actionlibs,
- Services cannot handle long computation steps that must be run in the background. When embedded in an actionlib, long computation steps can produce the result when available. The main node can query the result at any moment after it is ready, or wait for the result to be ready as in a service mode,
- Actionlibs permit to start multiple actions together, for instance several grasp generating nodes, and to take the result of the first node that finishes,
- Actionlibs can be nested provided that the pre-emption requests are propagated correctly in each server.

4.2 Coordination component

Requirements: The HANDLE project focused on in-hand manipulation and the main application was developed to show the advances in this domain. However, the main program was also responsible for the preparation phases such as recognizing and grasping the object to be in the situation of tasks execution with in-hand actions. That is why a coordination component, called the *master controller* (Fig. 12), was created that could process data flows, from the sensor to the respective components or from one component to another, in different situations (known/unknown objects, user choices, etc.). This step-based coordination component was also meant to be a debug tool that could easily re-run a failed step, without restarting the preparation phases. Finally a user interface was needed to monitor each step in simple and advanced display modes for debug.

Step structure: Once the flow chart was designed, a set of components was regrouped into steps. Each step is created as an actionlib calling other actionlibs or services (Fig. 12). For example, a step providing possible grasps associated to an object is necessary. This step combines several components such as a database grasp extraction node and several grasp generators. Depending on the object type, either one or another component is called. In the case of unknown objects several generators are started in parallel through the mechanism of actionlibs. When the first correct result is available the process continues whatever the generator that provided the result is.

Master controller: To coordinate and chain the steps in the correct fashion, the master controller node was created as an actionlib also. This node is responsible for preparing and distributing part of the data from one step to another and decides which step should be called depending on the previous result. It acts almost as a state machine but has less possible transitions, staying close to a pre-defined order of steps. Thanks to this node, it is easy to re-run a single step, re-using the same data that came out of the previous step. One can note that the pre-emption through the cancel topic is designed to be propagated from a step actionlib to the computing actionlibs and hence helps increase the reactivity of the system.

GUI: Controlling and monitoring the master controller is done through a GUI developed in rqt-gui[2] (Fig. 13). Interaction between the master controller actionlib and the GUI makes use of several key methods:

2. Although ROS Electric was used in the project, rqt-gui is on ROS Fuerte only but Electric and Fuerte run nicely together for actionlib interactions

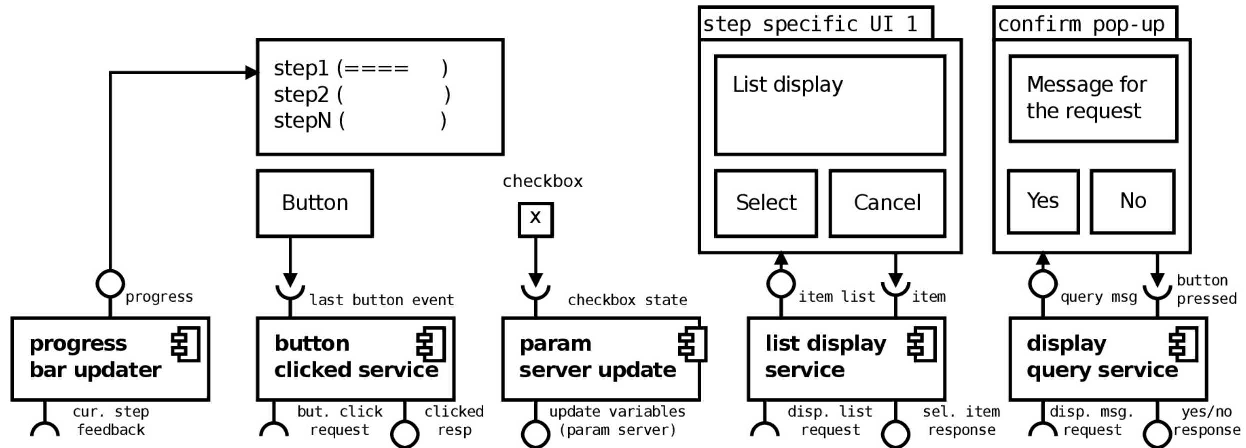


Fig. 13. A GUI concept providing services as interface.

- Requests from the master controller to the GUI, for example to ask the user a confirmation, is done through a service provided by the GUI. This service displays the request message in a pop-up window and sends the user selected button as a response to the caller.
- Buttons directly implement a service to get the master controller wait for the user to press a ”next” button for instance.
- Asynchronous data such as choices of check boxes/radio buttons are saved and exchanged between the GUI and the master controller through the parameter server. When the master needs to know the state of a checkbox, it queries the parameter server, which was updated by the GUI on click events.
- Progress bars in the GUI show the level of completion of each step, using the feedback mechanism of the actionlib. Indeed, step feedback messages were designed to contain a progress value among step-specific data.
- Step-specific feedback data are processed by step-specific UIs loaded in the main GUI. These specific UIs implement their own services or feedback processing units to exchange with a specific step.
- UIs and steps are developed together and are linked to the master through a configuration file with a standardized set of parameters.

All these mechanisms of interaction between the GUI and the master controller are not specifically designed for a GUI interaction. Since they rely on standard interfaces (service/actionlib/parameter server), any node capable of answering the requests could interact with the master. In fact, one can think of fake-user processes, or nodes generating fake (random) user data to automate/test the steps.

4.3 Integrating real and simulated worlds

With all the components ready and a master controller capable of running the main application, the whole set of nodes must be launched in different configurations such as simulated worlds or with the real platform. ROS includes a powerful mean to start nodes through an XML file called *launch file*.

Grouping by level: In the early integration or debug phases, the need to restart single nodes is frequent. However launch files usually configure and start several nodes together. For this reason, grouping components by level and also by steps proved to be practical (Fig. 14):

- low-level drivers were grouped, and then separated into vision, tactile and motor driver files. One can restart the hardware devices without breaking mid-level processing.
- mid-level nodes were grouped into processing unit files such as perception, planning, learning, database, utilities, etc. When developing one of these units, the nodes in it could be restarted keeping the others running.
- high-level applications were grouped to provide the manipulation stack functionality.
- monitoring-level GUI is one of the last node to be started.
- top-level steps actionlibs and the master controller node, which require all the components service/actionlibs to be ready, must be started at last.

Using ENV variables: Launch files also read ENV variables to configure parameters all over the included nodes. This feature was mainly used to distinguish simulation and real world runs. Some drivers or functions are not needed in simulation since all of the low-level drivers, tactile and vision plugins are started in a single application. The start of those drivers was therefore inhibited by an environment variable tested in the relevant launch files.

Machine parameter: Because of distributed computing, not all the components and nodes could be started on a single computer. ROS offers a method to start nodes through an ssh connection via a parameter called *machine*. The feature was used in most of the launch files, especially for drivers that are associated to hardware sitting on a dedicated machine.

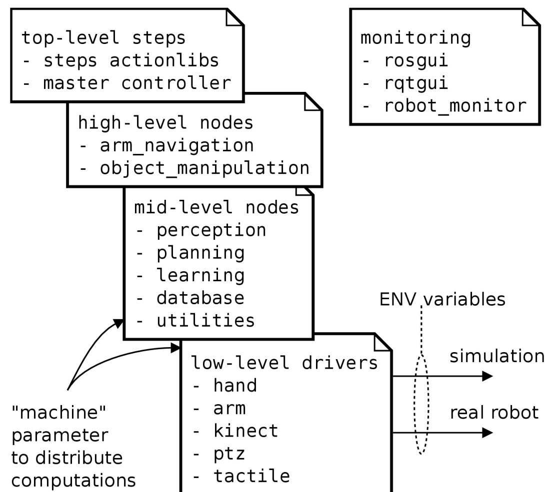


Fig. 14. Launch files regrouping components by level of action, using *machine* and *ENV variables* for further reconfigurability.

Overlaying: One possibility given by the ROS_PACKAGE_PATH variable was put into application in the project. To launch an application, ROS finds and executes the first corresponding node in the order of the PATH. It is possible to get a new package be called first provided that it has the same name as the package one wants to replace. This is called an overlay. As an example, the *sr_description* package, containing the URDF and tuning settings of the platform, is overlaid at different partners site, depending on the hardware at that location. With another version of the hand, overlaying the package and changing the model locally was rapid and safe for all surrounding nodes.

5 CONCLUSION

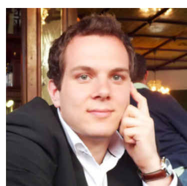
This paper showed the re-usability of ROS standard or community based packages in terms of flexibility, interoperability and reconfigurability. Other component-based specificities of ROS were highlighted such as distributed computing and multi-robot possibilities, multiplicity of the interfaces offered and easy coordination of components. The application of these features was detailed in examples of integration of the HANDLE project software architecture, showing the type of modifications that can be done to efficiently extend the original work to new robots or new purposes. A GUI and a step-based coordination component were described and useful hints about the integration process were given. The HANDLE project platform successfully proved the efficiency of the integration approach in the final demonstration combining more than a hundred of packages and components. As a future work, the nodes will be further transformed to be even more compatible with various hands with a different number of fingers.

REFERENCES

[1] Shadow Robot Company, “Shadow ethercat dual can motor hand.” http://www.shadowrobot.com. 1, 3.1
[2] D. Brugali and A. Shakhimardanov, “Component-based robotic engineering (part ii),” *IEEE Robotics and Automation Magazine*, vol. 17, no. 1, pp. 100–112, 2010. 1
[3] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in *ICRA Workshop on Open Source Software*, 2009. 2
[4] M. Ciocarlie, K. Hsiao, E. G. Jones, S. Chitta, R. B. Rusu, and I. A. Sucan, “Towards Reliable Grasping and Manipulation in Household Environments,” in *Intl. Symposium on Experimental Robotics (ISER)*, (New Delhi, India), 2010. 2, 2.3
[5] A. L. Christian Schlegel, Andreas Steck, *Introduction to Modern Robotics*, ch. Model-Driven Software Development in Robotics: Communication Patterns as Key for a Robotics Component Model. iConcept Press, 2012. 2
[6] M. Ciocarlie and K. Hsiao, “object manipulator stack.” http://www.ros.org/wiki/object_manipulator. 2.3, 4.1
[7] I. A. Sucan, M. Moll, and L. E. Kavraki, “The open motion planning library.,” *IEEE Robotics Automation Magazine*, vol. 19, no. 4, pp. 72–82, 2012. 2.3, 3.3
[8] R. Smits, “KDL: Kinematics and Dynamics Library.” http://www.orocos.org/kdl. 2.3, 3.1
[9] R. B. Rusu and S. Cousins, “3D is here: Point Cloud Library (PCL),” in *2011 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1–4, IEEE, May 2011. 2.3
[10] G. Bradski, “The OpenCV Library,” *Dr. Dobb’s Journal of Software Tools*, 2000. 2.3
[11] M. Ciocarlie, “household objects database.” http://www.ros.org/wiki/household_objects_database. 2.3
[12] Schunk GmbH & Co. KG, “Schunk Anthropomorphic Hand.” http://www.schunk.com/, May 2006. 3.1
[13] M. Grebenstein, A. Albu-Schaffer, T. Bahls, M. Chalon, O. Eiberger, W. Friedl, R. Gruber, S. Haddadin, U. Hagn, R. Haslinger, H. Hoppner, S. Jorg, M. Nickl, A. Nothhelfer, F. Petit, J. Reill, N. Seitz, T. Wimbock, S. Wolf, T. Wusthoff, and G. Hirzinger, “The dlr hand arm system,” in *ICRA, Shanghai, China, 2011*, pp. 3175–3182, IEEE, 2011. 3.1
[14] F. Ruiz, “Patches: Coupled joints, locked joints and python executable detection.” http://www.orocos.org/forum/rtt/rtt-dev/patches-coupled-joints-locked-joints-and-python-executable-detection, 2011. 3.1
[15] A. Miller and P. Allen, “Graspit! a versatile simulator for robotic grasping,” *Robotics Automation Magazine, IEEE*, vol. 11, pp. 110–122, Dec 2004. 3.1
[16] D. R. Faria, R. Martins, J. Lobo, and J. Dias, “Extracting data from human manipulation of objects towards improving autonomous robotic grasping,” *Robotics and Autonomous Systems*, vol. 60, no. 3, pp. 396 – 410, 2012. Autonomous Grasping. 3.1
[17] R. Krug and D. Dimitrovz, “Representing movement primitives as implicit dynamical systems learned from multiple demonstrations,” in *Advanced Robotics (ICAR), 2013 16th International Conference on*, pp. 1–8, Nov 2013. 3.1
[18] A. Bernardino, M. Henriques, N. Hendrich, and J. Zhang, “Precision grasp synergies for dexterous robotic hands,” in *Robotics and Biomimetics (ROBIO), 2013 IEEE International Conference on*, pp. 62–67, Dec 2013. 3.1
[19] N. Nicolas Burrus, M. Abderrahim, J. Garcia, and L. Moreno, “Object reconstruction and recognition leveraging an rgb-d camera,” in *The 12th IAPR Conference on Machine Vision Applications*, pp. 132–135, June 2011. 3.1
[20] CCNY Robotics Lab, “ccny_vision (including ar_pose) package.” http://www.ros.org/wiki/ccny_vision. 3.2
[21] H. Kato and M. Billinghurst, “Marker tracking and hmd calibration for a video-based augmented reality conferencing system,” in *Augmented Reality, 1999. (IWAR ’99) Proceedings. 2nd IEEE and ACM International Workshop on*, pp. 85–94, oct 1999. 3.2
[22] H. Liu, X. Song, J. Bimbo, K. Althoefer, and L. Senerivatne, “Intelligent fingertip sensing for contact information identification,” in *Advances in Reconfigurable Mechanisms and Robots I*, pp. 599–608, Springer London, 2012. 3.2

[23] ROS Industrial, "Ros packages for industrial robotics and automation applications." http://code.google.com/p/swri-ros-pkg. 3.3
[24] J. Corrales Ramon, V. Perdereau, and F. Torres Medina, "Multi-fingered robotic hand planner for object reconfiguration through a rolling contact evolution model," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pp. 625–630, May 2013. 3.3
[25] ROS-Concepts, "Ros topics." http://wiki.ros.org/Topics. 1
[26] ROS-Concepts, "Ros service." http://wiki.ros.org/Services. 2
[27] E. Marder-Eppstein and V. Pradeep, "Ros actionlib package documentation." http://www.ros.org/wiki/actionlib. 3
[28] I. Lütkebohle, R. Philippsen, V. Pradeep, E. Marder-Eppstein, and S. Wachsmuth, "Generic middleware support for coordinating robot software components: The task-state-pattern," *Journal of Software Engineering in Robotics*, vol. 1, no. 2, pp. 20–39, 2011. 3

Guillaume Walck received his Engineers degree in Embedded Systems from ENSEA Cergy-Pontoise in 2004, his M. Sc. degree in Robotics and Intelligent Systems and the Ph. D. degree in Robotics from Université Pierre et Marie Curie in 2006 and 2010, respectively. He was research engineer at Institut des Systèmes Intelligents et de Robotique in the HANDLE European project from 2009 to 2013 and since 2014 is lab system engineer in the CITEC of University of Bielefeld. His research interests cover robotics, control and software structure, embedded vision.



Ugo Cupcic received his Engineers degree in Bioinformatics from INSA Lyon in 2007 and his M. Sc. degree in Artificial Intelligence from Universite Pierre et Marie Curie in 2008. In 2008, he worked as a software engineer at Sword Group and is, since 2009, Senior Software Engineer at the Shadow Robot Company.

Toni Oliver Duran received his Engineers degree in Telecommunication from Universitat Politecnica de Catalunya in 2004. In 2002 he started working as embedded software and systems engineer at Amper, S.A. and is, since 2011, a software engineer at the Shadow Robot Company.

Veronique Perdereau is full professor at UPMC since 2003 and recognized as a IEEE senior member since 2001. She obtained her Electrical and Information engineering M. Sc in 1987 and Robotics and Automation Ph. D. degree in 1991. She is assistant professor at UPMC since 1987 and PhD advisor since 2000, full professor since 2003, chair of the IEEE Education France chapter. Her main research interests include Hybrid position/force control and Vision based control of robot manipulators, Multi-robot cooperation, Multi-fingered hand planning and control, Inverse kinematics, Redundant manipulator control, and Collision avoidance. She has published more than 70 scientific published papers. Since 2009, she is the coordinator of an FP7 funded IP, the HANDLE project. She is partner of the national research project ASSIST "Study and development of a two arms mobile manipulator for assistance of handicapped people". She is also responsible for the International Master Program "Mechatronic systems for Rehabilitation" at UPMC in collaboration with Brescia University.