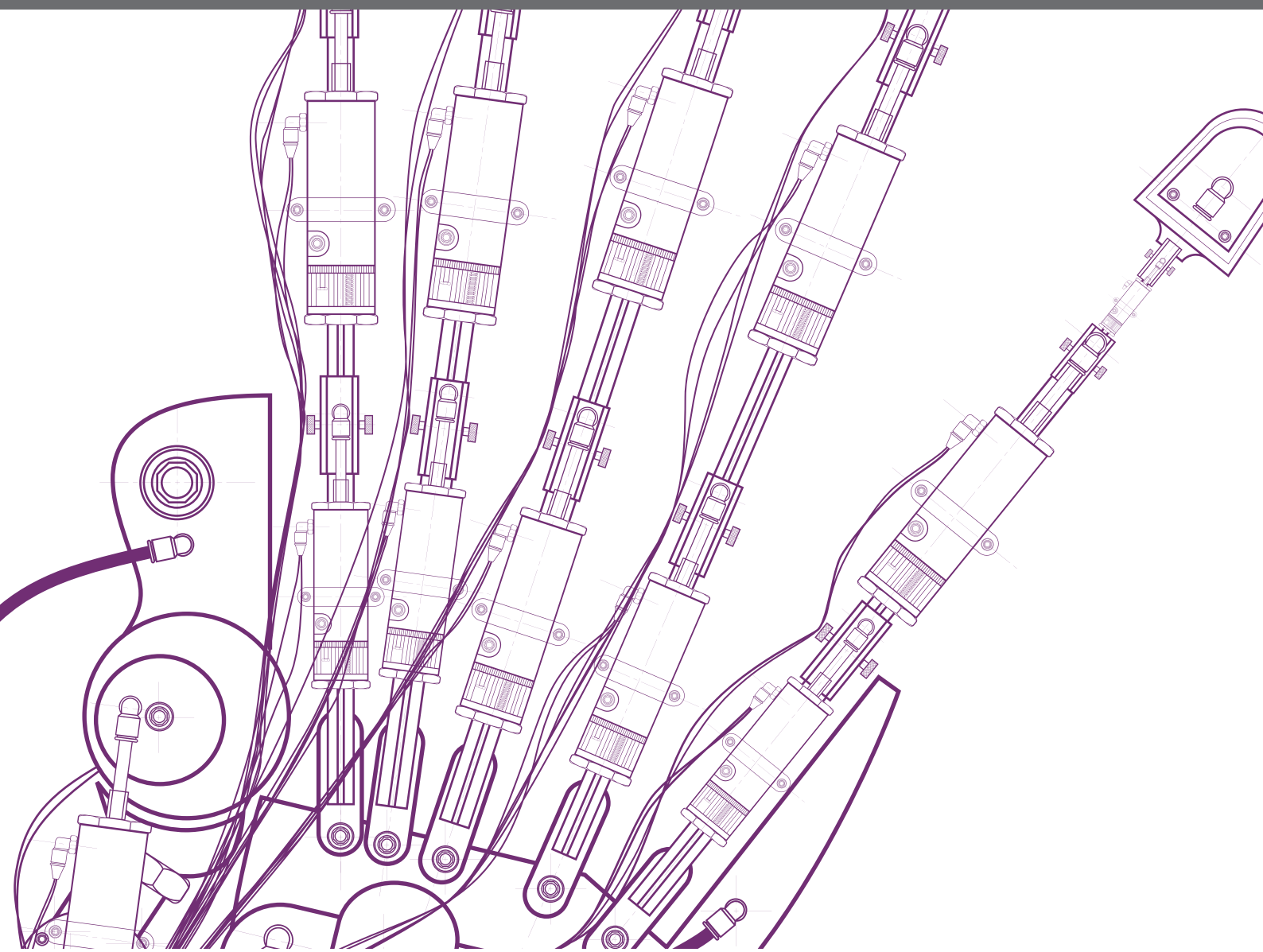
SOFTWARE ARCHITECTURES FOR HUMANOID ROBOTICS

EDITED BY: Lorenzo Natale, Tamim Asfour, Fumio Kanehiro and Nikolaus Vahrenkamp

PUBLISHED IN: Frontiers in Robotics and AI



frontiers Research Topics

frontiers

ISSN 1664-8714
ISBN 978-2-88945-590-4
DOI 10.3389/978-2-88945-590-4

About Frontiers

Frontiers is more than just an open-access publisher of scholarly articles: it is a pioneering approach to the world of academia, radically improving the way scholarly research is managed. The grand vision of Frontiers is a world where all people have an equal opportunity to seek, share and generate knowledge. Frontiers provides immediate and permanent online open access to all its publications, but this alone is not enough to realize our grand goals.

Frontiers Journal Series

The Frontiers Journal Series is a multi-tier and interdisciplinary set of open-access, online journals, promising a paradigm shift from the current review, selection and dissemination processes in academic publishing. All Frontiers journals are driven by researchers for researchers; therefore, they constitute a service to the scholarly community. At the same time, the Frontiers Journal Series operates on a revolutionary invention, the tiered publishing system, initially addressing specific communities of scholars, and gradually climbing up to broader public understanding, thus serving the interests of the lay society, too.

Dedication to Quality

Each Frontiers article is a landmark of the highest quality, thanks to genuinely collaborative interactions between authors and review editors, who include some of the world's best academicians. Research must be certified by peers before entering a stream of knowledge that may eventually reach the public - and shape society; therefore, Frontiers only applies the most rigorous and unbiased reviews.
Frontiers revolutionizes research publishing by freely delivering the most outstanding research, evaluated with no bias from both the academic and social point of view. By applying the most advanced information technologies, Frontiers is catapulting scholarly publishing into a new generation.

What are Frontiers Research Topics?

Frontiers Research Topics are very popular trademarks of the Frontiers Journals Series: they are collections of at least ten articles, all centered on a particular subject. With their unique mix of varied contributions from Original Research to Review Articles, Frontiers Research Topics unify the most influential researchers, the latest key findings and historical advances in a hot research area! Find out more on how to host your own Frontiers Research Topic or contribute to one as an author by contacting the Frontiers Editorial Office: researchtopics@frontiersin.org

SOFTWARE ARCHITECTURES FOR HUMANOID ROBOTICS

Topic Editors:
Lorenzo Natale, Fondazione Istituto Italiano di Technologia, Italy
Tamim Asfour, Karlsruher Institut für Technologie (KIT), Germany
Fumio Kanehiro, National Institute of Advanced Industrial Science and Technology (AIST), Japan
Nikolaus Vahrenkamp, Karlsruher Institut für Technologie (KIT), Germany

Citation: Natale, L., Asfour, T., Kanehiro, F., Vahrenkamp, N., eds. (2018). Software Architectures for Humanoid Robotics. Lausanne: Frontiers Media.
doi: 10.3389/978-2-88945-590-4

Table of Contents

frontiers in Robotics and AI

TECHNOLOGY REPORT
published: 13 October 2016
doi: 10.3389/frobt.2016.00057

Supervised Autonomy for Exploration and Mobile Manipulation in Rough Terrain with a Centaur-Like Robot

Max Schwarz, Marius Beul, David Droeschel, Sebastian Schüller, Arul Selvam Periyasamy, Christian Lenz, Michael Schreiber and Sven Behnke*

Autonomous Intelligent Systems, Institute for Computer Science VI, University of Bonn, Bonn, Germany

Planetary exploration scenarios illustrate the need for autonomous robots that are capable to operate in unknown environments without direct human interaction. At the DARPA Robotics Challenge, we demonstrated that our Centaur-like mobile manipulation robot Momaro can solve complex tasks when teleoperated. Motivated by the DLR SpaceBot Cup 2015, where robots should explore a Mars-like environment, find and transport objects, take a soil sample, and perform assembly tasks, we developed autonomous capabilities for Momaro. Our robot perceives and maps previously unknown, uneven terrain using a 3D laser scanner. Based on the generated height map, we assess drivability, plan navigation paths, and execute them using the omnidirectional drive. Using its four legs, the robot adapts to the slope of the terrain. Momaro perceives objects with cameras, estimates their pose, and manipulates them with its two arms autonomously. For specifying missions, monitoring mission progress, on-the-fly reconfiguration, and teleoperation, we developed a ground station with suitable operator interfaces. To handle network communication interruptions and latencies between robot and ground station, we implemented a robust network layer for the ROS middleware. With the developed system, our team NimbRo Explorer solved all tasks of the DLR SpaceBot Camp 2015. We also discuss the lessons learned from this demonstration.

Keywords: mapping, mobile manipulation, navigation, perception for grasping and manipulation, space robotics and automation

OPEN ACCESS

Edited by:
Fumio Kanehiro, National Institute of Advanced Industrial Science and Technology, Japan

Reviewed by:
John Nassour, Chemnitz University of Technology, Germany
Maja Borry Zorjan, Université de Versailles Saint-Quentin-en-Yvelines, France
Giuseppe Quaglia, Polytechnic University of Turin, Italy

****Correspondence:***
Max Schwarz
max.schwarz@uni-bonn.de

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *21 January 2016*
Accepted: *16 September 2016*
Published: *13 October 2016*

Citation:
Schwarz M, Beul M, Droeschel D, Schüller S, Periyasamy AS, Lenz C, Schreiber M and Behnke S (2016) Supervised Autonomy for Exploration and Mobile Manipulation in Rough Terrain with a Centaur-Like Robot. Front. Robot. AI 3:57. doi: 10.3389/frobt.2016.00057

1. INTRODUCTION

In planetary exploration scenarios, robots are needed that are capable to autonomously operate in unknown environments and highly unstructured and unpredictable situations. Since human workers cannot be deployed due to economic or safety constraints, autonomous robots have to robustly solve complex tasks without human intervention. To address this need, the German Aerospace Center (DLR) held the DLR SpaceBot Camp 2015.[1] Ten German research groups were supported to foster the development of robots, capable of autonomously solving complex tasks that are required in a typical planetary exploration scenario. During the SpaceBot Camp, the robots need to tackle these tasks:

- Find and identify three previously known objects in a planetary-like environment (cup, battery, and base station).

[1] http://www.dlr.de/rd/desktopdefault.aspx/tabid-8101/

- Take a soil sample of a previously known spot (optional).
- Pick up and deliver the cup and the battery to the base station.
- Assemble all objects.

All tasks had to be completed as autonomously as possible, including perception, manipulation, and navigation, in difficult terrain with slopes up to 15° that need to be traversed and larger untraversable slopes. The overall weight of the deployed robotic system was limited to 100 kg, and the total time for solving all tasks was 60 min. A rough height map with 50 cm resolution of the environment was known prior to the run. The use of any global navigation satellite system (GNSS) was prohibited. No line-of-sight between the robot and the crew was allowed, and communication between the robot and the operators was severely restricted. Data transmission was bidirectionally delayed by 2 s, resulting in a round trip time of 4 s – too large for direct remote control. Furthermore, the uplink connection was blocked entirely after 20 min and 40 min for 4 min each. More details on the SpaceBot Camp itself and our performance are provided in Section 11.

To address the tasks, we used the mobile manipulation robot Momaro (see **Figure 1**), which is configured and monitored from a ground station. Momaro is equipped with four articulated compliant legs that end in pairs of directly driven, steerable wheels. To perform a wide range of manipulation tasks, Momaro has an anthropomorphic upper body with two 7 degrees of freedom (DOF) manipulators that end in dexterous grippers. This allows for the single-handed manipulation of smaller objects, as well as for two-armed manipulation of larger objects and the use of tools. Through adjustable base height and attitude and a yaw joint in the spine, Momaro has a work space equal to the one of an adult person.

The SpaceBot Camp constitutes a challenge for autonomous robots. Since the complex navigation and manipulation tasks require good situational awareness, Momaro is equipped with a 3D laser scanner, multiple color cameras, and an RGB-D camera. For real-time perception and planning, Momaro is equipped with a powerful onboard computer. The robot communicates to a relay at the landing site via WiFi and is equipped with a rechargeable LiPo battery (details provided in Section 3).

The developed system was tested at the SpaceBot Camp 2015. Momaro solved all tasks autonomously in only 20:25 out of 60 min including the optional soil sample. No official ranking was conducted at the SpaceBot Camp, but since we were the only team solving all these tasks, we were very satisfied with the performance. We report in detail on how the tasks were solved. Our developments led to multiple contributions, which are summarized in this article, including the robust perception and state estimation system, navigation and motion–planning modules and autonomous manipulation and control methods. We also discuss lessons learned from the challenging robot operations.

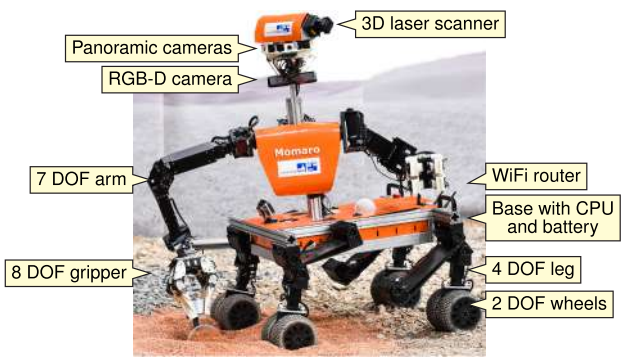


FIGURE 1 | The mobile manipulation robot Momaro taking a soil sample.

2. RELATED WORK

The need for mobile manipulation has been addressed in the past with the development of a variety of mobile manipulation systems, consisting of robotic arms installed on mobile bases with the mobility provided by wheels, tracks, or leg mechanisms. Several research projects exist that use purely wheeled locomotion for their robots (Mehling et al., 2007; Borst et al., 2009). In the previous work, we developed NimbRo Explorer (Stückler et al., 2015), a six-wheeled robot equipped with a 7 DOF arm designed for mobile manipulation in rough terrain, encountered in planetary exploration scenarios.

Wheeled rovers provide optimal solutions for well structured and relatively flat environments; however, outside of these types of terrains, their mobility quickly reaches its limits. Often they can only overcome obstacles smaller than the size of their wheels. Compared to wheeled robots, legged robots are more complex to design, build, and control (Raibert et al., 2008; Roennau et al., 2010; Semini et al., 2011; Johnson et al., 2015), but they have obvious mobility advantages when operating in unstructured terrains and environments. Some research groups have started investigating mobile robot designs that combine the advantages of both legged and wheeled locomotion, using different coupling mechanisms between the wheels and legs (Adachi et al., 1999; Endo and Hirose, 2000; Halme et al., 2003). In the context of the DARPA Robotics Challenge, multiple teams (beside ours) used hybrid locomotion designs (Hebert et al., 2015; Stentz et al., 2015). In particular, the winning team KAIST (Kim and Oh, 2010; Cho et al., 2011) used wheels on the knees of their humanoid robot to move quickly and safely between different tasks on flat terrain.

In 2013, DLR held a very similar SpaceBot competition which encouraged several robotic developments (Kaupisch et al., 2015). Heppner et al. (2015) describe one of the participating systems, such as the six-legged walking robot LAURON V. LAURON is able to overcome challenging terrain, although its six legs limit the locomotion speed in comparison to wheeled robots. As with our system, the software architecture is based on the Robot Operating System [ROS (Quigley et al., 2009)].

Sünderhauf et al. (2014) developed a cooperative team of two-wheeled robots, named Phobos and Deimos. The straightforward, rugged design with skid steering performed well, compared to more complicated locomotion approaches. We made the same observation in our participation at the SpaceBot Competition 2013 and opted to include wheels (opposed to a purely legged concept) in the Momaro robot. In the 2013 competition, Phobos and Deimos mainly had communication issues such that the ground station crew could neither stop Phobos from colliding

with the environment nor start Deimos to resume the mission. These problems highlight why we spent considerable effort on our communication subsystem (see Section 9) to ensure that the operator crew has proper situational awareness and is able to continuously supervise the robotic operation.

Schwendner et al. (2014) and Joyeux et al. (2014) discuss the six-wheeled Artemis rover. Artemis is able to cope with considerable terrain slopes (up to 45°) through careful mechanical design. In contrast, Momaro has to employ active balancing strategies (see Section 6) to prevent tipping over due to its high center of mass. The authors emphasize the model-driven design of both hardware and software. The latter is partly ROS based but also has modules based on the Rock framework. Artemis demonstrated its navigation capabilities in the 2013 competition, but eventually its navigation planners became stuck in front of a trench, again highlighting the need to design systems with enough remote access, so that problems can be diagnosed and fixed remotely.

A few articles on the SpaceBot Camp 2015 are already available. Kaupisch and Fleischmann (2015) describe the event and report briefly on the performances of all teams. Wedler et al. (2015) present the general design of their Lightweight Rover Unit (LRU), which competed in the SpaceBot Camp 2015, successfully solving all tasks except the optional soil sample task. The LRU is a four-wheeled rover with steerable wheels, similar to Momaro's drive. Comparable to our flexible legs, the suspension uses both active and passive mechanisms. However, the LRU wheels are rigidly coupled with pairs, and the base height cannot be adapted. Overall, the LRU seems geared toward building a robust and hardened rover for real missions, while Momaro's components are not suitable for space. On the other hand, Momaro can solve tasks requiring stepping motions and is capable of dexterous bimanual manipulation.

In our previous work, we describe the Explorer system used in the 2013 competition (Stückler et al., 2015) and its local navigation system (Schwarz and Behnke, 2014). Compared to the 2013 system, we improve on the

- capabilities of the mechanical design (e.g., execution of stepping motions or bimanual manipulation),
- grade of autonomy (execution of full missions, including assembly tasks at the base station),
- situational awareness of the operator crew, and
- robustness of network communication.

The local navigation approach has moved from a hybrid laser scanner and RGB-D system on three levels to a laser scanner-only system on two levels – allowing operation in regions where current RGB-D sensors fail to measure distance (e.g., in direct sunlight).

In contrast to many other systems, Momaro is capable of driving omnidirectionally, which simplifies navigation in restricted spaces and allows us to make small lateral positional corrections faster. Furthermore, our robot is equipped with six limbs, two of which are exclusively used for manipulation. The use of four legs for locomotion provides a large and flexible support polygon when the robot is performing mobile manipulation tasks. The Momaro system demonstrated multiple complex tasks under teleoperation in the DARPA Robotics Challenge (Schwarz et al., 2016).

Supervised autonomy has been proposed as a development paradigm by Cheng and Zelinsky (2001), who shift basic autonomous functions like collision avoidance from the supervisor back to the robot, while offering high-level interfaces to configure the functions remotely. In contrast to human-in-the-loop control, supervised autonomy is more suited toward the large latencies involved in space communications. Gillett et al. (2001) use supervised autonomy in the context of an unmanned satellite servicing system that must perform satellite capture autonomously. The survey conducted by Pedersen et al. (2003) not only highlights the (slow) trend in space robotics toward more autonomous functions but also points out that space exploration will always have a human component, if only as consumers of the data produced by the robotic system. In this manner, supervised autonomy is also the limit case of sensible autonomy in space exploration.

3. MOBILE MANIPULATION ROBOT MOMARO

3.1. Mechanical Design

Our mobile manipulation robot Momaro (see **Figure 1**) was constructed with several design goals in mind:

- universality,
- modularity,
- simplicity, and
- low weight.

In the following, we detail how we address these goals.

3.1.1. Universality

Momaro features a unique locomotion design with four legs ending in steerable wheels. This design allows to drive omnidirectionally and to step over obstacles or even climb. Since it is possible to adjust the total length of the legs, Momaro can manipulate obstacles on the ground, as well as reach to heights of up to 2 m. Momaro can adapt to the slope of the terrain through leg length changes.

On its base, Momaro has an anthropomorphic upper body with two adult-sized 7 DOF arms, enabling it to solve complex manipulation tasks. Attached to the arms are two 8-DOF dexterous hands consisting of four fingers with two segments each. The distal segments are 3D printed and can be changed without tools for easy adaption to a specific task. For the SpaceBot Camp, we designed distal finger segments that maximize the contact surface to the SpaceBot objects: the finger tips are shaped to clamp around the circumference of the cylindrical cup object (see **Figure 3**). The box-shaped battery object is first grasped using the proximal finger segments, and then locked in-place with the distal finger segments as soon as it is lifted from the ground.

The upper body can be rotated around the spine with an additional joint, thus increasing the workspace. Equipped with these various DOF, Momaro can solve most diverse tasks. If necessary, Momaro is even able to use tools. We showed this ability by taking a soil sample with a scoop at the SpaceBot Camp (see **Figure 2**).

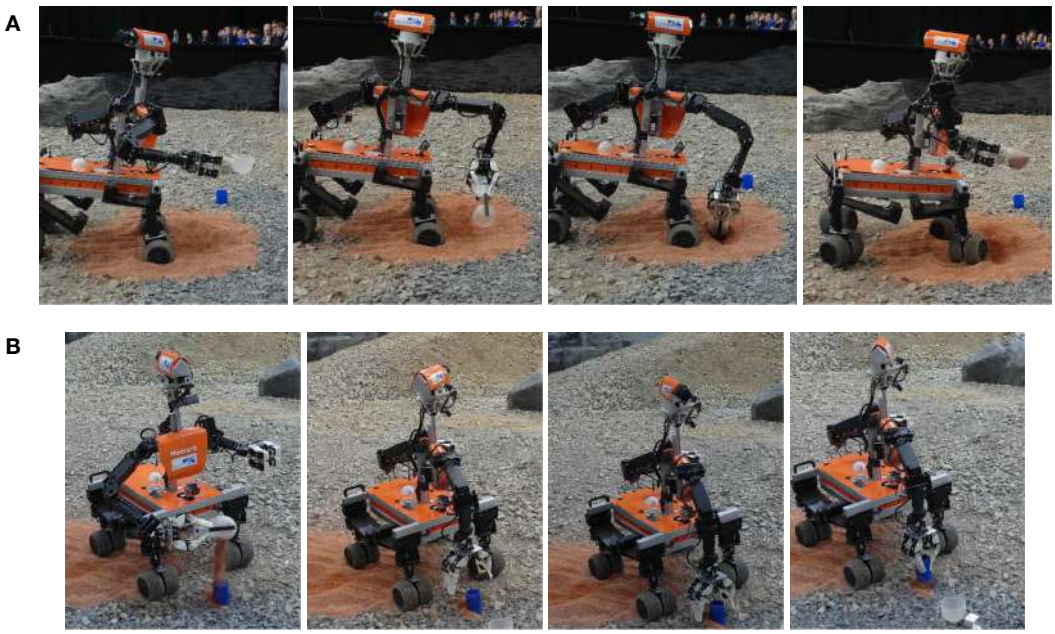


FIGURE 2 | Manipulation capabilities. **(A)** Momaro is using a scoop to take a soil sample. **(B)** After filling the blue cup with previously scooped soil, Momaro discards the scoop and grasps the cup to deliver it to a base station.

3.1.2. Modularity

All joints of Momaro are driven by Robotis Dynamixel actuators, which offer a good torque-to-weight ratio. While the finger actuators and the rotating laser scanner actuator are of the MX variant, all others are Dynamixel Pro actuators. **Figure 3** gives an overview of the DOF of Momaro. For detailed information on Momaro's actuators, we refer to Schwarz et al. (2016).

Using similar actuators for every DOF simplifies maintenance and repairs. For example, at the SpaceBot Camp, one of the shoulder actuators failed shortly before our run. A possibility could have been to repair the vital shoulder using a knee actuator, since the knees were hardly used in this demonstration. Fortunately, we acquired a spare actuator in time. Details can be found in Section 11.

3.1.3. Simplicity

For Momaro, we chose a four-legged locomotion design over bipedal approaches. The motivation for this choice was mainly the reduction in overall complexity, since balance control and fall recovery are not needed. Each leg has three degrees of freedom in hip, knee, and ankle. To reach adequate locomotion speeds on flat terrain, where steps are not needed, the legs are equipped with steerable wheel pairs. For omnidirectional driving, the wheel pairs can be rotated around the yaw axis, and each wheel can be driven independently. The legs also provide passive adaption to the terrain, as the leg segments are made from flexible carbon fiber and act as springs. The front legs have a vertical extension range of 40 cm. For climbing inclines, the hind legs can be extended 15 cm further. Using these features, obstacles lower than approximately 5 cm can be ignored.

3.1.4. Low Weight

Momaro is relatively lightweight (58 kg) and compact (base footprint 80 cm $\times$ 70 cm). During development and deployment, this is a strong advantage over heavier robots, which require large crews and special equipment to transport and operate. In contrast, Momaro can be carried by two people. In addition, it can be transported in standard suitecases by detaching the legs and torso.

3.2. Sensing

Momaro carries a custom-built 3D rotating laser scanner (see **Figure 3**) for simultaneous mapping and localization (see Section 5). As with previous robots (Stückler et al., 2015), a Hokuyo UTM-30LX-EW laser scanner is mounted on a slip ring actuated by a Robotis Dynamixel MX-64 servo, which rotates it around the vertical axis. For state estimation and motion compensation during a 3D scan, a PIXHAWK IMU is mounted close to the laser scanner.

For object detection, Momaro features an ASUS Xtion Pro Live RGB-D camera. Since Momaro's origins are in teleoperated scenarios (Schwarz et al., 2016), it also carries seven color cameras – three panoramic cameras and one downward-facing wide-angle camera mounted on the head, one camera mounted in each hand, and one wide-angle camera below the base. In a supervised autonomy scenario, these cameras are mainly used for monitoring the autonomous operation.

3.3. Electronics

Figure 3 gives an overview of the electrical components of Momaro. For onboard computation, an off-the-shelf mainboard with a fast CPU (Intel Core i7-4790K @4–4.4 GHz) and

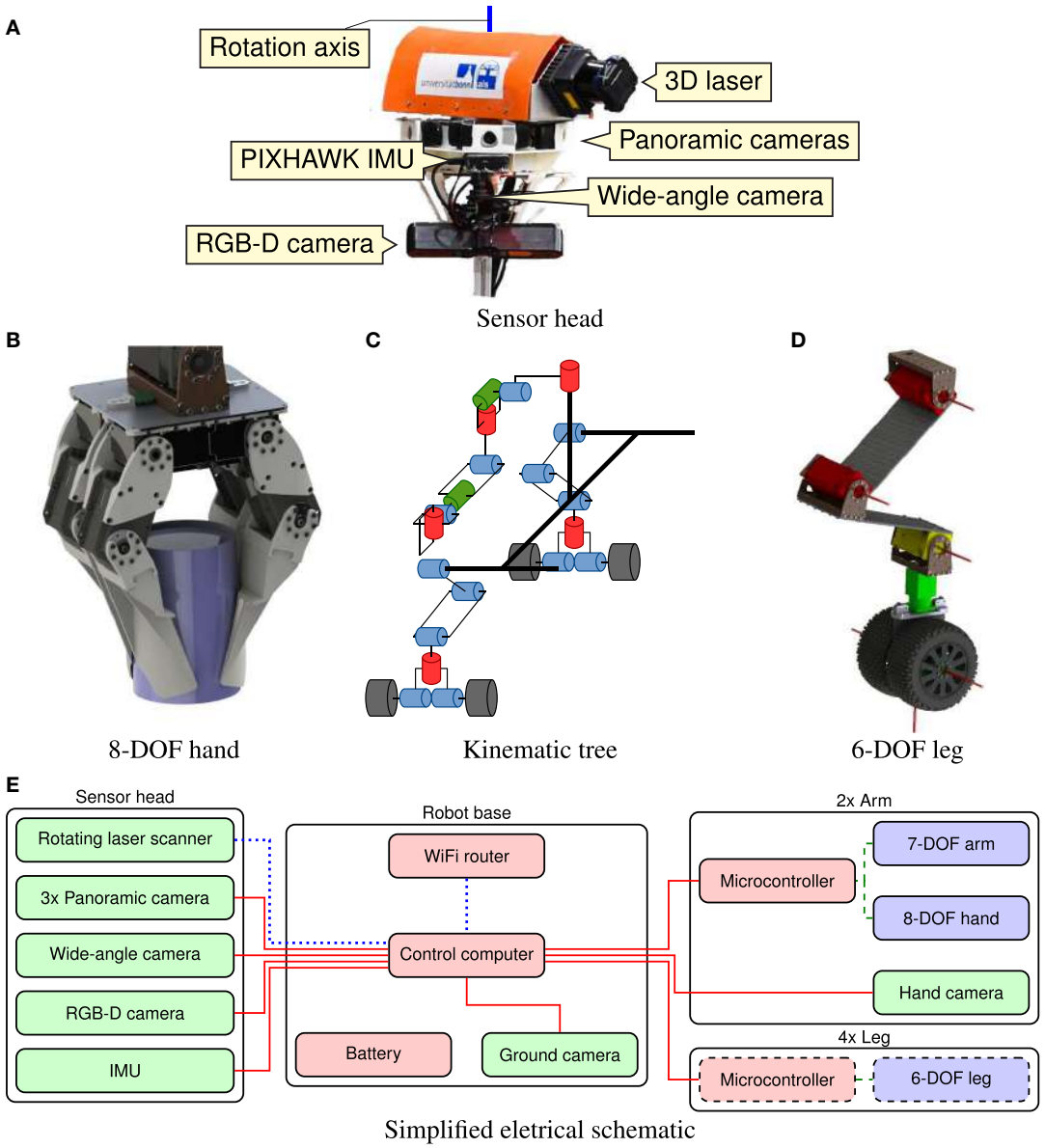


FIGURE 3 | Hardware components. **(A)** Sensor head carrying 3D laser scanner, IMU, four cameras, and an RGB-D camera. **(B)** The 8-DOF hand has specialized fingers for grasping the objects. **(C)** Kinematic tree of one half of Momaro. The hand is excluded for clarity. Proportions are not to scale. **(D)** The front left leg. The red lines show the axes of the six joints. **(E)** Simplified overview of the electrical components of Momaro. Sensors are colored green, actuators blue, and other components red. We show USB 2.0 data connections (red), LAN connections (dotted, blue), and the low-level servo bus system (dashed, green).

32 GB RAM is installed in the base. Communication with up to 1300 Mbit/s to the ground station is achieved through a NETGEAR Nighthawk AC1900 WiFi router. The hot-swappable six-cell 355 Wh LiPo battery yields around 1.5–2 h run time. Momaro can also run from a power supply for more comfortable development.

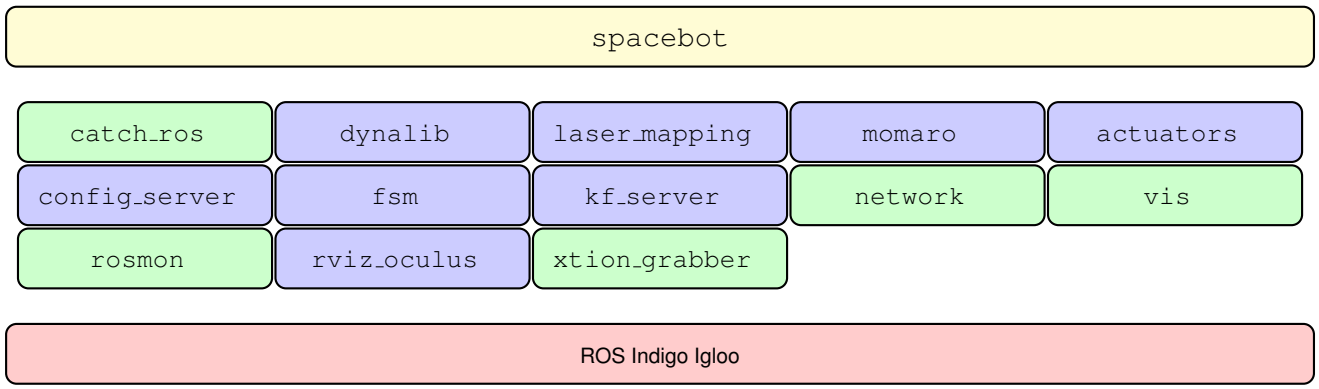
For more details on Momaro's hardware design, we refer to Schwarz et al. (2016).

4. SOFTWARE ARCHITECTURE

Both the Momaro robot and the scenarios we are interested will require highly sophisticated software. To retain modularity and maintainability and encourage code re-use, we built our software on top of the popular ROS [Robot Operating System (Quigley et al., 2009)] middleware. ROS provides isolation of software components into separate nodes (processes) and inter- and intraprocess communication via a publisher/subscriber scheme. ROS has seen widespread adoption in the robotics community and has a large collection of freely available open-source packages.

To support the multitude of robots and applications in our group,[2] we have a set of common modules, implemented as Git repositories. These modules (blue and green in **Figure 4**) are used across projects as needed. On top of the shared modules, we

[2]http://ais.uni-bonn.de/research.html



Package	Description
`fsm`	Finite state machine library
`kf_server`	Keyframe editing and interpolation, see Section 8
`laser_mapping`	Laser scanner SLAM using Multi-resolution Surfel Maps, see Section 5
`momaro`	Hardware support for the Momaro robot
`network`	Robust network transport for ROS, see Section 9
`robotcontrol`	Plugin-based real-time robot control node
`rosmon`	ROS process monitoring
`rviz_oculus`	Oculus Rift integration for RViz

FIGURE 4 | Organization of software modules. At the base, the ROS middleware is used. The blue colored boxes correspond to software modules, shared across robots, projects, and competitions. Finally, the spacebot module contains software, specific to the SpaceBot Camp. Modules colored in green have been released as open source, see https://github.com/AIS-Bonn.

have a repository for the specific application (e.g., DLR SpaceBot Camp 2015, yellow in **Figure 4**), containing all configuration, and code required exclusively by this application. The collection of repositories is managed by the `wstool` ROS utility.

Protection against unintended regressions during the development process is best gained through unit tests. The project-specific code is hard to test, though, since it is very volatile, on the one hand, and testing would often require full-scale integration tests using a simulator, on the other hand. This kind of integration tests have not been developed yet. In contrast, the core modules are very stable and can be augmented easily with unit tests. Unit tests in all repositories are executed nightly on a Jenkins server, which builds the entire workspace from scratch, gathers any compilation errors and warnings, and reports test results.

5. MAPPING AND LOCALIZATION

For autonomous navigation during a mission, our system continuously builds a map of the environment and localizes within this map. To this end, 3D scans of the environment are aggregated in a robot-centric local multiresolution map. The 6D sensor motion is estimated by registering the 3D scan to the map using our efficient surfel-based registration method (Droeschel et al., 2014a). In order to obtain an allocentric map of the environment – and to localize in it – individual local maps are aligned to each other using the same surfel-based registration method. A pose graph that connects the maps of neighboring key poses is optimized globally. **Figure 5** outlines our mapping system.

5.1. Preprocessing and 3D Scan Assembly

Before assembling 3D point clouds from measurements of the 2D laser scanner, we filter out the so-called *jump edges*. Jump edges arise at transitions between two objects and result in spurious measurements. These measurements can be detected by comparing the angle between neighboring measurements and are removed from the raw measurements of the laser scanner. The remaining measurements are then assembled to a 3D point cloud after a full rotation of the scanner. During assembly, raw measurements are undistorted to account for motion of the sensor during rotation.

We estimate the motion of the robot during a full rotation of the sensor from wheel odometry and measurements from the PIXHAWK IMU mounted in the sensor head. Rotational motions are estimated from gyroscopes and accelerometers, whereas linear motions are estimated by filtering wheel odometry with linear acceleration from the IMU. The resulting motion estimate is applied to the remaining measurements by means of spherical linear interpolation.

5.2. Local Mapping

The filtered and undistorted 3D point clouds are aggregated in a robot-centric multiresolution grid map as shown in **Figure 5**. The size of the grid cell increases with the distance from the robot, resulting in a fine resolution in the direct workspace of the robot and a coarser resolution farther away. The robot-centric property of the map is maintained by shifting grid cells according to the

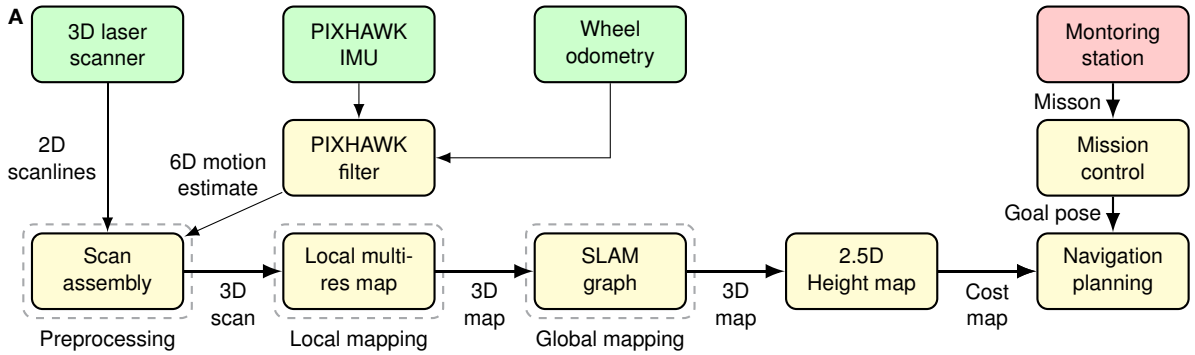


FIGURE 5 | SLAM and navigation architecture. **(A)** Overview of our mapping, localization, and navigation system. After filtering spurious measurements and assembling 3D point clouds (Section 5.1), measurements are aggregated in a robot-centric multiresolution map (Section 5.2) using surfel-based registration. Keyframe views of local maps are registered against each other in a SLAM graph (Section 5.3). A 2.5D height map is used to assess drivability. A standard 2D grid-based approach is used for planning (Section 6). **(B)** 3D points stored in the map on the robot. Color encodes height from ground. **(C)** The robot-centric multiresolution map with increasing cell size from the robot center. Color indicates the cell length from 0.25 m on the finest resolution to 2 m on the coarsest resolution.

robot motion – efficiently implemented by using circular buffers. Using robot-centric multiresolution facilitates efficiency in terms of memory consumption and computation time.

Besides 3D measurements from the laser scanner, each grid cell stores an occupancy probability – allowing to distinguish between occupied, free, and unknown areas. Similar to Hornung et al. (2013), we use a beam-based inverse sensor model and raycasting to update the occupancy probability of a cell. For every measurement in the 3D scan, we update the occupancy information of cells on the ray between the sensor origin and the endpoint.

After a full rotation of the laser, the newly acquired 3D scan is registered to the so far accumulated map to compensate for drift of the estimated motion. For aligning a 3D scan to the map, we use our surfel-based registration method (Droeschel et al., 2014a) – designed for this data structure, it leverages the multiresolution property of the map and gains efficiency by summarizing 3D points to surfels that are used for registration. Measurements from the aligned 3D scan replace older measurements in the map and are used to update the occupancy information.

5.3. Allocentric Mapping

We incorporate measurements from the wheel odometry, IMU, and local registration results to track the pose of the robot over a short period of time. To overcome drift and to localize the robot with respect to a fixed frame, we build an allocentric map from the robot-centric multiresolution maps acquired at different view poses (Droeschel et al., 2014b).

We construct a pose graph consisting of nodes, which are connected by edges. Each node corresponds to a view pose and its local multiresolution map. Nearby nodes are connected by edges, modeling spatial constraints between two nodes. Each spatial constraint is a normally distributed estimate with mean and covariance. An edge describes the relative position between

two nodes, arising from aligning two local multiresolution maps with each other. Similar to the alignment of a newly acquired 3D scan, two local multiresolution maps are aligned by surfel-based registration. Each edge models the uncertainty of the relative position by its information matrix, which is established by the covariance from registration. A new node is generated for the current view pose, if the robot moved sufficiently far.

In addition to edges between the previous node and the current node, we add spatial constraints between close-by nodes in the graph that are not in temporal sequence. By adding edges between close-by nodes in the graph, we detect loop closures. Loop closure allows us to minimize drift from accumulated registration errors, for example, if the robot traverses unknown terrain and reenters a known part of the environment.

From the graph of spatial constraints, we infer the probability of the trajectory estimate given all relative pose observations using the g^{2}o framework (Kuemmerle et al., 2011). Optimization is performed when a loop closure has been detected, allowing for online operation.

5.4. Localization

While traversing the environment, the pose graph is extended and optimized whenever the robot explores previously unseen terrain. We localize toward this pose graph during mission to estimate the pose of the robot in an allocentric frame. When executing a mission, e.g., during the SpaceBot Camp, the robot traverses goal poses w.r.t. this allocentric frame.

To localize the robot within the allocentric pose graph, the local multiresolution map is registered toward the closest node in the graph. By aligning the dense local map to the pose graph – instead of the relative sparse 3D scan – we gain robustness, since information from previous 3D scans is incorporated. The resulting registration transform updates the allocentric robot pose. To gain allocentric localization poses during acquisition of the scan, the 6D motion estimate from wheel odometry, and IMU is used to extrapolate the last allocentric pose.

During the SpaceBot Camp, we assumed that the initial pose of the robot was known, either by starting from a predefined pose or by means of manually aligning our allocentric coordinate frame with a coarse height map of the environment. Thus, we could navigate to goal poses in the coarse height map by localizing toward our pose graph.

5.5. Height Mapping

As a basis for assessing drivability, the 3D map is projected into a 2.5D height map, shown in **Figure 6**. In case multiple measurements are projected into the same cell, we use the measurement with median height. Gaps in the height map (cells without measurements) are filled with are local weighted mean if the cell has at least two neighbors within a distance threshold (20 cm in our experiments). This provides a good approximation of occluded terrain until the robot is close enough to actually observe it. After filling gaps in the height map, the height values are spatially filtered using the fast median filter approximation using local histograms (Huang et al., 1979). The resulting height map is suitable for navigation planning (see Section 6).

6. NAVIGATION

Our autonomous navigation solution consists of two layers: the global path planning layer and the local trajectory planning layer. Both planners are fed with cost maps calculated from the aggregated laser measurements.

6.1. Local Height Difference Maps

Since caves and other overhanging structures are the exception on most planetary surfaces, the 2.5D height map generated in Section 5.5 suffices for autonomous navigation planning.

The 2.5D height map H is transformed into a multi-scale height difference map. For each cell (x, y) in the horizontal plane, we calculate local height differences D_l at multiple scales l. We compute $D_l\ (x, y)$ as the maximum difference to the center cell (x, y) in a local l-window:

$$D_l(x,y) := \max_{\substack{|u-x|<l; u\neq x \\ |v-y|<l; v\neq y}} |H(x,y) - H(u,v)|\,. \qquad (1)$$

$H(u, v)$ values of NaN are ignored. In the cases where the center cell $H\ (x, y)$ itself is not defined, or there are no other defined l-neighbors, we assign $D_l\ (x, y) := \text{NaN}$. Small, but sharp obstacles

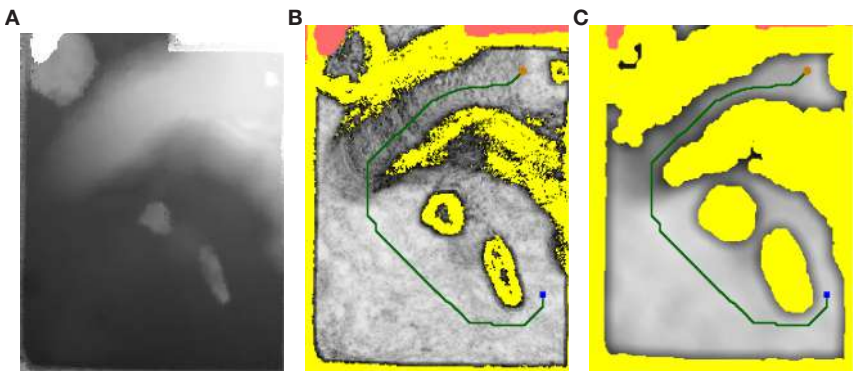


FIGURE 6 | Navigation planning. **(A)** 2.5D height map generated by projecting the 3D map. **(B)** Calculated traversability costs for each cell. **(C)** Inflated costs used for A* path planning. The orange dot represents the current robot position, the blue square represent the target position. Yellow regions represent absolute obstacles, red regions indicate missing measurements.

show up on the D_l maps with lower l scales. Larger inclines, which might be better to avoid, can be seen on the maps with a higher l value.

6.2. Path Planning

During the SpaceBot Camp, we used the standard ROS `navfn`[3] planner. Afterward, we replaced it with a custom A* planner to consider gradual costs fully, which the ROS planner was not designed to do. We transform the height difference map into a cost map that can be used for path planning.

A combined difference map, $\widetilde{D}$ is generated by linear combination of different D_l maps to comprise information about smaller obstacles and larger inclines. The summands from the D_3 and D_6 maps are constrained to a response of 1/2 to prevent the creation of absolute obstacles from a single scale alone. The smallest scale D_1 is allowed to create absolute obstacles, since sharp obstacles pose great danger to the robot:

$$\widetilde{D}(x,y) := \sum_{l\in\{1,3,6\}} \begin{cases} \lambda_l D_l & \text{if } l = 1 \\ min\{0.5;\, \lambda_l D_l\} & \text{otherwise.} \end{cases} \tag{2}$$

The λ_1, λ_3, and λ_6 parameter values for drivability computation were empirically determined as 2.2, 3.6, and 2.5, respectively.

6.2.1. Global Path Planning

For global path planning, we implemented an A* graph search on the 2D grid map. The Euclidean distance (multiplied with the minimum cost in the grid map) is used as the heuristic function for A*. This planning does not account for the robot foot print and considers the robot as just a point in the 2D grid. To ensure the generation of a safe path, we inflate obstacles in the cost map to account for the risk closer to obstacles. The inflation is done in two steps. The cells within the distance of robot radius from absolute obstacles are elevated to absolute obstacle cost, yielding cost map $\bar{D}$. Then for all other cells, we calculate local averages to produce costs D_D that increase gradually close to obstacles:

$$P(x,y) := \{(u,v) : (x-u)^2 + (y-v)^2 < r^2\}, \tag{3}$$

$$D_D(x,y) := \begin{cases} 1 & \text{if } \bar{D}(x,y) = 1 \\ \sum\limits_{(u,v)\in P(x,y)} \frac{\bar{D}(x,y)}{|P(x,y)|} & \text{otherwise.} \end{cases} \tag{4}$$

Figure 6 shows a planned path on the height map acquired during our mission at the SpaceBot Camp.

6.2.2. Local Trajectory Rollout

The found global path needs to be executed on a local scale. To this end, we use the standard ROS `dwa_local_planner`[4] package, which is based on the Dynamic Window Approach (Fox et al., 1997). The `dwa_local_planner` accounts for the robot foot print, so cost inflation is not needed.

In order to prevent oscillations due to imperfect execution of the planned trajectories, we made some modifications to the planner. The `dwa_local_planner` plans trajectories to reach the given goal pose (x, y, θ) first in 2D (x, y) and then rotates in-place to reach θ (this is called "latching" behavior). Separate cartesian and angular tolerances determine when the planner starts turning and when it reports navigation success. We modified the planner to keep the current "latching" state even when a new global plan is received (every 4 s), as long as the goal pose does not change significantly. We also wrote a simple custom recovery behavior that first warns the operator crew that the robot is stuck and then executes a fixed driving primitive after a timeout.

6.3. Omnidirectional Driving

The wheel positions $\mathbf{r}^{(i)}$ relative to the trunk determine not only the footprint of the robot but also the orientation and height of the robot trunk. During autonomous operation, the wheel positions are kept in a configuration with a base height.

Either autonomous navigation or manual operator input generates a velocity command $\mathbf{w} = (v_x, v_y, \omega)$ with horizontal linear velocity $\mathbf{v}$ and rotational velocity ω around the vertical axis. The velocity command is first transformed into the local velocity at each wheel i:

$$\begin{pmatrix} v_x^{(i)} \\ v_y^{(i)} \\ v_z^{(i)} \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \omega \end{pmatrix} \times \mathbf{r}^{(i)} + \dot{\mathbf{r}}^{(i)}, \tag{5}$$

where $\mathbf{r}^{(i)}$ is the current position of wheel i relative to the base. The kinematic velocity component $\dot{\mathbf{r}}^{(i)}$ allows simultaneous leg movement while driving. The wheels rotates to yaw angle $\alpha^{(i)} = \text{atan2}(v_y^{(i)}, v_x^{(i)})$ first and then moves with the velocity $||(v_y^{(i)}, v_x^{(i)})^T||$. While driving, the robot continuously adjusts the orientation of the ankle, using IMU information to keep the axis vertical and thus retains omnidirectional driving capability.

6.4. Base Orientation Control

To prevent the robot from pitching over on the high-incline areas in the arena, we implemented a pitch control mechanism. The pitch angle of the robot is continuously measured using the IMU. We then use a simple proportional controller to compensate for the disturbance. With the commanded angle w, disturbance z, controller gain K_p, plant gain K_s, and plant disturbance gain K_{sz}, the steady-state error e_b of the linearized proportional plant evolves with

$$e_b = \frac{1}{1 + K_s \cdot K_p} \cdot w - \frac{K_{sz}}{1 + K_s \cdot K_p} \cdot z. \tag{6}$$

Since the incline is directly measured, $K_s = 1$ and $K_{sz} = 1$. We found $K_p = 0.8$ to sufficiently stabilize for inclines present at the SpaceBot Camp. When driving up the ramp with $z \approx 15°$, and setpoint $w = 0°$ the resulting error (robot pitch) is $e_b \approx 8.3°$.

We found that this compensation enables Momaro to even overcome inclines greater than 20° without pitching over. Due to the lack of integral control, the robot is even ($eb = 0°$) only on a completely flat surface. Since this poses no balance problem, there is no need for integral control.

[3] http://wiki.ros.org/navfn
[4] http://wiki.ros.org/dwa_local_planner

7. OBJECT PERCEPTION

For approaching objects and adapting motion primitives to detected objects, RGB images, and RGB-D point clouds from the wide-angle camera and ASUS Xtion camera, mounted on the sensor head are used. We differentiate between object detection (i.e., determining an approximate object position) and object registration (i.e., determining the object pose accurately).

The objects provided by DLR are color coded. We classify each pixel by using a precomputed lookup table in YUV space. The lookup table is generated from a collection of ellipses for each color class in UV space (see **Figure 7**), and lower/upper limits in brightness (Y). Thus, we assume that the object color measurements are governed by a Gaussian mixture model in the UV plane. In practice, a single ellipse sufficed for each of the SpaceBot Camp objects.

When approaching an object, object detection is initially performed with the downward-facing wide-angle camera mounted on the sensor head (see **Figure 7**). Using the connected component algorithm, we obtain object candidate clusters of same-colored pixels. An approximate pinhole camera model calculates the view ray for each cluster. Finally, the object position is approximated by the intersection of the view ray with the local ground plane. The calculated object position is precise enough to allow approaching the object until it is in the range of other sensors.

As soon as the object is in range of the head-mounted ASUS Xtion camera, the connected component algorithm can also take Cartesian distance into account. We use the PCL implementation of the connected component algorithm for organized point clouds. Since the depth measurements allow us to directly compute the cluster centroid position, and the camera is easier to calibrate, we can approach objects much more precisely using the RGB-D camera.

When the object is close enough, we use registration of a CAD model to obtain a precise object pose (see **Figure 7**). Since color segmentation often misses important points of the objects, we perform a depth-based plane segmentation using RANSAC and Euclidean clustering as detailed by Holz et al. (2011) to obtain object clusters. The clusters are then registered using Generalized ICP (Segal et al., 2009).

ICP approaches often have problems with partially observed box shapes. For example, only the front and the top face of a box may be visible if the box is partially outside of the camera view

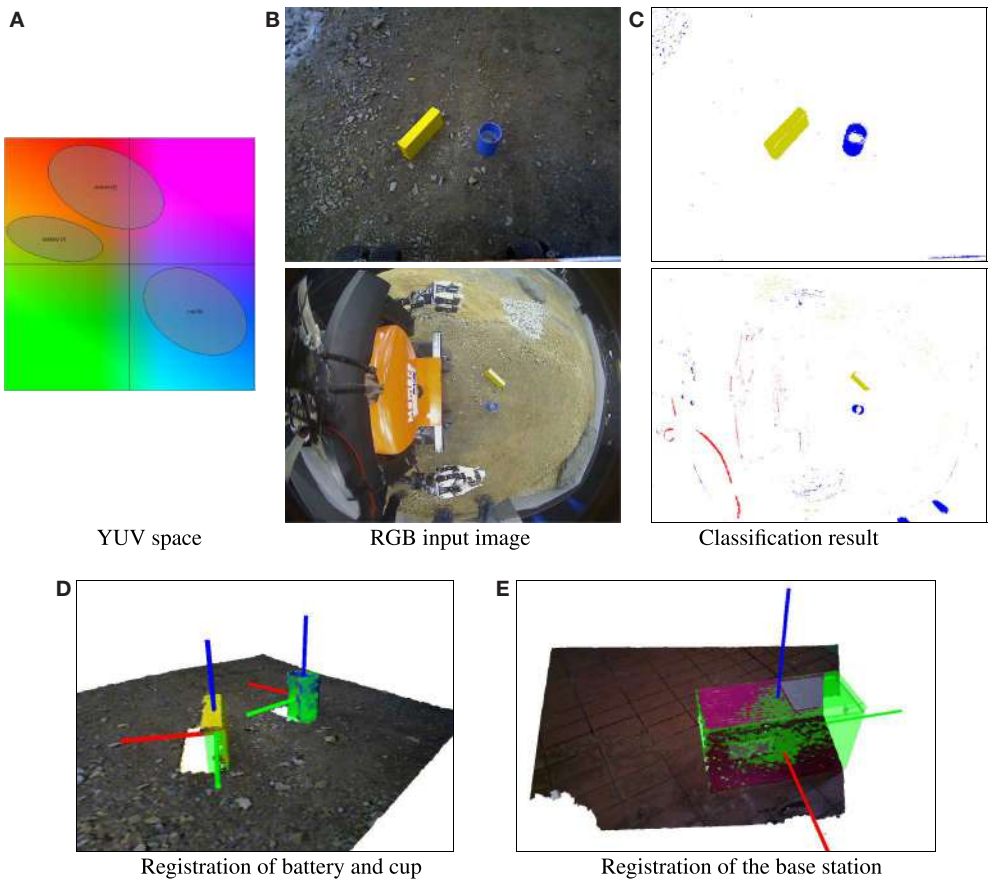


FIGURE 7 | Object perception. **(A)** Classification ellipses in UV space. **(B)** RGB input image (first row: Xtion camera, second row: RGB wide-angle camera). **(C)** Pixel classes (white = unknown). **(D)** RGB-D point cloud showing the cup and battery objects on SpaceBot Camp terrain. The registered models are shown in green. **(E)** Registration of the base station. Although neither the left nor the right face is visible, the pose ambiguity is resolved correctly.

frustum. To resolve the resulting ambiguity, we initialize the ICP pose using PCA under the assumption that the visible border of the object which is close to the image border is not an actual object border but is caused by the camera view frustum. In practice, this problem particularly occurs with the large base station object (see **Figure 7**).

The ICP pose is then normalized respecting the symmetry axes/planes of the individual object class. For example, the cup is symmetrical around the Z axis, so the X axis is rotated such that it points in the robot's forward direction (see **Figure 7**).

8. MANIPULATION

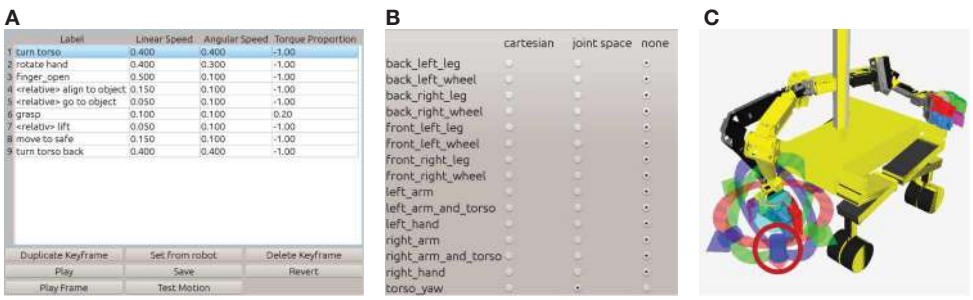
Since Momaro is a unique prototype, the time used for development and testing had to be balanced between individual submodules. To reduce the need for access to the real robot, we made extensive use of simulation tools. For manipulation tasks, we developed a Motion Keyframe Editor GUI to design motion primitives offline. Finished motions are then tested and finalized on the real robot with the original objects to be manipulated in the field. We show the Motion Keyframe Editor GUI in **Figure 8**. With its help, we designed dedicated motions for all specific tasks in the SpaceBot Camp. We give an overview of our custom motions and their purpose in **Figure 8**.

8.1. Kinematic Control

We use straightforward kinematic control for Momaro (see **Figure 9**). Both arms and the torso yaw joints are considered independently.

A goal configuration is specified by telemanipulation (see Section 10) or predefined keyframe sequences either in Cartesian or in joint space. To interpolate between current and goal configuration, the Reflexxes Motion Library (Kröger, 2011) is used. Goals for different limbs can be defined concurrently; the interpolation is configured in a way that goals for all limbs are reached simultaneously. Cartesian poses are converted to joint-space configurations, using inverse kinematics after interpolation. We use the selectively damped least squares approach (SDLS) described by Buss and Kim (2005) to calculate the inverse kinematics of the arms. Before the configurations are sent to the hardware controllers for execution, they are checked for self-collisions using the MoveIt! Library.[5] Detecting a collision will abort motion

[5]http://moveit.ros.org



Motion schedule | Interpolation selection | Motion simulation

D Designed motions

Motion	Purpose	Reference type
scoop	fill scoop tool with soil sample	absolute
fill cup	pour soil sample into the cup and discard scoop tool	relative (cup)
grasp cup right hand	grasp cup with right hand from above	relative (cup)
grasp battery left hand	grasp battery with left hand from above	relative (battery)
place cup	place cup on base station	relative (base station)
place battery	put battery into base station	relative (base station)
toggle switch	toggle switch on side of base station	relative (base station)
grasp abort {left,right} arm	motion to initial position when grasp is aborted	absolute
reset {left,right} arm	move all individual joints of the arm in defined position (resolves singularity-induced c-space ambiguities)	absolute
reset torso	move torso into initial position	absolute
cheer	cheer to the audience	absolute

FIGURE 8 | Keyframe editor GUI. **(A)** Motions are designed step by step and can be absolute or relative to perceived objects. **(B)** The user can select which joint groups are included in the currently edited keyframe and if interpolation between keyframes is Cartesian or joint space. **(C)** The real position of the robot is indicated in black. The currently edited keyframe target is shown in yellow. Interactive markers can be used to modify the keyframe pose in 6D (here only for the right hand). A model of the cup (blue, circled red) is placed in front of the robot to assist designing relative motions. **(D)** Overview of the custom motions designed for the SpaceBot Camp 2015.

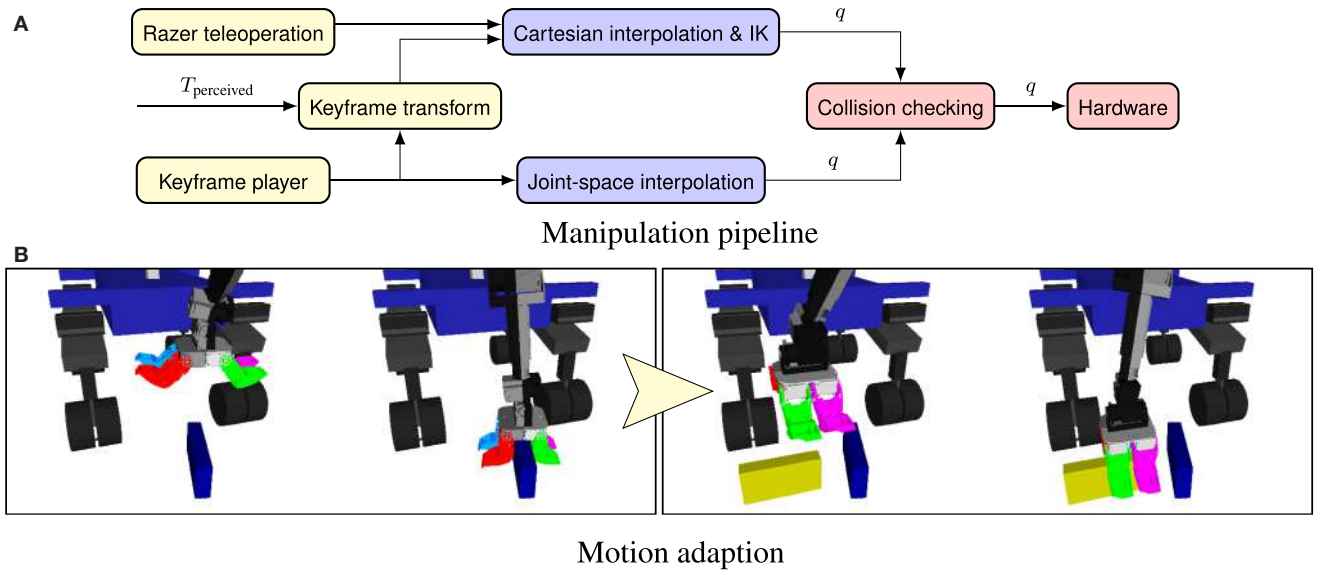


FIGURE 9 | Object manipulation. **(A)** Kinematic control architecture for Momaro. Joint configurations can be generated using magnetic trackers or the keyframe player. Cartesian poses in keyframes can be adapted to a measured pose p. The interpolated configurations $T_{\text{perceived}}$ are checked for collisions before they are sent to the hardware. **(B)** Grasping objects dynamically using motion adaption. Left: the blue reference object is grasped as the primitive was designed in the Keyframe editor. Right: the primitive is automatically adapted to the perceived pose of the yellow object.

execution. For safety reasons, different methods of manipulation control (i.e., telemanipulation and the keyframe player) will preempt each other.

8.2. Motion Adaption

Since it is often impossible or too slow to precisely approach an object in all 6 dimensions, we relax the assumption of absolute positioning. Motions can be designed around a reference object T_{refernce}. When the motion is executed, the predefined endeffector pose $T_{\text{endeffector}}$ is transformed in selected keyframes i to match the perceived object $T_{\text{perceived}}$:

$$T_{\text{relative}} = T^{(i)}_{\text{perceived}} (T_{\text{reference}})^{-1} T^{(i)}_{\text{endeffector}} \tag{7}$$

Figure 9 shows how a motion, designed relative to a reference object, is adapted to a perceived object pose to account for imprecise approach of the object.

As described in Section 7, the perceived objects are represented in a canonical form, removing all ambiguities resulting from symmetries in the original objects. For example, the rotation-symmetric cup is always grasped using the same yaw angle. After adaption, the Cartesian keyframes are interpolated as discussed earlier.

9. COMMUNICATION

Communication between the ground station and a planetary rover is typically very limited – in particular, it has high latency due to the speed of light and the large distances involved. The SpaceBot Camp addressed this limitation by imposing several constraints on the network link:

- Packets were delayed by 2 s in each direction, as expected to occur on a lunar mission,
- the uplink from the ground station to the robot could only be opened for 5 min at a time, and
- the 60-min schedule included two 4-min windows where uplink communication was not possible (e.g., due to planetary occlusions).

Furthermore, our system uses a wireless data link inside the arena, which introduces packet loss.

The main idea of our communication system is to minimize latency by exploiting the different characteristics of the local wireless link inside the arena and the simulated inter-planetary network.

9.1. Communication Architecture

Our communication architecture is shown in **Figure 10**. The DLR-provided network emulator is the central element limiting all communication between robot and operator crew. To be able to exploit the different link characteristics, we place an additional field computer between the network emulator and the robot. Thus, it is connected to the network emulator via a reliable Ethernet connection and communicates directly with the robot over WiFi. As the WiFi link is unreliable, but has low latency, while the network emulator link is reliable, but has high latency, this places the field computer in an ideal position to exploit both link characteristics.

As the network emulator allows communication only through a single port per direction, we use the Linux `tun` interface to create a network tunnel over two ports. For UDP tunneling, we adapted code from the `quicktun` project.[6] The tunnel wraps all packets in UDP packets, transmitted over the two designated ports.

[6]http://wiki.ucis.nl/QuickTun

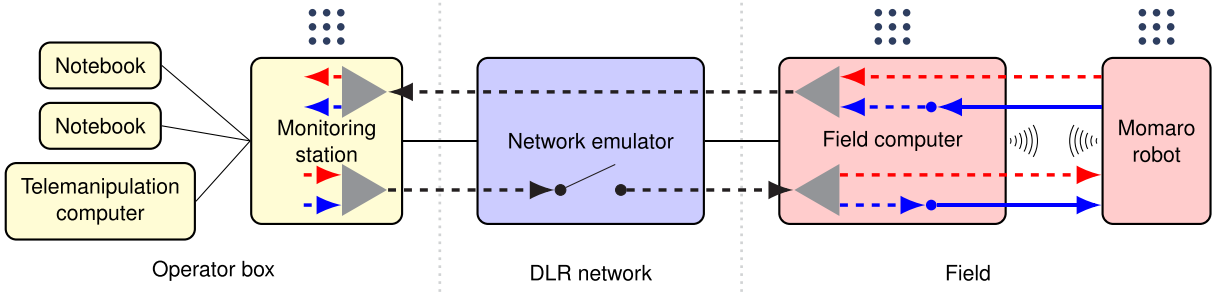


FIGURE 10 | Communication architecture. Components in the vicinity of the operators are shown in yellow, DLR-provided components in blue, and components in the "arena"-network in red. Solid black lines represent physical network connections. Thick lines show the different channels, which stream data over the network (dotted: UDP, solid: TCP). The ROS logo indicates a ROS master. UDP tunnel endpoints are designated by triangles. Streaming links (Section 9.1.2) are colored red, message links (Section 9.1.3) are shown in blue.

This allows us to use multiple communication channels without interference.

Separate ROS masters run on the robot, the field computer, and the ground station. Multiple operator computers can be connected to the ROS master running on the ground station to provide additional views and means for intervention.

9.1.1. Communication Software Module

Since our participation in the DLR SpaceBot Cup 2013 (Stückler et al., 2015), our group develops a robust software module (`nimbro_network`) for communication between multiple ROS masters over unreliable and high-latency networks. We used it with very good results in the DLR SpaceBot Cup 2013 and in the DARPA Robotics Challenge (Schwarz et al., 2016). Since the DRC, the module is now freely available[7] under BSD-3 license. In contrast to custom-engineered network stacks for a particular purpose, it allows the generic transport of ROS topics and services. The module is ideally suited for situations where the connection drops and recovers unexpectedly, since it avoids any configuration/discovery handshake.

Several specific transports and compression methods exist, such as a ROS log transport, `tf` snapshotting, or H264 video stream compression.

For large messages, a transparent BZip2 compression can be enabled. Automatic rate limiting with configurable upper and lower bounds ensures that bandwidth limits are met.

`nimbro_network` also allows forward error correction (FEC), i.e., augmenting the sent packets with additional packets allowing content recovery from arbitrary subsets of sufficient size of transmitted packets. Depending on the message size, a Reed–Solomon codec (Lacan et al., 2009) or a LDPC-Staircase codec (Roca et al., 2008) is chosen.

Note that in principle ROS offers built-in network transparency. Since this functionality heavily relies on the TCP protocol for topic discovery and subscription, even when the "UDPROS" transport is chosen, this is unsuitable for unreliable and high-latency networks.

[7]http://github.com/AIS-Bonn/nimbro_network

9.1.2. Streaming Data

Most high-bandwidth data from the robot are of *streaming* type. The key feature here is that lost messages do not lead to system failures, since new data will be immediately available, replacing the lost messages. In this particular application, it even would not make sense to repeat lost messages because of the high latencies involved. This includes

- video streams from the onboard cameras,
- transform information (TF),
- servo diagnostic information (e.g., temperatures),
- object detections, and
- other visualizations.

In the uplink direction, i.e., commands from the operator crew to the robot, this includes, e.g., direct joystick commands.

Consequently, we use the `nimbro_network` UDP transport for streaming data (red in **Figure 10**). The transport link between robot and field computer uses the FEC capability of `nimbro_network` with 25% additional recovery packets to compensate WiFi packet loss without introducing new latency.

9.1.3. Message Data

Other data are of the *message* type, including

- Laser pointclouds,
- SLAM maps,
- SLAM transforms,
- ROS action status messages, and
- ROS service calls.

Here, a message loss might be costly (e.g., SLAM maps are only generated on every scanner rotation) or might even lead to system failure (e.g., loss of a ROS action state transition). Therefore, the TCP transport is used for this kind of messages over the WiFi link to eliminate the possibility of packet loss. The link over the network emulator is still implemented with the UDP protocol, since there is no packet loss here and the high latencies prohibit TCP handshakes. The message links are colored blue in **Figure 10**.

10. MISSION CONTROL INTERFACES

For the operator crew, situational awareness is most important. Our system shows camera images, 3D visualization, and diagnosis information on a central ground station with four monitors (see **Figure 11**).

In order to cope with the degraded communication link, the system needs to be as autonomous as possible, while retaining the ability to interrupt, reconfigure, or replace autonomous behavior by manual intervention. To this end, our system provides three levels of control to the operator crew. On the highest level, entire missions can be specified and executed. The intermediate level allows configuration and triggering of individual autonomous behaviors, such as grasping an object. On the lowest level, the operators can directly control the base velocity using a joystick or move individual DOF of the robot.

The last aspect of our control paradigm is remote debugging. Operators need to be able to directly introspect, debug, and

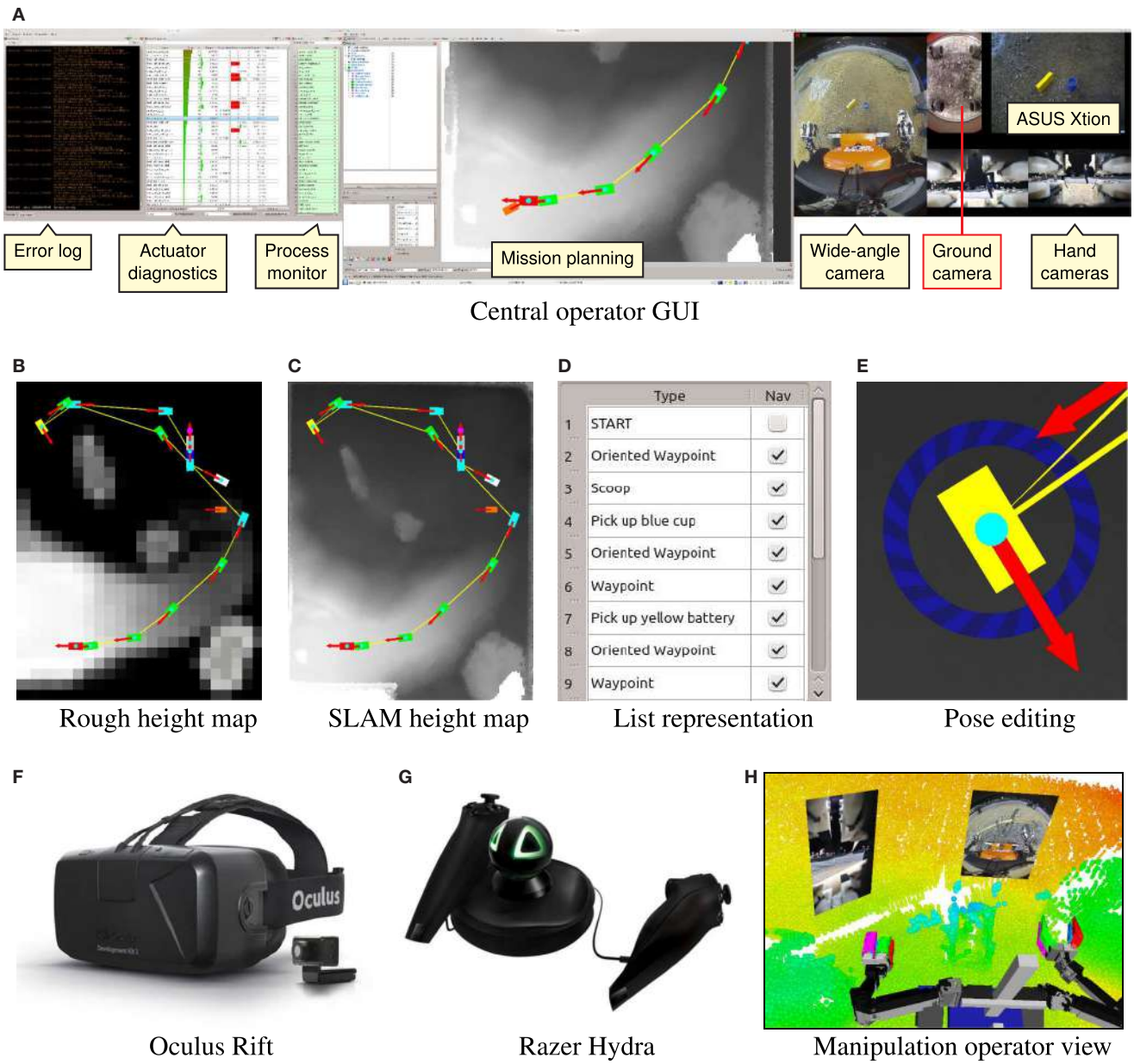


FIGURE 11 | Operator interfaces. **(A)** Overview of the GUI shown on the three lower screens of the main ground station. The left, center, and right screens are dedicated to system monitoring and diagnosis, mission planning, and camera images, respectively. **(B)** Mission plan on rough height map provided by DLR. **(C)** Mission plan on detailed height map generated from the SLAM map. **(D)** List representation of the first 8 poses. The "Nav" column can be used to disable navigation (e.g., start grasping an object immediately). **(E)** Pose editing using interactive marker controls. The position can be modified by dragging the rectangle. The pose is rotated by dragging on the blue circle. Teleoperation interfaces: operator uses **(F)**, oculus rift DK2 HMD, and **(G)** Razer Hydra 6 DOF controllers for immersive teleoperation. **(H)** 3rd person view of the scene rendered in the Oculus HMD during debris cleaning (see **Figure 13**).

manipulate the software on the robot in order to prevent relatively simple problems from escalating to mission failures.

We describe the developed operator interfaces in the following.

10.1. Mission Planning and Execution

Our mission control layer is able to execute all required tasks in the SpaceBot Camp specification. The mission can be specified fully in advance on a rough height map and can later be interactively refined as the mission progresses, and a more detailed map of the environment is created.

A specified mission consists of a list of 2D poses in the height map frame. Attached to each pose is an optional action, which is executed when the robot reaches the pose. Poses without an associated action are just used as navigation targets. Supported actions include:

- Taking a soil sample using the scoop in one hand,
- approaching and grasping the battery,
- approaching the cup, filling it with the soil sample and grasping it, and
- approaching the base station and performing all station manipulation tasks.

The mission can be configured and monitored using our Mission GUI (see **Figure 11**). During the mission, execution can be stopped at any time, mission updates can be performed, and the execution resumed. Missions can also be spliced in the sense that the currently performed action is carried out and then execution switches to a new mission.

In the case of a failure of the mission control level, or if the operator judges that the system will not be able to carry out the mission autonomously, the execution can be interrupted and the task in question can be carried out using the lower control levels. Afterward, the mission can be resumed starting after the completed task.

10.2. Semiautonomous Control

The semiautonomous control level gives direct access to all individual, less autonomous behaviors. This includes

- approaching an object,
- grasping an object,
- performing single manipulation tasks, and
- navigating to a goal pose.

10.3. Low-Level Control

If all autonomous behaviors fail, the operators can also directly teleoperate the robot. For manipulation, our operators can choose between on-screen teleoperation using 6D interactive markers in either Cartesian or joint space or immersive 3D telemanipulation (see **Figure 11**) using an Oculus Rift HMD and 6D magnetic trackers [see Rodehutskors et al. (2015) for details].

For navigation, the operator can use a joystick to directly control the base velocity. Teleoperation speed is of course limited by the high feedback latency, so that this method is only used if the navigation planners get stuck. Finally, several macros can be used to influence the robot posture or recover from servo failures such as overheating.

10.4. Remote Introspection and Debugging

To be able to react to software problems or mechanical failures, operators first need to be aware of the problem. Our system addresses this concern by

- providing direct access to the remote ROS log,
- showing the state of all ROS processes, and
- transmitting and displaying 3D visualization data from the autonomous behaviors.

Once aware of the problem, the operators can interact with the system through ROS service calls over our `nimbro_network` solution, parameter changes, or ROS node restarts through `rosmon`. In extreme cases, it is even possible to push small Git code patches over the network and trigger re-compilation on the robot. If everything else fails, the operators can access a remote command shell on the robot using the `mosh` shell (Winstein and Balakrishnan, 2012), which is specifically optimized for high-latency, low-bandwidth situations. The shell gives full access to the underlying Linux operating system.

11. EVALUATION

Momaro has been evaluated in several simulations and lab experiments as well as in the DARPA Robotics Challenge (DRC) Finals in June 2015, during the DLR SpaceBot Cup Qualification in September 2015, and the DLR SpaceBot Camp in November 2015 (Kaupisch et al., 2015). For details on our performance at the DRC Finals, we refer to Schwarz et al. (2016). Here, we will focus on our performance at the SpaceBot Qualification and Camp.

In preparation for the DLR SpaceBot finals, the SpaceBot Cup Qualification tested basic capabilities of the robotic system. To qualify, participants had to solve three tasks which involved exploration and mapping of an arena and manipulation of the cup and the battery, but no assembly. In contrast to the finals, the communication uplink time was unlimited, which lowered the required autonomy level. Using our intuitive telemanipulation approaches, our team was the only team to successfully qualify in the first attempt. Further information about our performance is available on our website.[8] Since only two other teams managed to qualify using their second attempt, the planned SpaceBot Cup competition was changed to an open demonstration, called the SpaceBot Camp.

The SpaceBot Camp required participants to solve mapping, locomotion, and manipulation tasks in rough terrain. As detailed in Section 1, the battery and cup (with soil sample) had to be found and transported to the base station object, where an assembly task was to be performed. The participants were provided with a coarse map of the environment that had to be refined by the robot's mapping system. As detailed in Section 9, the communication link to the operator crew was severely constrained both in latency (2 s per direction) and in availability.

11.1. Locomotion

While Momaro was mainly evaluated on asphalt at the DRC (Schwarz et al., 2016), the SpaceBot Camp arena included various

[8] http://www.ais.uni-bonn.de/nimbro/Explorer

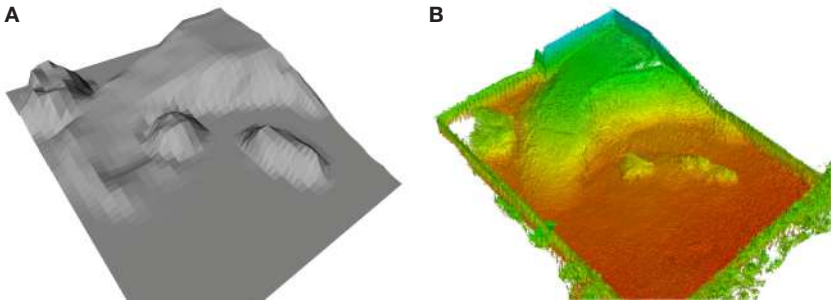


FIGURE 12 | Map refinement. **(A)** Rough map of the SpaceBot Camp 2015 arena. **(B)** The resulting global map from data acquired during the competition.

types of soil and stones (see **Figure 14**). We did not experience any problems on the main traversable area, which was covered with flattened soil mixed with stones. During our run, we avoided the gravel and sand areas. We also traversed the soil sample area (loose granulate), and parts of the slopes covered with gravel, as long as the inclination permitted. Testing after our run confirmed that Momaro's wheels were not suited for the fine sand areas on the edge of the ramp, causing the robot to get stuck.

While preparing for the SpaceBot Camp, we learned that our pitch stabilization control method works reliably, even under extreme conditions. Being able to reliably overcome ramps with inclines greater than 20°, we were confident that locomotion would not pose a problem during the competition. Unfortunately, we only employ stabilization in pitch direction. Turning around the yaw axis on a pitched slope can result in a dangerous roll angle. We dealt with this issue during our final run by placing enough waypoints on the primary slope in the course to ensure proper orientation (see **Figure 11**).

11.2. Mapping and Self-Localization

Our mapping system continuously built an allocentric map of the environment during navigation, guided by waypoints specified on the coarse height map. The coarse map and the allocentric map, generated from our mapping system, are shown in **Figure 12**. While showing the same structure as the coarse map, the resulting allocentric map is accurate and precisely models the environment. During a mission, the map is used for localization and to assess traversability for navigation. The estimated localization poses are shown in **Figure 14**.

Despite the challenging planetary-like environment, causing slip in odometry and vibrations of robot and sensor, our mapping system showed very robust and reliable performance. There was only one situation during the run where the operators had to intervene: due to traversing the abandoned scoop tool – used to take the soil sample – the robot was exposed to a fast and large motion, resulting in a distorted 3D scan. This distorted 3D scan caused spurious measurements in the map. The operators decided to clear the SLAM map using a remote service call to prevent localization failures. The map was rebuilt from this point on and successfully used for the rest of the mission.

11.3. Object Manipulation

While preparing our run, we found the battery slot in the base station to have a significant resistance due to a build-in clamping mechanism. Due to our flexible motion design workflow, we were able to alter the motion, so that Momaro would execute small up- and downward motions while pushing to find the best angle to overcome the resistance.

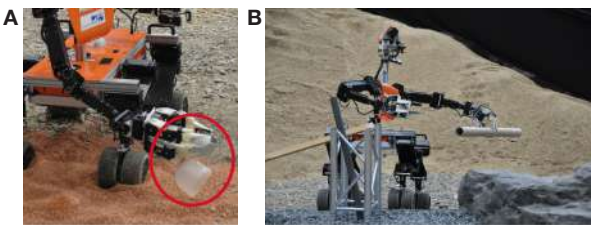


FIGURE 13 | Details of our run at SpaceBot camp. **(A)** Due to a failed finger actuator, Momaro failed to take the soil sample in the first attempt. **(B)** After finishing all tasks of the SpaceBot Camp, we showed Momaro's universal capabilities by removing debris from the terrain under teleoperation.

The insertion of the battery requires high precision. To account for inaccuracies in both battery and station pose, we temporarily place the battery on top of the station. After grasping the battery again, we can be sure that any offset in height is compensated.

Furthermore, we found it to be error prone to grasp the battery at the very end, which is necessary to entirely push it inside the slot. Instead, we push the battery in as far as possible until the hand touches the base station. After releasing the battery, we position the closed hand behind it and push it completely inside with part of the wrist and proximal finger segments.

Overall, our straightforward keyframe adaption approach proved itself to be very useful. Compared to motion–planning techniques, it lacks collision avoidance and full trajectory optimization, but it is sufficient for the variety of performed tasks.

11.4. Full System Performance at DLR SpaceBot Camp 2015

After a restart caused by a failed actuator (described below), Momaro solved all tasks of the SpaceBot Camp with supervised autonomy. Our team was the only one to demonstrate all tasks including the optional soil sample extraction. **Figure 14** gives an overview of the sequence of performed tasks. A video of our performance can be found online.[9] While overall the mission was

[9]https://youtu.be/q_p5ZO-BKWM

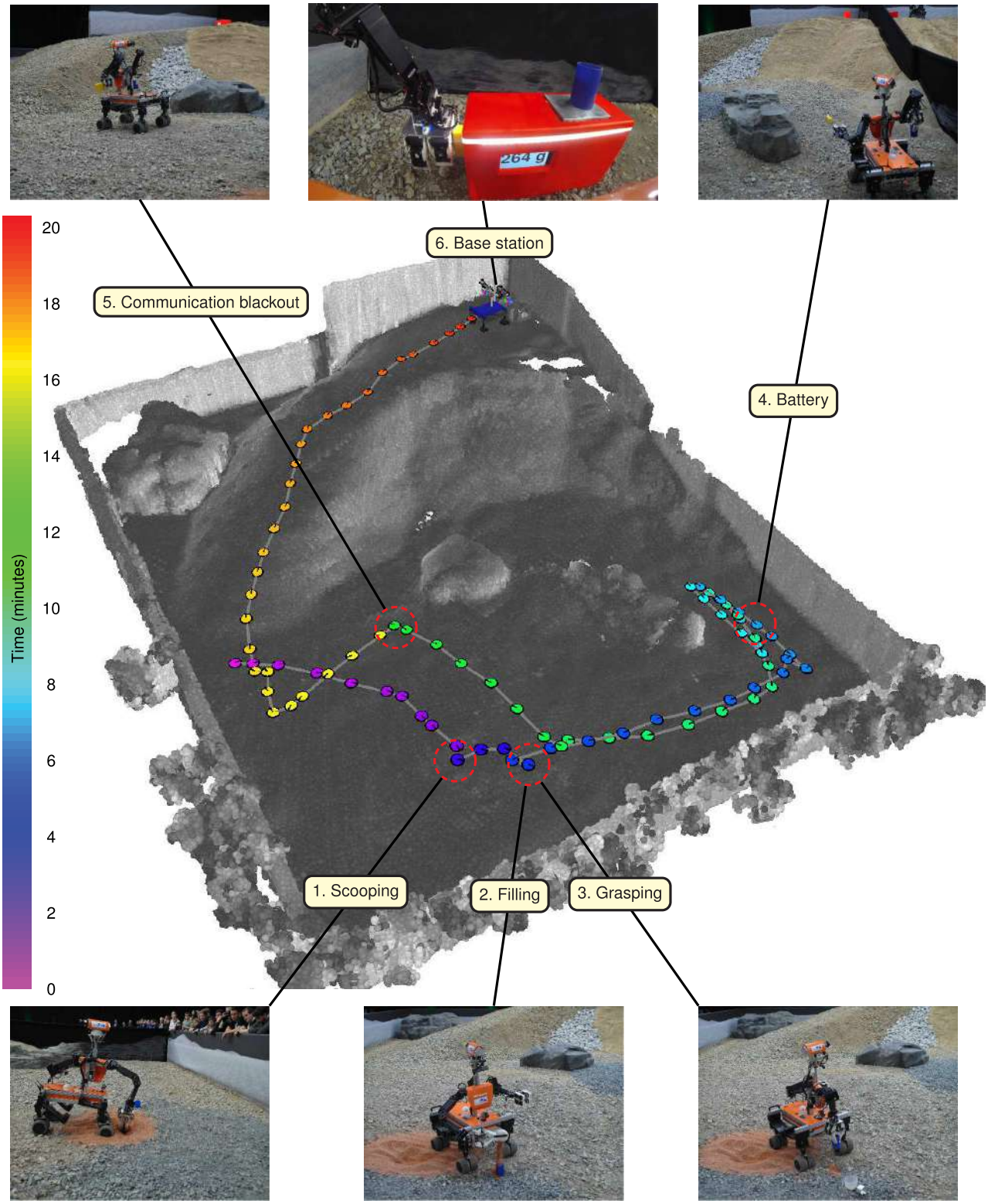


FIGURE 14 | Overview of the executed mission at SpaceBot Camp. The mission starts by scooping the soil sample, filling it into the cup and grasping the cup, then locating and grasping the battery pack. After waiting until the end of scheduled communication blackout, the mission is concluded by Base station assembly.

TABLE 1 | Timings of our run at the DLR SpaceBot Camp 2015.

Task	Start time (mm:ss)	End time (mm:ss)	Duration (mm:ss)
Soil sample collection	1:05	1:40	0:35
Fill and grasp cup	2:15	3:05	0:50
Grasp battery	7:00	7:40	0:40
Base station assembly	18:25	20:25	2:00
Total (including locomotion)	0:00	20:25	20:25

successful, we experienced a number of problems which will be discussed in detail.

In our run, Momaro failed to take the soil sample in the first attempt. During the vigorous scooping motion, the scoop turned inside the hand (cf. **Figures 2** and **13**). We found the problem to be a malfunctioning finger actuator in the hand holding the scoop. Since we were confident that Momaro would be able to solve all tasks even in the remaining 50:20 min, we restarted the whole run after performing a software reset on the affected finger and letting it cool down.

In the second attempt, scooping succeeded and Momaro was able to complete all remaining tasks as well. See **Figure 14** for detailed images of the subtasks. Timings of the run are listed in **Table 1**.

Although Momaro was able to complete all tasks, this was not possible fully autonomously. While approaching the battery, a timeout aborted the process. This built-in safety feature made operator interaction necessary to resume the approach. Without intervention, Momaro would have executed the remainder of the mission without the battery object.

As Momaro reached the main slope of the course, we also approached the time of the first communication blackout, because we lost time in the beginning due to the restart. The operator crew decided to stop Momaro at this point, as we knew that going up would be risky and intervention would have been impossible during the blackout. After the blackout, autonomous operation resumed and Momaro successfully went up the ramp to perform the assembly tasks at the base station (**Figure 14**). Although the operators paused autonomous navigation at one point on the slope to assess the situation, no intervention was necessary and navigation resumed immediately.

After finishing the course in 20:25 min, we used the remaining time to show some of Momaro's advanced manipulation capabilities by removing debris from the terrain with Momaro and our intuitive teleoperation interface (**Figure 13**).

12. LESSONS LEARNED

Our successful participation in the SpaceBot Camp was an extremely valuable experience, identifying strong and weak points of our system in a competitive benchmark within the German robotics community. Lessons learned include

- *Mechanical Design*. While the humanoid torso raised the center of gravity and thus caused stability concerns on high terrain inclines, it allowed us to perform bimanual manipulation. Being able to carry both objects in the hands allowed us to omit storing the objects in separate holders on the robot, saving time. Furthermore, our end effector design allowed us to use a scoop to take the soil sample. The soil extraction task was not attempted by any other team. In future work, we will further improve the robot balance control to operate in more difficult rough terrain. For instance, adaptive roll stabilization could advance Momaro's locomotion capabilities.
- *Actuator Monitoring*. Our system provides extensive diagnostic actuator feedback such as temperature and current consumption. Still, this was not enough to prevent the failure of the finger actuator during our run. Actuator monitoring and damage prevention should have a high priority during development.
- *Software Design: Autonomy Follows Teleoperation*. Our unique history of competing previously in the DARPA Robotics Challenge, a competition heavily focused on intuitive teleoperation, set us apart from other teams. In particular, resulting from the DRC competition, we had extensive intuitive teleoperation abilities before starting work on the higher autonomy required by the SpaceBot Camp. We suspect that most other teams followed the opposite approach, augmenting the autonomy later on with teleoperation facilities, which can be difficult if the system was not designed for teleoperation from the start. Treating the autonomy as an additional layer on a teleoperable system ensures that the operator crew has full control of the system at all time. Furthermore, this also accelerates development, since missing autonomous functionalities can be substituted by intuitive teleoperation. We demonstrated the ability of our telemanipulation solution after our run by removing debris and thus clearing the robot's path.
- *Intelligent Progress Monitoring*. Our mission control layer included some very basic error handling, e.g., fixed timeouts on certain actions. Unfortunately, one of these timeouts resulted in an early abort of the battery approach in our run, which had to be corrected by operator action. A more intelligent system, tracking the progress of the current task, would have noticed that the approach was still progressing and would have continued the approach. In future, we will investigate such resilient progress monitoring methods in more detail.

13. CONCLUSION

In this article, we presented the mobile manipulation robot Momaro and its ground station. We provided details on the software and hardware architecture of the integrated robot system and motivate design choices. The feasibility, flexibility, usefulness, and robustness of our design were evaluated with great success at the DLR SpaceBot Camp 2015.

Novelties include an autonomous hybrid mobile base combining wheeled locomotion with active stabilization in combination with fully autonomous object perception and manipulation in rough terrain. For situational awareness, Momaro is equipped with a multitude of sensors such as a continuously rotating 3D laser scanner, IMU, RGB-D camera, and a total of seven color cameras. Although our system was build with comprehensive autonomy in mind, all aspects from direct control to mission specification can be teleoperated through intuitive operator interfaces. Developed for the constraints posed by the SpaceBot Camp, our

system also copes well with degraded network communication between the robot and the monitoring station.

The robot localizes by fusing wheel odometry and IMU measurements with pose observations obtained in a SLAM approach using laser scanner data. Autonomous navigation in rough terrain is tackled by planning cost-optimal paths in a 2D map of the environment. High-level autonomous missions are specified as augmented waypoints on the 2.5D height map generated from SLAM data. For object manipulation, the robot detects objects with its RGB-D camera and executes grasps using parametrized motion primitives.

In the future, shared autonomy could be improved by automatic failure detection, such that the robot reports failures and recommends a suitable semiautonomous control mode for recovery. Currently, only vision-based manipulation is supported by the system. Additional touch and force-torque sensing could potentially lead to more robust manipulation capabilities.

AUTHOR CONTRIBUTIONS

MSchwarz was the team leader and main author. MB contributed in pose estimation and evaluation. DD developed the SLAM system. SS developed the motion adaption system. AP contributed the autonomous navigation planner. CL developed operator interfaces and designed motions. MSchreiber constructed the robot and provided mechanical design details. SB initiated and managed the project, mentored the team during the preparations and competition, and assisted in writing this article.

FUNDING

This work was supported by the European Union's Horizon 2020 Programme under Grant Agreement 644839 (CENTAURO) and by Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR) under Grant No. SORA1413.

REFERENCES

Adachi, H., Koyachi, N., Arai, T., Shimiza, A., and Nogami, Y. (1999). "Mechanism and control of a leg-wheel hybrid mobile robot," in *Proc. of the IEEE/RSJ Int. Conference on Intelligent Robots and Systems (IROS)*, Vol. 3 (Kyongju: IEEE), 1792–1797.

Borst, C., Wimbock, T., Schmidt, F., Fuchs, M., Brunner, B., Zacharias, F., et al. (2009). "Rollin' Justin – mobile platform with variable base," in *Proc. of the IEEE Int. Conference on Robotics and Automation (ICRA)* (Kobe: IEEE), 1597–1598.

Buss, S. R., and Kim, J.-S. (2005). Selectively damped least squares for inverse kinematics. *Graph. GPU Game Tools* 10, 37–49. doi:10.1080/2151237X.2005.10129202

Cheng, G., and Zelinsky, A. (2001). Supervised autonomy: a framework for human-robot systems development. *Auton. Robots* 10, 251–266. doi:10.1023/A:1011231725361

Cho, B.-K., Kim, J.-H., and Oh, J.-H. (2011). Online balance controllers for a hopping and running humanoid robot. *Adv. Robot.* 25, 1209–1225. doi:10.1163/016918611X574687

Droeschel, D., Stückler, J., and Behnke, S. (2014a). "Local multi-resolution representation for 6d motion estimation and mapping with a continuously rotating 3d laser scanner," in *Proc. of the IEEE Int. Conference on Robotics and Automation (ICRA)* (Hong Kong: IEEE), 5221–5226.

Droeschel, D., Stückler, J., and Behnke, S. (2014b). "Local multi-resolution surfel grids for mav motion estimation and 3d mapping," in *Proc. of the Int. Conference on Intelligent Autonomous Systems (IAS)*, Padova.

Endo, G., and Hirose, S. (2000). "Study on roller-walker (multi-mode steering control and self-contained locomotion)," in *Proc. of the IEEE Int. Conference on Robotics and Automation (ICRA)*, Vol. 3 (San Francisco: IEEE), 2808–2814.

Fox, D., Burgard, W., and Thrun, S. (1997). The dynamic window approach to collision avoidance. *IEEE Robot. Autom. Mag.* 4, 23–33. doi:10.1109/100.580977

Gillett, R., Greenspan, M., Hartman, L., Dupuis, E., and Terzopoulos, D. (2001). "Remote operation with supervised autonomy (ROSA)," in *Proceedings of the 6th International Conference on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS 2001)*, Montreal.

Halme, A., Leppänen, I., Suomela, J., Ylönen, S., and Kettunen, I. (2003). WorkPartner: interactive human-like service robot for outdoor applications. *Int. J. Robot. Res.* 22, 627–640. doi:10.1177/02783649030227011

Hebert, P., Bajracharya, M., Ma, J., Hudson, N., Aydemir, A., Reid, J., et al. (2015). Mobile manipulation and mobility as manipulation – design and algorithms of robosimian. *J. Field Robot.* 32, 255–274. doi:10.1002/rob.21566

Heppner, G., Roennau, A., Oberländer, J., Klemm, S., and Dillmann, R. (2015). *Laurope – Six Legged Walking Robot for Planetary Exploration Participating in the SpaceBot Cup*, Noordwijk: ESA.

Holz, D., Holzer, S., Rusu, R. B., and Behnke, S. (2011). "Real-time plane segmentation using RGB-D cameras," in *RoboCup 2011: Robot Soccer World Cup XV* (Istanbul: Springer), 306–317.

Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., and Burgard, W. (2013). OctoMap: an efficient probabilistic 3D mapping framework based on octrees. *Auton. Robots* 34, 189–206. doi:10.1007/s10514-012-9321-0

Huang, T., Yang, G., and Tang, G. (1979). A fast two-dimensional median filtering algorithm. *IEEE Trans. Acoust. Speech Signal Process.* 27, 13–18. doi:10.1109/TASSP.1979.1163188

Johnson, M., Shrewsbury, B., Bertrand, S., Wu, T., Duran, D., Floyd, M., et al. (2015). Team IHMC's lessons learned from the DARPA robotics challenge trials. *J. Field Robot.* 32, 192–208. doi:10.1002/rob.21571

Joyeux, S., Schwendner, J., and Roehr, T. M. (2014). "Modular software for an autonomous space rover," in *Proceedings of the 12th International Symposium on Artificial Intelligence, Robotics and Automation in Space (SAIRAS)*, Montreal.

Kaupisch, T., and Fleischmann, M. (2015). "Mind the robot – rovers leave tracks in the artificial planetary sands," in *COUNTDOWN – Topics from the DLR Space Administration*, Vol. 31, 38–43. Available at: http://www.dlr.de/rd/en/desktopdefault.aspx/tabid-4788/7944_read-45190/

Kaupisch, T., Noelke, D., and Arghir, A. (2015). "DLR spacebot cup – Germany's space robotics competition," in *Proc. of the Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA)*, Noordwijk.

Kim, M.-S., and Oh, J.-H. (2010). Posture control of a humanoid robot with a compliant ankle joint. *Int. J. HR* 07, 5–29. doi:10.1142/S0219843610001988

Kröger, T. (2011). "Opening the door to new sensor-based robot applications – the reflexxes motion libraries," in *Proc. of the IEEE Int. Conference on Robotics and Automation (ICRA)*, Shanghai.

Kuemmerle, R., Grisetti, G., Strasdat, H., Konolige, K., and Burgard, W. (2011). "G2o: a general framework for graph optimization," in *Proc. of the IEEE Int. Conference on Robotics and Automation (ICRA)*.

Lacan, J., Roca, V., Peltotalo, J., and Peltotalo, S. (2009). *Reed-Solomon Forward Error Correction (FEC) Schemes*. Technical Report RFC (5510). IETF.

Mehling, J., Strawser, P., Bridgwater, L., Verdeyen, W., and Rovekamp, R. (2007). "Centaur: NASA's mobile humanoid designed for field work," in *Proc. of the IEEE Int. Conference on Robotics and Automation (ICRA)* (Roma: IEEE), 2928–2933.

Pedersen, L., Kortenkamp, D., Wettergreen, D., and Nourbakhsh, I. (2003). "A survey of space robotics," in *Proceedings of the 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space* (NASA), 19–23.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., et al. (2009). "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, Vol. 3, (Kobe), 5.

Raibert, M., Blankespoor, K., Nelson, G., Playter, R., The Big-Dog Team. (2008). "BigDog, the rough-terrain quadruped robot," in *Proceedings of the 17th World Congress, The International Federation of Automatic Control* (Seoul, Korea: Elsevier), 10823–10825.

Roca, V., Neumann, C., and Furodet, D. (2008). *Low Density Parity Check (LDPC) Staircase and Triangle Forward Error Correction (FEC) Schemes*. RFC (5170). IETF.

Rodehutskors, T., Schwarz, M., and Behnke, S. (2015). "Intuitive bimanual telemanipulation under communication restrictions by immersive 3d visualization and motion tracking," in *Proc. of the IEEE-RAS Int. Conference on Humanoid Robots (Humanoids)*, Seoul.

Roennau, A., Kerscher, T., and Dillmann, R. (2010). "Design and kinematics of a biologically-inspired leg for a six-legged walking machine," in *3rd IEEE RAS and EMBS International Conference on Biomedical Robotics and Biomechatronics (BioRob)* (Tokyo: IEEE), 626–631.

Schwarz, M., and Behnke, S. (2014). "Local navigation in rough terrain using omnidirectional height," in *Proc. of the German Conference on Robotics (ROBOTIK)* (Munich: VDE).

Schwarz, M., Rodehutskors, T., Droeschel, D., Beul, M., Schreiber, M., Araslanov, N., et al. (2016). NimbRo rescue: solving disaster-response tasks through mobile manipulation robot Momaro. *J. Field Robot*. Available at: http://www.ais.uni-bonn.de/papers/JFR_NimbRo_Rescue_Momaro.pdf

Schwendner, J., Roehr, T. M., Haase, S., Wirkus, M., Manz, M., Arnold, S., et al. (2014). "The artemis rover as an example for model based engineering in space robotics," in *ICRA Workshop on Modelling, Estimation, Perception and Control of All Terrain Mobile Robots*, Hong Kong.

Segal, A., Haehnel, D., and Thrun, S. (2009). "Generalized-ICP," in *Proc. of Robotics: Science and Systems*, Seattle.

Semini, C., Tsagarakis, N., Guglielmino, E., Focchi, M., Cannella, F., and Caldwell, D. (2011). Design of HyQ-A hydraulically and electrically actuated quadruped robot. *Proc. Inst. Mech. Eng. I J. Syst. Control Eng.* 225, 831–849. doi:10.1177/0959651811402275

Stentz, A., Herman, H., Kelly, A., Meyhofer, E., Haynes, G. C., Stager, D., et al. (2015). CHIMP, the CMU highly intelligent mobile platform. *J. Field Robot.* 32, 209–228. doi:10.1002/rob.21569

Stückler, J., Schwarz, M., Schadler, M., Topalidou-Kyniazopoulou, A., and Behnke, S. (2015). NimbRo Explorer: semiautonomous exploration and mobile manipulation in rough terrain. *J. Field Robot.* 33, 411–430. doi:10.1002/rob.21592

Sünderhauf, N., Neubert, P., Truschzinski, M., Wunschel, D., Pöschmann, J., Lange, S., et al. (2014). "Phobos and deimos on mars – two autonomous robots for the dlr spacebot cup," in *Proceedings of the 12th International Symposium on Artificial Intelligence, Robotics and Automation in Space-i-SAIRAS'14* (Montreal: The Canadian Space Agency (CSA-ASC)).

Wedler, A., Rebele, B., Reill, J., Suppa, M., Hirschmüller, H., Brand, C., et al. (2015). "LRU – lightweight rover unit," in *Proc. of the 13th Symposium on Advanced Space Technologies in Robotics and Automation (ASTRA)*, Noordwijk.

Winstein, K., and Balakrishnan, H. (2012). "Mosh: an interactive remote shell for mobile clients," in *USENIX Annual Technical Conference* (Boston: USENIX Association), 177–182.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

frontiers in Robotics and AI

ORIGINAL RESEARCH
published: 23 June 2016
doi: 10.3389/frobt.2016.00033

The ArmarX Statechart Concept: Graphical Programing of Robot Behavior

Mirko Wächter, Simon Ottenhaus, Manfred Kröhnert, Nikolaus Vahrenkamp and Tamim Asfour*

High Performance Humanoid Technologies Lab (H^2T), Institute for Anthropomatics and Robotics (IAR), Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Programing sophisticated robots, such as service robots or humanoids, are still a complex endeavor. Although programming robotic applications requires specialist knowledge, a robot software environment should support convenient development, while maintaining full flexibility needed when realizing challenging robotics tasks. In addition, several desirable properties should be fulfilled, such as robustness, reusability of existing programs, and skill transfer between robots. In this work, we introduce the ArmarX statechart concept, which is used for describing control and data flow of robot programs. This event-driven statechart approach of ArmarX helps realizing important features, such as increased robustness through distributed program execution, convenient programming through graphical user interfaces, and versatility by interweaving dynamic statechart structure with custom user code. We show that using hierarchical and distributed statecharts increases reusability, allows skill transfer between robots, and hides complexity in robot programming by splitting robot behavior into control flow and functionality.

Keywords: robot software framework, robot programing, statecharts, graphical user interfaces, distributed processing

OPEN ACCESS

Edited by:
Jochen J. Steil,
Bielefeld University, Germany

Reviewed by:
Ali Paikan,
Istituto Italiano di Tecnologia (IIT), Italy
Christian Schlegel,
University of Applied Sciences Ulm, Germany

****Correspondence:***
Mirko Wächter
mirko.waechter@kit.edu

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *03 December 2015*
Accepted: *03 June 2016*
Published: *23 June 2016*

Citation:
Wächter M, Ottenhaus S, Kröhnert M, Vahrenkamp N and Asfour T (2016) The ArmarX Statechart Concept: Graphical Programing of Robot Behavior. Front. Robot. AI 3:33. doi: 10.3389/frobt.2016.00033

1. INTRODUCTION

Programing complex robots like humanoids is challenging and is often divided into at least two domains. One being, low-level control, which is essential for smooth execution, system stabilization, safety, and consideration of dynamic effects. On the other hand, high-level robot programing copes with perception, task and motion planning, user interaction, memory concepts, and reusability of robot skills. Well-designed robot software frameworks should support the development of complex robot programs on all system levels. Therefore, a framework needs to provide well-defined interfaces for all available robot components and the flexibility to additionally implement application- or task-specific behaviors. In addition, a basic set of robot skills (i.e., robot programs for a special behavior) should be available, which can be used to assemble more complex robot programs. One challenge in building a robot framework is to provide means for doing this in a robust and convenient way.

In this work, we focus on high-level robot programing and discuss how using hierarchical, distributed statecharts for encoding robot skills aid in achieving convenient programing and reusable, transferable robot behaviors. Possible candidates of statechart implementations must meet the following requirements to be considered eligible: full control over data flow and control flow, local scoping of data similar to encapsulation in programing, runtime-reconfigurability as

well as runtime introspection. It should also not be necessary to recompile programs upon structural or control flow changes. Furthermore, a graphical user interface is desirable in order to reduce the unavoidable complexity of describing robot behavior and to minimize development and comprehension efforts. This convenience feature should provide means for defining and parametrizing both control and data flow, online visualization of active states and transitions in running programs, and a convenient way to incorporate custom user code. Additionally, a code generator should be provided for enforcing type-safety and catching errors in user code as early as possible as well as allowing source code auto-completion in development environments of statechart-related data types and functions.

We will discuss the statechart concept of the robot development environment ArmarX (Vahrenkamp et al., 2015) in detail and show how it provides both reusability of high-level robot skills realized as distributed, hierarchical statecharts, and the possibility to add user code with access to the external robot components. **Figure 1** shows how statecharts are integrated in the basic structure of ArmarX.

In Section 2, we elaborate on the state of the art and compare it to our approach. Our statechart concept is presented in detail in Section 3. This is extended in Section 4 in regard to usability and integration in the robot development framework ArmarX. In Section 5, we show some use cases for the presented approach to give a better understanding of how it can be utilized. The discussion in Section 6 reflects our experience with the ArmarX statecharts, and Section 7 concludes the paper.

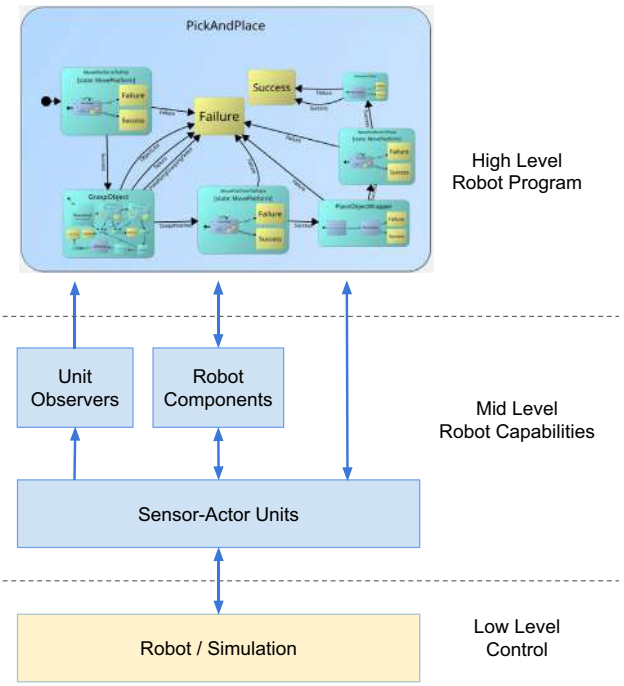


FIGURE 1 | Basic structure of ArmarX. The low-level hardware access is abstracted through the Sensor–Actor unit concept. These Sensor–Actor units realize hardware or simulator access and hide the low-level communication from higher-level layers of the robot software. On the mid-level, robot capabilities, such as perception, planning, and motion generation, are implemented in a network transparent way. The high-level layer comprises a set of robot skills, realized as statecharts, which are used for assembling complex robot programs. The arrows depict middleware communication, which can be local or remote. The decision if a communication channel is local or incorporates remote calls is transparently taken on the fly by the middleware based on the current deployment.

2. RELATED WORK

Robot Development Environments (RDEs) have coevolved with the increasing complexity and capabilities of modern robots. Taking a closer look at recent RDEs, there has been an agreement on the necessity of distributed processing for complex robotic systems [e.g., Scholl et al. (2001), Bruyninckx et al. (2003), Metta et al. (2006), Ando et al. (2008), Quigley et al. (2009)]. Communication in such distributed systems is often performed via middlewares, such as CORBA (2006) or Ice (Henning, 2004). In other cases, specialized middleware systems or messaging protocols have been developed based on task-specific requirements.

2.1. Robot Development Environments

Already several years ago, Schlegel and Wörz (1999) saw the necessity to develop modular and distributed frameworks for complex multi-sensorimotor systems and presented the software framework SmartSoft. Apart from distribution and communication, RDEs differ depending on which part of robot programming they target. For example, MiRPA (Finkemeyer et al., 2007) provides a low-level message-oriented real-time communication middleware. OpenRTM (Ando et al., 2008) is situated on the lower control level and provides a component model with input, output, and configuration interfaces as well as basic execution state machines (inactive, active, error states). MOOS (Newman, 2008) is located on a similar level than OpenRTM and provides a publish–subscribe-based communication and data exchange between MOOS applications via a central database. OpenRDK (Calisi et al., 2008) is also a low-level framework and uses agents as main abstraction, which dynamically instantiate modules-containing functionality. Modules communicate through a blackboard-type mechanism and can access input, output, and parameter data of any other module. YARP (Metta et al., 2006), being used for the iCub robots (Metta et al., 2008), provides low-level communication as a basis for higher-level robot capabilities implemented in the iCub software. Last, ROS (Quigley et al., 2009) and Orocos (Bruyninckx et al., 2003) lean toward the implementation of higher-level system capabilities. In ROS, software modules called *nodes* span a peer-to-peer network and send messages, whereas Orocos provides an explicit component model and separates the structure of the control system from its functionality.

In 2010, Bischoff et al. (2010) started an initiative to structure and formalize the robot development process by identifying and documenting best practices and refactoring existing components to increase reusability and robustness.

Schlegel et al. (2015) and Thomas et al. (2013) strive in their approaches to divide tasks into different complexity levels to reduce the knowledge required to adapt a robot to new but similar tasks.

A different type of architecture for robot skill specification was proposed by Nordmann et al. (2015). They fuse methods of software design and classical motion primitives to form a model-driven approach for complex motion control architectures.

2.2. Statecharts and Coordination Systems

Besides the original publications (Harel, 1987; Harel and Politi, 1998), there are many other publications (Coleman et al., 1992; Von der Beeck, 1994; Samek, 2002) and software projects (Angermann et al., 2014; EasyCODE, 2015; Yakindu, 2015) on statecharts for a variety of different use cases.

The concept of statecharts as new formalism to represent and describe complex systems was presented first by Harel (1987) and Harel and Politi (1998). The concept extends the finite state machines (FSM) proposed by Gill et al. (1962) to a powerful representation, which significantly reduces the complexity for system developers by introducing several notations features. Like FSMs, statecharts consist in their core of states and transitions between these states and extend FSMs by the following features. The most important addition is the introduction of hierarchically nested states. Harel introduced *inter-level-transitions* to allow direct transitions into sub-states as well as *orthogonality* to allow parallel execution of different states at the same statechart level. Moreover, a *history*-connector was added to provide states with a memory, which store the information about which sub-state should be reactivated when a state is revisited. *Condition*-connectors control to which subsequent state a transition leads. Finally, each state can be connected to actions being triggered during different phases of the state: entering, leaving, and an action that is executed repeatedly as long as the state is active.

Several general purpose frameworks exist, which can be used to specify the program flow based on statechart mechanisms. In late 2015, the W3 consortium released version 1.0 of an XML statechart notation [ScXML, World Wide Web Consortium (W3), 2015] to establish one format describing Harel statecharts. Similarly, the Object Management Group defined the UML StateMachines notations [Object Management Group (OMG), 2015]. While these specifications mainly focus on general purpose notations of the Harel formalism, the ArmarX statecharts aim at providing a ready-to-use statechart framework in the robotics context.

The well-known *de facto* extension of C++ Boost (Huber, 2007) contains a subproject called the Boost Statechart Library, which offers a statechart implementation close to the original formalism of Harel. It has the unique feature of specifying the statecharts with C++ templates and achieving compile-time statechart validation. While this is a valuable feature to ensure valid statecharts, it does not fit our requirements. For our purposes, we require runtime-reconfigurability and no recompilation on layout changes as well as runtime introspection, which is difficult to achieve if the structure is specified implicitly with C++ templates. On the side of graphical tools, the statechart graphical modeling tool QM (Quantum Leaps, 2015) provides means for designing and implementing event-driven low-level statecharts for embedded systems with a strong focus on traceability at the code level. The complete statecharts are generated into C++ code, meaning that for statechart structure changes recompilation is necessary. In our statecharts, we aim to generate code only to catch errors in the user code as early as possible and for IDE auto-completion purposes. In Yakindu (2015), another graphical statechart modeling tool is presented, aiming at usability and assistance inside the editor during typing. Though it seems to target low-level statecharts like QM with limited data flow control, which is of high importance in the ArmarX statecharts, as described later.

Statecharts are widely used in robotics to control behavior on a high level (Nilsson and Center, 1973; Merz et al., 2006; Billington et al., 2010; Bohren and Cousins, 2010; Klotzbücher and Bruyninckx, 2012), since they address several of the problems of robotics like state-based control and event-triggered execution. In the well-known RDE ROS (Quigley et al., 2009), an approach called SMACH (Bohren and Cousins, 2010) is employed that focuses on data flow in statecharts. However, scope of data flow in ROS SMACH is handled differently than in ArmarX. In ROS SMACH, a child state can access all data used by its parent state. This not only eases programing because it is easy to operate with data on several levels but also violates the principle of modularity of states and creates implicit data dependencies between states. A state using datafields of a parent state cannot easily be reused in another state, since it depends on the availability of specific datafields in a parent state. Due to this, we do not allow data scopes over several state levels in ArmarX and require explicit mapping of data between state levels. Also, ROS SMACH only supports graphical online visualization of states but does not provide any tool for graphical programing. In many aspects, the statecharts of ArmarX are similar to the restricted Finite State Machine (rFSM) (Klotzbücher and Bruyninckx, 2012) from Orocos (Bruyninckx et al., 2003). However, the statecharts in Orocos focus on coordination of components but offer only very limited support to specify transition-based data flow. They promote the "pure coordination" concept, where the coordination part of the framework should be strictly decoupled from the computation capabilities to avoid unresponsiveness and blocking. This resembles the state-phases of our approach, which are split into coordination and computation phases. Though, to give the developer the ability to easily create critical sections separation of coordination and computation is only encouraged and not enforced in the ArmarX statechart framework.

Stampfer and Schlegel (2014) present an aspect similar to our dynamic structure, where they modify the statechart-formalism to allow for dynamic replacement of states with alternatives from a "robot app store" to increase robustness and reduce complexity. This enables usage of different implementations of a state in the same context, which is usually needed if a different robot should be used. Further, they also provide means for controlling data flow in their statecharts. The main difference to our approach regarding data flow control is that Stampfer and Schlegel (2014) attach data directly to events, while in our approach a transition contains a parameter mapping, which defines the sources to be used to fill a target parameter on triggering of a transition (see Section 3.3.5).

Behavior-based systems [e.g., Arkin (1998), Nicolescu and Matarić (2002), Frank et al. (2012), Paikan et al. (2014)] are another way to specify high-level robot functionality. The most striking difference is that statecharts are state-based, and

behavior-based systems are rule-based. This means statecharts have an explicit current state, while behavior-based systems only have an implicit state. Additionally, behavior-based systems are inherently parallel, whereas statecharts are sequential. While behavior-based systems may be closer to behavior of humans or animals, we do not think they scale well for programing purposes. For the developer, an explicit state is easier to comprehend, and it eases the debugging process. Both are vital criteria for software development and maintenance.

2.3. Graphical Robot Programing

When developing high-level software on a robotic platform, it is desirable to configure and connect existing components using a graphical user interface to prevent writing repetitive and therefore error-prone source code. This allows new as well as experienced users to intuitively and efficiently combine mid- and high-level components in order to create a functional system structure. Since writing software is one of the main challenges in robotics for beginners, such as students, Graphical Robot Programing offers a great entry point. It removes the obstacle presented by syntax and control flow of a conventional programing language (Rahul et al., 2014). Graphical software development often combines complexity hiding by connecting modular components on a macroscopic scale with the option to write low-level software, facilitating control tasks on joint level or performing motions in Cartesian space (Pot et al., 2009). Graphical and tabular representations are an accessible way to model system behavior in the context of simulation, validation, and consistency checking of a system design before final implementation (MathWorks, 2015c). Hirzinger and Bauml (2006) are using Simulink (MathWorks, 2015b) in conjunction with MATLAB (MathWorks, 2015a) to graphically model subsystems to later generate executables running on a real-time target. The Microsoft Visual Programing Language (Microsoft, 2012b), as part of Microsoft Robotics Developer Studio (Microsoft, 2012a), proposes developing the complete logic and program flow in a visual development environment as it lowers the bar for beginner programmers. However, we decided to limit the visual development in ArmarX to the definition of structure, used data types, and data flow in our statecharts for the benefit that the user can write unrestricted C++ code. The RDE YARP (Metta et al., 2006) also offers means of graphical programing with the *gyarpbuilder* (Paikan, 2014), yet on another level. With *gyarpbuilder*, it is possible to connect continuous input and output data of components graphically and to insert arbitrators in these connections to manipulate data flow easily. *RtcLink* (AIST, 2015) from the OpenRTM project offers a GUI to operate on RT-Components existing in a network. It can activate and deactivate components as well as connect their ports. It leverages the capabilities of an established IDE by providing the GUI as an Eclipse plugin.

3. ArmarX STATECHARTS

The complexity of multicomponent systems can be challenging in terms of program and data flow. Hence, only skilled experts are capable of designing and realizing highly connected software systems, as they are needed on humanoid robots. The aim of the ArmarX statechart concept is to reduce such complexity and increase reusability of already created functionality.

With ArmarX, we provide a generic robotics software programing environment, which combines event-driven programing with distributed component-based robot applications. A robot framework in ArmarX consists of several distributed components providing access to sensors and actors (i.e., the hardware), offering computation functionality, and realizing a robot memory system as a common data source for the robot software. On top of these robot components, the ArmarX statechart mechanism can be employed to define the structure of the mid- to high-level robot behavior (i.e., the program flow). In order to gain full flexibility within the robot applications, the programmer can use well-defined entry points to implement user-specific source code. By separating structure from behavior, the task of building new robot software applications can be supported through graphical user interfaces, while maintaining full flexibility on source code level. ArmarX provides means of designing such statecharts textually and graphically with the possibility to link them with user code to perform custom operations. The graphical way is presented in Section 4.1 in detail.

In the following sections, we present the design principles we chose for statecharts in ArmarX and the resulting differences to Harel's formalism. The details of the ArmarX statechart concept are explained in the remainder of this section.

3.1. Design Principles

Key principles of the ArmarX statecharts are modularity, reusability, runtime-reconfigurability, decentralization, and state disclosure.

- *Modularity* in our statecharts comes naturally through the individual states and explicitly specified input and output. There is no direct interaction allowed between sub-states of different parent states.
- *Reusability* is ensured, since every state can be used as a sub-state in any other state and has a specific interface for interaction. The interface is specified with the state parameters like the parameters of a function.
- *Runtime-reconfigurability* means that a statechart can be defined in configuration files, and that the statechart structure can be changed completely at runtime.
- *Decentralization* means that a statechart does not need to be resided in one process, but can be spread over several processes and hosts. This enables load balancing and robustness. A crashed distributed state component would not crash the whole statechart but would just create an event for higher layers that this specific state has failed (see Section 5.1, for an example of crash recovery).
- *State disclosure* means that the current state and all its parameters can be inspected at runtime and logged for future behavior adaptation via a network interface (see Section 4.5).

3.2. Differences to Harel's formalism

The statecharts in ArmarX differ in several points from Harel's original formalism. We omitted some of Harel's features to

comply with our design principles and to simplify the statechart design process for the developer. We added one important aspect to our statechart, which is not covered in Harel's formalism: data flow specification and control during transitions. The *hierarchy* and *condition*-connectors are available like in the original statecharts. We do not allow direct *inter-level-transitions* to not violate the principle of modularity. The *history*-connector is not available, since it conflicts with the data flow specifications, and to reduce side effects during execution as well as to simplify the comprehension of the current state of the system during introspection. Each entering of a state with the same parameters must provide the same internal state. *Orthogonality* is currently available only in a smaller scope. Each active state can contain an asynchronous user code function executed in a separate thread. Thus, the different hierarchy levels can run in parallel.

3.3. ArmarX Statechart Internals

Statecharts in ArmarX are organized in groups (see **Figure 2**). Following the composite pattern, a statechart in ArmarX is a state itself. A state can contain sub-states and transitions between these sub-states. Every state can be nested in another state to construct state hierarchies. Transitions between sub-states are triggered by events. Transitions do not only specify control flow but also data flow by attaching a parameter mapping to each transition. This mapping contains instructions on how to fill the input parameters of the next state. Distribution of statecharts over multiple processes is possible with *Remote States*, which transparently represent states located in another process.

In the following sections, we are describing the main technical aspects of the ArmarX statecharts: sub-states, transitions, events, state phases, data flow, interfacing with external components, distributed statecharts, and the dynamic statechart structure.

3.3.1. Sub-State Types

Sub-states are not the same as states in ArmarX. States are templates, which are instantiated as sub-states of other states. Though, only one type of sub-states is direct instantiations of states. ArmarX statecharts consists of four different types of sub-states, each with a specific purpose.

3.3.1.1. LocalState

Local states are normal state instances with no special features.

3.3.1.2. EndState

EndStates trigger leaving the parent state immediately. They cannot contain sub-states or execute any user code. *EndStates* are one way to specify outgoing transitions of the parent state. The name of an *EndState* specifies the name of the outgoing transition of the parent state.

3.3.1.3. RemoteState

Remote states behave like local states but point internally to a specific state in another process.

3.3.1.4. DynamicRemoteState

Dynamic remote states are similar to remote states but are like generic pointers. On entering, a dynamic remote state morphs into a specific remote state based on parameters mapped during the transition.

3.3.2. Transitions

Transitions in ArmarX statecharts define control flow and data flow. Each transition is associated with one event that the corresponding source state can process. A transition is comprised of a source state, a destination state, the associated event, and a data mapping that defines the data flow between states during this transition.

Each state has exactly one initial transition if the parent state has at least one sub-state. The initial transition can be seen as the transition from the parent state to the first sub-state. This transition is triggered immediately when the parent state is entered. Thus, when the top-level state of a state hierarchy is entered,

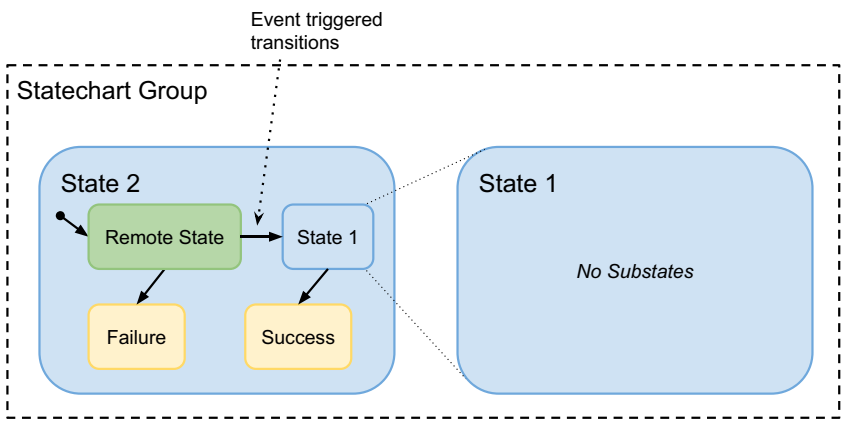


FIGURE 2 | Statecharts in ArmarX are organized in groups. States are comprised of transitions and sub-states. Sub-states are states themselves and can originate from any statechart group. If the statechart groups of parent state and sub-states differ, the sub-state is called a remote state (green state). A state can appear as a sub-state in multiple other states and can even occur multiple times within one state. Control flow is defined by transitions between sub-states. Transitions starting at the current state can be triggered by events. The control flow within a state is terminated if any end-state (yellow state) is reached. The parent state can also be left if an external event occurs.

initial sub-states on each level are entered recursively until the lowest level of the statechart is reached.

Each end-state defines one outgoing transition in the corresponding parent state. When the control flow reaches an end-state, the control flow within the parent state is terminated, and the associated transition of the parent state is triggered.

Transitions do not only describe the control flow but also carry data and define the data flow between states. The data flow during transitions is realized through a parameter mapping definition, which is attached to transitions (see Section 3.3.5). One important detail to mention is that transitions can only be created between sub-states of the same parent state, unlike in Harel statecharts. We decided to create this restriction to keep the modularity principle of states. If states would have transitions to other hierarchy levels or other parent states, the parent state could not be reused without disconnecting that transition.

3.3.3. Events

Transitions between sub-states can only be triggered by events. Events can be fired either by user code, if an end-state is reached, or if a certain condition is met. Events from user code or from end-states are fired immediately, while events from conditions are fired as soon as the condition is fulfilled.

Conditions are specified by terms based on Boolean algebra comprising literals and Boolean operators.

3.3.3.1. Event Generation with Conditions

A literal is defined by a data field of an observer, and a parametrized check that is to be performed on this data field. Conditions are installed in sensor-observers and are evaluated by the appropriate observer after each sensor update. To clarify the concept of distributed conditions, the following listing gives an example that will be explained in detail below.

The first statement in **Box 1** defines the literal `objectDistance` that describes the distance between the `hand` and `object2` and checks if this distance is below `10 mm`. `object2PoseRef` is a reference to the current pose of `object2` and is updated continuously. "`ObjectMemoryObserver.hand.pose`" describes the current pose of the hand within the `ObjectMemoryObserver`. The `poseDistance` check compares the position components of both poses and evaluates to true if the distance falls below the provided argument value (here `10 mm`).

The second statement defines `forceMagnitude`, which checks if the force in the right TCP is larger than the given threshold. Both literals are combined using a disjunction. So, if either of both conditions is true, the corresponding event `ObjectReachedEvent` is fired. The condition is evaluated in a distributed fashion. A central component called `ConditionHandler` distributes the literals to the appropriate observers. This approach avoids unnecessary transmission of high frequency sensor values, since only changes of the Boolean state of a literal are signaled by the observers. When the Boolean term of a condition evaluates to `TRUE`, the `ConditionHandler` fires the associated event. The middleware passes the event to the state that originally installed the condition.

BOX 1 | An exemplary definition of an event condition.

```
Literal objectDistance("ObjectMemoryObserver.hand.pose",
checks::poseDistance, {object2PoseRef, 10});
Literal forceMagnitude("ForceTorqueObserver.forces.TCP R",
checks::magnitudeLarger, {5.0});
installCondition("ObjectReachedEvent", objectDistance || forceMagnitude);
```

In the context of event processing, the ArmarX state disclosure concept is consistently realized, e.g., by providing an event inspection GUI, as shown in **Figure 9**. This GUI enables the developer to explore condition trees of currently active conditions, and it further allows inspecting the history of past conditions.

3.3.3.2. Event Processing

Arriving events are queued and processed sequentially by the receiving process. Due to the distributed and asynchronous nature of the software framework, processing of events need to be performed with caution in order to ensure stability and consistency. One aspect that needs to be considered is the fact that a state may already be left when an event arrives. To address this issue, all events contain the id of the destination state.

Additionally, special care needs to be taken to consistently consider parallelism. Since statecharts in ArmarX can be distributed over several processes, events can arrive and be processed in parallel. In order to deal with this situation, the ArmarX statechart framework protects critical sections, allowing concurrent multi-threaded access. Such critical sections are the event-processing function (one per statechart level) and the state phases, where the state coordination is performed (see next section for details). Thus, transitions can only be taken once, and states are only entered or exited once.

3.3.4. State Phases

During the visit of a state, different phases are passed through: *OnEnter*, *running*, *onBreak*, and *onExit*. To enable developers to execute own code in a state, each phase is linked to a user code function, i.e., C++ code. *OnEnter*, *onBreak*, and *onExit* are atomic coordination phases, while *running* is the computation phase of a state for complex, long-running computations. The order of execution of the phases is as follows: *onEnter*, *running*, and then *onBreak* or *onExit*. Before entering a state (i.e., phase *onEnter*), the parameters (explained in the next paragraph) are mapped or set to default values. In the *onEnter* phase, local variables can be set to be mapped into sub-states or prepared for later phases. When a transition is triggered, the *onExit* or *onBreak* phase is entered. Which phase is executed depends on the level where the transition was triggered: as aforementioned, statecharts are hierarchical. Thus, it is possible for a higher state to receive an event, although its sub-states are not finished yet. In this case, the sub-statecharts cannot finish in an expected manner. To give the developer an option to deal with this unexpected behavior, each state provides the *onBreak* phase. If no behavior is specified for the *onBreak* phase, the user code function of the *onExit* phase is executed. When a top-level state received an event, the complete stack of child states needs to exit first. This starts at the leaf-sub-state, a sub-state with no further sub-states, and proceeds up level by level.

Whenever a state is entered, its initial sub-state is entered as well. This means that after executing the *onEnter* phase of a state, the *onEnter* phase of the initial sub-state is also executed immediately afterward.

Since the user has freedom of implementation in the coordination phases, she/he is discouraged by warnings if computationally costly code is detected. After entering, the *running*-phase is launched in its own thread to allow the execution of computationally costly user code without interfering with the coordination. In the default-behavior, the coordination does not wait for the *run*-function to finish and ignores all results produced by the *running*-phase after the state was left. During each of these phases, the developer can access different parameter dictionaries in the user code functions, which are explained in the next paragraph.

Although C++ code is difficult to verify, we decided to employ C++, since all our algorithms and most robotics algorithms in general are written in C++.

3.3.5. Transition-Based Data Flow

One important, and to our best knowledge in this extend, unique feature of the ArmarX statecharts is the extensive control of the data flow in the statecharts, which eases accomplishing the modularity and reusability principles. All states are equipped with input and output parameter dictionaries to decouple states from external global data storage. Input parameters are read-only in user code functions and specify all parametrization the state needs for its computations. Output parameters can be set in the user code functions, contain the results of a state, and can be used as source for input parameters of the next state or mapped back to the parent's local or output parameters.

Additionally, so-called local parameters are provided and accessible for the user code. Local parameters are intended to be used for temporal local storage of parameters that are passed down to sub-states' input, passed up from sub-states' output, or passed between different state phases. Once a state is left, all parameters are reset in order to avoid side effects of previous visits.

Each parameter dictionary field consists of a string identifier and a variant data type that can manifest itself into arbitrary types. ArmarX already provides the basic types bool, integer, float, double, and string as well as several types associated with robotics, like vectors, matrices, 3D poses, or probability distributions. If needed, developers can implement new types easily.

These parameter dictionaries are defined by the developer and specify the interface of each state, i.e., which data it needs for execution. Each parameter can be optional, can have a default value,[1] and/or can be filled from several sources. We call this *parameter mapping*. When a state is used, its non-optional input parameters without default values need to be connected with other parameters of the same type. Thus, a parameter mapping for each of these input parameters needs to be created for each state instance. The developer can choose between mappings from the output of a previous state from the same hierarchy level, the input or local parameters of the parent state, or from a parameter attached to the transition-event. Additionally, developers can map values from the output of a state to the local or output parameters of the parent state. Later, when another sub-state needs the calculated value as an input parameter, the local parameter is mapped to that input parameter. For example, generic counter states can be implemented following this pattern, so that counting loop sequences of states can be defined without writing any additional specialized custom code. With this, it is possible to pass data from a sub-state to another state later in the chain more easily. Otherwise, the parameters would need to be mapped from state to state. **Figure 3** shows the different types of mappings during transitions.

[1]Consequently, if parameters have a default value, the optional flag does not make much sense any more, thus these two Boolean flags basically form a tri-state.

3.3.6. Interfacing with External Components

Statecharts that can only access functionality and data of themselves are not particularly useful for robotics. Therefore, they must be able to access all available components. Since ArmarX is a heavily distributed system, it cannot be assumed that required components are running in the same process or on the same host. Hence, states require network proxies to these components, and it should be ensured that a state is only started if all required components are available. Dependencies for a group of states can therefore be defined in a so-called *StatechartContext*, which manages dependencies and enables states to communicate with external components.

3.3.7. Distributed Statecharts

Another important feature of ArmarX is the possibility to distribute statecharts over several processes or hosts. To this end, states in ArmarX are organized in groups, which, for example, contain states that are semantically similar and share the same dependencies to external components. In this context, semantically similar means, states that share common aspects regarding their purpose. For example, all states for controlling holonomic

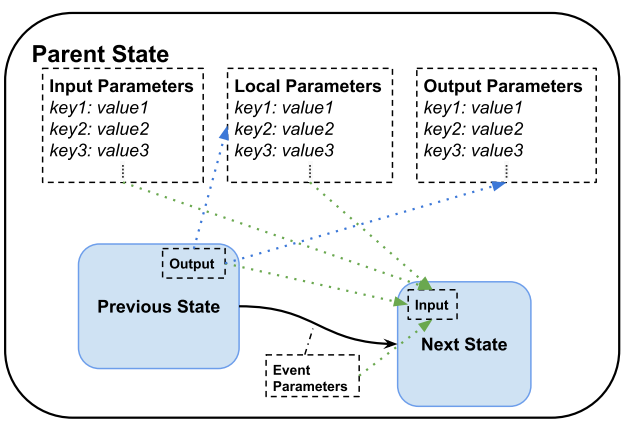


FIGURE 3 | Available types of parameter mapping during transitions. Each state has three parameter dictionaries: input, local, and output parameters. In the blue sub-states, only the relevant dictionaries for the mapping during the transition are shown. The green arrows show possible mappings to the input parameters of the next state. The blue arrows show possible mappings from the output of the previous state to the local and output parameters of the parent state. These mappings happen after leaving the previous state and before entering the next state.

platform movements, from a PD-controller to calls to a path planning component, should be encapsulated in one statechart group. Though, this is just a useful convention.

Each group is executed as one component in a so-called *RemoteStateOfferer*. These *RemoteStateOfferers* offer states to be used by others states as *RemoteStates* over the network. For robustness, each *RemoteStateOfferer* is located in its own process. Thus, a *RemoteState* is inserted whenever a state uses a state of another group as a sub-state. This process is completely transparent for the developer. The only difference to a local state is that the *RemoteStateOfferers* name needs to be specified in addition to the state name. Theoretically, each state could have its own group for maximized robustness. Since distributed statecharts are slower than local statecharts, developers need to decide carefully when to split statecharts in more than one group. Another advantage of distributed statecharts is the possibility to deploy them close to their components. A statechart that makes heavy use of the robot's memory should ideally be located on the same host as the database servers, whereas a visual servoing statechart should be close to the vision system and the host, where joint-level control takes place. **Figure 4** depicts the linkage between different statechart groups and *RemoteStates*.

Due to the sophisticated underlying middleware Ice, which transforms network communication into normal, transparent function calls, the step from local statecharts to distributed statecharts was fairly easy. Sub-states pointing to a remote state just use another implementation of the state interface, which reroutes all the function calls over the middleware. On the other side, there is the aforementioned *RemoteStateOfferer* component, which offers a network interface to the normally, local functions of a state. This way, consistency is assured in the same way as it is done locally, with mutexed access and storage of data only on the offerer side. Thus, synchronization of data is not needed.

3.3.8. Dynamic Statechart Structure

In most statechart frameworks, the structure of the statecharts is fixed, once it has been designed by the developer. This limits the usability of statecharts in a highly dynamic environment, e.g., in the context of humanoid service robots. In this context, a symbolic planner may be incorporated, which needs to be able to change the statechart structure on the fly, according to the currently planned program flow. ArmarX supports dynamic online statechart restructuring by offering so-called *DynamicRemoteStates*, which provide generic entry points for exchangeable statecharts. As the name suggests, a *DynamicRemoteState* connects to a state in another (or its own) process. It decides upon entering, into which state it is morphed based on specific parameters passed by the transition. Additionally, it can specify more parameters that are mapped into the connected state. The correctness and completeness of the parameters is verified at runtime, i.e., when the state is loaded.

3.4. Textual Statechart Specification

While the advised method to create statecharts is to use the Statechart Editor (see Section 4.1), it is also possible to specify statecharts textually, as shown in **Box 2**.

First, each state needs to be added with its state class (TemplateParameter) and the instance name (parameter of *addState()*). Afterward, transitions between these sub-states can be created by specifying the start and end-state, and on which event these transitions should be triggered.

4. THE STATECHART CONCEPT EMBEDDED INTO ArmarX

Statecharts can be implemented in various ways by using a lookup table for transitions, by implementing transition tables

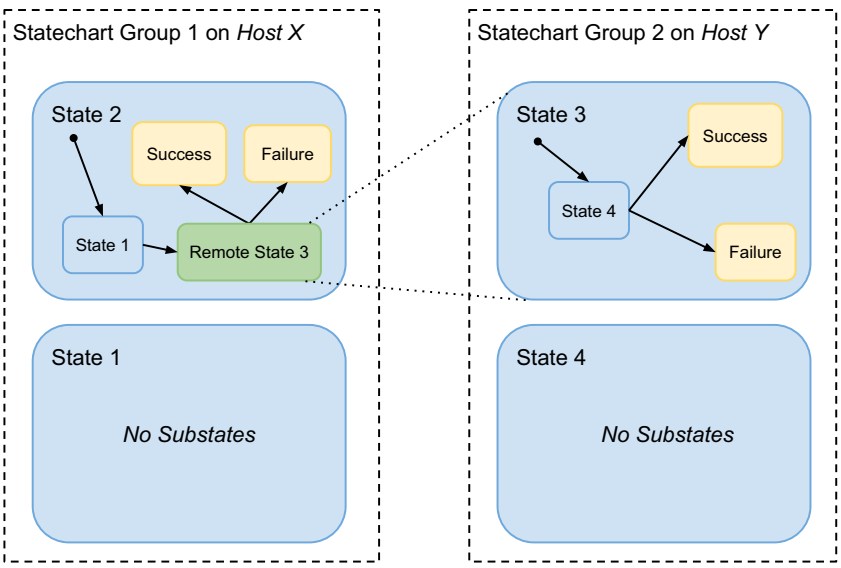


FIGURE 4 | Statecharts in ArmarX are organized in groups, which can be distributed over several processes and hosts. Each statechart group resides by default in one process. By creating *RemoteState* instances, it is possible to incorporate states of another group transparently into a statechart.

BOX 2 | An exemplary textual definition of a state.

```
void defineSubstates()
    {
    //add sub-states
    setInitState(addState < InitialState > ("Initial"));
    StateBasePtr finalSuccess = addState < SuccessState > ("Success");
    StateBasePtr finalFailure = addState < FailureState > ("Failure");
    //add transitions
    addTransition < Next > (getInitState(), getInitState());
    addTransition < TimerExpired > (getInitState(), finalFailure);
    addTransition < Success > (getInitState(), finalSuccess);
    }
```

via switch-case statements, by implementing an object-oriented state pattern, etc. Since all these approaches are based on writing code to perform the state transitions, a lot of repetitive textual description is usually necessary to define large statecharts. This textual description becomes rapidly incomprehensible for other developers. To overcome this tedious and error-prone work, a graphical statechart editor was developed for ArmarX statecharts.

4.1. Statechart Editor: Defining Control and Data Flow

The goal of the statechart editor is to enable all users to create new statecharts with sub-states, to define input and output parameters, and to connect states with transitions. The editor covers all major use cases related to editing a statechart: creation of structure, definition of control flow, and definition of data flow during transitions. The user is not required to write any custom code to create a functional statechart. We decided to store the statechart definition in a custom xml-based format.

Figure 5A shows the main window of the statechart editor. Statecharts are organized in statechart groups. A statechart group can contain multiple statecharts and sub-states. For further organization of statecharts, folders and sub-folders are available. All statechart groups are listed on the left side of the main window. The user can open any statechart from the state library for graphical editing or reuse a statechart by including it as a sub-state within another statechart.

When a statechart is opened for graphical editing, it is displayed on the right side of the editor. The editor offers a variety of options to edit a statechart, including specialized dialogs and context menus.

4.1.1. Sub-States

By dragging a statechart from the statechart library into the right editing area, a sub-state is created. A state can be reused multiple times as a sub-state within a statechart. The editor displays sub-states in two different colors: states from within the same statechart group are colored blue; states from different statechart groups are displayed in turquoise (*RemoteState*). *DynamicRemoteStates* are violet.

4.1.2. End-States

End-states are special sub-states, which are colored yellow. Each end-state implicitly creates an outgoing event/transition. When the statechart transitions to an end-state, the execution within the statechart is terminated. Additionally, the corresponding event/transition is triggered so that control flow moves back to the parenting statechart where execution is continued. When transitioning to an end-state, a statechart completes by terminating its execution entirely if no parenting statechart is present, i.e., the statechart in question is the top-level statechart.

4.1.3. Events and Transitions

As mentioned before, an end-state implicitly creates an event, which in turn implicitly creates an outgoing transition. When a statechart is initially added as a sub-state, all outgoing transitions of this sub-state are displayed as detached transitions. Transitions can be connected to other sub-states by dragging them onto the target sub-state. To create a valid statechart, all transitions have to be connected from a source state to a target state. The target state can be another sub-state, end-state, or the source state itself in case of a reflexive transition. When no detached transitions remain, the transitioning behavior of the statechart is fully defined, which implies that no event is left unhandled. Additional events can be specified in the state properties, which are fired from the code directly or on fulfilled conditions.

4.1.4. State Parameters

Each state has a list of input, output, and local parameters. A parameter is defined by its name, data type, and an optional default value. **Figure 5B** displays the input parameters of the *PlaceObjectSkill*, as it is used in ArmarX. Role and usage of the three parameter types is similar to those of parameters, return values, and local variables of functions in imperative programing languages.

4.1.5. Data Flow

A transition can be accompanied by several data mappings that define the data flow within the statechart during this transition. The statechart editor does not support global data storage, since global data storage breaks the concept of data encapsulation. Data are only passed between states during a transition. Data can be passed in 6 different ways, as depicted in **Figure 3**. The editor ensures that only parameters of the same type are mapped, while parameter mappings are edited in the transition dialog. An example is given in **Figure 5C**.

For many use cases, it is possible to compose a complete statechart by combining the capabilities listed above and by using existing states from the state library. Writing any additional source code in C++ is not required in these cases.

More complex applications may require implementing custom behavior of states using source code. For these cases, the editor offers the option to jump directly into the source code of any state. Additionally, the source code of a state can be viewed in the bottom panel below the graphical editing area (see **Figure 5A**).

4.2. Linking Implementation and Control Flow

Using a graphical definition for statecharts implies that all parameters, parameter types, parameters mappings, events, and transitions are identified via names. Since states are reusable

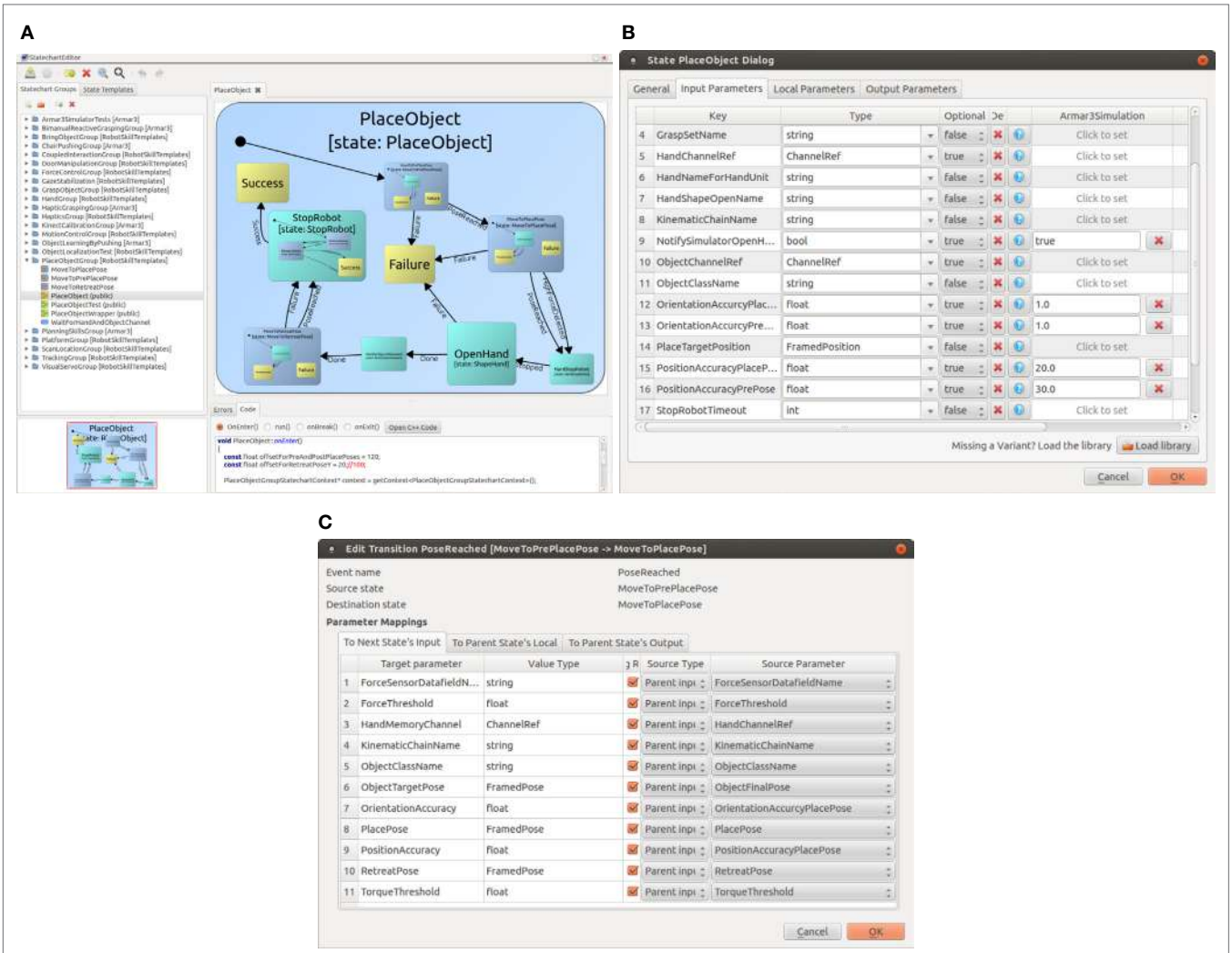


FIGURE 5 | **Dialogs of the Statechart Editor**. **(A)** The Statechart Editor is a graphical programming tool to conveniently design the control and data flow of a statechart. **(B)** The parameters of a statechart including their types and default values are defined in the Statechart Editor. **(C)** Transitions between states contain a parameter mapping which can be easily specified in the transition dialog of the Statechart Editor.

and do not store any information about previous or following states, all states have to share the same basic interface for passing input and output data. We decided to define this interface using string-Variant maps, as described in Section 3.3.5. Additionally, the state functions *OnEnter*, *Run*, *OnBreak*, and *OnExit* can be implemented in C++. Since C++ is a statically typed language without reflection, accessing an input parameter would look similar to this:

```
float myInput = ((FloatContainer*)
getInput("MyValue"))- > get();
```

The resulting code overhead to access input parameters and to write output parameters is substantial, if one takes into consideration that not only basic types but also lists and maps of any data type are supported. Furthermore, the identification of parameters by strings and run time casts can lead to run time errors that could have been detected during the compile time. Instead, accessing an input without self-written overhead code should look like this:

```
float myInput = getMyValue();
```

To achieve this type safe and auto-completion friendly interface, we employ a code generator that generates custom wrapper functions to access inputs and outputs. Inside a generated function, the parameter is referenced by name, and necessary casts are applied. Since these functions are generated automatically, access by name and the casts will never lead to run time errors. Instead, all possible errors related to parameter accessing occur at compile time. Detecting these kinds of errors before executing the statechart saves a lot of time during development.

4.3. Connecting Statecharts and ArmarX Components

One of the main aspects of statecharts in ArmarX is to interact with components. Since different statecharts for different tasks often require different sets of components, each statechart depends on a set of components. The statechart editor generates a complete list of all available ArmarX components from component meta information. The user can pick any number of components from this list and add them to the dependencies of the statechart, as shown in **Figure 6A**. Every selected component can then be accessed inside the states via a proxy object. Also, additional code is automatically generated so that the statechart registers these dependencies within the ArmarX framework before start-up. Then, the dependency resolver in ArmarX ensures that all necessary components are running before the statechart starts execution.

The list of component proxies for a state can be interpreted as the interface of this state to the ArmarX framework. Similar to object-oriented development, our goal is to keep these interfaces small. For example, a pick and place statechart requires components to operate the robotic platform, the arms, the hands, do visual servoing, etc. Without encapsulation of proxies, this would lead to a very wide interface for high-level tasks.

To approach this challenge, we offer a wrapping statechart group for each important component. Each state within a group encapsulates a common task of the encapsulated component. For example, the *HandGroup* offers states to open or close the hands. A high-level statechart can then use these wrapper states to indirectly interact with components without the need of a direct dependency on all components. For example, the *PlaceObjectGroup* needs to control the arms and hands as well as to perform visual servoing to increase accuracy. This demands interaction with the *KinematicUnit*, *HandUnit*, and *MemoryX*, among others. Each of these units is encapsulated by a statechart group, namely the *MotionControlGroup*, *HandGroup*, and *VisualServoGroup*. The *PlaceObjectGroup* uses these statechart groups to indirectly interact with the encapsulated components, as shown in **Figure 6B**.

4.4. Statechart Profiles and State Cloning: Reusing Statecharts for Different Robots

When developing a new skill for a robot, we usually start in simulation. During the transfer of the statechart to the real robot, a lot of parameters usually need to be adapted. For example, when picking up objects from a table, the height of the table might be different in simulation and in reality or the force torque sensor thresholds differ. But, these are just differences in parametrization and not on source code level. Thus, our goal is to have the same source code working in simulation as well as on the real robot.

To meet this requirement, we introduce the concept of profiles. When working with the statechart editor, the user first selects which profile he or she wants to work with. Every parameter of every state can have specialized values in different profiles, but it is also possible to define default values that apply to multiple profiles if no specialized value is set.

Figure 7 displays the parameter edit dialog for the place object example mentioned above. The parameter *ObjectName* is set to "GreenCup" and is applied in simulation as well as on the real robot. The parameter *TableHeight* is set to 900 mm for simulation and to 800 mm for the real robot. The statechart will be executed with the appropriate parameters depending on the selected profile.

When reusing statecharts for new robots, simple parameter adjustment is often not sufficient. The underlying behavior implementation of states might require adaptation or the statecharts need to communicate with different components altogether. To cover these cases, the statechart editor offers the option of cloning complete statecharts, including all sub-states as well as cloning all

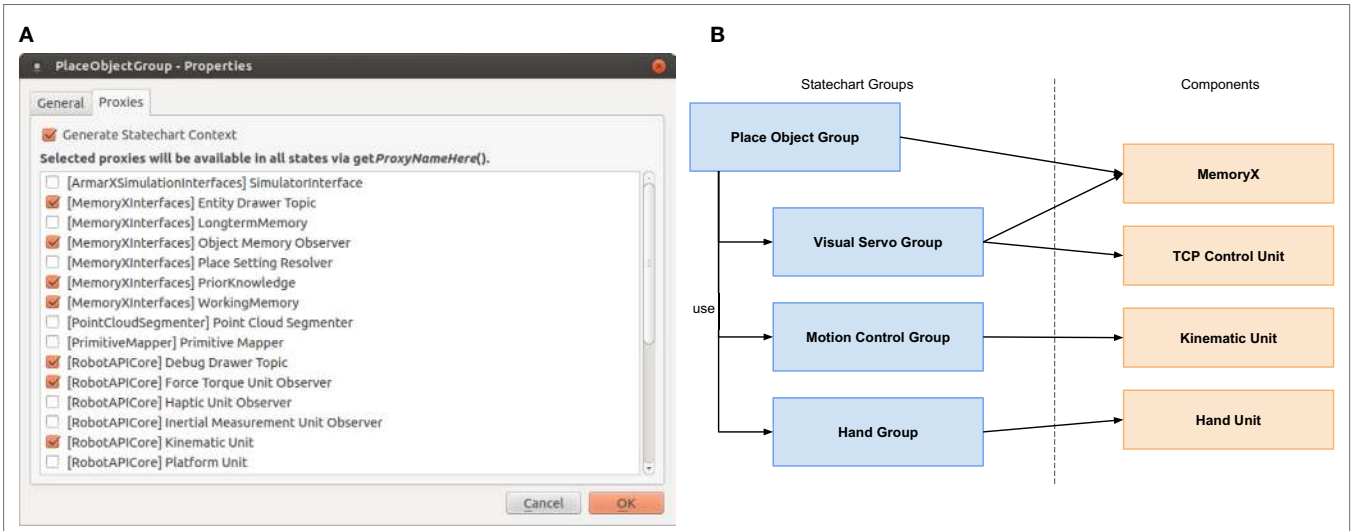
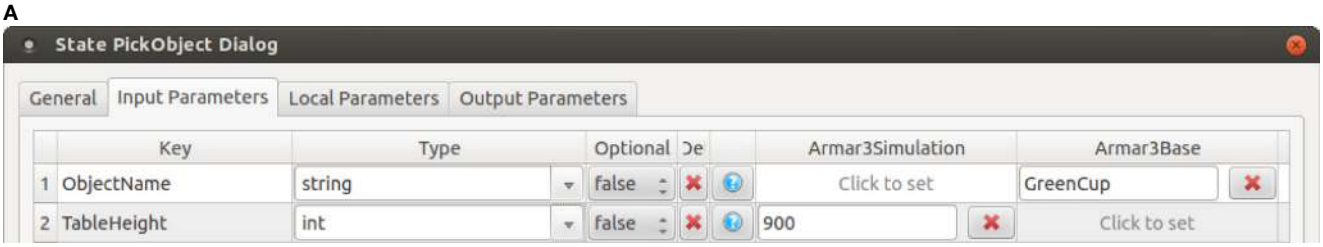


FIGURE 6 | Interaction of Statecharts and Components in ArmarX. **(A)** Components which are used by statecharts can be selected in the Statechart Editor. This defines the dependencies of a statechart to external components. **(B)** Statechart groups encapsulating components to reduce interface width on high level statecharts.

A



Parameters for simulation

B

Parameters for the real robot ARMAR-IIIb

FIGURE 7 | **The figure shows statechart profiles for simulation (A), and a real robot (B) to compensate for differences between simulation and robot**.

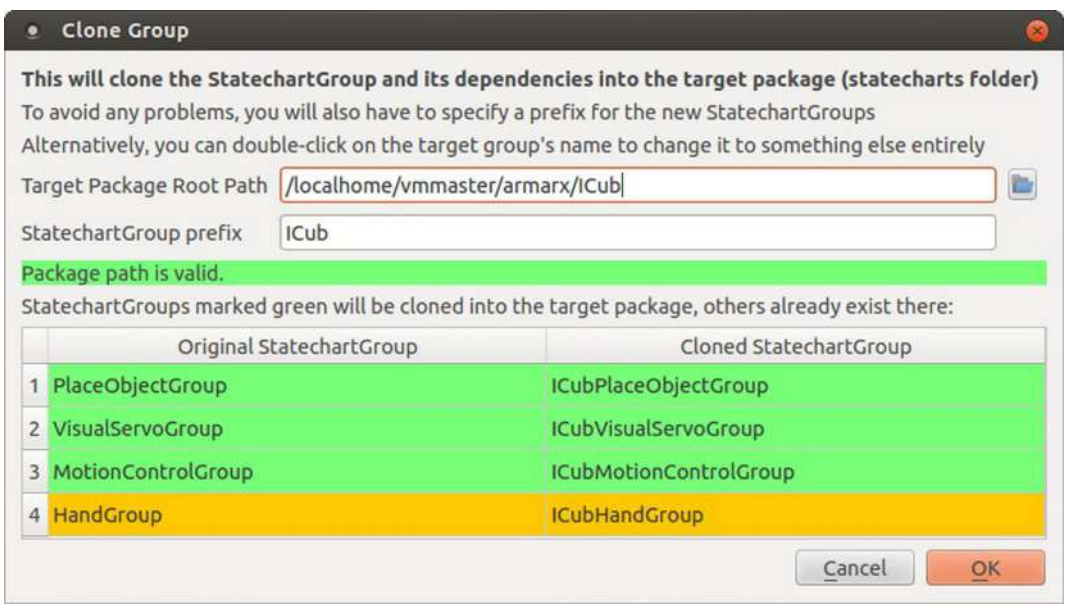


FIGURE 8 | **Cloning the *PlaceObjectGroup* for the iCub**.

state-dependencies of the statechart in question. Dependencies are determined by finding all external statecharts that are used in the statechart to be cloned. This process is applied recursively until the list of dependencies is complete. In addition, the statechart editor checks if some of the dependencies have already been cloned for the target robot and omits these states while cloning accordingly. When cloning states, it is possible to apply a prefix to all new states to avoid later confusion. Additionally, all necessary C++ source code files are copied, renamed, and modified to match the new names. Statecharts yielded by the cloning process can be compiled and executed without any manual adaptations or amending of source code.

Figure 8 shows an exemplary use case, in which the statechart group for placing objects (*PlaceObjectGroup*) is cloned to be adapted for the iCub robot. In this example, the *HandGroup* has already been cloned previously and has been adapted for the iCub under the name *ICubHandGroup*. The editor recognizes that the *ICubHandGroup* already exists inside the ArmarX iCub package. All newly cloned states that have a dependency to the *HandGroup* will use the adapted *ICubHandGroup* instead of the original.

4.5. System State Disclosure

Disclosing the state of a robotic system is one of the key features of ArmarX for diagnosing problems at runtime and inspecting the internal state during development. Programmers are able to access data of many parts of the system required for debugging, monitoring, and profiling purposes. Different built-in framework mechanisms provide this information, which includes sensor data, conditions, statechart-related events, as well as component dependencies, and the execution state of statecharts and components. Specialized visualizations are available for presenting and inspecting these different aspects. Textual output is presented as a time-stamped log, memory contents are displayed in a 3D view, and a plotter is provided for one dimensional sensor data. Statecharts, their control flow, and active states are visualized in the StatechartViewer (see **Figure 10**). Within statecharts, conditions are used to generate events based on sensor data and can be viewed as Boolean expression trees, as shown in **Figure 9**.

Additionally, ArmarX discloses the system state on a very low level for determining bottlenecks or providing hints for partitioning the distributed application. On the component level, CPU-, memory-, and network utilization data are accessible via the observer mechanism [see Vahrenkamp et al. (2015)] for easy visualization with the graphical plotter. On the statechart level, state transitions and timing information about state durations are available. To enable later processing and evaluation, this low-level data can be stored persistently in the memory structure provided by ArmarX.

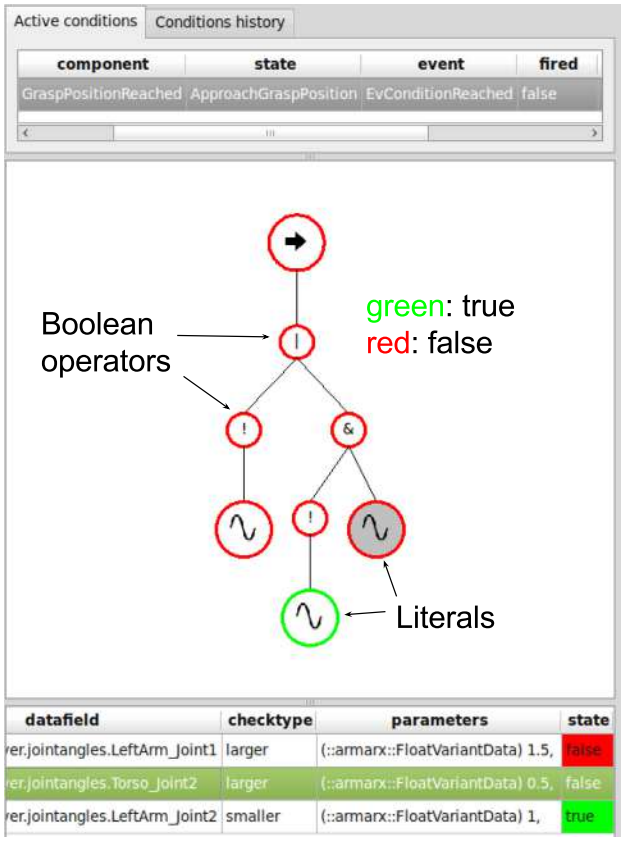


FIGURE 9 | Plugin for visualizing active and already expired conditions based on Boolean terms. The top part of the figure shows available conditions and their origin in respect to components and/or states as well as the corresponding event and its status. In the middle, a visualization of the condition as a binary tree is shown. Each sub-term of the expression tree is colored green upon fulfillment and red otherwise. The leafs of the tree are literals of the condition term. They are Boolean predicates and correspond to the table in the bottom part of the image. Each predicate consists of a datafield, a check-type, and values it is compared to.

4.6. Validation

Validation is always an important point in software development. Since generic formal validation of a statechart with arbitrary user code is difficult, we supply the possibility to create statechart test cases like unit tests. Since ArmarX statecharts usually interact with robot components, the user can specify a simulation environment that should be started alongside the statechart test. In the statechart test, the output parameters of a state or whole statechart can be validated, or the status of robot components like the memory can be checked.

5. APPLICATIONS AND USE CASES

In this section, several applications and use cases realized with ArmarX will be presented, which show how distributed statecharts support robustness and provide both convenient usage and flexibility.

5.1. Robustness and Fault Recovery

In this use case, we show how fault recovery concepts are realized within ArmarX. This is important, since most robotics software is written in C++, which allows writing program, crashing implementations easily. Hence, a robust robot framework must be able to deal with crashing applications in a way that other components are informed but not affected by a component fault. Further, fault recovery mechanisms should be provided for high- and low-level robot control.

Several concepts support robustness in ArmarX.

- Dependency management: due to the distributed nature of ArmarX, crashing components do not affect other components in a non-deterministic way. If component *A* depends on another component *B*, the dependency manager of ArmarX only sets *A* to the state *connected* after *B* is fully initialized and connected. If component *B* stops working (i.e., crashes), *A* is informed and reset to its prior *initialized* state. If the system is capable of restarting *B*, *A* will be set to *connected* again.
- Automatic restart: the deployment mechanisms of the distributed Ice middleware can be used to automatically check for running applications. In case an application (an ArmarX component) stopped working, it can be automatically started again.
- High-level fault recovery: if an implementation of a robot statechart is erroneous and causes the statechart to crash, the encapsulating statechart is automatically informed that the

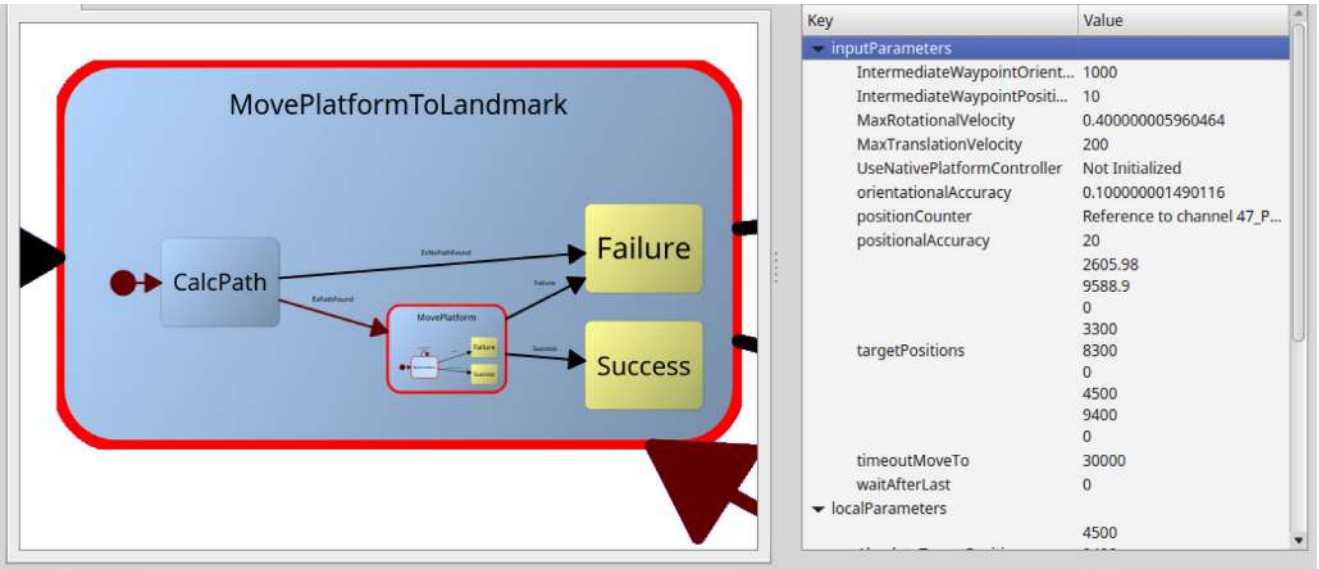


FIGURE 10 | The current state of an executed statechart can be inspected live in the StatechartViewer. The statecharts are layouted on the fly. The red state border signals that this state is active. On the right, current state parameters of the selected state can be examined. Executed transitions are highlighted as well in red, which fades to black over a few seconds to visualize the transition trace.

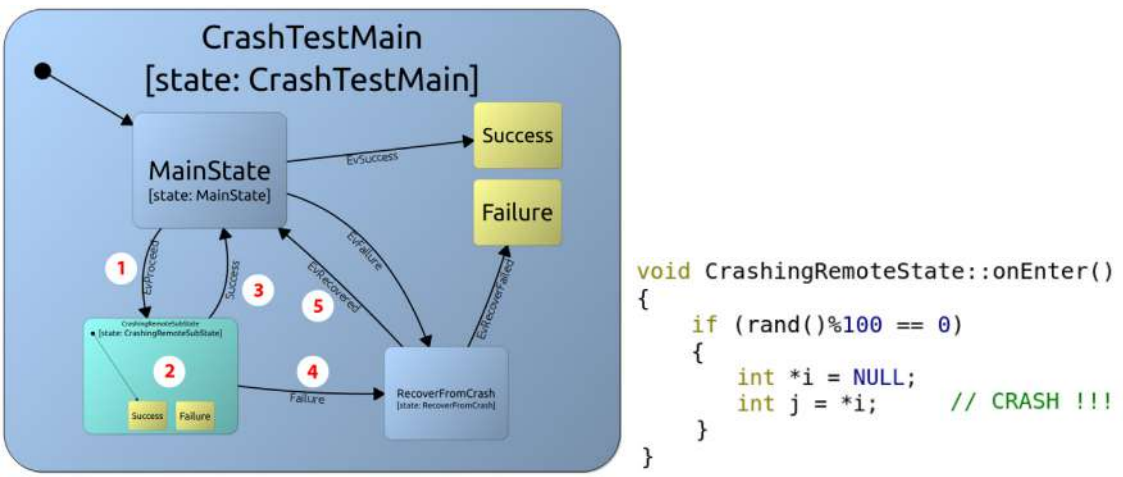


FIGURE 11 | Distribution of statecharts increases robustness of the system. Left: the *CrashTestMain* state encapsulates a remote sub-state, which faults from time to time due to a *segmentation fault*. Right: the C++ code of the enter method of the state *CrashingRemoteState*, which causes a *segmentation fault* error in a non-deterministic way. The error results in an immediate termination of the application that executes the sub-state.

execution of its sub-state resulted in a failure. Hence, the high-level robot program can consistently handle defective parts in the robot program, which could result in a non-deterministic behavior of the robot otherwise.

In the following section, we will show how a crashing sub-statechart can be handled by the robot program. In **Figure 11**, a statechart is depicted on the left. The execution of the statechart starts with the *MainState*, which emits the event *EvProceed* (1 in **Figure 11**) causing the execution to pass to the *CrashingRemoteState* statechart (2 in **Figure 11**). A normal execution would result in a success event (3 in **Figure 11**), but as shown in **Figure 11** on the right, the statechart crashes in a non-deterministic way due to a segmentation fault. Such a segmentation fault results in an immediate termination of the application executing *CrashingRemoteState*. The encapsulating statechart *CrashTestMain* automatically gets informed by the ArmarX runtime system via the *Failure* event (4 in **Figure 11**) and can recover from this faulty behavior in a deterministic manner (5 in **Figure 11**).

5.2. Generic Robot Skills

ArmarX provides a library of generic skills, implemented as statecharts, which can be configured and used for a wide variety of robots. The skills cover most basic capabilities needed to setup a robot skill library. In addition to these skills, robot-specific statecharts can be implemented to account for specific features of the platform. The set of generic skills currently provided by the ArmarX framework is listed in **Table 1**.

Generic skills can be applied to a specific robot by configuring their parameters and by providing robot-specific components on the mid-level of the ArmarX architecture (see **Figure 1**). Hence, statecharts provide a dependency list of components, which must be running before execution is possible. For example, the *ShapeHand* skill needs a *HandUnit* to be running, and the skill parameters must specify which shapes are available for execution.

5.2.1. Use Case: Generic Skills on Different Robots

To show how skills can be applied to different robots, we present a use case for YouBot (Kuka, 2015) and ARMAR-4 (Asfour et al., 2013), showing the required steps to use the skills *MoveTCP* and *MoveJoints* on different robots.

In general, two steps are needed to program a robot platform with ArmarX. First, a basic set of (robot) components must be configured in order to realize the mid-level structure of the robot software, as shown in **Figure 1**. Second, the initial set of skills has to be configured, defining the basic capabilities the robot programmer can use to build robot applications.

5.2.1.1. Robot Components

Initially, several components must be realized for the different robots. Beforehand, the robot's visualization, kinematics, and dynamics properties must be defined. In ArmarX, these properties are specified with the Simox (Vahrenkamp et al., 2012) robot file format. The minimal set of components needed for the *MoveJoints* and *MoveTCP* skills is listed below:

- *KinematicUnit*: encapsulates access on joint level. In the following examples, the robots are simulated with kinematic simulation units provided by ArmarX. On a real robot, this component is connected to the robot's hardware layer. In case of ARMAR-4, the *KinematicUnit* connects to the ArmarX-RT layer to communicate with the motors and sensors (Vahrenkamp et al., 2014).
- *KinematicUnitObserver*: observes the raw joint data in order to trigger events.
- *RobotStateComponent*: a network transparent representation of the robot used for forward and inverse kinematics.
- *TCPControlUnit*: allows control of the tool center point (TCP) in Cartesian space.

Access to the real robot (i.e., to the drivers) needs to be implemented via the *KinematicUnit* component, while all these components are already available in simulation and can be configured for use with a new robot. Hence, a basic framework can be quickly realized by configuring provided ready-to-use components of ArmarX.

5.2.1.2. Robot Skills

Once all components are set up for the specific robot, high-level robot program can be implemented. As a starting point, several skills can be taken from the ArmarX skill template library and configured to be used on the robot. In this example, the *MoveTCP* and *MoveJoints* skills are used, and a waving statechart is programed via the *Statechart Editor* tool. As shown in **Figures 12** and **13**, the realization can take advantage of the ready-to-use skill library of ArmarX on such different robots as ARMAR-4 and YouBot. In addition, the waving statechart that is used in **Figure 13** at the top can be directly executed on the real ARMAR-4, as shown in **Figure 13** at the bottom. Such a skill transfer for the complex reactive grasping skill (similar to 3) is also shown in the work presented by Paikan et al. (2015).

TABLE 1 | Generic set of skills available for use with different robots.

Skill	Description
MoveJoints	Moves joints either in position or velocity control mode
MoveTCP	Moves the tool center point to a Cartesian target
VisualServo	Implements a position-based visual servo approach
MovePlatform	Moves a platform-based robot along a graph or to a specific point
LookTo	Centers a Cartesian position with the head
GraspObject	Picks up an object with an end effector
BringObject	Picks up an object and delivers it to a specified location
ZeroForce	Enables zero force control for an end effector
StopRobot	Stops all movements
PlaceObject	Puts down a grasped object
ScanForObject	Applies a scanning strategy to search for an object
TrackObject	Tries to track an object
ViewSelection	Changes view direction, according to an automatic attention mechanism
Open/Close/Shape Hand	Move hand to specific shapes

5.3. Reactive Grasping of Unknown Objects

In the context of the Xperience (2011) Project, we developed a statechart and extended accompanying components to perform Reactive Grasping based on vision and haptics on the humanoid robot ARMAR-III (Asfour et al., 2006). This use case demonstrates reusability of ArmarX statecharts through extension of the programed behavior and the incorporation of sensor feedback on different hierarchy levels. The approach presented in Schiebener et al. (2011) was used to initially learn an object hypothesis and pose. The pose and the forward kinematics are not perfectly exact. Therefore, correcting actions during grasping are necessary. Guidance of the hand during the grasping approach phase is based on visual servo. To accommodate for inaccuracies, we need an extended visual servoing that reacts on collisions of the hand with the object. Instead of implementing a specialized version of visual servoing, we created a wrapping statechart called *Visual Servo with Collision Detection*, which is used in our Reactive Grasping (see **Figure 14**).

In parallel to the statechart execution, three different collision detection components are running to detect visual collisions, tactile collisions, and collisions inferred from proprioceptive data. These components run independently, monitor different

FIGURE 12 | The waving statechart executed on YouBot, while running a kinematic simulation.

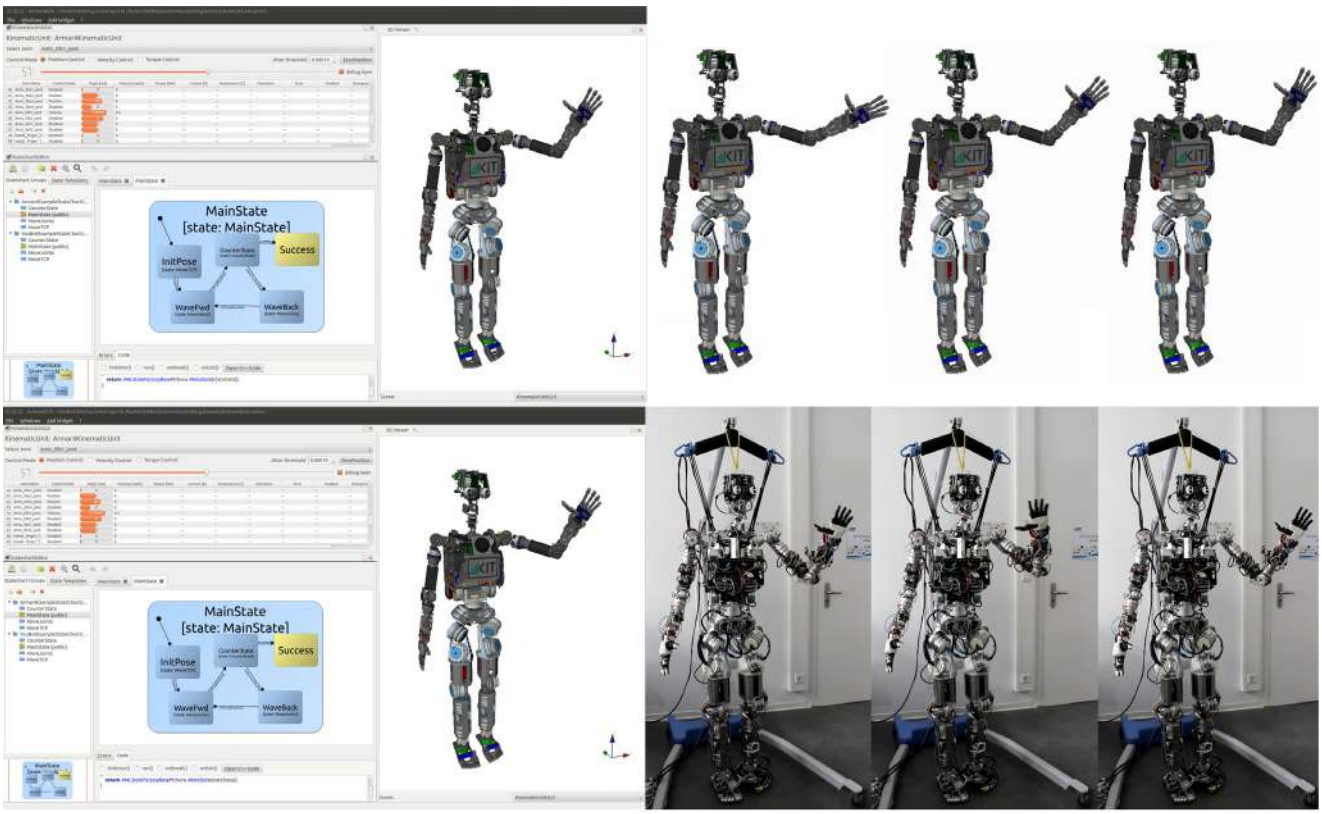


FIGURE 13 | ARMAR-4 executing a waving motion with the same statechart in simulation and on the real robot.

sensors of the robot, and offer event notifications usable in statecharts. The wrapping statechart *Visual Servo with Collision Detection* monitors the output of the collision detection components by installing conditions with given thresholds on the output data. Then, the visual servoing statechart is started as a sub-state. If any of the conditions is met during servoing, the appropriate event is fired. The wrapping state *Visual Servo with Collision Detection* is exited, and the execution of all sub-states is stopped. Hereby, the visual servoing is interrupted, and the collision can be handled appropriately by correcting the grasp pose. After correcting the pose, the statechart transitions back to the extended visual servoing.

By wrapping the visual servoing skill in a statechart, we can reuse and extend the visual servoing without modifying it. The used visual servoing skill is the standard visual servoing from the ArmarX statechart library.

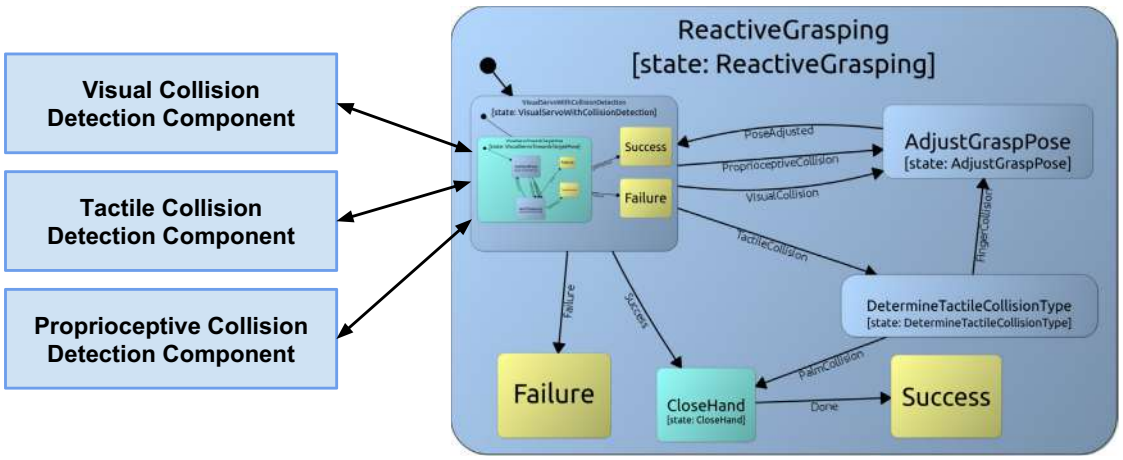


FIGURE 14 | Simplified statechart for Reactive Grasping in the Xperience project.

5.4. Dynamic State Replacement

One use case for the dynamic state replacement feature of ArmarX is the combination of a symbolic task planning system with ArmarX statecharts for execution. To connect the planning system to statecharts, a control statechart, as shown in **Figure 15**, was built around one *DynamicRemoteState* (depicted in violet). Since statecharts do not offer an interface for remote procedure calls, it is not possible to communicate with states directly. States react on external changes by observing changes in datafields. Thus, we inserted an additional component, the plan step observer, on which the statechart can install conditions to receive an event (*EvNextStepPlanned*) on changes related to the current planning step. The planning system manages this datafield containing the current action and its parameters. After the event was received, the desired skill statechart is loaded into the *DynamicRemoteState* and is directly executed. With this powerful mechanism, it is possible to implement interactive and dynamic robotic applications in a consistent and robust way.

6. DISCUSSION

In the following section, we are discussing our experiences with implementing and developing robot programs with the ArmarX statechart framework. Since we realized a large number of robot programs for a wide variety of applications for the robots of the ARMAR series, we gained rich experience that allows us to elaborate on advantages and disadvantages of the proposed concept. The presented statechart approach has extensively been used not only to demonstrate simple tasks like the examples in this paper but also for complex skills applied in real world scenarios, including grasping, opening and closing doors, mixing, or pouring as presented in Ovchinnikova et al. (2015).

We think that the decision to restrict the ArmarX statecharts to a subset of Harel's original statechart definition has been shown to benefit our statechart concept, since the removed features (inter-level-transitions, history-connector) were rarely missed but improved comprehension and reusability significantly.

Compared to the framework (Scholl et al., 2001) we used before, in which robot behaviors were also encoded by state machines, we see the advantages of now having a clear structure, advanced graphical tools, and a consistent concept for defining the data flow. In particular, the explicit definition of the data flow, i.e., specifying input and output parameters of a state with a defined and clear scope, helps immensely with understanding and reusing existing states. Another effect of this explicit data flow definition is that implicit data dependencies to other states are not possible, which ensures that entering a state with the same set of input parameters leads to the same result. Naturally, specifying the data flow explicitly and in detail is development overhead, but we are sure that it is worth the effort in the long run. Though, specifying and inspecting data flow with graphical tools simplifies this process greatly.

Such graphical tools are not only useful for defining the data flow but also indispensable for developing complex state machines (although ArmarX allows defining statechart structures manually, it is infeasible to realize complex robot programs this way). Hence, the graphical Statechart Editor is one of the most important tools of the ArmarX framework, which supports the convenient development of robot programs.

From our experience, we can confirm the necessity seen by Harel of introducing the concept of hierarchies into state machines. Hierarchies are essential for developing complex state machines and for maintaining reusability. For example, the grasping skill consists of up to six hierarchy levels, where some of the sub-states are used several times. Unrolling this into one hierarchy level results into a statechart that is practically impossible to design due to the number of required states.

The ArmarX statecharts proved to be applicable for use cases from low-level to high-level. An example of a low-level statechart is a controller for holonomic platform movements, where the leaf state is the PD-controller (using the asynchronous user code *run*-function), and the level above decides on the waypoints. An example for a high-level statechart is the statechart from Section 4, which is used for symbolic plan execution.

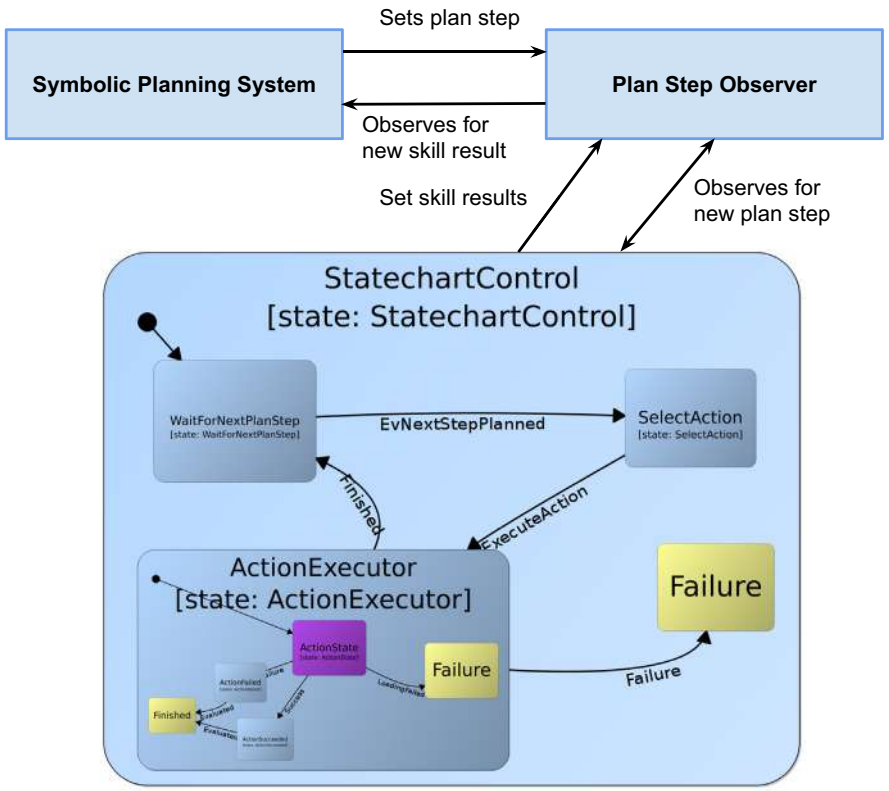


FIGURE 15 | **Planning statechart with a *DynamicRemoteState* (violet) that can be changed at runtime**.

Currently, there are many small decider or preparation states, performing some minor, but necessary calculations like coordinate transformations. This introduces clutter, since new states need to be created frequently. In the future, we plan to improve this by the possibility to attach *conversion*-functions to transitions to perform such minor calculations.

7. CONCLUSION

We presented the statechart concept of the robot development environment ArmarX and showed how high-level robot programing can be realized in a robust and convenient way. The event-driven statechart approach within ArmarX helps realizing important features, such as increased robustness through distributed program execution, convenient programing through graphical user interfaces, and versatility by interweaving dynamic statechart structure with custom user code. Additionally, we extended the original statechart concept by Harel with the possibility to explicitly specify data flow between states. These features build a solid base for implementing higher-level robot programs, which is accompanied by advanced framework capabilities, such as reusable robot programs and the presented ability to transfer skills to different robots.

In future work, we will improve the framework in terms of high-level robot program development, validation, and debugging. Therefore, we will introduce orthogonality into the statechart concept to enable parallel statechart structures. Currently, parallel execution is supported only between hierarchy levels, but there are use cases where orthogonal skill execution eases the design of a high-level robot program. In addition, we will work on automatic statechart validation in order to eliminate faults in robot programing and to speed up the development process. Furthermore, we plan to offer break points in statecharts, which will greatly improve debugging on statechart level.

AUTHOR CONTRIBUTIONS

TA identified based on his experience in developing humanoid robots the need for robot software frameworks which link sensorimotor execution and high-level planning. MW and TA developed the proposed statechart concept and MW implemented it in the ArmarX framework. MW, SO, MK, and NV realized the graphical user interface and applications of these proposed statecharts. All authors wrote and revised the manuscript.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union's Seventh Framework Programme under grant agreement no. 270273 (Xperience) and from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 643950 (SecondHands). The authors would like to thank all members and students of the Humanoids group at KIT for their various contributions to this work.

REFERENCES

AIST. (2015). *RtcLink*. Available at: http://openrtm.org/openrtm/en/content/rtclink-0

Ando, N., Suehiro, T., and Kotoku, T. (2008). "A software platform for component based rt-system development: Openrtm-aist," in *Proceedings of the 1st International Conference on Simulation, Modeling, and Programming for Autonomous Robots, SIMPAR '08* (Berlin; Heidelberg: Springer-Verlag), 87–98.

Angermann, A., Beuschel, M., Rau, M., and Wohlfarth, U. (2014). *MATLAB-Simulink-Stateflow: Grundlagen, Toolboxen, Beispiele*. München: Walter de Gruyter.

Arkin, R. C. (1998). *Behavior-Based Robotics*. Cambridge: MIT Press.

Asfour, T., Regenstein, K., Azad, P., Schröder, J., Vahrenkamp, N., and Dillmann, R. (2006). "ARMAR-III: an integrated humanoid platform for sensory-motor control," in *IEEE/RAS International Conference on Humanoid Robots (Humanoids)* (Genova), 169–175.

Asfour, T., Schill, J., Peters, H., Klas, C., Bücker, J., Sander, C., et al. (2013). "ARMAR-4: a 63 dof torque controlled humanoid robot," in *IEEE/RAS International Conference on Humanoid Robots (Humanoids)* (Atlanta), 390–396.

Billington, D., Estivill-Castro, V., Hexel, R., and Rock, A. (2010). "Modelling behaviour requirements for automatic interpretation, simulation and deployment," in *SIMPAR, Volume 6472 of Lecture Notes in Computer Science* (Darmstadt: Springer), 204–216.

Bischoff, R., Guhl, T., Prassler, E., Nowak, W., Kraetzschmar, G., Bruyninckx, H., et al. (2010). "BRICS – best practice in robotics," in *Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK)* (Munich: VDE), 1–8.

Bohren, J., and Cousins, S. (2010). The SMACH high-level executive [ROS news]. *IEEE Robot. Autom. Mag.* 17, 18–20. doi:10.1109/MRA.2010.938836

Bruyninckx, H., Soetens, P., and Koninckx, B. (2003). "The real-time motion control core of the Orocos project," in *IEEE International Conference on Robotics and Automation (ICRA)* (Taipei), 2766–2771.

Calisi, D., Censi, A., Iocchi, L., and Nardi, D. (2008). "Openrdk: a modular framework for robotic software development," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Nice: IEEE), 1872–1877.

Coleman, D., Hayes, F., and Bear, S. (1992). Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Trans. Softw. Eng.* 18, 8–18. doi:10.1109/32.120312

CORBA, O. M. G. (2006). *Corba Component Model 4.0 Specification. Specification Version 4.0*. CORBA Object Management Group.

EasyCODE. (2015). *EasyCODE*. Available at: http://www.easycode.de

Finkemeyer, B., Kröger, T., Kubus, D., Olschewski, M., and Wahl, F. M. (2007). "MiRPA: middleware for robotic and process control applications," in *Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware at the IEEE/RSJ International Conference on Intelligent Robots and Systems* (San Diego, CA), 78–93.

Frank, M., Leitner, J., Stollenga, M., Harding, S., Förster, A., and Schmidhuber, J. (2012). "The modular behavioral environment for humanoids and other robots (mobee)," in *ICINCO (2)* (Rome: Citeseer), 304–313.

Gill, A. (1962). *Introduction to the Theory of Finite-State Machines*. New York: McGraw-Hill.

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.* 8, 231–274. doi:10.1016/0167-6423(87)90035-9

Harel, D., and Politi, M. (1998). *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, Inc.

Henning, M. (2004). A new approach to object-oriented middleware. *IEEE Internet Comput.* 8, 66–75. doi:10.1109/MIC.2004.1260706

Hirzinger, G., and Bauml, B. (2006). "Agile robot development (ard): a pragmatic approach to robotic software," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Beijing: IEEE), 3741–3748.

Huber, A. (2007). *Boost Statechart Library*. Available at: http://www.boost.org

Klotzbücher, M., and Bruyninckx, H. (2012). Coordinating robotic tasks and systems with rFSM statecharts. *J. Softw. Eng. Robot.* 3, 28–56.

Kuka. (2015). *YouBot Webpage*. Available at: http://www.youbot-store.com

MathWorks. (2015a). *MATLAB*. Available at: http://www.mathworks.com/products/matlab/

MathWorks. (2015b). *Simulink*. Available at: http://www.mathworks.com/products/simulink/

MathWorks. (2015c). *Stateflow*. Available at: http://www.mathworks.com/products/stateflow/

Merz, T., Rudol, P., and Wzorek, M. (2006). "Control system framework for autonomous robots based on extended state machines," in *ICAS* (Silicon Valley: IEEE Computer Society), 14.

Metta, G., Fitzpatrick, P., and Natale, L. (2006). YARP: yet another robot platform. *Int. J. Adv. Robot. Syst.* 3, 43–48. doi:10.5772/5761

Metta, G., Sandini, G., Vernon, D., Natale, L., and Nori, F. (2008). "The icub humanoid robot: an open platform for research in embodied cognition," in *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems, PerMIS '08* (New York, NY: ACM), 50–56.

Microsoft. (2012a). *Robotics Developer Studio*. Available at: https://msdn.microsoft.com/en-us/library/bb648760.aspx

Microsoft. (2012b). *Visual Programming Language*. Available at: https://msdn.microsoft.com/en-us/library/bb483088.aspx

Newman, P. M. (2008). Moos-mission orientated operating suite. *Mass. Inst. Technol. Tech. Rep.* 2299, 1–77.

Nicolescu, M. N., and Matarić, M. J. (2002). "A hierarchical architecture for behavior-based robots," in *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 1* (New York, NY: ACM), 227–233.

Nilsson, N. J., and Center, A. I. (1973). *A Hierarchical Robot Planning and Execution System*. Menlo Park, CA: Stanford Research Institute.

Nordmann, A., Wrede, S., and Steil, J. (2015). "Modeling of movement control architectures based on motion primitives using domain-specific languages," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on* (Seattle, WA: IEEE), 5032–5039.

Object Management Group (OMG). (2015). *OMG Unified Modeling Language Version 2.5*.

Ovchinnikova, E., Wächter, M., Wittenbeck, V., and Asfour, T. (2015). "Multi-purpose natural language understanding linked to sensorimotor experience in humanoid robots," in *IEEE/RAS International Conference on Humanoid Robots (Humanoids)* (Seoul), 365–372.

Paikan, A. (2014). *Enhancing Software Module Reusability and Development in Robotic Applications [Dissertation]*. Bergamo: Istituto Italiano di Tecnologia.

Paikan, A., Metta, G., and Natale, L. (2014). "A representation of robotic behaviors using component port arbitration," in *5th International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob)*, Bergamo.

Paikan, A., Schiebener, D., Wächter, M., Asfour, T., Metta, G., and Natale, L. (2015). "Transferring object grasping knowledge and skill across different robotic platforms," in *International Conference on Advanced Robotics (ICAR)* (Istanbul), 498–503.

Pot, E., Monceaux, J., Gelin, R., and Maisonnier, B. (2009). "Choregraphe: a graphical tool for humanoid robot programming," in *Robot and Human Interactive Communication, 2009. RO-MAN 2009. The 18th IEEE International Symposium on* (Toyama: IEEE), 46–51.

Quantum Leaps. (2015). *QM Statemachines*. Available at: http://www.state-machine.com

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., et al. (2009). "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, Vol. 3. Kobe, 5.

Rahul, R., Whitchurch, A., and Rao, M. (2014). "An open source graphical robot programming environment in introductory programming curriculum for undergraduates," in *MOOC, Innovation and Technology in Education (MITE), 2014 IEEE International Conference on* (Patiala: Thapar University), 96–100.

Samek, M. (2002). *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*. Boca Raton, FL: CRC Press.

Schiebener, D., Ude, A., Morimotot, J., Asfour, T., and Dillmann, R. (2011). "Segmentation and learning of unknown objects through physical interaction," in *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on* (Bled: IEEE), 500–506.

Schlegel, C., Lotz, A., Lutz, M., Stampfer, D., Inglés-Romero, J. F., and Vicente-Chicote, C. (2015). Model-driven software systems engineering in robotics: covering the complete life-cycle of a robot. *Info. Technol.* 57, 85–98. doi:10.1515/itit-2014-1069

Schlegel, C., and Wörz, R. (1999). "The software framework SMARTSOFT for implementing sensorimotor systems," in *Intelligent Robots and Systems, 1999. IROS'99. Proceedings. 1999 IEEE/RSJ International Conference on*, Vol. 3 (Kyongju: IEEE), 1610–1616.

Scholl, K.-U., Albiez, J., and Gassmann, B. (2001). "Mca-an expandable modular controller architecture," in *3rd Real-Time Linux Workshop*.

Stampfer, D., and Schlegel, C. (2014). Dynamic state charts: composition and coordination of complex robot behavior and reuse of action plots. *Intell. Serv. Robot.* 7, 53–65. doi:10.1007/s11370-014-0145-y

Thomas, U., Hirzinger, G., Rumpe, B., Schulze, C., and Wortmann, A. (2013). "A new skill based robot programming language using uml/p statecharts," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on* (Karlsruhe: IEEE), 461–466.

Vahrenkamp, N., Kröhnert, M., Ulbrich, S., Asfour, T., Metta, G., Dillmann, R., et al. (2012). "Simox: a robotics toolbox for simulation, motion and grasp planning," in *International Conference on Intelligent Autonomous Systems (IAS)* (Jeju Island), 585–594.

Vahrenkamp, N., Wächter, M., Kröhnert, M., Kaiser, P., Welke, K., and Asfour, T. (2014). "High-level robot control with ArmarX," in *INFORMATIK Workshop on Robot Control Architectures*, 1–12.

Vahrenkamp, N., Wächter, M., Kröhnert, M., Welke, K., and Asfour, T. (2015). The ArmarX framework – supporting high level robot programming through state disclosure. *Info. Technol.* 57, 99–111. doi:10.1515/itit-2014-1066

Von der Beeck, M. (1994). "A comparison of statecharts variants," in *Formal Techniques in Real-time and Fault-Tolerant Systems* (Lübeck: Springer), 128–148.

World Wide Web Consortium (W3). (2015). *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. Available at: https://www.w3.org/TR/scxml/

Xperience. (2011). *The Xperience Project*. Available at: http://www.xperience.org

Yakindu. (2015). *Yakindu Statechart Editor Tools*. Available at: http://www.yakindu.org

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

frontiers in Robotics and AI

TECHNOLOGY REPORT
published: 07 June 2016
doi: 10.3389/frobt.2016.00031

A Comprehensive Software Framework for Complex Locomotion and Manipulation Tasks Applicable to Different Types of Humanoid Robots

Stefan Kohlbrecher[1], Alexander Stumpf[1], Alberto Romay[1], Philipp Schillinger[1], Oskar von Stryk[1] and David C. Conner[2]*

[1] Simulation, Systems Optimization and Robotics Group, Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany, [2] Capable Humanitarian Robotics and Intelligent Systems Laboratory, Department of Physics, Computer Science and Engineering, Christopher Newport University, Newport News, VA, USA

While recent advances in approaches for control of humanoid robot systems show promising results, consideration of fully integrated humanoid systems for solving complex tasks, such as disaster response, has only recently gained focus. In this paper, a software framework for humanoid disaster response robots is introduced. It provides newcomers as well as experienced researchers in humanoid robotics a comprehensive system comprising open source packages for locomotion, manipulation, perception, world modeling, behavior control, and operator interaction. The system uses the Robot Operating System (ROS) as a middleware, which has emerged as a *de facto* standard in robotics research in recent years. The described architecture and components allow for flexible interaction between operator(s) and robot from teleoperation to remotely supervised autonomous operation while considering bandwidth constraints. The components are self-contained and can be used either in combination with others or standalone. They have been developed and evaluated during participation in the DARPA Robotics Challenge, and their use for different tasks and parts of this competition are described.

Keywords: urban search and rescue, humanoid robots, mobile manipulation, human–robot interaction, motion planning

OPEN ACCESS

Edited by:
Fumio Kanehiro, National Institute of Advanced Industrial Science and Technology, Japan

Reviewed by:
Mehmet Dogar, University of Leeds, UK
Wenzeng Zhang, Tsinghua University, China

****Correspondence:***
Stefan Kohlbrecher
kohlbrecher@sim.tu-darmstadt.de

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *07 December 2015*
Accepted: *13 May 2016*
Published: *07 June 2016*

Citation:
Kohlbrecher S, Stumpf A, Romay A, Schillinger P, von Stryk O and Conner DC (2016) A Comprehensive Software Framework for Complex Locomotion and Manipulation Tasks Applicable to Different Types of Humanoid Robots. Front. Robot. AI 3:31. doi: 10.3389/frobt.2016.00031

1. INTRODUCTION

The 2015 DARPA Robotics Challenge (DRC) Finals showed that robotic systems provide promising capabilities for providing assistance in disaster scenarios that necessitate complex locomotion and manipulation abilities (see **Figure 1**). At the same time, the competition showed that there are still numerous research challenges that have to be solved before robot systems are capable and robust enough for use in real disasters.

Toward this goal, we present our ROS-based framework for solving complex locomotion and manipulation tasks. To our knowledge, it is the first fully open-sourced framework featuring documentation that allows other researchers to replicate the provided functionality and results in simulation or, after necessary interfacing, on their own robot systems. Our framework is based on ROS (Quigley et al., 2009), which has evolved to be the *de facto* standard robotics middleware within the robotics research community and parts of the robotics industry.

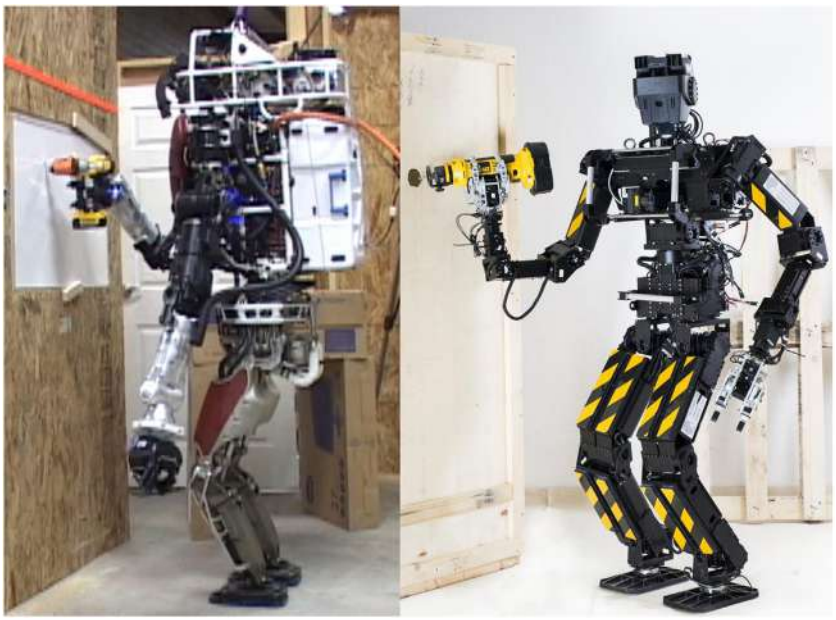


FIGURE 1 | Two of the robot systems used. The Boston Dynamics Inc. (BDI) Atlas robot and the Robotics THOR-MANG robot.

The contribution of this work is twofold:

- The framework and architecture of our approach for enabling complex humanoid robots to fulfill challenging tasks in disaster environments are detailed.
- We provide a detailed discussion of different components for perception, locomotion, and manipulation contributing to achieve the overall task of flexible disaster response.

2. RELATED WORK

While humanoid robotics is an active research area, the DRC program demonstrated the wealth of open research challenges in areas, such as controls, planning, and human–robot interaction. For the first time, humanoids had to fulfill a variety of tasks in a common competition setup, which shifted focus from concentration on specialized research topics toward the realization of humanoid (and other) systems that provide integrated perception, locomotion, and manipulation capabilities.

After the DRC Trials, publications by multiple teams described their approaches, but the majority of teams did not make their software available as open source that would allow for reproduction of the presented results. The MIT DRC team uses optimization-based planning and control (Fallon et al., 2015), LCM (Huang et al., 2010) as a middleware, and the Matlab-based Drake system as a planning and control backend.[1] Team IHMC uses a proprietary middleware based on Java (Johnson et al., 2015). Both teams provide significant parts of their software as open source software, but do not provide instructions and a setup that allows running their full setup as used for the DRC in simulation. We provide an overview of our DRC related research in Kohlbrecher et al. (2015) and detail aspects in separate publications on footstep planning (Stumpf et al., 2014), manipulation (Romay et al., 2014, 2015), and behavior control (Schillinger et al., 2016).

In Du et al. (2014), a manipulation approach used with the BDI Atlas robot is described, focusing on some of the DRC tasks. In Banerjee et al. (2015) another human-supervised manipulation control approach is described with a focus on the door DRC task.

For manipulation, bilateral teleoperation approaches allow teleoperation by the operator, while the robot simultaneously provides force feedback. Although demonstrations show the approach to be highly promising where applicable, there are potential stability issues when using bilateral approaches (Willaert et al., 2012) that make their use infeasible with constrained and significantly varying communications conditions, such as those considered in this work.

A relevant account by various teams of "What happened at the DRC" is available online (DRC-Teams, 2015). This gives a brief summary of issues and results from many teams.

3. ARCHITECTURE

The goal of this work is to provide a comprehensive and reusable software ecosystem to enable humanoid robot systems to perform complex locomotion and manipulation tasks. To provide compatibility with a wide range of robot systems and to reduce integration effort with existing open source software, the system uses ROS as middleware.

3.1. Requirements

The ability to leverage existing developments and software in a way that allows users to avoid the duplication of efforts and

[1] https://github.com/RobotLocomotion/drake

spending time re-implementing approaches is highly relevant for advancing the field of robotics research. While this requirement is not as relevant for mature commercialized robotic systems, using standard software for functional system components allows new users to reproduce results quickly. This is major driver for accelerating research in robotics and, thus, a key factor for accelerating the development of disaster response robots; that is, developing supervised autonomous systems that are deployable in real disaster situations.

The achievable complexity of robotic system architectures is limited unless the architectural design allows a transparent exchange of functional components (e.g., for manipulation or footstep planning) and also can be extended by additional functional components. Modularity, re-usability, and extensibility are key properties of the architectural design needed to enable sustainable robotic system development.

While robots can be considered expendable in the sense that a loss is acceptable (in contrast to human responders), high reliability and resilience are important aspects that disaster response robotic systems have to provide. Failures in disaster situations can have grave consequences; for instance, when a robot gets stuck or otherwise unresponsive, it can then block future access for responders, or tie up responders that could be required elsewhere.

As communications in disaster environments can be degraded, the possibility of delayed, reduced bandwidth, intermittent, or even completely absent communication has to be considered in the system design. Appropriate measures have to be taken to be tolerant of variations in communication link quality. This also motivates the need for autonomous capabilities. Autonomous performance under ideal (communications) conditions might actually be inferior to a human expert using teleoperation; however, under constrained communication conditions with outages or very high latencies, teleoperation might become impossible to use. In that case, leveraging autonomous functionality, for instance, for motion planning and control, is the only possible way to proceed.

3.2. System Architecture

To achieve high reliability, as discussed for the coactive design concept (Johnson et al., 2014) observability, predictability, and directability of the robotic system are required. When considering the human supervisor and robot as a team, the members, thus, have to allow each other to understand the state of the other side (observability). They also have to be able to predict and understand the intent of the other side (predictability). Lastly, team members have to be able to communicate meaningful and accurate commands (directability).

The capability of informing the operator about the robot state using appropriate information and visualization must be considered (Kohlbrecher et al., 2015). Predictability is achieved by visualizing action outcomes prior to the command being sent to the robot. Achieving directability requires interfaces that allow for efficient and reliable interaction. These concepts will be revisited in following sections.

As noted previously, to achieve high reliability and versatility, the capability to flexibly change control and interaction modes between autonomous and teleoperated operation is crucial. While autonomous and assistive functions promise to reduce workload of operators and in some cases higher reliability, they can be brittle in real-world scenarios, where unexpected situations and failures can foil prior mission plans. In such cases, the capability of flexible switching between modes can significantly improve the reliability of the system, as the human supervisor has a toolbox of options at her disposal and can dynamically switch between them, adapting to the situation.

As the lowest level of interaction between operator and robot, teleoperation should always be available, communication permitting. Bypassing autonomous functions, this interaction mode shifts burden to the operator. Importantly, connectivity between robot and operator has to be sufficient in both directions; otherwise teleoperation becomes slow, unsafe, or even impossible.

With currently fielded robotic systems, these good communications conditions have to be met, as otherwise the robot becomes inoperable. Once autonomous assistance functionality is in more widespread use, the capability to fall back to teleoperation can be impeded by communication constraints, allowing for new applications. As teleoperation is the last fallback mode in case autonomous components fail, availability of it, no matter how limited, is important for overall reliability as it provides the ability to recover from unexpected scenarios.

In supervised autonomy mode, the operator provides task-level goals to the robot that are then followed autonomously using onboard systems. The operator observes actions, and generally provides permission to proceed at significant steps. This reduces reliance on connectivity and low latency communication, as the robotic system can follow task-level goals even when communication is intermittent; however, such an approach requires sophisticated sensing and planning capabilities for onboard systems. Using full autonomy, the human operator only specifies the mission and associated tasks and provides a start command, monitors data provided by the robot to maintain situation awareness, and either reacts to requests from the robot or switches to a lower autonomy mode on her own discretion. The clear advantage of full autonomy is that there is no need for communications as long as everything works well. The onboard autonomy system leverages the capabilities for task-solving used in the supervised autonomy mode and also makes use of planning capabilities, either directly or via task-level autonomy functionality.

It is crucial that when using a flexible level of interaction, the system stays in a well-defined state. For instance, when teleoperation commands are sent, autonomous control components have to be notified of the switch in interaction level as to not cause undefined behavior when commands both from the operator and autonomous executive are executed at the same time. This is discussed in Section 6.

3.3. Middleware

Developing a modular system requires a common communication framework, or middleware. To satisfy the research-level requirements on reproducibility and modularity, ROS is chosen as the underlying middleware. The nearly ubiquitous proliferation of ROS in the research community allows for using established standard interfaces and the ROS infrastructure allows for the development of highly modular code. With a large user

base, the barrier of entry for other researchers to use open source developments is much lower, which is highly advantageous considering the goal of advancing research for challenging applications, such as versatile disaster response robots.

While ROS provides solutions for many common robotics tasks, there are capabilities that received less attention by the research community than others. This is also true for disaster response using humanoid robots. The following areas were identified as requiring significant contributions to enable robot to perform complex disaster response tasks:

- Communication over constrained connections. ROS does not provide built-in facilities for communication over a degraded link.
- Footstep planning for locomotion in challenging terrain.
- Operator guided manipulation.

In the remainder of this work, components that address these shortcomings are detailed. It should be noted that the focus is not on low-level control of humanoid robots; it is assumed that basic control and locomotion capabilities are provided. The presented contributions leverage and interface with these basic control capabilities to achieve flexible high-level control.

3.4. Constrained Communications

While ROS provides transparent capability for distributing components over different machines by means of the network-based TCP/IP-based transport, communication constraints can impose additional challenges that make using ROS standard transports not feasible in some highly constrained scenarios. For those, specialized communication bridge tools need to be used, separating the ROS networks of the onboard and operator control station OCS sides. Such software has been developed by Team ViGIR during participation in the DRC (Kohlbrecher et al., 2015). In the sections that follow, we reference communications across the comms bridge; therefore, this section provides a basic description of the functionality.

The ROS middleware presumes a connection to a centralized communications manager (ROS master). Furthermore, communication with the ROS master requires a non-trivial amount of communication as modules come on line. As the degraded communications allowed by the DRC rules did not permit such unrestricted communications, Team ViGIR used a dual master setup between the OCS side and the robot *onboard* side.

The communication bridge system (comms bridge) developed by Team ViGIR uses mirrored components on either side that pass data across dedicated network channels. The components subscribe to messages on one side, compress them using custom encodings, send them across to the other side for uncompressing, and republish them as standard ROS messages. The messages use consistent names on each side to allow the system to also run transparently as a single ROS network without the comms bridge.

As the communication channels and compression are optimized for the specific rules of the DRC, and contain certain proprietary data for the Atlas robot, we have not open sourced the comms bridge and, therefore, it is not the focus of this paper. The general idea of a comms bridge is generally applicable, so that this paper describes several of the approaches to data communication over constrained links in the sections that follow.

4. PERCEPTION AND STATE ESTIMATION

The worldmodel system has to provide state estimation and situational awareness (SA) to the supervisor–robot team. To effectively leverage the human supervisor's cognitive and decision-making capabilities, a state estimate of both the internal and external state of the system has to be made available via the often constrained communication link between robot and operator. With current state of the art sensors often providing sensor data at rates in excess of 100 MB/s, this is both crucial and challenging.

The type of communication constraints under which the perception system has to work depends on used hardware and encountered scenario. They can include limited bandwidth, significant latency, and intermittent communication outages. The worldmodel system is designed to provide situational awareness and state estimation for the operator under all of these conditions. To achieve reliable and efficient manipulation with a remote operator in the loop, obtaining 3D geometry data is crucial. In the following sections, the approach and components for providing SA to both human supervisors and the robot are described.

4.1. Worldmodel Server

The worldmodel server[2] component preprocesses, collects, and aggregates sensor data and makes it available to both onboard and OCS system components. Leveraging established open source libraries, such as PCL (Rusu and Cousins, 2011) and octomap (Hornung et al., 2013), the worldmodel server allows queries of information about the environment with flexible level of detail and bandwidth consumption.

Three-dimensional sensing is provided by onboard sensors, providing point cloud data. A frequently used setup used here is a LIDAR and optionally a RGB-D type camera system. As RGB-D sensing generally has a smaller field of view, is sensitive to lighting conditions, and has less consistent measurement accuracy, LIDAR data are used as the default main source for creating a 3D geometry model of the environment onboard the robot. To achieve this, the planar scans of the LIDAR have to be preprocessed and aggregated, so full 3D point clouds can be generated from them. The following preprocessing steps are employed:

First, scan data are filtered for spurious measurements commonly called "mixed pixels" that occur at depth discontinuities (Tuley et al., 2005; Tang et al., 2007), using the shadow point filter available as a ROS package.

The filtered scan is then converted to a point cloud representation. During this process, motion of the LIDAR on the relative to the robot is considered and a high fidelity projection is employed, transforming every scan endpoint separately.

In a last step, parts belonging to the robot have to be filtered out of LIDAR data. To increase robustness against errors in kinematics calibration, a specialized robot geometry model uses

[2] https://github.com/team-vigir/vigir_perception/tree/master/vigir_worldmodel_server

simplified and enlarged collision geometries for self-filtering purposes.

LIDAR scans are saved in a ring buffer along with snapshots of coordinate frames used within the system. By employing this method, aggregate point clouds relative to different coordinate frames can be provided on request. A ROS API allows querying the world model via both ROS topics or services and flexibly retrieving region of interest point cloud or octomap data relative to different coordinate frames. This capability can be employed by both onboard and OCS system components.

The primary onboard 3D geometry model is created using octomap, a volumetric, probabilistic approach using an octree as a back-end. Using this approach, the environment representation maintained onboard can be updated efficiently and in a probabilistically sound way. Even in case of changes in the environment or drift in state estimation, the environment model is updated accordingly and maintains a useful representation.

The octomap environment model provides the main geometry representation and is used for multiple purposes. Using ray casting, distances to geometry can easily be determined. This feature can be used from within the OCS to perform ray cast distance queries against onboard geometry. In this case, only the ray cast information has to be transmitted to the robot and the distance information is transmitted back, utilizing only very low bandwidth.

The capability to request ROI data of the environment model allows to transfer small ROI geometry over the constrained connection on supervisor demand and also makes geometry available to other modules on request, like the footstep planning system. Similarly, it is possible to request 2D grid map slices of the octomap representation, aggregating 3D data into a 2D grid map. Using compression during transmission, this representation is very compact and often sufficient for supervisors to gain SA.

4.2. LIDAR Data Compression

In case of intermittent communication, the approach for querying the onboard worldmodel for data from the OCS as described in the previous section can fail, as no data can be transmitted in periods of communication loss. Instead, it is desirable to transmit all geometry information available onboard to the OCS side as long as a communication window is available. A mirror of the worldmodel can then be queried on the OCS side instead of relying on a connection to the remote onboard worldmodel. The approach described in the following is available online.[3]

In case of intermittent communication between supervisors and robot, two instances of the worldmodel server are used: one for the onboard/robot side and one for the OCS side. As direct transmission of point cloud data is error prone when experiencing packet loss, additional processing on LIDAR data is performed to make each packet compact enough to fit within a standard 1500-B UDP packet and compress it as to be able to transmit a maximum of data during a communications burst.

[3]https://github.com/team-vigir/vigir_manipulation_planning/tree/master/vigir_lidar_octomap_updater

For compression of LIDAR data, the GIS research community developed solutions for large-scale airborne LIDAR datasets (Isenburg, 2013), but these significantly differ in structure from those by small planar scanners. For this reason, an approach leveraging the special structure of data provided by planar scanners is presented here.

Direct transmission of point cloud data generated onboard the robot would cause prohibitive bandwidth cost as a point cloud representation with at least three floating point values for each Cartesian point is not a compact one. For this reason, the natural and compact representation of a laser scan as an array of range values is leveraged and used instead. To fully reconstruct the 3D geometry captured by a single scan, a high fidelity projection of the scan has to be performed, however, taking into account motion of the LIDAR mirror during the data capture process. If this motion is not considered, scan data show visible skew and ghosting (double walls) once it gets converted to a point cloud representation. The following approach is thus utilized:

- Perform a 3D high fidelity projection onboard the robot and perform self-filtering. The onboard octomap and worldmodel are updated simultaneously.
- Compress the scan data by writing the range values to a 2-Byte array representing millimeters and also encoding self-filtering information. Threshold and map intensity information to a single Byte.
- Add information about the scanner transform in world frame, one transform for the start of the scan and one for the end. This information allows performing a high fidelity projection of the scan after unpacking on the OCS side.
- Split the compressed scan into chunks that are small enough to be compressible to <1500 B. A schematic of this approach is available in **Figure 2**. By using this approach, each compressed scan packet is a self-contained unit and can be unpacked and used on the receiver side without the need for packet reassembly.

On the OCS side, the compression process is reversed, and resulting scan data are used to update the OCS world model. The size of a LaserScan message is dominated by the range and intensity fields. A typical Hokuyo LIDAR, for instance, provides 1080 measurements per scan. For compression, floating point range values in meters are converted to millimeters and stored in an unsigned 16 bit number. Self-filtering of robot parts from LIDAR data requires knowledge of the whole transform tree of the robot and, thus, has to be performed on the onboard side if transmission of high bandwidth transform data to the OCS side is to be avoided. Per default, self-filtering is, thus, performed onboard and compressed laser scan data are annotated with a single bit per scan point, indicating if the self-filter determined that it belongs to the robot or objects attached to the robot.

Intensity data are converted from a floating point intensity to an unsigned 8 bit number. Here, a loss in fidelity is acceptable as intensity is mainly used for visualization and a range of 2^8 values is sufficient for presentation to the human supervisors.

Table 1 shows the different scan representation and their relative size. In **Figure 3**, the setup using one worldmodel instance each on the onboard and OCS sides is visualized. The

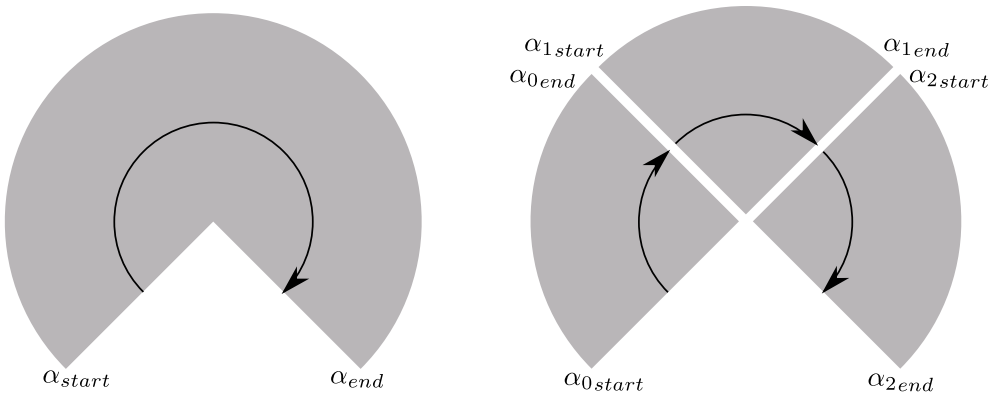


FIGURE 2 | Splitting LIDAR scans for compression. A schematic view from the top is shown here and the rotation direction indicated by an arrow. A LIDAR scan can be described by the start angle α_{star} and end angle α_{end}. With a known angular resolution, scan points can be projected. To achieve a small packet size, the scan is split and intermediate start and end angles computed.

TABLE 1 | Different LIDAR scan representations and the associated data size.

Data	LaserScan (Bytes)	LocalizedLaserScan (Bytes)	Compressed (Bytes)
Header	≥16	–	–
Metadata	7 × 4	–	–
Ranges	4 × 1080	2 × 1080	$<\frac{1}{3}\times2\times1080$
Intensities	4 × 1080	1080	$<\frac{1}{3}\times2\times1080$
Total	8684	3240	<1080

As shown, the compressed size results in a packet size below the 1500 B of a standard size UDP packet.

synchronization is performed using the previously described compressed scan transmission mechanism.

4.3. Sensor Data Processing for Situation Awareness

To provide the supervisor(s) with the necessary SA for complex manipulation tasks, not only geometry but also image and texture data are crucial. In this section, components allowing for the processing of sensor data to achieve suitable representations and visualizations for obtaining supervisor SA are discussed.

4.3.1. Region of Interest Image Data

As images are readily compressible using standard compression methods, providing such data to the operator is often possible and can be feasible even when bandwidth is constrained. Often, only a limited region of interest in the full image is required. Examples are visually inspecting the quality of a grasp or the accuracy of end-effector positioning. To provide this capability, the operator can request full image and region of interest independently, making it possible to show coarse resolution full images, but high-resolution regions of interest. To minimize communication requirements, an optional video frame-rate is part of the request and images can be sent at a fixed rate without need for bi-directional communication.

4.3.2. Mesh Generation

To provide a high fidelity visualization for 3D geometry data, an infrastructure for generating meshes from both LIDAR point clouds and camera or LIDAR-based depth images was developed.[4] Compared to plain point cloud visualization, this approach allows for a clear view of geometry and texturing of mesh surfaces, which allows for easier scene understanding by human supervisors.

Figure 4 shows a schematic of the mesh generation data flow. As indicated by the light blue OR gates, the mesh generation process can be based on different kinds of input data. Based on depth images, a mesh can be generated using a FastMesh (Holz and Behnke, 2013) approach. The depth image can either be provided by a RGB-D type camera or it can be generated from LIDAR data. In the latter case, data have to be aggregated over time, however. Instead of depth images, LIDAR-based point clouds can also be used for mesh generation; in this case, the mesh is generated from LIDAR point cloud data directly. This approach does not have the restricted field of view of the depth image-based one.

An example of generating meshes based on stereo camera RGB and depth data is shown in **Figure 5**. Three novel rendered viewpoints are shown, demonstrating how the approach combines the fidelity of image data with 3D geometry.

4.3.3. Fisheye Camera

The Atlas robot could not rotate the Multisense sensor head around the yaw axis, greatly limiting the field of view of the main sensor system. With early versions of the Atlas robot, this was a severe issue, as the volume of good manipulability for the arms was outside the Multisense sensor field of view. To remedy this issue, a system for rectification the Fisheye lenses of the fisheye cameras was developed.[5] Using a ROS-integrated version of the OCamLib library (Scaramuzza and Siegwart, 2007), the fisheye

[4]https://github.com/team-vigir/vigir_perception/tree/master/vigir_point_cloud_proc
[5]https://github.com/team-vigir/vigir_wide_angle_image_proc

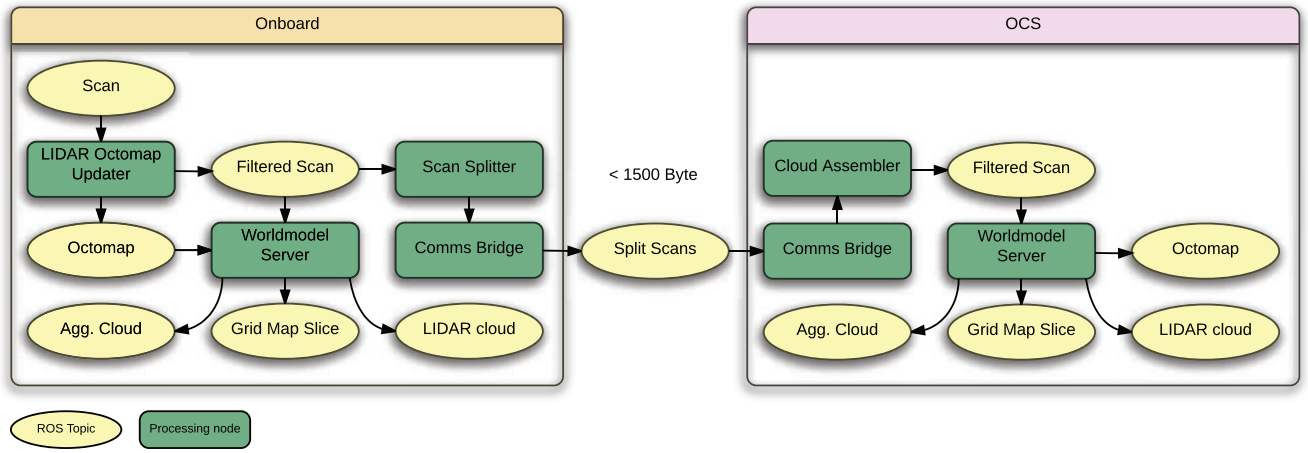


FIGURE 3 | **Overview of the Worldmodel server setup**. Worldmodel information is synchronized via compressed LIDAR data. One instance of the worldmodel server is running on the onboard and OCS side each.

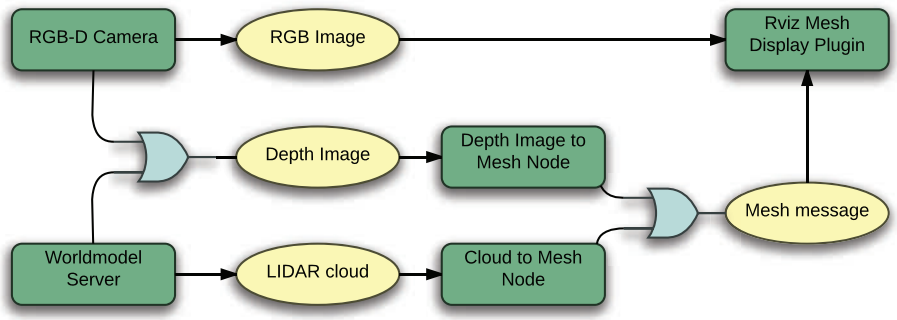


FIGURE 4 | **Options for generating a mesh representation of the environment**. The RGB camera image gets texture mapped on a mesh generated from LIDAR or depth image data. The depth image is either provided by a camera directly or can be generated from aggregated LIDAR point cloud data.

distortion is calibrated. This allows generating novel rectified views from fisheye images not exhibiting severe distortion that otherwise makes judging of spatial relations difficult for operators.

Recomputing the rectification online, the system can track arbitrary frames on the robot or in the environment. It is, thus, possible to create a virtual pinhole camera that, for instance, tracks an end effector of the robot.

5. PLANNING

For manipulation, motions to move manipulators into desired configurations for grasping or other tasks need to be generated. As it can reduce operator workload considerably, a crucial capability is automated collision avoidance, both considering self-collisions of the robot (e.g., arm coming in contact with torso) and collision of robot parts with the environment. When performing manipulation in contact with the environment, motion must not lead to unplanned high internal forces acting on the robot, as these can quickly lead to damage to the robot, especially if it loses balance as a result. While force or admittance control approaches can reduce this risk, they are often difficult to implement due to limited force sensing and control performance on real systems. Preventing unintended contact in the first place thus serves as a risk reduction measure.

As high latency limits the usefulness of otherwise promising approaches for teleoperation of end effectors that rely on real-time feedback (Leeper et al., 2013), direct control is not feasible. Instead, the supervisor(s) specify goal joint configurations or Cartesian goal poses and requests robot onboard systems to reach them.

The system described in this section is available as open source.[6]

5.1. Previewing Manipulation

As described in Chapter 4, the worldmodel server provides the supervisor(s) with the necessary tools to achieve situational awareness of the environment state in a variety of different bandwidth

[6]https://github.com/team-vigir/vigir_manipulation_planning

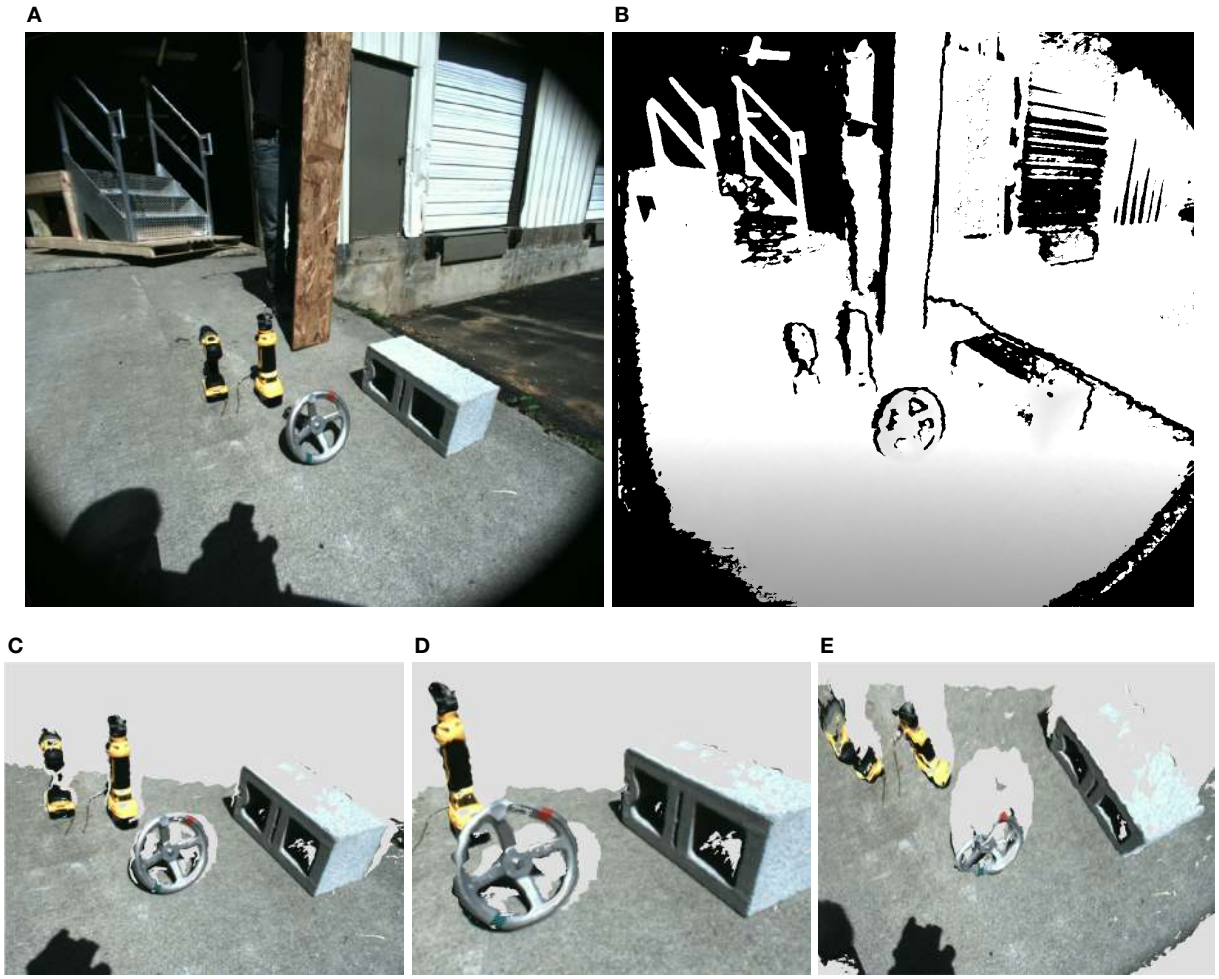


FIGURE 5 | **Rendering novel views based on textured mesh data**. **(A)** RGB image, **(B)** depth image, and **(C–E)** novel view points rendered based on applying texture to a mesh generated from the depth image.

conditions. To be able to reliably perform manipulation, an approach for predictive visualization of how the robot interacts and likely will interact with the environment in the future is required.

With the high number of DOF of humanoid systems and the challenges of balance control, judging the reachability and manipulability of the robot for a given task can be much more difficult than for more conventional robots. While inverse reachability approaches show promising results in the literature (Vahrenkamp et al., 2013; Burget and Bennewitz, 2015), they do not consider constraints beyond kinematics and self collisions. Such additional constraints are for instance sensor visibility constraints or control-related constraints due to appendage control performing better in some configurations than others. It would be possible to incorporate those into inverse reachability analysis, but this remains a largely unsolved topic for research at this time.

To provide an intuitive interface to human operators, the so called "ghost robot" is used. This is a interactive puppet robot that can be used to predictively simulate the kinematics of manipulation tasks. The state of the ghost robot can be modified in the user interface without effects on the real robotic system. Once the supervisor is satisfied with ghost robot based planning, planning and motion requests can be generated based on the ghost robot state using variety of different options detailed below.

The ghost robot is an essential tool for teleoperation and supervised autonomy and is used for the full range of manipulation and locomotion control. While it remains possible to move the robot by sending joint angles directly, this is discouraged due to the high risk involved in such actions.

As shown in **Figure 6** the ghost robot state can be modified based via a ROS API that allows for the following options:

- Joint angles. The ghost robot can externally be set to be in a desired joint angle configuration. Importantly, a subset of joints can be used here.
- Cartesian goals for end effectors. The ghost robot end effectors can be moved to Cartesian goals. In this case, an IK solver is used internally to solve for the joint positions.

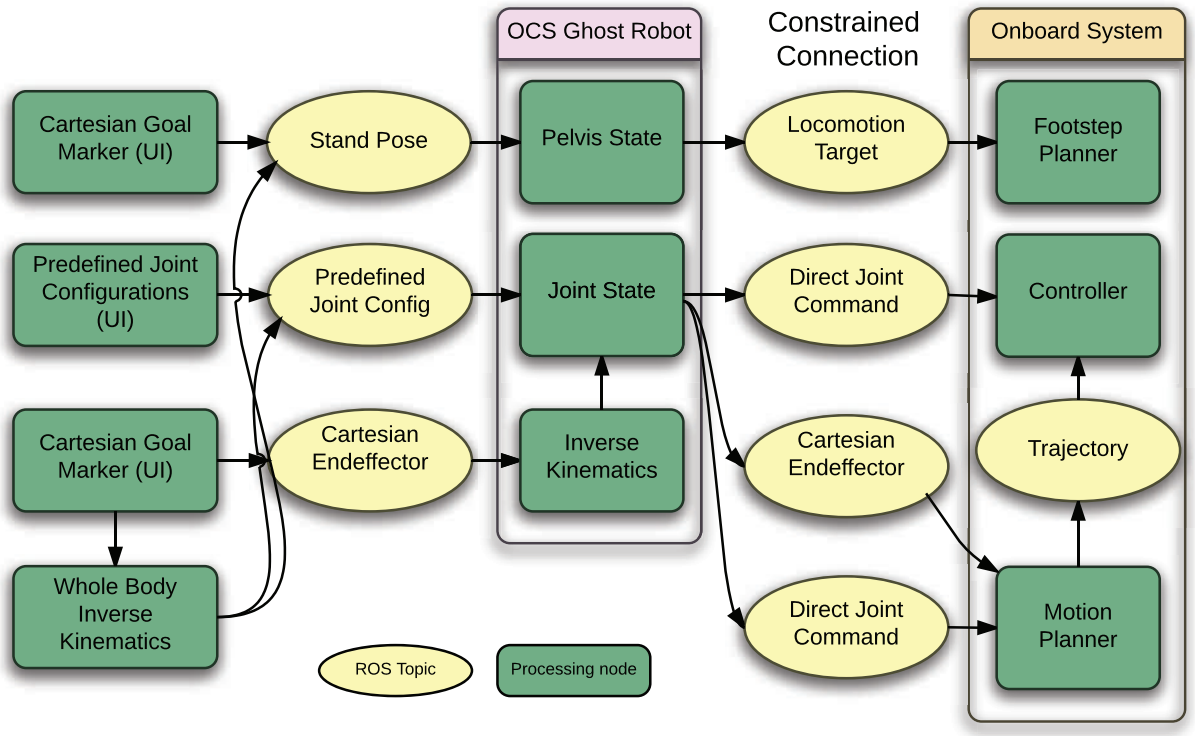


FIGURE 6 | Schematic showing inputs and outputs for the ghost robot that is used for pre-planning manipulation tasks.

- Cartesian goals for the robot pose. The ghost robot root frame (frequently the pelvis in case of a humanoid) can be moved to a desired Cartesian goal pose.

If a whole body IK solver is used externally, the ghost can also be set to a desired state by jointly using the joint angle and Cartesian robot pose interfaces simultaneously.

Based on the ghost robot state, the following types of commands can be generated to be executed on the real robot:

- A goal pose for the footstep planner based on the ghost robot pelvis position in the global frame.
- The joint configuration of one of the ghost's appendage groups can be sent to the onboard controller as a motion target.
- The same joint configuration can be sent to the onboard motion planner, which then generates a collision free trajectory for it.
- The Cartesian end-effector pose can be sent to the onboard motion planner, which then generates a collision free trajectory to reach it.

It should be noted that the last two options are not equivalent on most humanoid robots, as balance control generally will shift the pelvis pose when the arm configuration of the robot changes, resulting in an offset for the first option.

Figure 7 shows use of the ghost robot during the DRC Trials. It is used for determining a stand pose for the robot on the left and for planning manipulation of a valve on the right.

5.2. Planning System Details

Manipulation for disaster response often incorporates prolonged contact situations, for instance when opening a door or turning a valve. Especially in disaster response applications, cluttered environments present a challenge, as obstacles have to be avoided during motion planning.

The manipulation planning system is based on the MoveIt![7] (Chitta et al., 2012) motion planning framework available for ROS. This framework provides a powerful API for planning and different planning components.

The system enables planning to goal joint configurations and to goal end-effector poses and thus is directly compatible with the ghost robot approach described in the previous section. Two planning modes are available: the default mode is unconstrained planning, with joints free to move between the start and goal joint configurations. The other mode is a constrained motion mode. Here, motion is constrained to follow a Cartesian path between the start and goal end-effector pose. In this case, waypoints are generated based on linear interpolation between start and goal position and orientations for waypoints are generated using slerp (Shoemake, 1985) between start and goal end-effector quaternions. More complex constrained motions such as circular motion for turning a valve are generated by concatenating multiple short linearly interpolated Cartesian paths.

[7] http://moveit.ros.org/

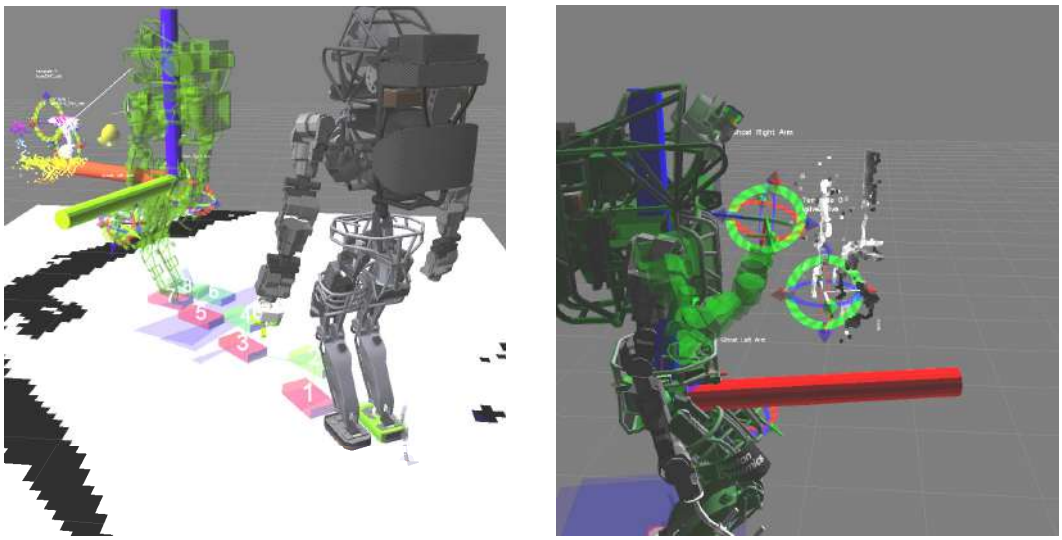


FIGURE 7 | Two examples of using the ghost robot for previewing manipulation. **(A)** The ghost robot is used to preview the stand pose before performing manipulation. **(B)** Previewing arm motion during the valve task at the DRC Trials. The solid robot is the current true state, while the translucent green one is the ghost robot.

For obstacle avoidance, the volumetric octomap representation as described in Chapter 4 is used. As contact with the environment is required in many manipulation tasks, collision checking between end effectors and the environment can optionally be disabled by the supervisor(s). For instance, collision avoidance is needed to safely bring the robot hand into a position to pick up a drill. In order to grasp the drill, collisions between the palm and fingers of the hand and the drill handle must be allowed, however.

In challenging conditions, noise in sensor data that lead to geometric artifacts, preventing successful planning due to spurious collisions cannot be ruled out completely. To cope with such situations, collision checking against the octomap environment model can also be disabled for the complete robot geometry; in this case, the ghost robot changes color to warn the operator.

For motion planning, the number of joints (DOF) to use can be selected by the supervisor(s). For instance, on Atlas, planning can be performed using either 7 DOF with the arms only, or by including the torso joints and using up to 10 DOF. As the 10 DOF planning mode tends to result in higher control error or oscillation in some joint configurations, the operator can lock a selection of torso joints to restrict the planning space. The same approach can be used on other robotic systems transparently.

To allow for safety and robustness, the ability to select the desired trajectory execution speed with every planning request was introduced. Using standard MoveIt! functionality, trajectories were previously time parameterized according to the velocity limits supplied in the URDF robot model. This approach turned out to be not flexible enough for challenging manipulation in contact that might require moving appendages slow for safety.

5.3. Planning Interface

To implement the described manipulation back-end, the MoveIt! API was used and DRC-specific capabilities were implemented in a separate *move_group* capability plugin. This offers the advantage of retaining standard MoveIt! library planning features, while simultaneously allowing the development of extended capabilities specific for disaster response manipulation tasks.

As shown in **Figure 8**, the planning system is exposed via a ROS Action server interface and, thus, provides feedback about the planning and plan execution process. The Action interface is the sole entry point for requesting and executing motion plans and (in order of increasing autonomy) used for teleoperation, affordance-based manipulation planning and for motion plan requests generated by the behavior executive. For teleoperation, an onboard node translates compressed and compact motion requests by the operator into an Action request that then gets forwarded to the planning system.

5.4. Supervised and Autonomous Control

The described planning system offers a powerful API that can be used to plan for complex manipulation tasks. In the preceding sections, both the teleoperation interface and the planning back-end are described.

To achieve both task-level supervised operation and autonomous control, two additional software components for manipulation use the described planning system as a back-end for performing manipulation: an object template framework and the FlexBE behavior engine.

Figure 9 shows an overview of how the different system components interact to achieve the full range of capability from teleoperation to full autonomy in interaction with one or more human supervisors.

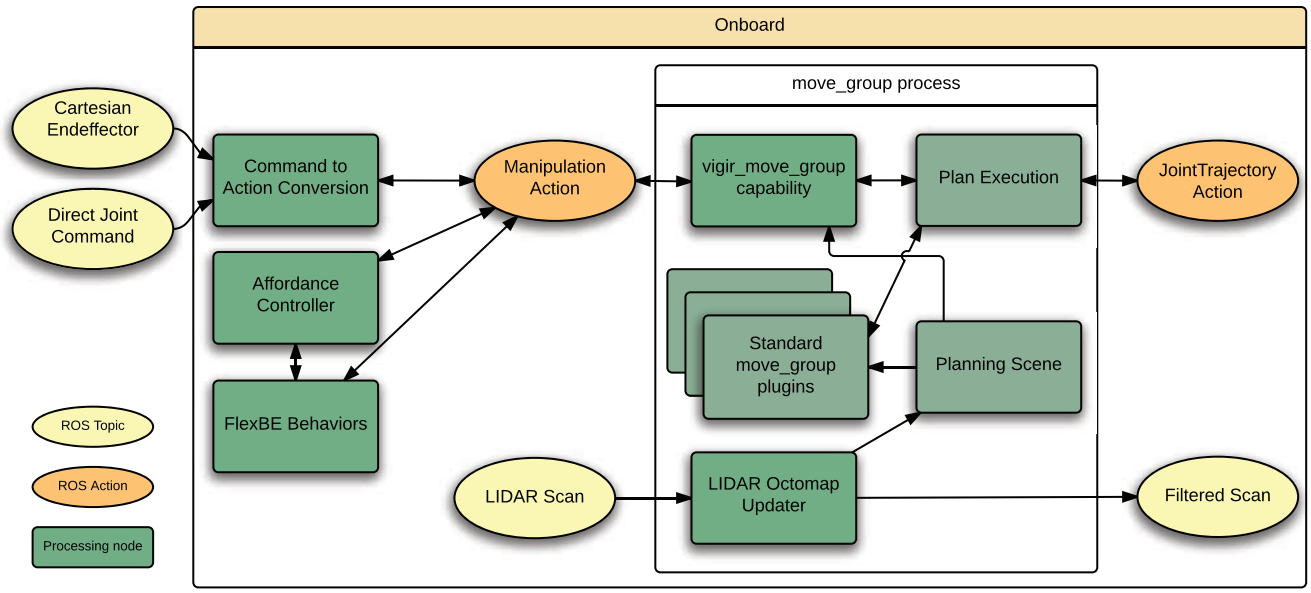


FIGURE 8 | **Overview of the planning back-end**. Both the planning interface and the LIDAR octomap updater are loaded into the standard MoveIt! move_group process as plugins. Using this approach, existing functionality provided by MoveIt! is kept, but extended.

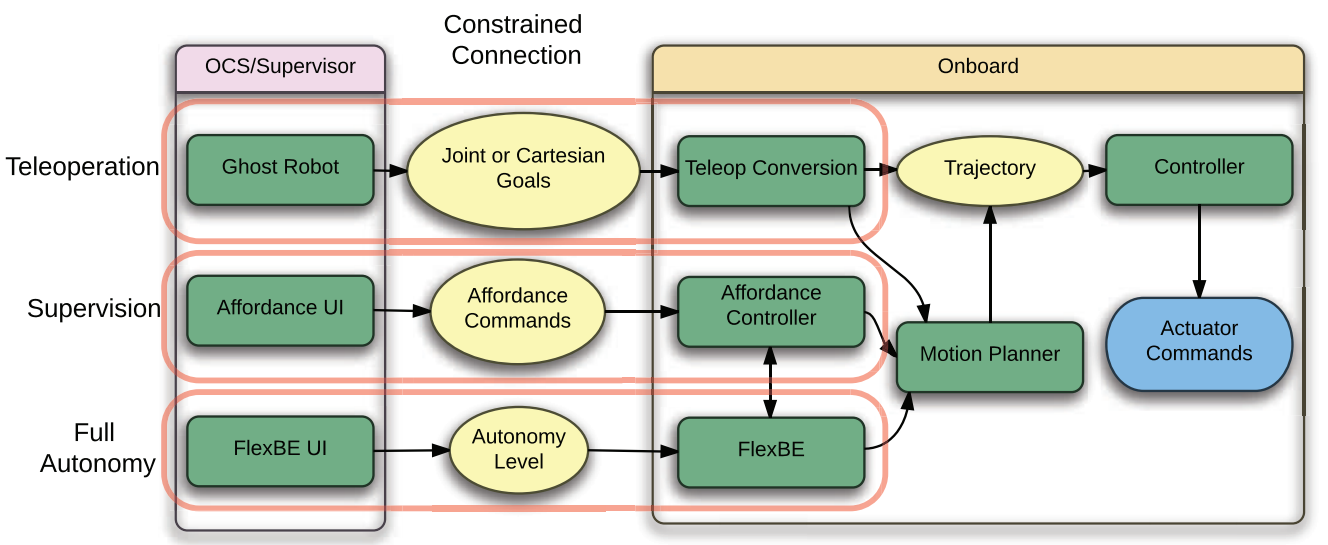


FIGURE 9 | **Supporting multiple levels of interaction for manipulation capable avatar robots**.

5.4.1. Object Templates

Instead of directly controlling appendages, the object template-based approach for manipulation (Romay et al., 2014, 2015) uses models of objects to be manipulated, the so-called object templates. These are placed by the human operator in the virtual environment model where 3D sensor data of the environment are visualized and serve as references to achieve manipulation task at a higher level of abstraction.

Object Templates contain relevant information about the objects they represent, such as physical and abstract information. With this, the operator can provide the robot with potential standing poses, grasp poses, usable parts of the object, and manipulation skills or affordances (Gibson, 1977) to manipulate the object. With each template offering a set of affordances, motion can be specified by the operator on the affordance level. A door opening motion can, for instance, be commanded by using the "open"

affordance defined for the door handle and the "push" affordance defined by the door hinge.

The information that object templates provide can also be abstracted by higher system layers, such as autonomous behaviors.

5.4.2. Automatic Behavior Control

For autonomous execution of complex manipulation and locomotion tasks, the Flexible Behavior Engine (FlexBE) has been developed during the DRC. A detailed overview is provided in Section 6. The object template system is also used within FlexBE to represent manipulatable objects. The behavior executable can, thus, take over responsibility for coordinating complex tasks from remote human supervisors where applicable.

5.5. Whole-Body Planning

While the developed motion planning system performs well for many manipulation tasks requiring only upper body motion, sampling-based planning falls short for planning whole-body motions that require the consideration of balance constraints. To also support this, the optimization-based planning approach available as part of the Drake framework (Fallon et al., 2015) has been integrated with the Team ViGIR planning system. Planning using Drake can transparently be used by specifying the plan request. Drake has also been integrated with the ghost robot on the OCS side and the operator can use Drake-based whole-body inverse kinematics to pre-plan tasks, such as reaching toward the ground for picking up objects.

5.6. Footstep Planning

A key challenge of the DRC was enabling the robot be able to tackle locomotion tasks, such as the traversal of sloped stairs, ramps, and rubble. While Team ViGIR depended on a manufacturer supplied footstep controller for stepping and stability, the specification of footstep placements remained a significant challenge; Team ViGIR extended an existing planner for 2D environments to handle this more complex 3D terrain.

The footstep planner has to satisfy two main requirements: the planner has to solve the navigation problem of finding the shortest safe path in a given environment. Second, it has to generate a feasible sequence of footstep placements, which can be executed by the robot with minimal risk of failure. Additionally, the DRC competition discouraged the use of slow footstep planning approaches due to mission time limits. Here, operator performance highly depends on the speed and performance of the used footstep planning system, so planning efficiency becomes important. It is desirable that the planning system provides all parameters of the walking controller for each step, so that the complex low-level walking controller interface is completely hidden from the operator to reduce the chance of operator error. This kind of footstep planning systems has not been applied to human-size real robots in complex terrain scenarios, such as the DRC before.

5.6.1. Overview

Our footstep planning approach satisfies the requirements mentioned above and requires the operator to only provide a goal pose to start planning. During the DRC competition, we have introduced the first search-based footstep planner capable of generating sequences of footstep placements in full 3D under planning time constraints and using an environment model based on online sensor data (Stumpf et al., 2014). The planner solves the navigation problem of finding shortest and collision-free paths in difficult terrain scenarios while simultaneously computing footstep placements appropriate for a given walking controller. A 3D terrain generator allows to generate terrain models for the footstep planning system online. It is able to efficiently compute the full 6 DoF foot pose for foot placements based on 3D scans of the environment. In addition, a novel collision check strategy based on ground contact estimation allows the planner to consider overhanging steps, significantly enhancing performance in rough terrain scenarios.

The described approach has been successfully applied to three different biped humanoid robots during the DRC Finals. As the only team at the DRC Trials, we demonstrated that our approach is able to generate suitable footstep plans over entire obstacles that had been executed without interruptions (see **Figure 10**).

5.6.2. Terrain Modeling

Planning in difficult terrain scenarios needs a suitable 3D terrain model that can efficiently be generated and utilized by the footstep planner. Therefore, a terrain model generator[8] was implemented which analogously to the worldmodel server (see section 4.1)

[8] see https://github.com/team-vigir/vigir_terrain_classifier

FIGURE 10 | Atlas traversing chevron hurdles based on computed footstep plan.

accumulates all incoming LIDAR scans given as point clouds. All data are stored in a discrete octree to reduce the amount of needed memory and enable efficient data fusion.

For each point in an incoming point cloud, a normal estimation[9] with respect to the point neighborhood is immediately performed. Afterwards, the octree is updated with this new information. Each node within the octree, thus, provides the 3D position of the scan point and the estimated surface normal. Through the sparse laser scan updates of the spinning LIDAR, this operation can be performed in real-time on a single core of a CPU. In general, performing this operation with stereo vision or RGB-D systems is possible too, but needs further investigation as they generate more noisy data.

The described approach allows to run real-time terrain model generation on a robotic system as long as it is capable of providing point clouds given in an absolute world frame.

5.6.3. Footstep Planning Framework

The main objective is to provide an integrated footstep planning framework that may be deployed easily into an existing ROS setup. Providing a framework, the planner has to be expandable for new features but closed for modifications. Any user of the framework should only have to implement and extend robot-specific elements to use the planning system instead of developing a modified version of an existing planner or even starting from scratch each time. Already implemented and, thus, proven algorithms are kept untouched, reducing the likelihood of errors and saving implementation effort. Although the framework must generalize well, it has to be able to solve difficult terrain task problems and utilize the versatile locomotion capabilities of robot-specific walking controllers.

In order to meet this objective, parameter (*vigir_generic_params*)[10] and plugin (*vigir_pluginlib*)[11] management systems have been implemented.

5.6.3.1. Parameter System

In real-world applications, different terrain scenarios need to be tackled (e.g., flat surface, stairs or sloped terrain). The footstep planner can perform best if an appropriate set of parameters is defined for each kind of terrain scenario. This allows the operator to easily switch between different planning behaviors. Furthermore, it is desirable to be able to modify a parameter set if the situation requires it. In general, these requirements can be solved using the available ROS message infrastructure. Frameworks, such as the presented footstep planner, however, are supposed to be extended with new features. The structure of parameter sets may vary during runtime that is in conflict to ROS messages requiring a static structure. A simple solution would be separate configuration files and well as user interfaces for each plugin. Due to high maintenance effort this, however, is undesirable.

5.6.3.2. Plugin System

The *vigir_pluginlib* provides the capability to manage versatile plugins that can be also used outside of the footstep planning domain. The approach is based on *pluginlib* that already allows for dynamically loading plugins using the ROS build infrastructure. We have extended the package into a semantic plugin management system. The practical implementation consists of two parts: the plugin base class and the plugin manager.

5.6.3.3. Framework Overview

The plugin and parameter management systems form the infrastructure base of the footstep planning framework.[12,13,14] The footstep planner pipeline has multiple injection points where a user might want to customize the behavior of the planner. For each of those, a semantic base class has been introduced as follows:

- *CollisionCheckPlugin*: basic collision check for a given state or transition,
- *CollisionCheckGridMapPlugin*: specialized CollisionCheckPlugin for occupancy grid maps,
- *HeuristicPlugin*: computes heuristic value from current state to goal state,
- *PostProcessPlugin*: allows performing additional computation after each step or step plan has been computed,
- *ReachabilityPlugin*: check if transition between two states is valid,
- *StepCostEstimatorPlugin*: estimates cost and risk for given transition,
- *StepPlanMsgPlugin* (unique): marshaling interface for robot specific data, and
- *TerrainModelPlugin* (unique): provides 3D model of environment.

The last two semantic base classes are defined to be unique; only a single instance might be running instance at a time. **Figure 11** shows the use of plugins within the planner pipeline. For a quick deployment of the framework, concrete plugin implementations for common cases already exist for all semantic-based classes.

A major goal is maintaining footstep planner efficiency. Therefore, the computational overhead of the plugin system must be kept to a minimum. It obviously is inefficient to reload needed plugins in each single iteration of the planning process. For this reason, the planner loads all plugins only once and sets their parameters once before starting planning. Additionally, a mutex locks all critical callback functions of the planning system. The footstep planner is, thus, protected against any changes of the plugin as well as parameter manager during the planning process.

Advanced walking controllers usually need very specific data to allow for performing complex locomotion tasks. For instance, these data could be intermediate trajectory points of the foot or the convex hull of expected ground contact. The framework has been designed to be able to provide this capability. The plugin system allows to inject additional computation needed by the

[9] http://pointclouds.org/documentation/tutorials/normal_estimation.php
[10] https://github.com/team-vigir/vigir_generic_params
[11] https://github.com/team-vigir/vigir_pluginlib
[12] https://github.com/team-vigir/vigir_footstep_planning_msgs
[13] https://github.com/team-vigir/vigir_footstep_planning_basics
[14] https://github.com/team-vigir/vigir_footstep_planning_core

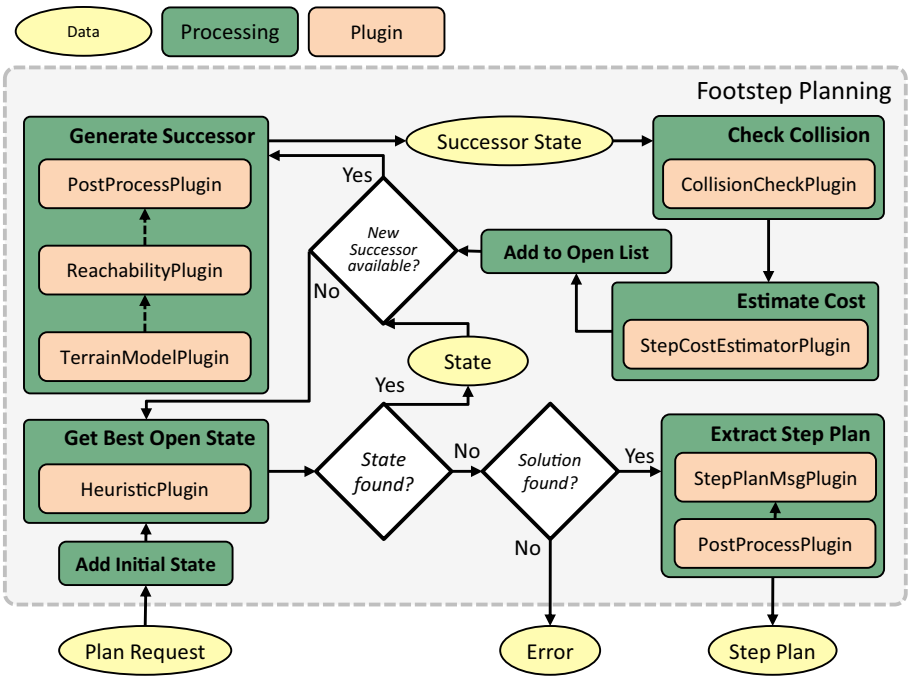


FIGURE 11 | **Plugins embedded into Footstep Planning Pipeline**.

walking controller. Analogously to the parameter management system, all custom data can be carried as a byte stream within regular step plan messages. Marshaling algorithms already available for basic data types can be applied here as well. Marshaling for complex data types has to be implemented as a customized StepPlanMsgPlugin. The framework is, thus, able to pack all custom data into the generic step plan message and send it to the hardware adapter, where it gets unpacked and forwarded to the walking controller. The framework, thus, supports any kind of walking controller via the plugin system without required modifications to the framework code base.

5.6.4. Interactive Footstep Planning

During the DRC Trials, the inability to refine generated footstep plans was identified as a shortcoming. Even though the planner is able to generate feasible plans, the possibility that the resulting plan contains undesirable steps due to noisy sensor data remains. In this case, the operator previously had to request a new step plan in the hope to get a better result. For this reason, the footstep planning system was extended to provide multiple services to manage footstep plans. These services can be used by user interfaces to enable interactive footstep planning, allowing full human in the loop planning. This mode allows for plan stitching, plan revalidation, and editing single steps with assistance of the footstep planner. The operator is able to quickly adjust single steps, while the planner will automatically update the 3D position of the new foot pose if enabled and provides immediate feedback if the modified step sequence is still feasible for the walking controller.

6. BEHAVIOR EXECUTIVE

Combination of multiple, complex software components on a high level is an often underestimated issue when composing robot systems. Existing solutions are often very application specific and require expert developers for implementing mission specifications. Thus, in order to provide a suitable task-level layer of control for full or assisted robot autonomy, the behavior engine FlexBE[15] (Schillinger, 2015) has been developed. It is based on SMACH (Bohren and Cousins, 2010) and extends it by several helpful capabilities in order to facilitate both development and execution of high-level behaviors. Furthermore, FlexBE provides an extensive user interface, enabling even non-expert users to compose and execute mission specifications within short time frames. During runtime, execution can be monitored and controlled remotely and the robot autonomy level can be flexibly adjusted.

6.1. Component Interface

FlexBE (standing for "flexible behavior engine") adapts the concept of hierarchical state machines similar to the implementation in SMACH. Each state corresponds to an action executed by the robot and transitions reflect the possible outcomes of these actions while data gathered during runtime can be passed between states. This approach enables to focus on the internal state of the robot (i.e., the current state of action execution). Knowledge about the external environment is only considered

[15]http://flexbe.github.io

implicitly when designing a behavior, but is not needed to be exactly known during execution. Especially in disaster response, where the environment cannot be precisely modeled and events are typically the result of own actions, this concept helps to effectively define high-level behaviors.

Each action state is given by a class definition implementing a specific interface. Depending on the situation, a state returns one of its outcomes, leading to a transition in the respective containing state machine. Furthermore, states can declare input and output keys for sharing common data during runtime. As depicted by the concept overview in **Figure 12**, these state implementations form the atomic building blocks from which behaviors are composed. Each action state refers to a well-defined and encapsulated high-level capability of the robot, for example, closing fingers, planning footsteps, or executing a trajectory.

6.2. Behavior Development

In order to support the user in composing state instantiations to a complete behavior, FlexBE provides a comprehensive graphical user interface, including a state machine editor. **Figure 13** shows a screenshot of this editor displaying a behavior as used for the DRC task of turning the valve. Yellow boxes denote states, white boxes are embedded state machines, and the boxes in pink refer to other complete stand-alone behaviors, which can be embedded as well. Transitions are given by arrows between the states, where their labels refer to the outcomes of the outgoing states, i.e., under which condition the respective transition is taken. Their color corresponds to the required level of autonomy, which can be selected by the user.

The editor also provides a set of useful tools for making sure that states are composed in the correct way. For example, a dataflow graph can be displayed in order to check how data will be accessed and potentially modified by the different robot capabilities during runtime. More importantly, each time a behavior is saved, or on demand, consistency of a behavior is validated. This includes checks such as that each outcome corresponds to a well-defined transition and no runtime data are potentially accessed before being written. FlexBE allows to make behavior modifications even during runtime and for updating behaviors while they are executed. With the static check functionality, the state machine editor ensures that such modifications do not lead to runtime failure.

Experience from designing task-level behaviors for the DRC showed that the usage of these concepts and the related tools not only helped facilitating the definition of complex behaviors since state machines could be modeled graphically instead

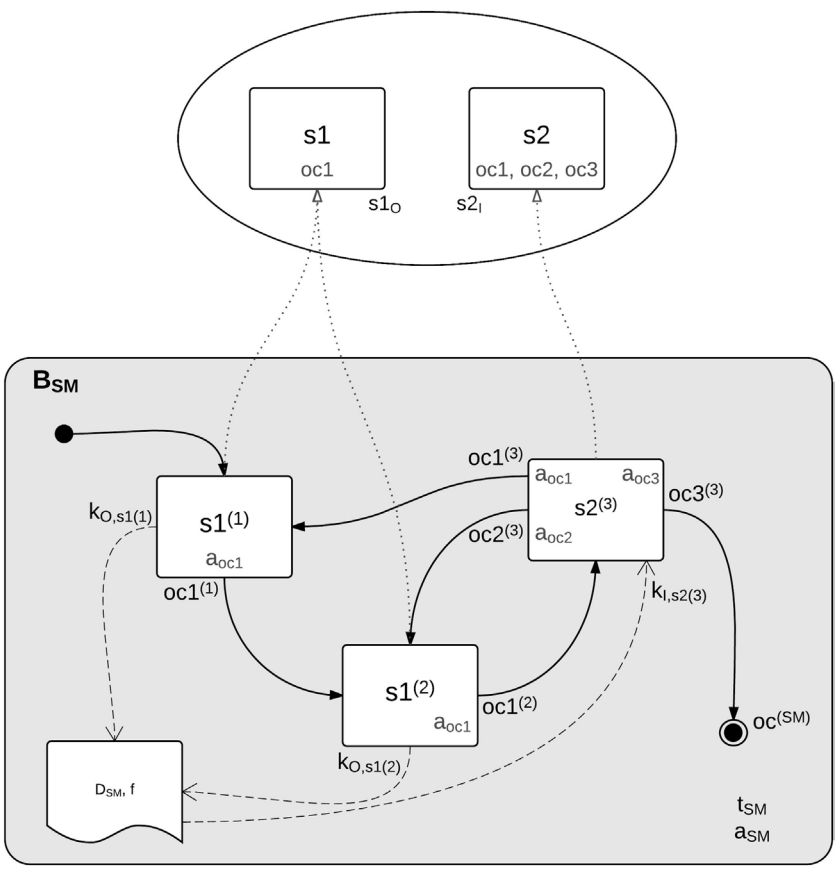


FIGURE 12 | Conceptual overview of a behavior composition. Each behavior is defined by a hierarchical state machine that instantiates and connects (lower part) re-usable state implementations (upper part).

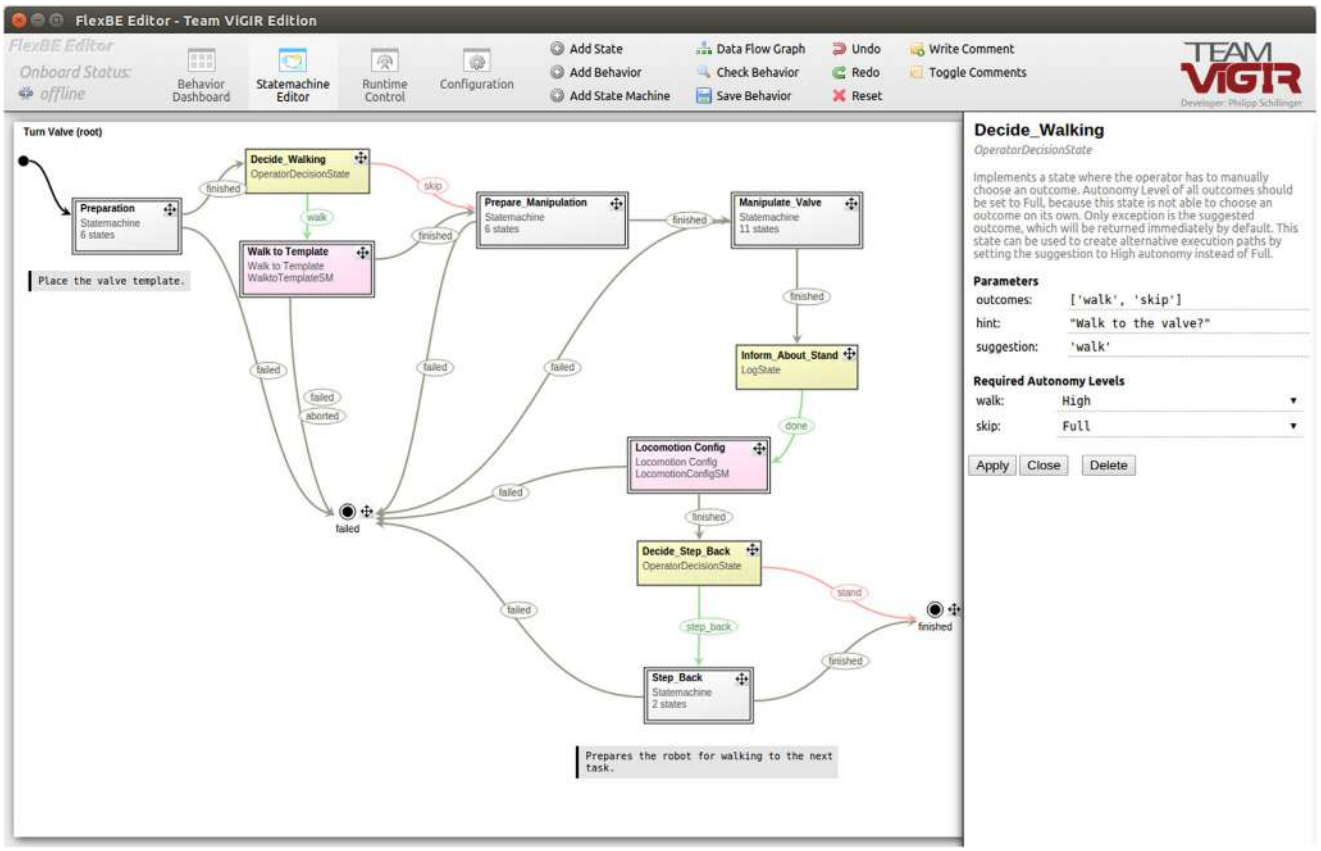


FIGURE 13 | Behavior to solve the DRC task of turning the valve, visualized by FlexBE's state machine editor. Even during execution, the structure can easily be re-arranged with just a few mouse-clicks and without manually writing code.

of manually writing code, but also because a well-defined encapsulation of robot capabilities and the re-usability of all parts encouraged proper software engineering practices, such as modularity and a clear separation of control and data flow. Furthermore, the augmentation of states by detailed documentation was helpful when working with a multitude of different capabilities.

6.3. Behavior Execution

Execution of behaviors is embedded in the graphical user interface as well. During runtime, the currently active state is displayed in the center, with transitions pointing to the possible successor states. While a robot would always proceed to the next state whenever possible in full autonomy, the operator is able to limit the autonomy level of the robot in order to prevent wrong transitions in phases of limited situational awareness. If a transition requires more autonomy than allowed by the operator, this transition will be highlighted in the runtime control user interface and the operator is asked to either confirm or decline this decision. The operator can also force a completely different one at any time.

Although this concept of collaborative autonomy is helpful for the control flow of behaviors, the operator also needs to be able to provide specific data to the robot as required during runtime if the robot fails to retrieve it on its own. For this purpose, an input data server is running as part of the OCS. Whenever requested by the robot, the operator can assist and provide the required data manually, for example, place an object template. This is invoked by using an input state, which is implemented as a robot capability like any other perception state.

In order to account for constrained communication, robot–operator collaboration is carefully designed to be bandwidth efficient. A behavior mirror component runs on the OCS and mimics the onboard behavior, requiring only minimal runtime updates. It is, thus, possible to abstract from the fact that the behavior is not running locally and components, such as the user interface, can work on this mirror in order to retrieve the data they need for monitoring the runtime status.

7. EXPERIMENTS

7.1. DRC Finals

The DRC Finals took place at Pomona, CA, USA on June 5th and 6th 2015. In the DRC Finals, three teams used the software described in this work, demonstrating the claimed flexibility and modularity.

Unlike in the previously held DRC Trials, tasks were not attempted separately. Instead teams had 60 min time to score as many of the 8 tasks as they could. Each team was allowed two runs in the competition, one on the first and one on the second competition day. The first objective was reaching the door, behind which the other tasks were situated. This could be done either by starting the robot in a Polaris Ranger vehicle and letting the robot drive up to the goal line close to the door, or by starting outside the vehicle and letting the robot walk the whole distance. Scoring awarded 0 points for walking, 1 point for driving, and 1 point for egress from the vehicle. Teams could opt out from performing egress. In this case, a reset had to be called and the robot manually extracted from the vehicle, resulting in a 10 min reset penalty and no point for egress. Traversing the door was the next task, with one point for full traversal through the door frame.

After traversing the door, communication constraints went into effect, meaning that the high bandwidth connection for perception data had pseudo-random dropouts of up to 30 s length, with 1 s windows of communication in-between. Fifteen minutes before run end, the drop outs stopped, allowing for full communication again.

7.1.1. Team Hector

Team Hector used a THOR-MANG robot. While the system showed promising capabilities during the qualification for the DRC Finals and prior to them during testing, slope of the ground at the Trials and hardware problems resulted in the robot falling in both Final runs.

The driving task was performed reliably, but on both days the robot fell while attempting to perform the door task.

7.1.2. Team Valor

Team VALOR used the ESCHER robot in the DRC Finals. The team decided to not attempt the driving task. ESCHER was one of two robots that successfully walked the complete distance from the start point up to the door. The attempt at opening the door was not successful due to encountered hardware issues.

7.1.3. Team ViGIR

Team ViGIR used the most recent, untethered version of the BDI Atlas robot. Originally, the team intended to skip the driving task. When it became clear that it would be allowed to not perform egress, but instead call for a reset, a decision was made to attempt the driving task. The performance for both competition days is briefly described the next two paragraphs.

7.1.3.1. Finals Day One

Starting in the Polaris Ranger vehicle, teleoperation was used to drive the robot down the vehicle course. A worldmodel of the course was obtained through LIDAR and cameras, and it was visualized in the OCS as described in Section 4. With this perception information, operators were able to use a driving controller system that generated the necessary joint motions to actuate the steering wheel and actuate the gas pedal. Details on the driving controller system will be described in Section 7.2. After completing the driving course, an intervention (with an associated 10-min penalty pause as specified in the DRC rules) was then used to manually extract the robot from the car.

After the penalty time, the door task was attempted. During the attempt to perform the door task, the supervisor team noticed that high-level behavior execution did not work as intended. This was later traced back to a faulty setup of the communications bridge system and increased saturation of the wireless links used in the competition. The supervisor team, thus, switched from using assisted autonomy via FlexBE behaviors to using object templates and teleoperation. Operators inserted the door template in the OCS where the sensor data of the door was displayed and commanded the robot to walk to the pre-defined stance pose for opening the door. After locomotion was performed, the operators attempted to open the door using the "open" affordance defined in the door object template. The robot hand slipped away from the door handle and, thus, the autonomous execution of the affordance failed. For this reason, the operators switched to Cartesian-space teleoperation. Using this approach, the door was successfully opened as seen in **Figure 14A**. After opening the door, the operators manually commanded the robot to walk toward a stance pose to open the valve. The valve task was solved using mainly the affordance-level control provided by the valve object template (see **Figure 14B**). Finally, before being able to actuate the switch in the surprise task, time ran out, ending the run. A video is available online.[16]

7.1.3.2. Finals Day Two

The second-day mission again started by the supervisor team using teleoperation for driving the Polaris vehicle. Due to erratic network connectivity and possible operator error, a barrier was touched and a reset had to be called. In the second attempt, the driving task was performed successfully. The door opening task was performed using object template and automated behavior control. After the door was opened, the pump of the robot shut down for unknown reasons and the robot fell. After this forced reset, another attempt at traversing the door was made, resulting in another fall. A video is available online.[17]

7.1.4. Discussion

The perception system worked as designed during the competition, providing image and full LIDAR-based environment geometry data. It provided the necessary data also under communication constraints after traversing the door only allowed intermittent communication over the 300MBit high data rate connection from the robot.

All three teams using Team ViGIR's software were able to leverage the contributions described in this work, which enabled them to perform supervised locomotion and manipulation with highly diverse bipedal robot systems. Due to encountered issues, the full potential, however, could not be demonstrated at the DRC Finals. The DRC competition operated on a tight schedule. This meant that delays in hardware availability presented significant challenges,

[16]https://youtu.be/VEsUICAa4rg
[17]https://youtu.be/Whw-tG0Wh9U

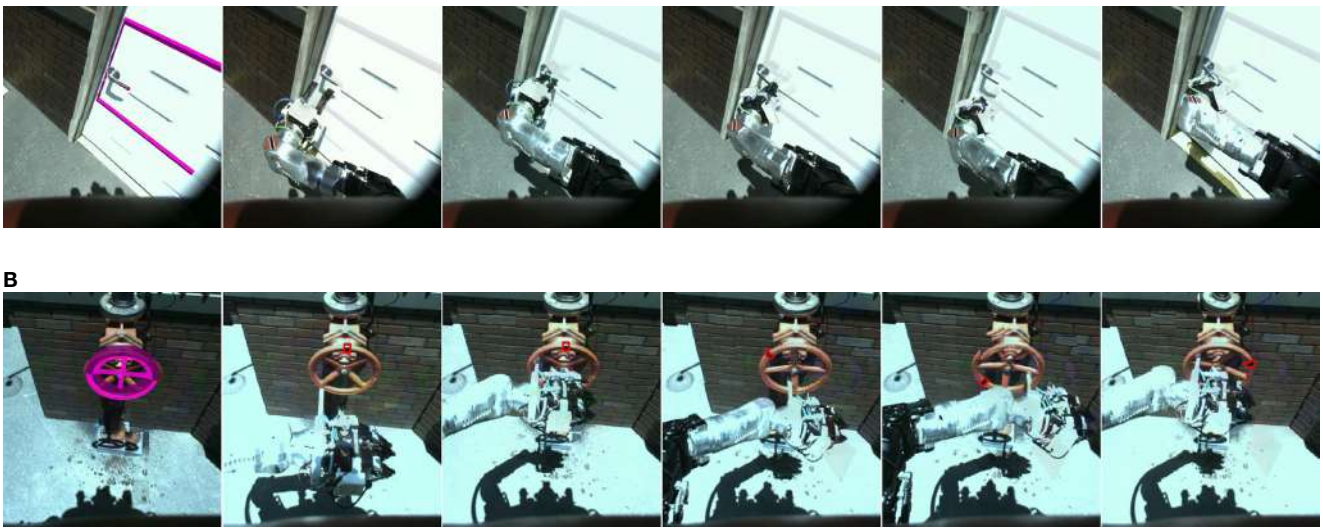


FIGURE 14 | Team ViGIR Atlas robot view of performing tasks at the DRC Finals. (A) Door task. From left to right: Door Template aligned, Pre-grasp, Grasp, Turn Clockwise affordance, Push affordance (fails to open), door opened after Cartesian teleoperation. **(B)** Valve task. From left to right: Valve Template aligned, Pre-grasp, Grasp, Open affordance 45°, 135°, and 270°. Images courtesy of Alberto Romay.

as they would reduce the time available for testing software and training operators. This general constraint is a contributing factor to the issues encountered, such as the communications setup issue experienced, during the first day by Team ViGIR.

Open source whole-body controllers for the ATLAS and Thor-MANG robot were not available; instead manufacturer-provided libraries were used for low-level control of these robots. While capabilities offered by these libraries proved useful, their closed source nature provided little transparency and did not allow for tighter coupling between high-level and low-level control. The fact that ATLAS teams who developed their own controller based on prior research scored higher than those who used the BDI-supplied one supports this assertion.

While using a sliding level of autonomy up to full teleoperation worked well, offloading the task of object perception and pose estimation from supervisors is an aspect that has not been focused on so far. Instead, the described approach relied on providing supervisors with the necessary situational awareness to perform these tasks. Reliable automated solutions have the potential to improve performance and speed at which complex tasks can be performed.

7.2. Driving a Vehicle

Demonstrating the applicability of the framework to different robot systems and tasks, we focus here on the realization of the driving task for both the ATLAS and Thor-MANG robot as an example.

7.2.1. Controlling the Vehicle

To control the vehicle, both the steering wheel and gas pedal have to be actuated. Depending on the robot used, this can be challenging due to factors, such as size, strength, and sensing capabilities. To increase robustness, adapters that can be added to the Polaris Ranger XP900 vehicle were developed. As shown in **Figure 15A**, a knob attached to the steering wheel with a spring was used for steering control. While allowing for actuation of the steering wheel, the spring adds compliance to the setup and prevents high forces being exerted on either robot or vehicle. For the pedal shown in **Figure 15B**, an adapter was used that limits pedal travel as to limit the maximum speed of the vehicle. For the ATLAS robot, the adapter also had to mechanically transfer the steering command from the passenger side of the vehicle to the pedal, as the robot was too big to fit at the driver side. As a safety measure, a spring was added to the pedal adapter that always brings the adapter back into the idle position when not pressed down.

7.2.2. Perception

As generic and robust ego-motion estimation for humanoid robots placed in vehicles is highly challenging and prone to failure, a teleoperation-based approach was used. Steering angle and gas pedal angle are set by the operator. As a safety measure, the system automatically stops if no commands have been received within a threshold duration.

7.2.3. Results

In the DRC Finals, for both ATLAS and THOR-MANG, the capability to drive a car as required in the DRC Finals rules was demonstrated. A video is available online.[18]

[18]https://www.youtube.com/watch?v=noxAK7YdJUE

FIGURE 15 | **Hardware attachment for driving a car**. **(A)** Compliant steering wheel adapter. **(B)** Pedal adapter for the Thor-MANG robot.

7.3. Simulation

Due to the high cost of complex humanoid robot systems, it is highly desirable to be able to simulate them. This allows performing research and experiments when real systems are not available.

7.3.1. Simulation of Humanoids

The components described in this work are available as open source software and an example setup using the THOR-MANG robot in Gazebo simulation can be reproduced easily using available install instructions.[19] A tutorial video showing the use of the example setup is available online.[20]

7.3.2. Example of Use with a Non-Biped Robot

Demonstrating the flexibility and modularity of the provided architecture, we show how manipulation capabilities can be added to a robot system that combines the proven mobility of a tracked base with a humanoid upper body.

The robot is capable of fully autonomous exploration of environments using the software components described in Kohlbrecher et al. (2014). In a demonstration video,[21] it first explores parts of the environment fully autonomously, with the supervisor observing progress. When the supervisor notices that the closed door prevents the robot from continuing exploration, she uses the manipulation capabilities of the robot to open the door using teleoperation or affordance-level control using the contributions described in this work. Afterwards, the supervisor can command the robot to keep exploring the environment autonomously or continue operating in a lower autonomy mode. Instructions for install and use of the shown system are available online.[22]

8. CONCLUSION

This work discusses a comprehensive software framework for performing complex disaster response tasks using humanoid robots with human supervisors in the loop. System architecture design considerations are detailed and comprehensive contributions toward different aspects, such as communication, perception, manipulation and footstep planning, and behavior control, are detailed.

The described contributions are available as open source software[23] for ROS. In contrast to other approaches, it has been successfully used on three different highly complex humanoid systems, demonstrating the flexibility and modularity of the system.

As discussed in the Section 7.1.4, while abstraction and decoupling from the low-level control system provided by robot systems can be considered a strength, achieving highest possible performance with a biped robot system requires full integration with and leveraging the capabilities of a whole-body control system. The realization of this is a subject of future work.

AUTHOR CONTRIBUTIONS

SK: Perception and manipulation; AS: Footstep planner; AR: Object/affordance template approach; PS: FlexBE behavior engine; OS: Advisor, overall design; and DC: Advisor, overall design.

[19] https://github.com/team-vigir/vigir_install/wiki/Install-thor-mang-vigir-gazebo
[20] https://www.youtube.com/watch?v=6fS89HGPEf4
[21] https://youtu.be/6ko27gKZGdA
[22] https://github.com/tu-darmstadt-ros-pkg/centaur_robot_tutorial
[23] https://github.com/team-vigir/vigir_install

REFERENCES

Banerjee, N., Long, X., Du, R., Polido, F., Feng, S., Atkeson, C. G., et al. (2015). "Human-supervised control of the atlas humanoid robot for traversing doors," in *15th IEEE-RAS International Conference on Humanoid Robots (Humanoids)*. Seoul.

Bohren, J., and Cousins, S. (2010). The SMACH high-level executive [ROS news]. *IEEE Robot. Automat. Mag.* 17, 18–20. doi:10.1109/MRA.2010.938836

Burget, F., and Bennewitz, M. (2015). "Stance selection for humanoid grasping tasks by inverse reachability maps," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, Seattle. 5669–5674.

Chitta, S., Sucan, I., and Cousins, S. (2012). MoveIt! *IEEE Robot. Automat. Mag.* 19, 18–19. doi:10.1109/MRA.2011.2181749

DRC-Teams. (2015). *What Happened at the DARPA Robotics Challenge?* Available at: http://www.cs.cmu.edu/~cga/drc/events/

Du, C., Lee, K.-H., and Newman, W. (2014). "Manipulation planning for the atlas humanoid robot," in *Robotics and Biomimetics (ROBIO), 2014 IEEE International Conference on*, 1118–1123.

Fallon, M., Kuindersma, S., Karumanchi, S., Antone, M., Schneider, T., Dai, H., et al. (2015). An architecture for online affordance-based perception and whole-body planning. *J. Field Robot.* 32, 229–254. doi:10.1002/rob.21546

Gibson, J. J. (1977). "The theory of affordances," in *Perceiving, Acting, and Knowing*, eds R. Shaw and J. Bransford (Madison, WI), 67–82.

Holz, D., and Behnke, S. (2013). "Fast range image segmentation and smoothing using approximate surface reconstruction and region growing," in *Intelligent Autonomous Systems 12, Volume 194 of Advances in Intelligent Systems and Computing*, eds S. Lee, H. Cho, K.-J. Yoon, and J. Lee (Berlin, Heidelberg: Springer), 61–73.

Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., and Burgard, W. (2013). Octomap: an efficient probabilistic 3D mapping framework based on octrees. *Auton. Robots* 34, 189–206. doi:10.1007/s10514-012-9321-0

Huang, A. S., Olson, E., and Moore, D. C. (2010). "LCM: Lightweight communications and marshalling," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. Taipei: IEEE.

Isenburg, M. (2013). Laszip. *Photogramm. Eng. Remote Sens.* 79, 209–217. doi:10.14358/PERS.79.2.209

Johnson, M., Bradshaw, J. M., Feltovich, P. J., Jonker, C. M., Van Riemsdijk, M. B., and Sierhuis, M. (2014). Coactive design: designing support for interdependence in joint activity. *J. Hum. Robot Interact.* 3, 2014. doi:10.5898/JHRI.3.1. Johnson

Johnson, M., Shrewsbury, B., Bertrand, S., Wu, T., Duran, D., Floyd, M., et al. (2015). Team IHMC's lessons learned from the DARPA robotics challenge trials. *J. Field Robot.* 32, 192–208. doi:10.1002/rob.21571

Kohlbrecher, S., Meyer, J., Graber, T., Petersen, K., Klingauf, U., and von Stryk, O. (2014). "Hector open source modules for autonomous mapping and navigation with rescue robots," in *RoboCup 2013: Robot World Cup XVII, Volume 8371 of Lecture Notes in Computer Science*, eds S.Behnke, M.Veloso, A.Visser, and R.Xiong (Berlin, Heidelberg: Springer), 624–631.

Kohlbrecher, S., Romay, A., Stumpf, A., Gupta, A., von Stryk, O., Bacim, F., et al. (2015). Human-robot teaming for rescue missions: team ViGIR's approach to the 2013 DARPA robotics challenge trials. *J. Field Robot.* 32, 352–377. doi:10.1002/rob.21558

Leeper, A., Hsiao, K., Ciocarlie, M., Sucan, I., and Salisbury, K. (2013). "Methods for collision-free arm teleoperation in clutter using constraints from 3D sensor data," in *IEEE Intl. Conf. on Humanoid Robots* (Atlanta, GA).

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., et al. (2009). "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*. Kobe.

Romay, A., Kohlbrecher, S., Conner, D. C., Stumpf, A., and von Stryk, O. (2014). "Template-based manipulation in unstructured environments for supervised semi-autonomous humanoid robots," in *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on* (Madrid: IEEE), 979–986.

Romay, A., Kohlbrecher, S., Conner, D. C., and von Stryk, O. (2015). "Achieving versatile manipulation tasks with unknown objects by supervised humanoid robots based on object templates," in *IEEE-RAS Intl. Conf. on Humanoid Robots*. Seoul.

Rusu, R. B., and Cousins, S. (2011). "3D is here: point cloud library (PCL)," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on* (Shanghai: IEEE), 1–4.

Scaramuzza, D., and Siegwart, R. (2007). "A practical toolbox for calibrating omnidirectional cameras," in *Vision Systems Applications*. Vienna: I-Tech Education and Publishing.

Schillinger, P. (2015). *An Approach for Runtime-Modifiable Behavior Control of Humanoid Rescue Robots*. Master's thesis. Technische Universität Darmstadt, Darmstadt.

Schillinger, P., Kohlbrecher, S., and von Stryk, O. (2016). "Human-robot collaborative high-level control with application to rescue robotics," in *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*.

Shoemake, K. (1985). Animating rotation with quaternion curves. *ACM SIGGRAPH Comput. Graph.* 19, 245–254. doi:10.1145/325165.325242

Stumpf, A., Kohlbrecher, S., Conner, D. C., and von Stryk, O. (2014). "Supervised footstep planning for humanoid robots in rough terrain tasks using a black box walking controller," in *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on* (Madrid: IEEE), 287–294.

Tang, P., Huber, D., and Akinci, B. (2007). "A comparative analysis of depth-discontinuity and mixed-pixel detection algorithms," in *D Digital Imaging and Modeling, 2007. 3DIM'07. Sixth International Conference on* (Montreal: IEEE), 29–38.

Tuley, J., Vandapel, N., and Hebert, M. (2005). "Analysis and removal of artifacts in 3-D ladder data," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on* (Barcelona: IEEE), 2203–2210.

Vahrenkamp, N., Asfour, T., and Dillmann, R. (2013). "Robot placement based on reachability inversion," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on* (Karlsruhe: IEEE), 1970–1975.

Willaert, B., Van Brussel, H., and Niemeyer, G. (2012). "Stability of model-mediated teleoperation: discussion and experiments," in *Haptics: Perception, Devices, Mobility, and Communication* (Tampere: Springer), 625–636.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

frontiers in Robotics and AI

ORIGINAL RESEARCH
published: 25 May 2016
doi: 10.3389/frobt.2016.00026

A Modular Software Framework for Eye–Hand Coordination in Humanoid Robots

Jürgen Leitner[1], Simon Harding[2], Alexander Förster[3] and Peter Corke[1]*

[1] Australian Centre for Robotic Vision, Queensland University of Technology, Brisbane, QLD, Australia, [2] Machine Intelligence Ltd., South Zeal, UK, [3] Institute for Artificial Intelligence, Universität Bremen, Bremen, Germany

We describe our software system enabling a tight integration between vision and control modules on complex, high-DOF humanoid robots. This is demonstrated with the *iCub* humanoid robot performing visual object detection and reaching and grasping actions. A key capability of this system is reactive avoidance of obstacle objects detected from the video stream while carrying out reach-and-grasp tasks. The subsystems of our architecture can independently be improved and updated, for example, we show that by using machine learning techniques we can improve visual perception by collecting images during the robot's interaction with the environment. We describe the task and software design constraints that led to the layered modular system architecture.

Keywords: humanoid robots, software framework, robotic vision, eye–hand coordination, reactive reaching, machine learning

OPEN ACCESS

Edited by:
Lorenzo Natale,
Istituto Italiano di Tecnologia, Italy

Reviewed by:
Matthias Rolf,
Oxford Brookes University, UK
Nikolaus Vahrenkamp,
Karlsruhe Institute of Technology, Germany
Giovanni Saponaro,
Instituto Superior Técnico, Portugal

****Correspondence:***
Jürgen Leitner
j.leitner@roboticvision.org

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *29 November 2015*
Accepted: *15 April 2016*
Published: *25 May 2016*

Citation:
Leitner J, Harding S, Förster A and Corke P (2016) A Modular Software Framework for Eye–Hand Coordination in Humanoid Robots. Front. Robot. AI 3:26. doi: 10.3389/frobt.2016.00026

1. INTRODUCTION

In the last century, robots have transitioned from science fiction to science fact. When interacting with the world around them robots need to be able to reach for, grasp, and manipulate a wide range of objects in arbitrary positions. Object manipulation, as this is referred to in robotics, is a canonical problem for autonomous systems to become truly useful. We aim to overcome the limitations of current robots and the software systems that control them, with a focus on complex bi-manual robots. It has previously been suggested that better perception and coordination between sensing and acting are key requirements to increase the capabilities of current systems (Kragic and Vincze, 2009; Ambrose et al., 2012). Yet with the increasing complexity of the mechanical systems of modern robots programing these machines can be tedious, error prone, and inaccessible to non-experts. Roboticists are increasingly considering learning over time to "program" motions into robotic systems. In addition, continuous learning increased the flexibility and provides the means for self-adaptation, leading to more capable autonomous systems. Research in artificial intelligence (AI) techniques has led to computers that can play chess on a level good enough to win against (and/or tutor) the average human player (Sadikov et al., 2007). Robotic manipulation of chess pieces on a human-level of precision and adaptation is still beyond current systems.

The problem is not with the mechanical systems. Sensory feedback is of critical importance for acting in a purposeful manner. For humans particularly, vision is an important factor in the development of reaching and grasping skills (Berthier et al., 1996; McCarty et al., 2001). The essential challenge in robotics is to create a similarly efficient perception system. For example, NASA's Space Technology Roadmap is calling for the development of autonomously calibrating hand-eye systems enabling successful off-world robotic manipulation (Ambrose et al., 2012). This ability is fundamental for

humans and animals alike, leading to many experimental studies on how we perform these actions (Posner, 1989; Jeannerod, 1997). The process is still not fully understood but basic computational models for how humans develop their reaching and grasping skills during infancy exist (Oztop et al., 2004). Where 14-month-old infants can imitate and perform simple manipulation skills (Meltzoff, 1988), robots can only perform simple, pre-programed reaching and grasping in limited scenarios. Our ability to adapt during motion execution to changing environments is lacking in robots right now. Yet this adaptation is important as even if the environment can be perceived precisely, it will not be static in most (interesting) settings.

Coming back to the chess example, for an autonomous system to pick up a chess piece, it needs to be able to perceive the board, detect the right piece, and locate the position accurately, before executing a purposeful motion that is safe for the robot and its environment. These sub-problems have turned out to be much harder than expected a few decades ago. With the progress in mechanical design, motion control, and computer vision, it is time to revisit the close coupling between those systems to create robots that perform actions in day-to-day environments.

1.1. Motion and Action: Interacting with the Environment

In the chess example, even if the state of the board and its location are known perfectly, moving a certain chess piece from one square to another without toppling other pieces is a non-trivial problem. Children, even at a very young age, have significantly better (more "natural," smoother) hand movements than almost all currently available humanoid robots. In humans, the development of hand control starts at an early age, albeit clumsily, and the precision grasp is not matured until the age of 8–10 years (Forssberg et al., 1991). Even after manipulation skills have been learnt, they are constantly adapted by a perception–action loop to yield desired results during action execution. Vision and action are closely integrated in the human brain. Various specializations develop also in the visual pathways of infants related to extracting and encoding information about the location and graspability of objects (Johnson and Munakata, 2005).

To enable robots to interact with objects in unstructured, cluttered environments, a variety of reactive approaches have been investigated. These quickly generate control commands based on sensory input – similar to reflexes – without sampling the robot's configuration space and deliberately searching for a solution (Khatib, 1986; Brooks, 1991; Schoner and Dose, 1992). Generally such approaches apply a heuristic to transform local information (in the sensor reference frame) to commands sent to the motors, leading to fast, reflex-like obstacle avoidance. Reactive approaches have become popular in the context of safety and human-robot interaction (De Santis et al., 2007; Dietrich et al., 2011) but are brittle and inefficient at achieving global goals. A detailed model of the world enables the planning of coordinated actions. Finding a path or trajectory is referred to as the path planning problem. This search for non-colliding poses is generally expensive and increasingly so with higher DOF. Robots controlled this way are typically slow and appear cautious in their motion execution. These reactive approaches started to appear in the 1980s as an alternative to the "think first, act later" paradigm. Current robotic systems operate in a very sequential manner. After a trajectory is planned, it is performed by the robot, before the actual manipulation begins. We aim to move away from such brittle global planner paradigms and have these parts overlap and have continuous refining based on visual feedback. The framework provides quick and reactive motions, as well as, interfaces to these for so higher-level agents or "opportunistic" planners can control the robot safely.

Once the robot has moved its end-effector close enough to an object, it can start to interact with it. In recent years, good progress has been made in this area thanks to the development of robust and precise grippers and hands and the improvement of grasping techniques. In addition, novel concepts of "grippers" appeared in research, including some quite ingenious solutions, such as the granular gripper (Brown et al., 2010). As alternative to designing a grasping strategy, it may be possible to learn it using only a small number of real-world examples, where good grasping points are known, and these could be generalized or transferred to a wide variety of previously unseen objects (Saxena et al., 2008). An overview of the state of research in robot grasping is found in Carbone (2013). Our framework provides an interface to "action repertoires." In one of the examples later on, we show how we use a simple grasping module that is triggered when the robot's end-effector is close to a target object. While vision may be suitable for guiding a robot to an object, the very last phase of object manipulation – the transition to contact – may require the use of sensed forces.

1.2. Robotic Vision: Perceiving the Environment

For a robot to pick a chess piece, for example, finding the chess board and each of the chess pieces in the camera image or even just to realize that there is a chess board and pieces in the scene is critical. An important area of research is the development of artificial vision systems that provide robots with such capabilities. The robot's perception system needs to be able to determine whether the image data contain some specific object, feature, or activity. While closely related to computer vision, there are a few differences mainly in how the images are acquired and how the outcome will provide input for the robot to make informed decisions. For example, visual feedback has extensively been used in mobile robot applications for obstacle avoidance, mapping, and localization (Davison and Murray, 2002; Karlsson et al., 2005). Especially in the last decade, there has been a surge of computer vision research. A focus is put on the areas relevant for object manipulation[1] and the increased interest in working around and with humans.

Robots are required to detect objects in their surroundings even if they were previously unknown. In addition, we require them to be able to build models so they can re-identify

[1]In recent years, various challenges have emerged around this topic, such as the Amazon Picking Challenge and RoboCup@Home.

and memorize them in the future. Progress has been made on detecting objects – especially when limiting the focus on specific settings – the interested reader is referred to current surveys (Cipolla et al., 2010; Verschae and Ruiz-del Solar, 2015). A solution for the general case, i.e., detecting arbitrary objects in arbitrary situations, is elusive though (Kemp et al., 2007). Environmental factors, including changing light conditions, inconsistent sensing, or incomplete data acquisition seem to be the main cause of missed or erroneous detection (Kragic and Vincze, 2009) (see also the environmental changes in **Figure 1**). Most object detection applications have been using hand-crafted features, such as SIFT (Lowe, 1999) or SURF (Bay et al., 2006), or extensions of these for higher robustness (Stückler et al., 2013). Experimental robotics still relies heavily on artificial landmarks to simplify (and speed-up) the detection problem, though there is recent progress specifically for the *iCub* platform (Ciliberto et al., 2011; Fanello et al., 2013; Gori et al., 2013). Many AI and learning techniques have been applied to object detection and classification over the past years. Deep-learning has emerged as a promising technology for extracting general features from ever larger datasets (LeCun et al., 2015; Schmidhuber, 2015). An interface to such methods is integrated in our framework and has been applied to autonomously learn object detectors from small datasets (only 5–20 images) (Leitner et al., 2012a, 2013a; Harding et al., 2013).

Another problem relevant to eye–hand coordination is estimating the position of an object with respect to the robot and its end-effector. "Spatial Perception," as this is known, is a requirement for planning useful actions and build cohesive world models. Studies in brain- and neuro-science have uncovered trends on *what* changes, when we learn to reason about distances by interacting with the world, in contrast *how* these changes happen is not yet clear (Plumert and Spencer, 2007). In robotics, to obtain a distance measure multiple camera views will provide the required observations. Projective geometry and its implementation in stereo vision systems are quite common on robotic platforms. An overview of the theory and techniques can be found in Hartley and Zisserman (2000). While projective geometry approaches work well under carefully controlled experimental circumstances, they are not easily transferred to robotics applications though. These methods are falling short as there are either separately movable cameras (such as in the case of the *iCub*, which can be seen in the imprecise out-of-the-box localization module (Pattacini, 2011)) or only single cameras available (as with Baxter). In addition, the method needs to cope with separate movement of the robot's head, gaze, and upper body. A goal for the framework was also to enable the learning of depth estimation from separately controllable camera pairs, even on complex humanoid robots moving about (Leitner et al., 2012b).

1.3. Integration: Sensorimotor Coordination

Although there exists a rich body of literature in computer vision, path planning, and feedback control, wherein many critical sub-problems are addressed individually, most demonstrable behaviors for humanoid robots do not effectively integrate elements from all three disciplines. Consequently, tasks that seem trivial to humans, such as picking up a specific object in a cluttered environment, remain beyond the state-of-the-art in experimental robotics. A close integration of computer vision and control is of importance, e.g., it was shown that to enable a 5 DOF robotic arm to pick up objects just providing a point-cloud generated model of the world was not sufficient to calculate reach and grasp behaviors on-the fly (Saxena et al., 2008). The previously mentioned work by Maitin-Shepard et al. (2010) was successful, manipulating towels due to a sequence of visually guided re-grasps. "Robotics, Vision, and Control" (Corke, 2011) puts the close integration of these components into the spotlight and describes common pitfalls and issues when trying to build systems with high levels of sensorimotor integration.

Visual Servoing (Chaumette and Hutchinson, 2006) is a commonly used approach to create a tight coupling of visual perception and motor control. The closed-loop vision-based control can be seen as a very basic level of eye–hand coordination. It has been shown to work as a functional strategy to control robots without any prior calibration of camera to end-effector transformation (Vahrenkamp et al., 2008). A drawback of visual servoing is that it requires the robust extraction of visual features; in addition,

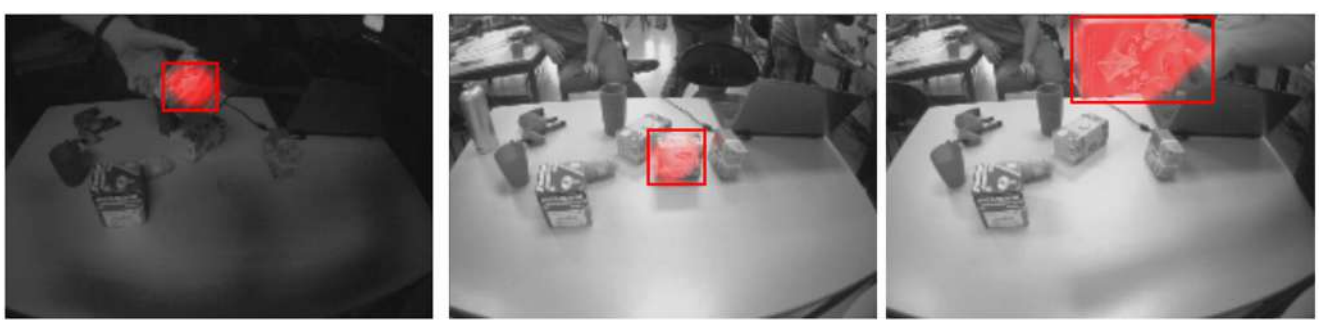


FIGURE 1 | During a stereotypical manipulation task, object detection is a hard but critical problem to solve. These images collected during our experiments show the changes in lighting, occlusions, and pose of complex objects. (Note: best viewed in color) We provide a framework that allows for the easy integration of multiple, new detectors (Leitner et al., 2013a).

the final configuration of these features in image space needs to be known *a priori*.

Active vision investigates how controlling the motion of the camera, i.e., where to look at, can be used to create additional information from a scene. (Welke et al., 2010), for example, presented a method that creates a segmentation out of multiple viewpoints of an object. These are generated by rotating an object in the robot's hand in front of its camera. These exploratory behaviors are important to create a fully functioning autonomous object classification system and are highlighting one of the big differences between computer and robotic vision.

Creating a system that can improve actions by using visual feedback, and vice versa improve visual perception by performing manipulation actions, necessitates a flexible way of representing, learning, and storing visual object descriptions. We have developed a software framework for creating a functioning eye–hand coordination system on a humanoid robot. It covers quite distinct areas of robotics research, namely machine learning, computer vision, and motion generation. Herein, we describe and showcase this modular architecture that combines those areas into an integrated system running on a real robotic platform. It was started as a tool for *iCub* humanoid but thanks to its modular design it can and has been used with other robots, most recently on a *Baxter* robot as well.

1.3.1. Robotic Systems Software Design and Toolkits

Current humanoid robots are stunning feats of engineering. With the increased complexity of these systems, the software to run these machines is increasing in complexity as well. In fact, programing today's robots requires a big effort and usually a team of researchers. To reduce the time needed to setup robotics experiments and to stop the need to repeatedly invent the wheel, good system level tools are needed. This has led to the emergence of many open source projects in robotics (Gerkey et al., 2003; van den Bergen, 2004; Metta et al., 2006; Jackson, 2007; Diankov and Kuffner, 2008; Fitzpatrick et al., 2008; Quigley et al., 2009). State-of-the art software development methods have also been translated into the robotics domain. Innovative ideas have been introduced in various areas to promote the reuse of robotic software "artifacts," such as components, frameworks, and architectural styles (Brugali, 2007). To build more general models of robot control, robotic vision and their close integration robot software needs to be able to abstract certain specificities of the underlying robotic system. There exists a wide variety of middleware systems that abstract the specifics of each robot's sensors and actuators. Furthermore, such systems need to provide the ability to communicate between modules running in parallel on separate computers.

ROS (Robot Operating System) (Quigley et al., 2009) is one of the most popular robotics software platforms. At heart, it is a component-based middleware that allows computational nodes to publish and subscribe to messages on particular topics, and to provide services to each other. Nodes communicate via "messages," i.e., data blocks of pre-defined structure, and can execute a networked distributed computer system and the connections can be changed dynamically during runtime. ROS also contains a wider set of tools for computer vision (OpenCV and point-cloud library PCL), motion planning, visualization, data logging and replay, debugging, system startup as well as drivers for a multitude of sensors, and robot platforms. For the *iCub* YARP (Yet Another Robotics Platform) (Metta et al., 2006) is the middleware of choice. It is largely written in C++ and uses separately running code instances, titled "modules." These can be dynamically and flexibly linked and communicate via concise and pre-defined messages called "bottles," facilitating component-based design. There is a wide range of other robotic middleware systems available, such as ArmarX (Vahrenkamp et al., 2015), OROCOS (Soetens, 2006), and OpenRTM (Ando et al., 2008), all with their own benefits and drawbacks, see (Elkady and Sobh, 2012) for a comprehensive comparison.

The close integration of vision and control has been addressed by VISP (Visual Servoing Platform) developed at INRIA (Marchand et al., 2005). It provides a library for controlling robotic systems based on visual feedback. It contains a multitude of image processing operations, enabling robots to extract useful features from an image. By providing the desired feature values, a controller for the robot's motion can be derived (Hutchinson et al., 1996; Chaumette and Hutchinson, 2006). The framework presented here is building on these software systems to provide a module-based approach to tightly integrate computer vision and motion control for reaching and grasping on a humanoid robot. The architecture grew naturally over the last few years and was initially designed for the *iCub* and, hence, used YARP. While there exists also a "bridge" component in YARP allowing it to communicate with ROS topics and nodes, it was easy to port it to ROS and Baxter. Furthermore, there is currently a branch being developed aimed to be fully agnostic to the underlying middleware.

2. THE EYE–HAND FRAMEWORK

The goal of our research is to improve the autonomous skills of humanoid robots by providing a library giving a solid base of sensorimotor coordination. To do so, we developed a modular framework that allows to easily run and repeat experiments on humanoid robots. To create better perception and motion, as well as a coordination between those, we split the system into two subsystems: one focusing on action and the other one on vision (our primary sense). To deal with uncertainties, various machine learning (ML) and artificial intelligence (AI) techniques are applied to support both subsystems and their integration. We close the loop and perform grasping of objects, while adapting to unknown, complex environments based on visual feedback, showing that combining robot learning approaches with computer vision improves adaptivity and autonomy in robotic reaching and grasping.

Our framework, sketched in **Figure 2A**, provides an integrated system for eye–hand coordination. The Perception (green) and Action (yellow) subsystems are supported by Memory (in blue) that enables the persistent modeling of the world. Functionality has grown over time and the currently existing modules that have been used in support of eye–hand coordination framework for cognitive robotics research (Leitner, 2014, 2015) are as follows:

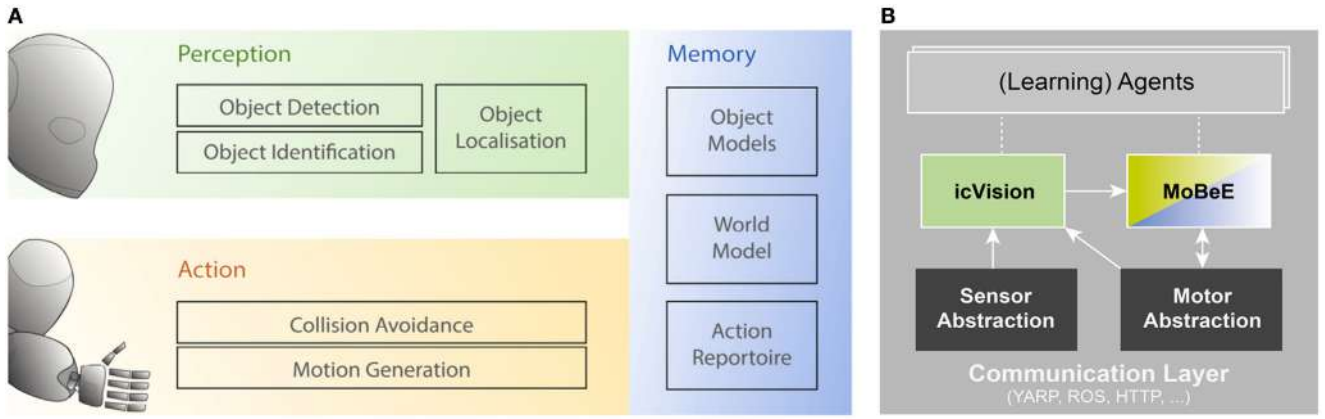


FIGURE 2 | **(A)** Overview of the common subsystems for a functional eye–hand coordination on humanoid robots. In broader terms, one can separate the (visual) perception side (in green) from the action side (yellow). In addition to these, a memory subsystem (blue) allows to build-up an action repertoire and a set of object models. **(B)** The presented framework herein consists of a modular way of combining perception tasks, encapsulated in the *icVision* subsystem [green, as in **(A)**], with the action side and a world model, represented by the *MoBeE* subsystem (in yellow and blue). In addition, agents can interface these systems to generate specific behaviors or to learn from the interaction with the environment (see Results). To allow portability, the system uses a communication layer and a robot abstraction middleware, e.g., ROS or YARP.

- *Perception: Object Detection and Identification:* as mentioned above, the detection and identification of objects is a hard problem. To perform object detection and identification, we use a system called *icVision*. It provides a modular approach for the parallel execution of multiple object detectors and identifiers. While these can be hard-coded (e.g., optical flow segmentation of moving object, or simple color thresholding), the main advantage of this flexible system is that it can be interfaced by a learning agent (sketched in **Figure 2B**). In our case, we have successfully used Cartesian Genetic Programing for Image Processing (CGP-IP) (Harding et al., 2013) as an agent to learn visual object models in both a supervised and unsupervised fashion (Leitner et al., 2012a). The resulting modules perform specific object segmentation of the camera images.
- *Perception: Object Localization: icVision* also provides a module for estimating the location of an object detected by multiple cameras – i.e., the two eyes in the case of the *iCub*. In this case, again the flexibility of the perception framework allows for a learning agent to predict object positions with a technique based on genetic programing and an artificial neural network estimators (Leitner et al., 2012b). These modules can be easily swapped or run in parallel, even on different machines.
- *Action: Collision Avoidance and Motion Generation: MoBeE* is used to safeguard the robot from collisions both with itself and the objects detected. This is implemented as a low-level interface to the robot and uses virtual forces based on the robot kinematics to generate the robot's motion. A high-level agent or planner can provide the input to this system (more details in the next section).
- *Memory: World Model:* In addition to modeling the kinematics of the robot to *MoBeE* also keeps track of the detected object in operational space. It is also used as a visualization for the robot's current belief state by highlighting (impeding) collisions (see Section 2.2).
- *Memory: Action Repertoire:* a light-weight, easy-to-use, one-shot grasping system is used. It can be configured to perform a variety of grasps, all requiring to close the fingers in a coordinated fashion. The *iCub* incorporates touch sensors on the fingertips, but due to the high noise, we use the error reported by the PID controllers of the finger motors to know when they are in contact with the object.

Complex, state-of-the art humanoid robots are controlled by a distributed system of computers most of which are not onboard the robot. On the *iCub* (and similarly on *Baxter*), an umbilical provides power to the robot and a local-area-network (LAN) connection. **Figure 3** sketches the distributed computing system used to operate a typical humanoid robot: very limited on-board computing, which mainly focuses on the low-level control and sensing, is supported by networked computers for computational intensive tasks. The *iCub*, for example, employs an on-board PC104 controller that communicates with actuators and sensors using CANBus. Similarly, *Baxter* has an on-board computing system (Intel i7) acting as the gateway to joints and cameras. More robot-specific information about setup and configuration, as well as the code base, can be found on the *iCub* and *Baxter* Wiki pages,[2] where researchers, from a large collection of research labs using the robot, contribute and build up a knowledge base.

All the modules described communicate with each other using a middleware framework (depicted in **Figure 2B**). The first experiments were performed on the *iCub*; therefore, the first choice for the middleware was YARP. A benefit of using a robotic middleware is that actuators and sensors can be abstracted, i.e., the modules that connect to *icVision* and *MoBeE* do not require to know the robot specifics. Another benefit of building on existing robotics middleware is the ability to distribute modules across

[2] *iCub* Wiki URL: http://wiki.icub.org *Baxter* Wiki URL: http://api.rethinkrobotics.com

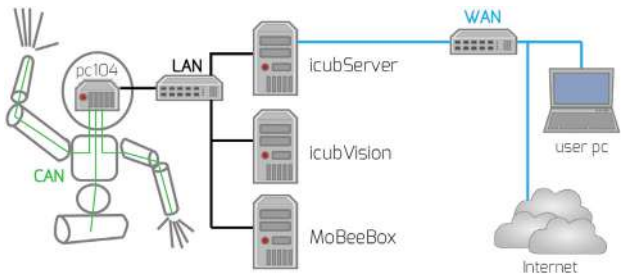


FIGURE 3 | A sketch of the computing setup we used to operate the *iCub* at the IDSIA Robotics Lab. The `pc104` handles the on-board data processing and controls the motors via CAN-bus. The `icubServer` is running the YARP server and is the router into the IDSIA-wide network and the internet. Dedicated computers for vision (`icubVision`) and collision avoidance (`MoBeeBox`) are used.

multiple computers. In our setup, the various computational tasks were implemented as nodes, which were then distributed throughout the network of computers. For the experiments on the *iCub*, a separate computer was used to run multiple object detection modules in parallel, while another computer (`MoBeEBox`) performed the collision avoidance and visualizing the world model. During the development of new modules, an additional user PC was connected via Ethernet to run and debug the new modules. Component-level abstraction using middleware increases portability across different robotic systems. For example, running *MoBeE* with different robot arms is easily done by simply providing the new arm's kinematic model as an XML file. Transferring to other middleware systems is also possible, though a bit more intricate. We have ported various parts of the architecture to ROS-based modules allowing to interact with ROS-based robots, such as *Baxter*.

Humanoid robots, and the *iCub* in particular, have a high DOF, which allows for complex motions. To perform useful actions, many robots need to be controlled in unison requiring robust control and planning algorithms. Our framework consists of an action subsystem, which in turn contains collision avoidance and grasping capabilities.

2.1. Object Detection and Localization Modules: icVision

Our humanoid robot should be able to learn how to perceive and detect objects from very few examples, in a manner similar to humans. It should have the ability to develop a representation that allows it to detect the same object again and again, even when the lighting conditions change, e.g., during the course of a day. This is a necessary prerequisite to enable adaptive, autonomous behaviors based on visual feedback. Our goal is to apply a combination of robot learning approaches, artificial intelligence, and machine learning techniques, with computer vision, to enable a variety of proposed tasks for robots.

icVision (Leitner et al., 2013c) was developed to support current and future research in cognitive robotics. This follows a "passive" approach to the understanding of vision, where the actions of the human or robot are not taken into account. It processes the visual inputs received by the cameras and builds (internal) representations of objects. This computation is distributed over multiple modules. It facilitates the 3D localization of the detected objects in the 2D image plane and provides this information to other systems, e.g., a motion planner. It allows to create distributed systems of loosely coupled modules and provides standardized interfaces. Special focus is put on object detection in the received input images. **Figure 4** shows how a simple red detection can be added as a separate running module. Specialized modules, containing a specific model, are used to detect distinct patterns or objects. These specialized modules can be connected and form pathways to perform, e.g., object detection, similarly to the hierarchies in the visual cortex. While the focus herein is on the use of single and stereo camera images, we are confident that information from RGB-D cameras (such as the Microsoft Kinect) can be easily integrated.

The system consists of different modules, with the core module providing basic functionality and information flow. **Figure 5** shows separate modules for the detection and localization and their connection to the core, which abstract the robot's cameras and the communication to external agents. These external agents are further modules and can do a wide variety of tasks, for example, specifically test and compare different object detection or localization techniques. *icVision* provides a pipeline that connects visual perception with world modeling in the *MoBeE* module (dashed line in **Figure 5**). By processing the incoming images from the robot with a specific filter for each "eye," the location of the specific object can be estimated by the localization module and then communicated to *MoBeE* (**Figure 6** depicts the typical information flow).

2.2. Robot and World Modeling for Collision Avoidance: MoBeE

MoBeE (Modular Behavior Environment for Robots) is at the core of the described framework for eye–hand coordination. It is a solid, reusable, open-source[3] toolkit for prototyping behaviors on the *iCub* humanoid robot. *MoBeE* represents the state-of-the-art in humanoid robotic control and is similar in conception to the control system that runs DLR's Justin (De Santis et al., 2007; Dietrich et al., 2011). The goal of *MoBeE* is to facilitate the close integration of planning and motion control (sketched in **Figure 2B**). Inspired by Brooks (1991), it aims to embody the planner, provide safe and robust action primitives, and perform real-time re-planning. This facilitates exploratory behavior using a real robot with *MoBeE* acting as a supervisor preventing collisions, even between multiple robots. It consists of three main parts all implemented in C++: a kinematic library with a visualization, and a controller, running in two separate modules. These together provide the "collision avoidance" (yellow) and "world model" (blue) as depicted in **Figure 2A**. **Figure 5** shows the connections between the various software entities required to run the full eye–hand coordination framework. *MoBeE* communicates

[3]URL: https://github.com/kailfrank/MoBeE

```
#include <string>
#include <yarp/os/all.h>
#include <yarp/sig/all.h>

#include "icFilterModule.h"

using namespace std;
using namespace yarp::os;
using namespace yarp::sig;

class RedFilterModule : public icFilterModule {

public:
    RedFilterModule() { setName("RedFilterTest"); }
    ~RedFilterModule() {}

    /* This is our main function. Will be called periodically every getPeriod() s */
    bool TestModule::updateModule()
    {
        if(! isRunning ) return false;

        ImageOf<PixelBgr> *left_image = leftInPort.read();  // read an image
        ... // check for valid image

        // very simple test for red-ness
        for (int x=0; x < left_image->width(); x++)
            for (int y=0; y < left_image->height(); y++) {
                PixelBgr& pixel = left_image->pixel(x,y);
                // make sure red level exceeds blue and green by a factor and a threshold
                if ( pixel.r > pixel.b * 1.5 + 10 && pixel.r > pixel.g * 1.5 + 10 )
                    output_image[x][y] = 1;
                else
                    output_image[x][y] = 0;
            }

        outputPort.write();         // write the output image
        return isRunning;
    }
};
```

FIGURE 4 | Little coding is required for a new module to be added as filter to *icVision*. This shows a simple red filter being added. The image acquisition, connection of the communication ports, and cleanup are all handled by the superclass.

with the robot and provides an interface to other modules. One of these is the perception side *icVision*.

In its first iteration, *MoBeE* provided virtual feedback for a reinforcement learning experiment. This was necessary as most current robotic systems lack a physical skin that would provide sensory information to perform reflexive motions. It was intended to enforce constraints in real time while a robot is under the control of any arbitrary planner/controller. This led to a design based on switching control, which facilitated experimentation with pre-existing control modules. A kinematic model is loaded from an XML file using "Zero Position Displacement Notation" (Gupta, 1986).

When the stochastic or exploratory agent/controller (light gray at the top in **Figure 5**) does something dangerous or undesirable, *MoBeE* intervenes. Collision detection is performed on the loaded kinematic robot model consisting of a collection of simple geometries to form separate body parts (see **Figure 7**). These geometries are created as C++ objects that inherit functionality from both the fast geometric intersection library and the visualization in OpenGL. The joint encoders provided by the robot abstraction layer are used to calculate collisions, i.e., intersecting body parts. In the first version, this collision signal was used to avoid collisions by switching control, which was later abandoned in favor of a second-order dynamical system (Frank, 2014). Constraints, such as impeding collisions, joint limits, or cable lengths, can be addressed by adding additional forces to the system. Due to the dynamical system, many of the collisions encountered in practice no longer stop the robot's action, but rather deflect the requested motion, bending it around an obstacle.

MoBeE continuously mixes control and constraint forces to generate the robot motion in real time and results in smoother, more intuitive motions in response to constraints/collisions (**Figure 8**). The effects of sensory noise are mitigated passively by the controller. The constraint forces associated with collisions are proportional to their penetration depth; in the experimentation, it was observed that the noise in the motor encoder signal has a minimal effect on collision response. The sporadic shallow collisions, which can be observed when the robot is operating close to an obstacle, such as the other pieces of a chess board, generate tiny forces that only serve to nudge the robot gently away from the obstacle. *MoBeE* in addition can be used for adaptive roadmap planning (Kavraki et al., 1996; Stollenga et al., 2013), the dynamical approach means that the planner/controller is free to explore continuous spaces, without the need to divide them into safe and unsafe regions.

The interface for external agents is further simplified by allowing to subscribe to specific points of interest in the imported models (seen in yellow in **Figure 7**). These markers can be defined both on static or moving objects or the robots. The marker positions or events, such as the body part being in a colliding pose,

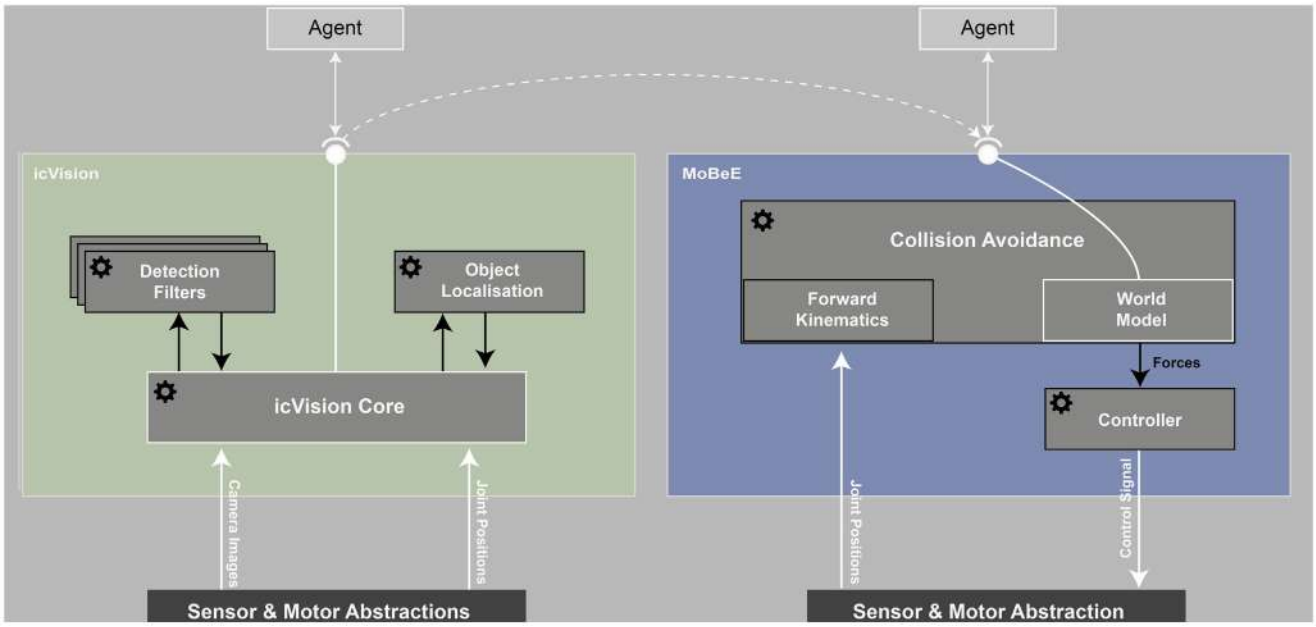


FIGURE 5 | The framework described consists of multiple software entities (indicated by cogwheels), all connected via a communication layer, such as YARP or ROS. The provided robot abstraction is used by the main modules. On the perception side, *icVision* processes the incoming camera images in its core module and sends them to (possibly multiple) separately running detection filters. Another separate entity is performing the localization based on the detected objects and robot's pose. The *icVision* Core also provides interfaces for agents to query for specific objects or agents that learn object representations (such as a CGP-IP based learner). The interface also provides the objects location to *MoBeE*. There a world model is created by calculating the forward kinematics from the incoming joint positions. The same entity performs the collision avoidance between separate body parts or the objects that have been detected by *icVision*. The controller is independent and translates the virtual forces created by *MoBeE* or provided by higher level planning agents into motor commands.

are broadcast via the interface allowing connected agents to react, e.g., to trigger a grasp primitive. More details about the whole *MoBeE* architecture and how it was used for reach learning can be found in Frank (2014). Additionally, we have published multiple videos of our robotic experiments while using *MoBeE* foremost: "Toward Intelligent Humanoids."[4]

2.3. Action Repertoire: LEOGrasper Module

Robotic grasping is an active and large research area in robotics. A main issue is that in order to grasp successfully the pose of the object to be grasped has to be known quite precisely. This is due to the grasp planners required to plan the precise placement and motion of each individual "finger" (or gripper). Several methods for robust grasp planning exploit the object geometry or tactile sensor feedback. However, object pose range estimation introduces specific uncertainties that can also be exploited to choose more robust grasps (Carbone, 2013).

A different approach is used in our implementation that does use a more reactive approach. Grasp primitives are triggered from *MoBeE*, which involve the controlling the five digit *iCub* hand. These primitives consist of target points in joint space to be reached sequentially during grasp execution. Another problem is to realize when to stop grasping. The *iCub* has touch sensors on the palm and finger tips. To know when there is a successful grasp, these sensors need to be calibrated for the material in use. Especially for objects as varied as plastic cups, ceramic mugs, and tin cans, the tuning can be quite cumbersome and leads to a lower signal-to-noise ratio. We decided to overcome this by using the errors from the joint controllers directly. This approach allows to provide feedback whether a grasp was successful or not to a planner or learning system.

LEOGrasper is our light-weight, easy-to-use, one-shot grasping system for the *iCub*. The system itself is contained in one single module using YARP to communicate. It can be triggered by a simple command from the command line, network, or as in our case from *MoBeE*. The module can be configured for multiple grasp types, these are loaded from a simple text file, containing some global parameters (such as the maximum velocity) as well as the trajectories. Trajectories are specified by providing positions for each joint individually, containing multiple joints per digit as well as abduction, spread, etc. on the *iCub*. We provide power and pinch grasp and pointing gestures. For example, to close all digits in a coordinated fashion, at least two positions need to be defined, the starting and end position (see **Figure 9**). For more intricate grasps, multiple intermediate points can be provided. The robot's fingers are controlled from the start point to each consecutive point, when an `open` signal is received. For `close`, the points are sent in reverse order.

[4]Webpage: http://Juxi.net/media/ or direct video URL: http://vimeo.com/51011081

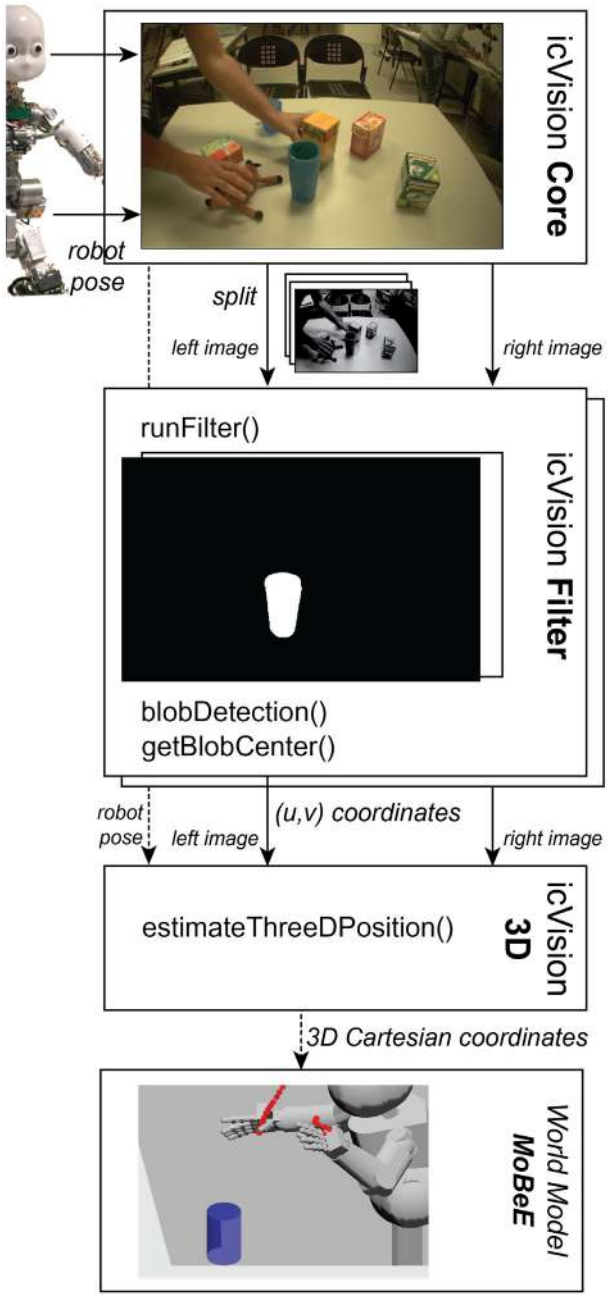


FIGURE 6 | To provide information about the 3D location of an object to *MoBeE*, the following is performed: at first, camera images are received by the *core* from the hardware via a communications layers. The images are split into channels and made available to each individual *icVision* filters that is currently active. These then perform binary segmentation for a specific object. The objects (center) location in the image frame, *(u,v)* is then communicated to a 3D localization module. Using the joint encoder values and the object's location in both eyes, a location estimate is then sent to the *MoBeE* world model (Leitner et al., 2013a).

LEOGrasper has been used extensively in our robotics lab and selected successful grasps are shown in **Figure 10**.[5] The existing trajectories and holding parameters were tuned through experimentation; in the future, we aim at learning these primitives using human demonstrations or reward signals.

3. METHOD OF INTEGRATING ACTION AND VISION: APPLYING THE FRAMEWORK

The framework has been extensively used over the last few years in our experimental robotics research. Various papers have been published during the development of the different subsystems and their improvements. **Table 1** provides an overview. *MoBeE* can be pre-loaded with a robot model using an XML file that describes the kinematics based on "Zero Position Displacement Notation" (Gupta, 1986). **Figure 11** shows a snippet from the XML describing the Katana robotic arm. In addition a pre-defined, marked-up world model can be loaded from a separate file as well. This is particularly useful for stationary objects in the world or to restrict the movement space of the robot during learning operations.

Through the common interface to *MoBeE* object properties of each object can be modified, through an RPC call, following YARP standard and is accessible from the command line, a webpage, or any other module connecting to it. These objects are placed in the world model by either loading from a file at start-up or during runtime by agents, such as the *icVision* core. Through the interface an object can also be set as an *obstacle*, which means repelling forces are calculated, or as a *target*, which will attract the end-effector. In addition, objects can be defined as ghosts, leading to the object being ignored in the force calculation.

As mentioned earlier on, previous research suggests that connections between motor actions and observations exist in the human brain and describes their importance to human development (Berthier et al., 1996). To interface and connect artificial systems performing visual and motor cortex-like operations on robots will be crucial for the development of autonomous robotic systems. When attempting to learn behaviors on a complex robot, such as the *iCub* or Baxter, state-of-the-art AI and control theories can be tested (Frank et al., 2014) and shortcomings of these learning methods can be discovered (Zhang et al., 2015) and addressed. For example, Hart et al. (2006) showed that a developmental approach can be used for a robot to learn to reach and grasp. We developed modules for action generation and collision avoidance and their interfaces to the perception side. By having the action and motion side tightly coupled, we can use learning algorithms that require also negative feedback. We can create this without actually "hurting" the robot.

3.1. Example: Evolving Object Detectors

We previously developed a technique based on Cartesian Genetic Programing (CGP) (Miller, 1999, 2011) allowing for the automatic generation of computer programs for robot vision tasks, called Cartesian Genetic Programming for Image Processing (CGP-IP) (Harding et al., 2013). CGP-IP draws inspiration from previous work in the field of machine learning and combines it with the available tools in the image processing discipline, namely in the form of OpenCV functions. OpenCV is an open-source

[5] Source code available at: https://github.com/Juxi/iCub/

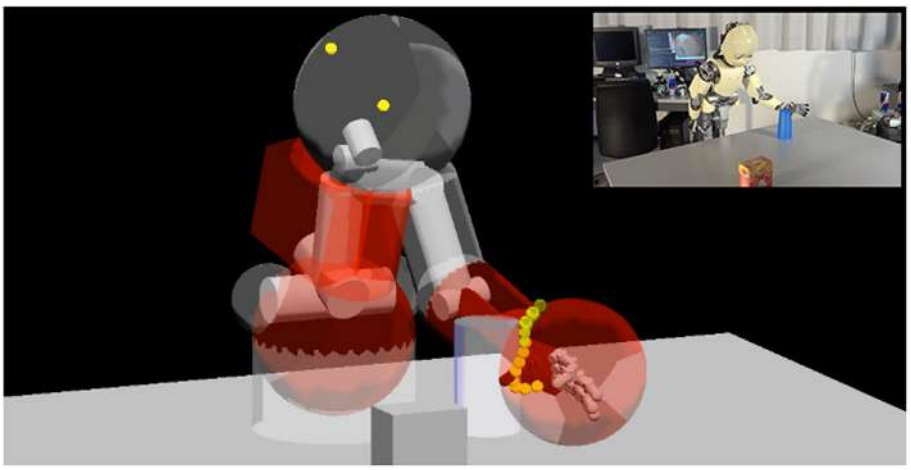


FIGURE 7 | A scene of the *iCub* avoiding an object (inset) during one of our experiments (Leitner et al., 2014b) and its corresponding visualization of the *MoBeE* model. Red body parts are highlighting impeding collisions with either another body part (as in the case of the hip with the upper body) or an object in the world model (hand with the cup). (See video: https://www.youtube.com/watch?v=w_qDH5tSe7g).

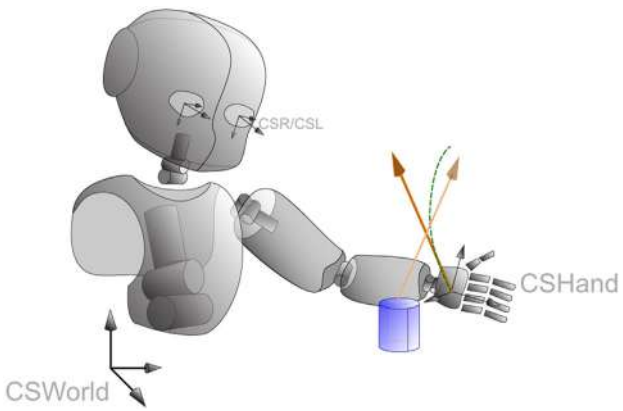


FIGURE 8 | The virtual forces created by the dynamical system within *MoBeE*. It continuously mixing control and constraint forces (orange vectors) to generate the robot motion in real time. It results in a smoother, more intuitive motions in response to constraints/collisions (dashed green line). To calculate the force, the distance of the object to the hand in its coordinate frame CSHand is used. *MoBeE* handles the transformation from the coordinate systems of the cameras (CSR/CSL) to the world frame CSWorld and CSHand.

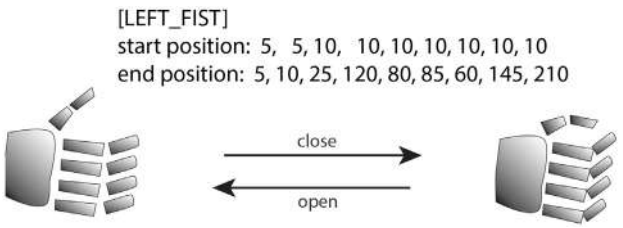


FIGURE 9 | Defining a grasp in *LEOGrasper* is simple: defining a start and end position in joint space is all that is required. The `open` command will revert the hand into the start state (left), `close` will attempt to reach the end state (right). For more complex grasps intermediate states can be provided.

framework providing a mature toolbox for a variety of image processing tasks. This domain knowledge is integrated into CGP-IP allowing to quickly evolve object detectors from only a small training set – our experiments showed that just a handful of (5–20) images per object are required. These detectors can then be used to perform the binary image segmentation within the *icVision* framework. In addition, CGP-IP allows for the segmentation of color images with multiple channels, a key difference to much of the previous work focusing on gray scale images. CGP-IP deals with separate channels and splits incoming color images into individual channels before they can be used at each node in the detector. This leads to the evolutionary process selecting which channels will be used and how they are combined.

CGP-IP manages a population of candidates, which consists of individual *genes*, representing the nodes. Single channels are used as inputs and outputs of each node, while the action of each node is the execution of an OpenCV function. The full candidate can be interpreted as a computer program performing a sequence of image operations on the input image. The output of each candidate filter is a binary segmentation. GPs are supervised, in the sense that a fitness will need to be calculated for each candidate. For scoring each individual, a ground truth segmentation needs to be provided. A new generation of candidates is then created out of the fittest individuals. An illustrative example of a CGP-IP candidate is shown in **Figure 12**. CGP-IP can directly create C# or C++ code from these graphs. The code can be executed directly on the real hardware or pushed as updates to existing filter modules running within *icVision*. CGP-IP includes classes for image operations, the evolutionary search and the integration

FIGURE 10 | The *iCub* hand during grasp execution with a variety of objects, including tin cans, tea boxes, and plastic cups (Leitner et al., 2014b).

TABLE 1 | Overview of experiments facilitated by parts of the architecture presented herein.

Experiment description	Framework	Reference
Autonomous object detection	icVision, CGP-IP	Leitner et al. (2012a, 2013b)
Multi robot collision avoidance	MoBeE (vSkin)	Leitner et al. (2012b,c)
Safe policy learning	MoBeE (vSkin)	Pathak et al. (2013)
Object detection and localisation	icVision	Leitner et al. (2013a)
Spatial perception learning	MoBeE, icVision	Leitner et al. (2013d)
Learning object detection	CGP-IP	Leitner et al. (2013e)
Humanoid motion planning	MoBeE	Stollenga et al. (2013)
Reinforcement learning for reaching	MoBeE	Frank et al. (2014)
Improving vision through interaction	Full system	Leitner et al. (2014a)
Reactive reaching and grasping	Full system	Leitner et al. (2014b)
Cognitive and developmental robots	Full system	Leitner (2015)

with the robotic side through a middleware. Currently, we are extending the C# implementation to run on various operating systems and be integrated into a distributed visual system, such as DRVS (Chamberlain et al., 2016).

CGP-IP allows not just for a simple reusable object detection but also provides a simple way of learning these based on only very small training sets. In connection with our framework, these data can be collected on the real hardware and the learned results directly executed. For this, an agent module was designed that communicates with the *icVision* core through its interface. Arbitrary objects are then placed in front of the robot and images are collected while the robot is moving about. The collected images are then processed by the agent and a training set is created. With the ground truth of the location known, so is the location of the object in the image. A fixed size bounding box around this location leads to the ground truth required to evolve an object detector. This way the robot (and some prior knowledge of the location) can be used to autonomously learn object detectors for all the objects in the robot's environment (Leitner et al., 2012a).

3.2. Example: Reaching While Avoiding a Moving Obstacle

The inverse kinematics problem, i.e., placing the hand at a given coordinate in operational space, can be performed with previously available software on the *iCub*, such as the existing operational space controller (Pattacini, 2011) or a roadmap-based approach (Stollenga et al., 2013). These systems require very accurate knowledge of the mechanical system to lead to precise solutions, requiring a lengthy calibration procedure. These systems also tend to be brittle when change in the robot's environment requires adapting the created motions.

To overcome this problem, the framework, as described above, creates virtual forces based on the world model within *MoBeE* to govern the actual movement of the robot. Static objects in the environment, such as, e.g., the table in front of the robot, can be added directly into the model via an XML file. Once in the model, actions and behaviors are adapted due to computed constraint forces. This way we are able to send arbitrary motions to our system, while ensuring the safety of our robot. Even with just these static objects, this has been shown to provide an interesting way to learn robot reaching behaviors through reinforcement (Pathak et al., 2013; Frank et al., 2014). The presented system has the same functionality also for arbitrary, non-static objects.

For this after the detection in both cameras, the object's location is estimated and updated in the world model. The forces are continually recalculated to avoid impeding collisions even with moving objects. **Figure 7** shows how the localized object is in the way of the arm and the hand. To ensure the safety of the rather fragile fingers, a collision sphere around the end-effector was added – seen in red, indicating a possible collision due to the sphere intersecting with the object. The same can be seen with the lower arm. The forces push the intersecting geometries away from each other, leading to a movement of the

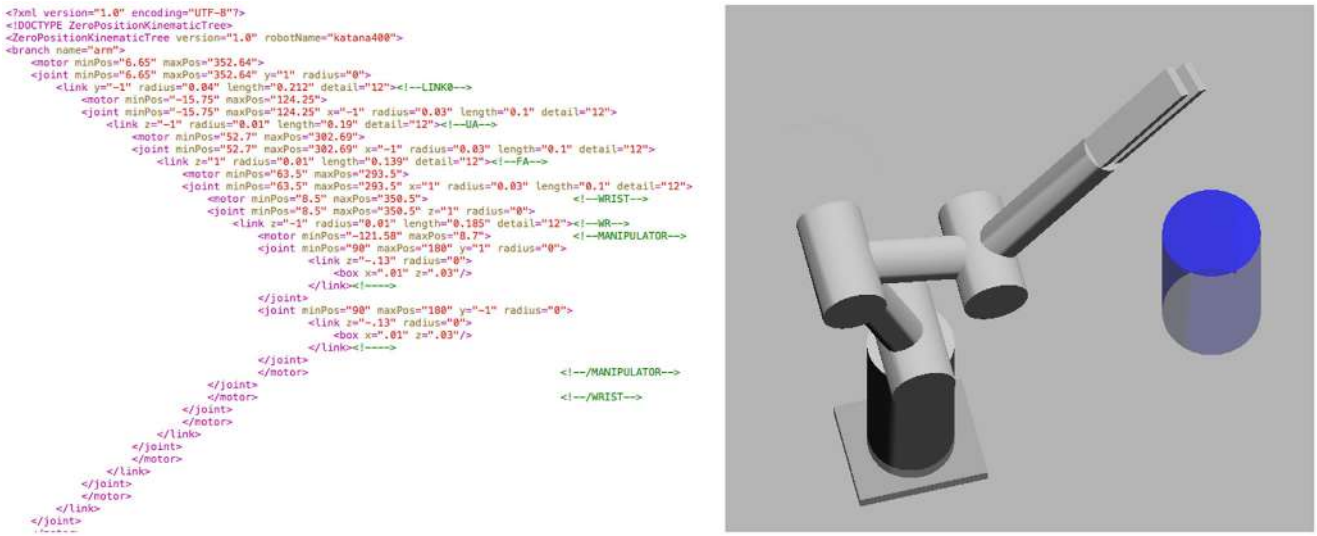


FIGURE 11 | (Left) The XML files used to describe the kinematics of the Katana arm. (Right) Visualization of the same XML file in *MoBeE*.

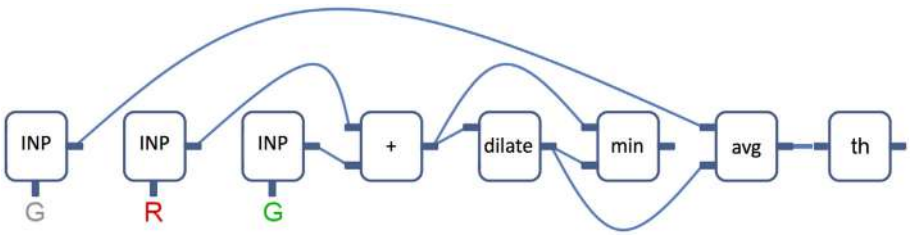


FIGURE 12 | Example illustration of a CGP-IP genotype, taking three input channels of the current image. A sequence of OpenCV operations is then performed before thresholding occurs to produce a binary segmented output image.

end-effector away from the obstacle. **Figure 13** shows how the robot's arm is "pushed" aside when the cup is moved close to the arm, therefore avoiding a non-stationary obstacle. It does so until the arm reaches its limit, then the forces cumulate and the end-effector is "forced" upwards to continue avoiding the obstacle. Similarly the reaching behaviour is adapted while the object is moved. Without an obstacle, the arm starts to settle back into its resting pose q^*. By simply sending a signal through the interface the type of the object within the world model can be changed from `obstacle` into `target`. This leads to the calculated forces now being attracting not repelling. *MoBeE* also allows to trigger certain responses when collisions occur. In the case when we want the robot to pick-up the object, we can activate a grasp subsystem whenever the hand is in the close vicinity of the object. We are using a prototypical power grasp style hand-closing action, which has been used successfully in various demos and videos.[6] **Figure 10** shows the *iCub* successfully picking up (by adding an extra upwards force) various objects using our grasping subsystem, executing the same action. Our robot frameworks are able to track multiple objects at the same time, which is also visible in **Figure 7**, where both the cup and the tea box are tracked. By simply changing the type of the object within *MoBeE*, the robot reaches for a certain object while avoiding the other.

3.3. Example: Improving Robot Vision by Interaction

The two subsystems can further be integrated for the use of higher level agents controlling the robot's behavior. Based on the previous section, the following example shows how an agent can be used to learn visual representations (in CGP-IP) by having a robot interact with its environment. Building on the previously mentioned evolved object detectors, we extended the robot's interaction ability to become better at segmenting objects. Similar to the experiment by Welke et al. (2010), the robot was able to curiously rotate the object of interest with its hand. Additional actions were added for the robot to perform, such as poke, push, and a simple viewpoint change by leaning left and right. Furthermore, a baseline image dataset is collected, while the robot (and the object) is static. In this experiment, we wanted to measure the impact of specific actions on the segmentation

[6]See videos at: http://Juxi.net/media/

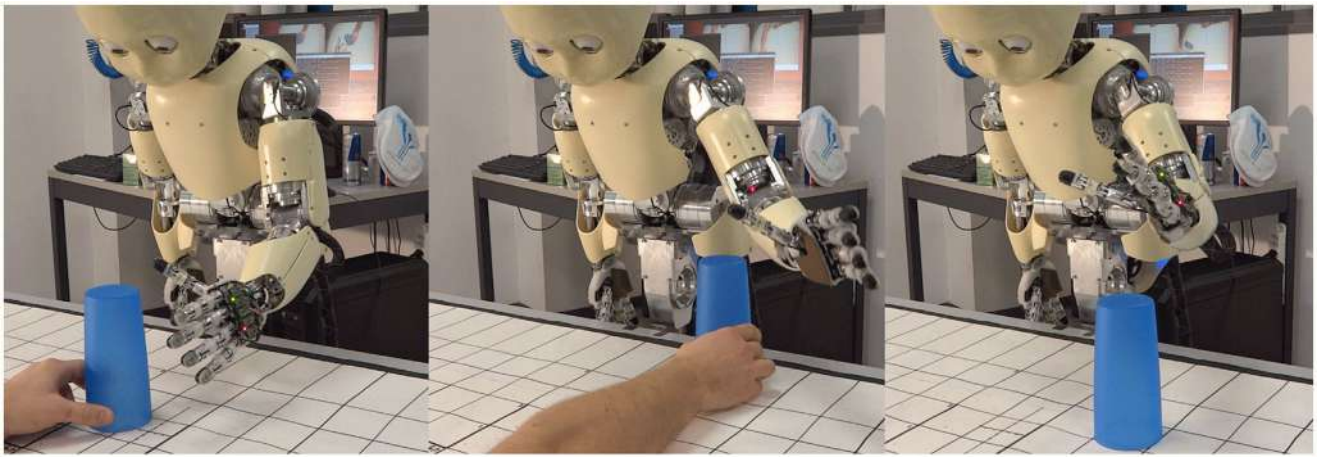


FIGURE 13 | The *iCub*'s arm is controlled by *MoBeE* to stay in a non-colliding pose of the moving obstacle and the table by using reactive virtual forces. (See video: https://www.youtube.com/watch?v=w_qDH5tSe7g).

performance. After the robot performed one of the four pre-programed actions, a new training set was collected, which contains the images from the static scenario and the images during action execution. While more data mean generally better results, we could also see that some actions were leading to better results than others. **Figure 14** shows visually how the improvement leads to better object segmentation, in validation images. On the left is the original camera image, in the middle the segmentation performed by an evolved filter solely based on the "static scene baseline," and on the right is the segmentation when integrating the new observations during a haptic exploration action.

By providing a measurable improvement, the robot can select and perform the action yielding the best possible improvement for a specific detector. Interaction provides robots with a unique possibility (compared to cameras) to build more accurate and robust visual representations. Simple leaning actions change the camera viewpoint sufficiently to collect a different dataset. This does not just help with separating geometries of the scene but also creates more robust and discriminative classifiers. Active scene interaction by e.g., applying forces to objects enables the robot to start to reason about relationships between objects, such as "are two objects (inseparably) connected," or, find out other physical properties, like, "is the juice box full or empty," We are planning to add more complex actions and abilities to learn more object properties and have started to investigate how to determine an object's mechanical properties through interaction and observation (Dansereau et al., 2016).

4. DISCUSSION

Herein, we present our modular software framework applied in our research toward autonomous and adaptive robotic manipulation with humanoids. A tightly integrated sensorimotor system, based on subsystems developed over the past years, enables a basic level of eye–hand coordination on our robots. The robot detects objects, placed at random positions on a table, and performs a visually guided reaching before executing a simple grasp.

Our implementation enables the robot to adapt to changes in the environment. It safeguards complex humanoid robots, such as the *iCub*, from unwanted interactions – i.e., collisions with the environment or itself. This is performed by integrating the visual system with the motor side by applying attractor dynamics based on the robot's pose and a model of the world. We achieve a level of integration between visual perception and actions not previously seen on the *iCub*. Our approach, while comparable to visual servoing, has the advantage of being completely modular and the ability to take collisions (and other constraints) into account.

The framework has grown over recent time and has been used in a variety of experiments mainly with the *iCub* humanoid robot. It has since then been ported in parts to work with ROS with the aim of running pick and place experiments on Baxter; the code will be made available on the authors webpage at: http://juxi.net/projects/VisionAndActions/. The overarching goal was to enable a way of controlling a complex humanoid robot, which combines motion planning with low-level reflexes from visual feedback. *icVision* provides the detection and localization of objects in the visual stream. For example, it will provide the location of a chess board on a table in front of the robot. It can also provide the position of chess pieces to the world model. Based on this, an agent can plan a motion to pick up a specific piece. During the execution of that motion, *MoBeE* calculates forces for each chess piece, attracting for the target piece, repelling forces for all the other pieces. These forces are updated whenever a new object (or object location) is perceived, yielding a more robust execution of the motion due to a better coordination between vision and action.

The current system consists of a mix of pre-defined and learned parts, in the future, we plan to integrate further machine learning techniques to improve the object manipulation skills of robotic systems. For example, learning to plan around obstacles, including improved prediction and selection of actions. This will lead to a more adaptive, versatile robot, being able to work in unstructured, cluttered environments. Furthermore, it might be of interest to investigate an even tighter sensorimotor coupling,

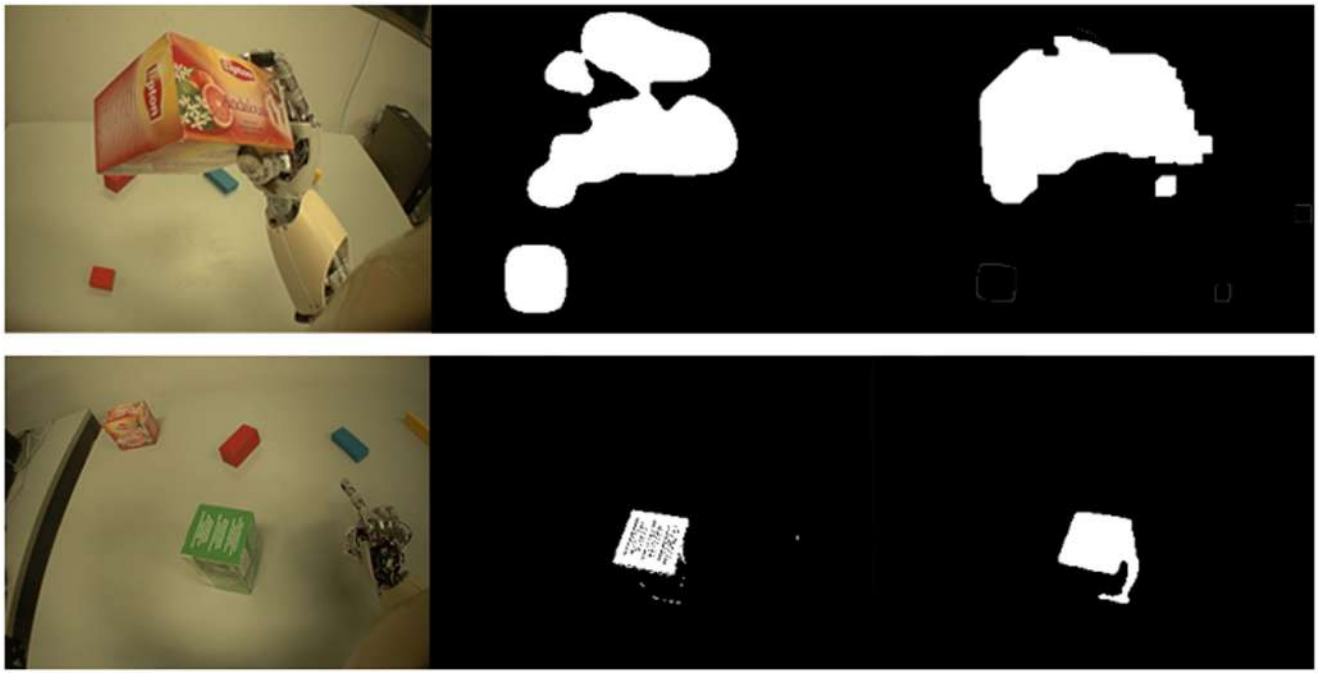


FIGURE 14 | Segmentation improvements for two objects after interaction. On the left, the robot's view of the scene. The middle column shows the first segmentation generated from the "static scene baseline." The last column shows the improved segmentation after learning continued with new images collected during the manipulation of the object.

e.g., by working directly in the image space – similar to image-based visual servoing approaches (Chaumette and Hutchinson, 2006) – this way avoiding to translate 2D image features into operational space locations.

In the future, we are aiming to extend the capabilities to allow for the quick end-to-end training of reaching (Zhang et al., 2015) and manipulation tasks (Levine et al., 2015), as well as, easy transition from simulation to real-world experiments. We are also looking at developing agents that interface this framework to learn the robot's kinematics and adapt to changes occurring due to malfunction or wear, leading to self calibration of a robot's eye–hand coordination.

AUTHOR CONTRIBUTIONS

JL is the main contributor both in the research and software integration of the framework. SH designed the CGP-IP software framework and the related experiments. AF designed and contributed to the research experiments on the iCub. PC contributed to the manuscript and the research experiments on Baxter.

ACKNOWLEDGMENTS

The authors would like to thank Mikhail Frank, Marijn Stollenga, Leo Pape, Adam Tow, and William Chamberlain for their discussions and valued inputs to this paper and the underlying software frameworks presented herein.

FUNDING

Various European research projects (IM-CLeVeR #FP7-IST-IP-231722, STIFF #FP7-IST-IP-231576) and the Australian Research Council Centre of Excellence for Robotic Vision (#CE140100016).

REFERENCES

Ambrose, R., Wilcox, B., Reed, B., Matthies, L., Lavery, D., and Korsmeyer, D. (2012). *NASA's Space Technology Roadmaps (STRs): Robotics, Tele-Robotics, and Autonomous Systems Roadmap*. Technical Report. Washington, DC: National Aeronautics and Space Administration (NASA).

Ando, N., Suehiro, T., and Kotoku, T. (2008). "A software platform for component based RT-system development: Openrtm-aist," in *Simulation, Modeling, and Programming for Autonomous Robots*, eds S. Carpin, I. Noda, E. Pagello, M. Reggiani and O. von Stryk (Springer), 87–98.

Bay, H., Tuytelaars, T., and Van Gool, L. (2006). "SURF: speeded up robust features," in *Computer Vision ECCV 2006*, eds A.Leonardis, H.Bischof, and A.Pinz (Berlin Heidelberg: Springer), 404–417.

Berthier, N., Clifton, R., Gullapalli, V., McCall, D., and Robin, D. (1996). Visual information and object size in the control of reaching. *J. Mot. Behav.* 28, 187–197. doi:10.1080/00222895.1996.9941744

Brooks, R. (1991). Intelligence without representation. *Artif. Intell.* 47, 139–159. doi:10.1016/0004-3702(91)90053-M

Brown, E., Rodenberg, N., Amend, J., Mozeika, A., Steltz, E., Zakin, M., et al. (2010). Universal robotic gripper based on the jamming of granular

material. *Proc. Natl. Acad. Sci. U.S.A.* 107, 18809–18814. doi:10.1073/pnas.1003250107

Brugali, D. (2007). *Software Engineering for Experimental Robotics*, Vol. 30. Berlin; Heidelberg: Springer.

Carbone, G. (2013). *Grasping in Robotics, Volume 10 of Mechanisms and Machine Science*. London: Springer.

Chamberlain, W., Leitner, J., and Corke, P. (2016). "A distributed robotic vision service," in *Proceedings of the International Conference on Robotics and Automation (ICRA)*, Stockholm.

Chaumette, F., and Hutchinson, S. (2006). Visual servo control, part I: basic approaches. *IEEE Robot. Autom. Mag.* 13, 82–90. doi:10.1109/MRA.2006.250573

Ciliberto, C., Smeraldi, F., Natale, L., and Metta, G. (2011). "Online multiple instance learning applied to hand detection in a humanoid robot," in *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*. San Francisco, CA.

Cipolla, R., Battiato, S., and Farinella, G. M. (2010). *Computer Vision: Detection, Recognition and Reconstruction*, Vol. 285. Berlin Heidelberg: Springer.

Corke, P. (2011). *Robotics, Vision and Control, Volume 73 of Springer Tracts in Advanced Robotics*. Berlin; Heidelberg: Springer.

Dansereau, D. G., Singh, S. P. N., and Leitner, J. (2016). *Proceedings of the International Conference on Robotics and Automation (ICRA)*, Stockholm.

Davison, A. J., and Murray, D. W. (2002). Simultaneous localization and map-building using active vision. *IEEE Trans. Pattern Anal. Mach. Intell.* 24, 865–880. doi:10.1109/TPAMI.2002.1017615

De Santis, A., Albu-Schäffer, A., Ott, C., Siciliano, B., and Hirzinger, G. (2007). "The skeleton algorithm for self-collision avoidance of a humanoid manipulator," in *Advanced Intelligent Mechatronics, 2007 IEEE/ASME International Conference on* (Zürich: IEEE), 1–6.

Diankov, R., and Kuffner, J. (2008). *Openrave: A Planning Architecture for Autonomous Robotics*. Tech. Rep. CMU-RI-TR-08-34. Pittsburgh, PA: Robotics Institute, 79.

Dietrich, A., Wimbock, T., Taubig, H., Albu-Schaffer, A., and Hirzinger, G. (2011). "Extensions to reactive self-collision avoidance for torque and position controlled humanoids," in *Proceedings of the International Conference on Robotics and Automation (ICRA)* (Shanghai: IEEE), 3455–3462.

Elkady, A., and Sobh, T. (2012). Robotics middleware: a comprehensive literature survey and attribute-based bibliography. *J. Robot.* 2012. doi:10.1155/2012/959013

Fanello, S. R., Ciliberto, C., Natale, L., and Metta, G. (2013). "Weakly supervised strategies for natural object recognition in robotics," in *Proceedings of the International Conference on Robotics and Automation (ICRA)*. Karlsruhe.

Fitzpatrick, P., Metta, G., and Natale, L. (2008). Towards long-lived robot genes. *Rob. Auton. Syst.* 56, 29–45. doi:10.1016/j.robot.2007.09.014

Forssberg, H., Eliasson, A., Kinoshita, H., Johansson, R., and Westling, G. (1991). Development of human precision grip I: basic coordination of force. *Exp. Brain Res.* 85, 451–457. doi:10.1007/BF00229422

Frank, M. (2014). *Learning to Reach and Reaching to Learn: A Unified Approach to Path Planning and Reactive Control through Reinforcement Learning*. Ph.D. thesis, Universitá della Svizzera Italiana, Lugano.

Frank, M., Leitner, J., Stollenga, M., Förster, A., and Schmidhuber, J. (2014). Curiosity driven reinforcement learning for motion planning on humanoids. *Front. Neurorobot.* 7:25. doi:10.3389/fnbot.2013.00025

Gerkey, B., Vaughan, R. T., and Howard, A. (2003). "The player/stage project: tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th International Conference on Advanced Robotics, Volume 1* (Coimbra), 317–323.

Gori, I., Fanello, S., Odone, F., and Metta, G. (2013). "A compositional approach for 3D arm-hand action recognition," in *Proceedings of the IEEE Workshop on Robot Vision (WoRV)*. Clearwater, FL.

Gupta, K. (1986). Kinematic analysis of manipulators using the zero reference position description. *Int. J. Robot. Res.* 5, 5. doi:10.1177/027836498600500202

Harding, S., Leitner, J., and Schmidhuber, J. (2013). "Cartesian genetic programming for image processing," in *Genetic Programming Theory and Practice X, Genetic and Evolutionary Computation*, eds R.Riolo, E.Vladislavleva, M. D.Ritchie, and J. H.Moore (New York; Ann Arbor: Springer), 31–44.

Hart, S., Ou, S., Sweeney, J., and Grupen, R. (2006). "A framework for learning declarative structure," in *Proceedings of the RSS Workshop: Manipulation in Human Environments*. Philadelphia, PA.

Hartley, R., and Zisserman, A. (2000). *Multiple View Geometry in Computer Vision*, 2nd Edn. Cambridge, UK: Cambridge University Press.

Hutchinson, S., Hager, G. D., and Corke, P. I. (1996). A tutorial on visual servo control. *IEEE Trans. Robot. Automat.* 12, 651–670. doi:10.1109/70.538972

Jackson, J. (2007). Microsoft robotics studio: a technical introduction. *IEEE Robot. Autom. Mag.* 14, 82–87. doi:10.1109/M-RA.2007.905745

Jeannerod, M. (1997). *The Cognitive Neuroscience of Action*. Blackwell Publishing. Available at: http://au.wiley.com/WileyCDA/WileyTitle/productCd-0631196048.html

Johnson, M. H., and Munakata, Y. (2005). Processes of change in brain and cognitive development. *Trends Cogn. Sci.* 9, 152–158. doi:10.1016/j.tics.2005.01.009

Karlsson, N., Di Bernardo, E., Ostrowski, J., Goncalves, L., Pirjanian, P., and Munich, M. (2005). "The vSLAM algorithm for robust localization and mapping," in *Proceedings of the International Conference on Robotics and Automation (ICRA)*. Barcelona.

Kavraki, L. E., Švestka, P., Latombe, J.-C., and Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Robot. Autom. Mag.* 12, 566–580. doi:10.1109/70.508439

Kemp, C., Edsinger, A., and Torres-Jara, E. (2007). Challenges for robot manipulation in human environments [grand challenges of robotics]. *IEEE Robot. Autom. Mag.* 14, 20–29. doi:10.1109/MRA.2007.339604

Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots. *Int. J. Robot. Res.* 5, 90. doi:10.1177/027836498600500106

Kragic, D., and Vincze, M. (2009). Vision for robotics. *Found. Trends Robot.* 1, 1–78. doi:10.1561/2300000001

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature* 521, 436–444. doi:10.1038/nature14539

Leitner, J. (2014). *Towards Adaptive and Autonomous Humanoid Robots: From Vision to Actions*. Ph.D. thesis, Universitá della Svizzera italiana, Lugano.

Leitner, J. (2015). "Chapter 10, a bottom-up integration of vision and actions to create cognitive humanoids," in *Cognitive Robotics* (CRC Press), 191–214. Available at: https://www.crcpress.com/Cognitive-Robotics/Samani/9781482244564

Leitner, J., Chandrashekhariah, P., Harding, S., Frank, M., Spina, G., Förster, A., et al. (2012a). "Autonomous learning of robust visual object detection and identification on a humanoid," in *Proceedings of the International Conference on Development and Learning and Epigenetic Robotics (ICDL)*. San Diego, CA

Leitner, J., Harding, S., Frank, M., Förster, A., and Schmidhuber, J. (2012b). Learning spatial object localization from vision on a humanoid robot. *Int. J. Adv. Robot. Syst.* 9.

Leitner, J., Harding, S., Frank, M., Förster, A., and Schmidhuber, J. (2012c). "Transferring spatial perception between robots operating in a shared workspace," in *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*. Villamoura.

Leitner, J., Förster, A., and Schmidhuber, J. (2014a). "Improving robot vision models for object detection through interaction," in *International Joint Conference on Neural Networks (IJCNN)*. Beijing.

Leitner, J., Frank, M., Förster, A., and Schmidhuber, J. (2014b). "Reactive reaching and grasping on a humanoid: towards closing the action-perception loop on the icub," in *Proceedings of the International Conference on Informatics in Control, Automation and Robotics (ICINCO)* (Vienna), 102–109.

Leitner, J., Harding, S., Chandrashekhariah, P., Frank, M., Förster, A., Triesch, J., et al. (2013a). "Learning visual object detection and localization using icVision," in *Biologically Inspired Cognitive Architectures 2012. Extended versions of selected papers from the Third Annual Meeting of the BICA Society (BICA 2012)*, eds. A. Chella, R. Pirrone, R. Sorbello and R. K. Jóhannsdóttir (Berlin Heidelberg: Springer), 29–41.

Leitner, J., Harding, S., Frank, M., Förster, A., and Schmidhuber, J. (2013b). "ALife in humanoids: developing a framework to employ artificial life techniques for high-level perception and cognition tasks on humanoid robots," in *Workshop on 'Artificial Life Based Models of Higher Cognition' at the European Conference on Artificial Life (ECAL)*. Taormina

Leitner, J., Harding, S., Frank, M., Förster, A., and Schmidhuber, J. (2013c). "An integrated, modular framework for computer vision and cognitive robotics research (icVision)," in *Biologically Inspired Cognitive Architectures 2012, Volume 196 of Advances in Intelligent Systems and Computing*, eds A. Chella, R. Pirrone, R. Sorbello, and K. Jóhannsdóttir (Berlin; Heidelberg: Springer), 205–210.

Leitner, J., Harding, S., Frank, M., Förster, A., and Schmidhuber, J. (2013d). "Artificial neural networks for spatial perception: towards visual object localisation in humanoid robots," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)* (Dallas, TX: IEEE), 1–7.

Leitner, J., Harding, S., Frank, M., Förster, A., and Schmidhuber, J. (2013e). "Humanoid learns to detect its own hands," in *Proceedings of the IEEE Conference on Evolutionary Computation (CEC)* (Cancun), 1411–1418.

Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2016). End-to-End training of deep visuomotor policies. *J. Mac. Learn. Res.* 17, 1–40.

Lowe, D. (1999). "Object recognition from local scale-invariant features," in *Proceedings of the International Conference on Computer Vision (ICCV)* Kerkyra.

Maitin-Shepard, J., Cusumano-Towner, M., Lei, J., and Abbeel, P. (2010). "Cloth grasp point detection based on multiple-view geometric cues with application to robotic towel folding," in *Proceedings of the International Conference on Robotics and Automation (ICRA)* (Anchorage, AK), 2308–2315.

Marchand, E., Spindler, F., and Chaumette, F. (2005). Visp for visual servoing: a generic software platform with a wide class of robot control skills. *IEEE Robot. Autom. Mag.* 12, 40–52. doi:10.1109/MRA.2005.1577023

McCarty, M., Clifton, R., Ashmead, D., Lee, P., and Goubet, N. (2001). How infants use vision for grasping objects. *Child Dev.* 72, 973–987. doi:10.1111/1467-8624.00329

Meltzoff, A. (1988). Infant imitation after a 1-week delay: long-term memory for novel acts and multiple stimuli. *Dev. Psychol.* 24, 470. doi:10.1037/0012-1649.24.4.470

Metta, G., Fitzpatrick, P., and Natale, L. (2006). YARP: yet another robot platform. *Int. J. Adv. Robot. Syst.* 3, 43–48. doi:10.5772/5761

Miller, J. (1999). "An empirical study of the efficiency of learning Boolean functions using a Cartesian genetic programming approach," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)* (Orlando, FL), 1135–1142.

Miller, J. F. (ed.). (2011). *Cartesian Genetic Programming: Natural Computing Series*. Berlin; Heidelberg: Springer.

Oztop, E., Bradley, N., and Arbib, M. (2004). Infant grasp learning: a computational model. *Exp. Brain Res.* 158, 480–503. doi:10.1007/s00221-004-1914-1

Pathak, S., Pulina, L., Metta, G., and Tacchella, A. (2013). "Ensuring safety of policies learned by reinforcement: reaching objects in the presence of obstacles with the icub," in *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*. Tokyo.

Pattacini, U. (2011). *Modular Cartesian Controllers for Humanoid Robots: Design and Implementation on the iCub*. Ph.D. thesis, Italian Institute of Technology, Genova.

Plumert, J., and Spencer, J. (2007). *The Emerging Spatial Mind*. Oxford: Oxford University Press.

Posner, M. (1989). *Foundations of Cognitive Science*. Cambridge, MA: The MIT Press.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., et al. (2009). "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*. Kobe.

Sadikov, A., Možina, M., Guid, M., Krivec, J., and Bratko, I. (2007). "Automated chess tutor," in *Computers and Games, Volume 4630 of Lecture Notes in Computer Science*, Vol. 13–25, eds H.Herik, P.Ciancarini, and H.Donkers (Berlin; Heidelberg: Springer), 13–25.

Saxena, A., Driemeyer, J., and Ng, A. (2008). Robotic grasping of novel objects using vision. *Int. J. Robot. Res.* 27, 157. doi:10.1177/0278364907087172

Schmidhuber, J. (2015). Deep learning in neural networks: an overview. *Neural Netw.* 61, 85–117. doi:10.1016/j.neunet.2014.09.003

Schoner, G., and Dose, M. (1992). A dynamical systems approach to task-level system integration used to plan and control autonomous vehicle motion. *Rob. Auton. Syst.* 10, 253–267. doi:10.1016/0921-8890(92)90004-I

Soetens, P. (2006). *A Software Framework for Real-Time and Distributed Robot and Machine Control*. Ph.D. thesis, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium. Available at: http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf

Stollenga, M., Pape, L., Frank, M., Leitner, J., Förster, A., and Schmidhuber, J. (2013). "Task-relevant roadmaps: a framework for humanoid motion planning," in *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, Tokyo.

Stückler, J., Badami, I., Droeschel, D., Gräve, K., Holz, D., McElhone, M., et al. (2013). "Nimbro@home: winning team of the robocup@home competition 2012," in *Robot Soccer World Cup XVI* (Berlin-Heidelberg: Springer), 94–105.

Vahrenkamp, N., Wächter, M., Kröhnert, M., Welke, K., and Asfour, T. (2015). The robot software framework armarx. *Inform. Tech.* 57, 99–111. doi:10.1515/itit-2014-1066

Vahrenkamp, N., Wieland, S., Azad, P., Gonzalez, D., Asfour, T., and Dillmann, R. (2008). "Visual servoing for humanoid grasping and manipulation tasks," in *Proceedings of the International Conference on Humanoid Robots* (Daejon), 406–412.

van den Bergen, G. (2004). *Collision Detection in Interactive 3D Environments*, Taylor & Francis Group, 277.

Verschae, R., and Ruiz-del Solar, J. (2015). Object detection: current and future directions. *Front. Robot. AI* 2:29. doi:10.3389/frobt.2015.00029

Welke, K., Issac, J., Schiebener, D., Asfour, T., and Dillmann, R. (2010). "Autonomous acquisition of visual multi-view object representations for object recognition on a humanoid robot," in *IEEE International Conference on Robotics and Automation (ICRA)* (Anchorage, AK: IEEE), 2012–2019.

Zhang, F., Leitner, J., Milford, M., Upcroft, B., and Corke, P. (2015). "Towards vision-based deep reinforcement learning for robotic motion control," in *Australasian Conference on Automation and Robotics (ACRA)*, Canberra.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

frontiers in Robotics and AI

ORIGINAL RESEARCH
published: 10 May 2016
doi: 10.3389/frobt.2016.00025

The Walk-Man Robot Software Architecture

Mirko Ferrati[1], Alessandro Settimi[1,2], Luca Muratore[2], Alberto Cardellino[3], Alessio Rocchi[2], Enrico Mingo Hoffman[2], Corrado Pavan[1], Dimitrios Kanoulas[2], Nikos G. Tsagarakis[2], Lorenzo Natale[3] and Lucia Pallottino[1]*

[1] Centro di Ricerca "E. Piaggio", University of Pisa, Pisa, Italy, [2] Department of Advanced Robotics (ADVR), Istituto Italiano di Tecnologia, Genova, Italy, [3] Department of Robotics, Brain and Cognitive Sciences (RBCS), Istituto Italiano di Tecnologia, Genova, Italy

A software and control architecture for a humanoid robot is a complex and large project, which involves a team of developers/researchers to be coordinated and requires many hard design choices. If such project has to be done in a very limited time, i.e., less than 1 year, more constraints are added and concepts, such as modular design, code reusability, and API definition, need to be used as much as possible. In this work, we describe the software architecture developed for Walk-Man, a robot participant at the Darpa Robotics Challenge. The challenge required the robot to execute many different tasks, such as walking, driving a car, and manipulating objects. These tasks need to be solved by robotics specialists in their corresponding research field, such as humanoid walking, motion planning, or object manipulation. The proposed architecture was developed in 10 months, provided boilerplate code for most of the functionalities required to control a humanoid robot and allowed robotics researchers to produce their control modules for DRC tasks in a short time. Additional capabilities of the architecture include firmware and hardware management, mixing of different middlewares, unreliable network management, and operator control station GUI. All the source code related to the architecture and some control modules have been released as open source projects.

Keywords: software architecture, humanoid robot, modular design, DRC challenge, teleoperation, robotic middlewares, emergency response

OPEN ACCESS

Edited by:
Giuseppe Carbone, University of Cassino and South Latium, Italy

Reviewed by:
Paolo Boscariol, University of Udine, Italy
Matthias Rolf, Oxford Brookes University, Japan

****Correspondence:***
Mirko Ferrati
mirko.ferrati@gmail.com

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *26 November 2015*
Accepted: *12 April 2016*
Published: *10 May 2016*

Citation:
Ferrati M, Settimi A, Muratore L, Cardellino A, Rocchi A, Mingo Hoffman E, Pavan C, Kanoulas D, Tsagarakis NG, Natale L and Pallottino L (2016) The Walk-Man Robot Software Architecture. Front. Robot. AI 3:25. doi: 10.3389/frobt.2016.00025

1. INTRODUCTION

In this paper, we describe the design decisions and the resulting software architecture of the Walk-Man robot, developed for the participation to the DARPA Robotics Challenge (DRC).

The goal of the DRC was to develop robots (not necessarily humanoid) capable to operate in a disaster scenario and to perform tasks, such as search and rescue, usually done by humans. During the challenge, the robot was required to perform different tasks, such as walk, drive a car, grasp and use objects, open doors, and rotate valves. The operator was located far from the robot, without line-of-sight, and not necessarily with a high bandwidth connection to the robot, so that direct teleoperation was not possible and a semi-autonomous approach was required. Thus, the operator was responsible for choosing the order and the timing of commands to solve the DRC tasks, depending on the level of autonomy of the robot. For example, the main task of opening a door might be handled by the operator with the following actions: reach the door handle, grasp it, turn and finally release it, or just with a single command: "open that door."

The Walk-Man team was composed by engineers and technicians with different background and fields of expertise ranging from compliant manipulation, walking pattern generation, control, to artificial vision. Each contribution have been integrated in the software architecture thanks to the proposed design so that also non-experts in software engineering and code development were able to develop dedicated modules. This choice has avoided the necessity of training the whole team and, hence, reduced the time effort. In this paper, we use the terms "software users" and "control module developers" interchangeably to refer to the users of the software architecture we developed. Our design choices are motivated by the needs of such users, and they aim to maximize the output of any developer through a well-tailored software infrastructure, inter-process communication facilities, and shared libraries with various tools.

While some of our implementation decisions may not apply to large long-term projects, we followed many principles that are in common with the state of the art of robotic software development: specifically, we adopted a component-based approach that focuses on modularity to support code reuse and rapid development (Brugali and Shakhimardanov, 2010), for example, with similar requirements in the ROS ecosystem refer to Coleman et al. (2014) and Walck et al. (2014).

Following the standard approach of YARP (Metta et al., 2006) and ROS (Quigley et al., 2009) middlewares, we decided to build a distributed network of applications (nodes), so that each different process is independent and users can have more freedom when developing their own modules. To save development time and to focus on the specific tasks for the DRC, we relied as much as possible on software components available within existing frameworks. A first design choice has been the development of a *Generic YARP Module* (GYM) to provide a set of libraries for control purpose both to enhance code reuse and to have a common interface to manage the modules execution flow.

We decided to adopt the YARP middleware for the development of our software architecture, and in particular for the design of the software interface between the robot hardware and the nodes related to motor control. This choice was motivated by our direct expertise in the development of YARP and because YARP has proved to be quite reliable in experimental settings (Hammer and Bäuml, 2013). In addition, YARP provides functionalities for setting channel prioritization using QoS and different communication protocols. These features, at the time of writing, are not yet available with ROS (although the upcoming version of ROS will provide similar functionalities with the adoption of Data Distribution Service at the transport layer).

To get advantage of the large codebase available in the ROS ecosystem, we designed a mixed architecture that integrates ROS nodes. In the final architecture, ROS was used in the high-level operator GUI and for the 3D perception. The operator graphic interface is a fundamental component of the architecture, it allows the remote control of the robot by enriching the pilot awareness with the data coming from the robot. The single components of the GUI inherit basic functionalities from a *Generic Widget*, i.e., the graphical interface of a GYM.

Similar works have been developed by teams participating in the DRC Trials, such as Johnson et al. (2015) and Yi et al. (2015), and Hebert et al. (2015). Most of these works have a custom low-level communication library, or middleware, which ensures a real-time control loop and a high level inter-process communication system (such as ROS, Orocos, OpenRTM, and PODO). Given the requirements on the network bandwidth imposed by DARPA, a custom manager was used to connect the operator control station to the robot computer, usually using TCP and UDP protocols without any abstraction layer, with two middleware servers (e.g., RosCores) in the operator station and the robot. The same solution has been adopted by the Walk-Man team whose architecture is based both on ROS and YARP that did not properly handle unreliable channels at the time of the DRC. Indeed, centralized servers are limiting for unreliable networks, and a custom bridge communicating with different reliable networks (robot, pilot station, etc.) using a TCP/UDP protocol is required. To cope with such problems, a custom network bridge that handles both protocols with a custom, optimized serialization of messages has been developed.

The aim of this paper is to describe the software with particular attention on how the proposed architecture helped the Walk-Man team and, in turn, how the team feedback affected the architecture design. The main contributions of this paper are as follows:

- a generic module template that captures a development pattern of robot control modules, avoiding the need to write the same boilerplate code multiple times in each module,
- an hybrid communication middleware architecture that includes ROS and YARP, along with a custom bridge used to handle both unreliable networks and environments with multiple nameserver (i.e., roscore and yarpserver), and
- the integration of the generic module template into the operator GUI, which is developed as a generic reconfigurable GUI capable of adapting to the DRC tasks as well as to future demos and lab projects.

The source code for the software described in this work can be found here: https://gitlab.robotology.eu/groups/walkman-drc

1.1. Robot Platform

The Walk-Man robot (see **Figure 1**) is a humanoid robot with 33 DoFs, each actuated by an electric series elastic actuator whose design is described in Negrello et al. (2015). Each motor is controlled by its own electronic board at a frequency of 1 kHz. These boards are connected to a shared *ethercat* network with 1 Gb bandwidth, used to send and receive joint position reference, along with other information such as temperatures, torques, and PID values. Five additional electronic boards in the ethercat network provide readings from the robot IMU and the four Force/Torque sensors, located in the wrists and ankles. A *control pc* with a quad-core i7-3612 runs the control software and is configured as the ethercat master.

Finally, a Multisense SL head, which contains a stereocamera and a LIDAR, is connected through its own 1 Gb network to a

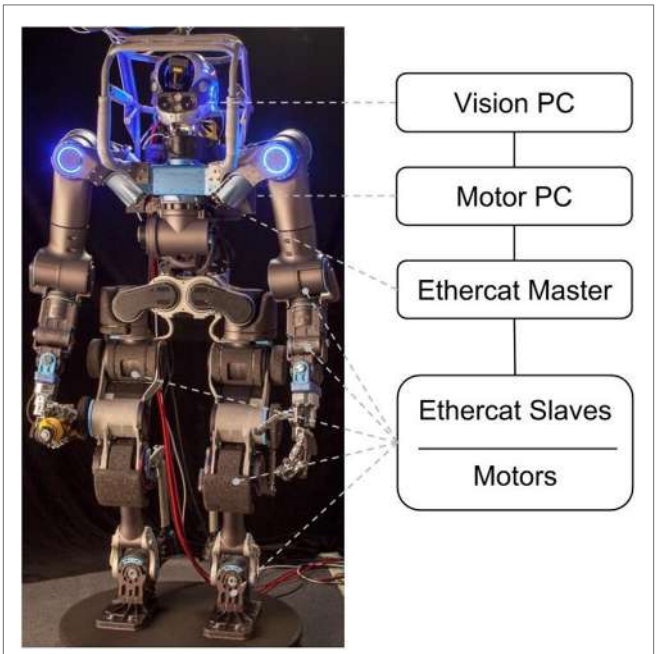


FIGURE 1 | The Walk-Man robot and the main network configuration.

vision pc, with the same hardware components as the control pc. An external *pilot pc* is connected through a wireless network to the control pc and vision pc.

The operating system on all the computers is Ubuntu 14.04 modified with Xenomai, all the code is written in C++ except for the firmware, which uses a subset of C. See **Figure 2** for an overview of the hardware and networks. It is worth mentioning that the whole robotic platform has been assembled for the first time 4 months prior of the DRC. As a consequence, all the motion control software tests on the hardware have been delayed until few weeks before the DRC. On the other hand, at the time of assembly, the proposed architecture was in an advanced stage of development and testing. Such asynchronous development of software control algorithms and the architecture has led to the necessity of a highly flexible and modular implementation of the latter.

1.2. Design Choices Overview

In this section, the strategies used to design the Walk-Man software architecture. A complete software stack has been built for the DRC consisting in a custom firmware, control modules tackling different tasks, a remote pilot graphical interface, and the whole architecture to manage and connect the different applications.

Due to the limited time constraint (around 10 months) and the variety of programing skills among our robotics researchers, our design choices were oriented to:

- avoid code duplicates and enhance code reuse;
- provide common shared C++ classes and utilities to the software users;
- ease and speed up the production of significant code by hiding code complexity in simple APIs;
- fast testing and debugging leveraging on simulators.

Following these principles, our core developers focused on low level interfaces, middleware management, and network and performance optimization.

We devised a layered component-based architecture, where each task of the DRC is handled by a single control module and modules interact with the hardware and each other through well-defined APIs. Once a rough and primitive API was defined, modules could be developed in parallel; in the meantime, shared functionalities could be improved under the hood of the high level control software without requiring code changes.

The YARP middleware has been chosen to obtain an abstraction layer for the hardware of the robot (sensors and motors) together with the set of interfaces. This abstraction layer allows to write code that can seamlessly interface to simulators or to the real robot (either remotely through the network or on the same machine using inter process communication). The initial phase of the development focused on the implementation of this abstraction layer for the simulated robot in Gazebo (Hoffman et al., 2014). This allowed to start performing experiments early on during the project. The same interface was implemented on the real robot allowing to transfer the code developed on the simulator with only minimal parameters changes.

2. SOFTWARE ARCHITECTURE

The Walk-Man architecture has been organized into four software layers (see **Figure 3**).

- The top layer is the operator control unit, named *pilotInterface*.
- A network bridge connects the pilot to the robot, where various *control and perception modules* form another layer.
- An *hardware abstraction* layer remotizes the robot hardware and provides to the control modules a set of shared libraries (*GYM*) used to interact with the remote driver, called *Ethercat Master*.
- The lowest layer is represented by the firmware running in embedded boards, each controlling one actuator.

2.1. Firmware-Ethercat

At the lowest level, each joint of Walk-Man is controlled by a PID position loop in a distributed embedded electronic system with one board per joint. Our main aim was to have a hard real-time loop in the firmware: the execution time of each firmware function was measured and tuned so that a 1 kHz loop could be implemented. In the software architecture, we developed for the Coman platform, the communication from the control pc to each board was performed on an ethernet BUS using a combination of UDP and TCP packets. The lack of synchronization between boards led to conflicts and consequent packets loss with UDP and delays with TCP protocols. We decided to move to an ethercat implementation, which allows synchronized communication and, therefore, much better control on the data traveling on the BUS. In particular, we measured the maximum number of bytes that each board

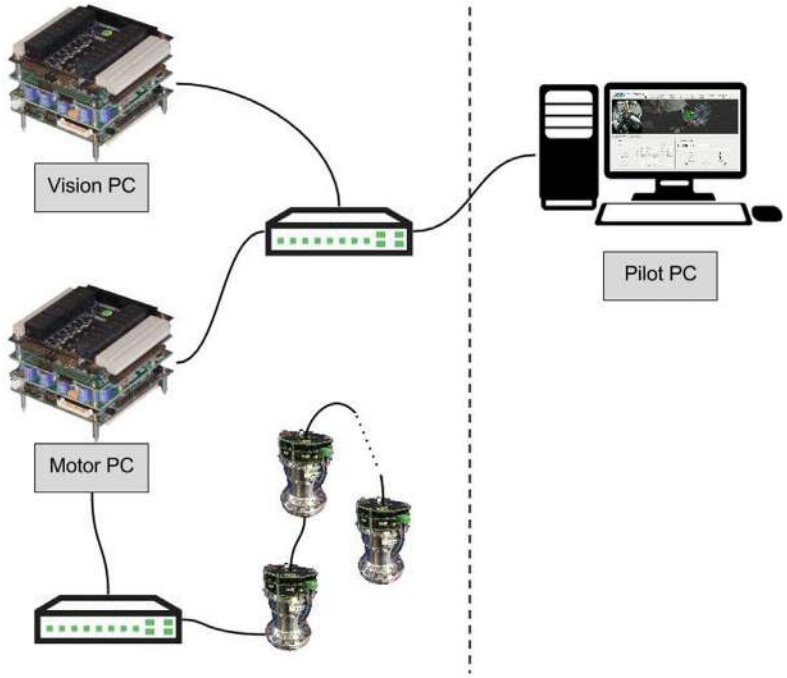


FIGURE 2 | **Walk-Man network**.

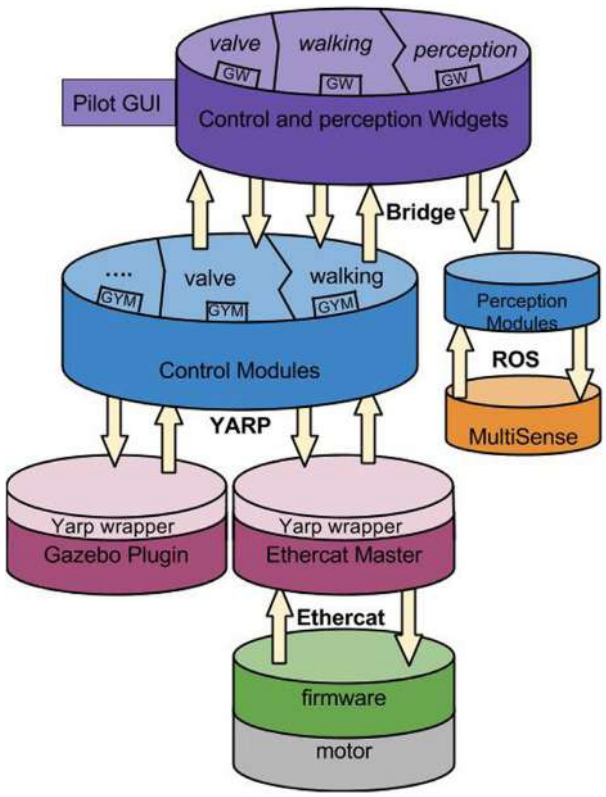


FIGURE 3 | **Complete view of the software architecture**.

Control board data Tx	
Position desired	4
Fault	2
PGain	2
DGain	2
Timestamp	2

Control board data Rx	
Position link	4
Reference pos	4
Max Temperature	2
Torque link	2
Fault	2
Round Trip Time	2

Sensor board data	
Fx	4
Fy	4
Fz	4
Tx	4
Ty	4
Tz	4
Fault	2
Round Trip Time	2
Timestamp	2

FIGURE 4 | **Dimension of various packet fields (bytes)**.

could handle at 1 kHz and defined a standard packet size with standard information, as shown in **Figure 4**.

This standard packet definition is an example of the various interfaces between software levels that will be described in this work and that allow software decoupling and testing.

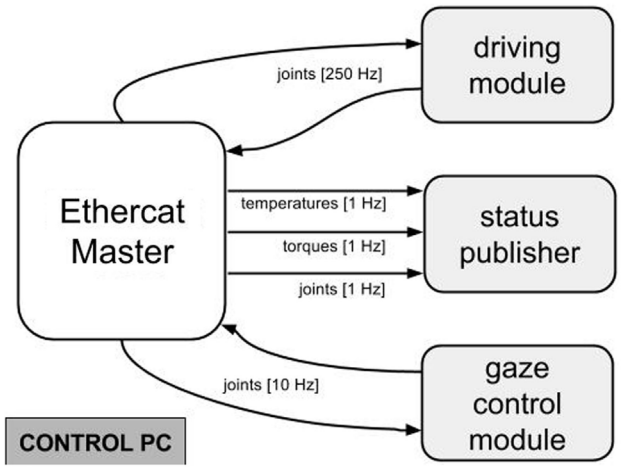


FIGURE 5 | As an example, during the DRC driving task three modules were using joint readings at 250, 10, and 1 Hz, one required torques and temperatures at 10 Hz, two were controlling the joints at 250 and 10 Hz.

2.2. Ethercat Master – YARP

In the real robot, the hardware manager runs on the control pc and is called *Ethercat Master*. It manages Ethercat slaves (i.e., the electronic boards), keeps them synchronized, and sends/receives position references in real-time.

The *Master* can be seen as a hardware robot driver, which handles low level communication and exposes a simpler and asynchronous API to the higher levels. Writing real-time code requires expertise that were not available in all the software developers. Therefore, a separation between the Master and the modules implementing higher-level behaviors has been introduced. This separation was achieved through the YARP middleware using the remotization functionalities that it provides for the robot abstraction layer. Usually, developers write software that communicates with the Master through the network; this has been achieved using asynchronous communication with the YARP middleware. This decoupling was beneficial because it allows stopping and starting modules without interfering with the Master. More importantly, it prevents modules that behave erratically to affect the real-time performance of the Master.

The Master creates an input and output YARP port for each control module and for each type of information required by them. In **Figure 5**, the modules running during the DRC driving task are reported together with communication frequencies.

2.3. Hardware/Simulation Abstraction Layer

The Ethercat-Master exposes the robot sensors and actuators in a YARP network by *remotizing* the robot with a set of YARP communication channels (this is achieved in YARP using special objects called *network wrappers*). An additional set of libraries, named *WholeBodyInterface*, hides YARP channels from control modules, and relieves the developers from the bureaucracy required to prepare and parse the messages to and from the robot.

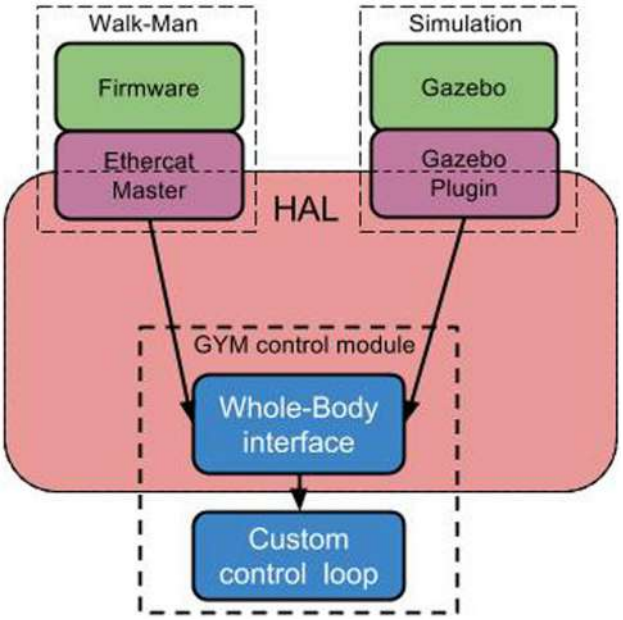


FIGURE 6 | Detailed low level software stack, including robot hardware abstractions and a whole-body class used by the Generic Yarp Module.

The composition of the YARP wrapper in the Ethercat-Master and the whole-body libraries realizes a two-tier Habstraction Layer (HAL) for the robot. This abstraction layer between the hardware driver and the control modules allowed us to easily switch between simulation and the real robot, since the Gazebo plug-ins for the Walk-Man robot implements exactly the same YARP classes and interfaces as the Master (see **Figure 6**).

In the simulation case, the Gazebo Plugin substitutes the HAL standalone application and it is fully compatible with the same set of shared libraries.

The two-tier abstraction layer implements a whole-body interface on top of the robot interface defined by YARP. The main difference between the two layers is that the latter separates joints in kinematic chains and implements interfaces for individual sensors; for practical reasons, the logical separation of the kinematic chains at this level is subject to fluctuations (for example, it affects how joint states are broadcast on the network). The whole-body interface groups all joints and associated sensors in a single kinematic chain. The advantage of this separation is that it exposes to the user the whole-body interface, which is stable because it is defined solely by the number of joints of the robot.

As an extreme example, 15 days before the DRC, we had to intentionally break the functions responsible for moving the robot joints. To reduce resource usage (and reduce jitter due to CPU overload), we changed how joints are grouped and transmitted on the network; all the required changes affected the YARP abstraction layer and remain limited to the implementation of the whole-body interface. All the user code remained untouched. The simulation, the real robot, and all the control modules were updated in just 2 days.

As suggested by Johnson et al. (2015), we fully understand (and wish) that in a long-term project APIs must not be modified, especially few days before the demo. However, we are convinced that, in a research environment APIs may need to be changed in critical moments, and the proposed approach is a way to mitigate the effect of such changes.

Finally, an advantage of this two-layer architecture is that it separates control modules from the middleware. This will allow to change the communication layer (i.e., the middleware) without affecting the control code.

2.4. Generic Control Module Template

A control module software can be summarized as a *sense-compute-move* loop, where *sense* receives all the inputs from the robot, the inputs are used by *compute* in order to implement the control law of the module. Finally, *move* sends to the robot the newly computed desired position of the joints. In reality, developers usually spend a part of development effort into initialization code: i.e., reading control parameters, starting the communication facilities, reading a description of the robot kinematics, and so on. We provided explicit support for this implementation pattern in the Generic YARP Module (GYM). The GYM has been designed as a C++ abstract class that provides a common and standard way to execute these initialization steps, along with a *sense* and *move* default implementation that provide boilerplate code required to initialize the YARP remotization interfaces. The source code of GYM can be found here: https://github.com/robotology-playground/GYM

GYM functions handle all the required YARP communication between a module, the Master, and the PilotInterface, effectively hiding YARP communication mechanisms and classes. GYM was iteratively improved driven by the effort to remove duplicated code across modules and based on the team feedback (10 developers) which helped revising the specifications and debugging.

Our experience showed that the adoption of GYM reduced duplicated code significantly. In addition GYM provides another separation between the code and the middleware. In fact, a Generic ROS Module is currently in development and complies with the GYM API, so that any module using GYM could also be used in the ROS system.

GYM is organized in two threads: a watchdog running at 1 Hz and a main control loop running in a range of frequencies between 100 and 500 Hz (**Figure 7**). Developers can write their own code inside the control loop function *run()*, they also have access to a set of helper function providing a standard kinematic description of the robot based on the robot URDF. The watchdog thread is not customizable and listens for standard commands from the pilot, through one of the standard communication interfaces (*switch interface*) described in the next section.

The GYM C++ class that needs to be inherited by the user has the following signature:

```
class generic_thread
{
public:
   /**
    * @brief custom initialization function: called before
      run(),
    * must be overrided by sub-classes
    */
   virtual bool custom_init()=0;

   virtual bool custom_pause(){
     return true;
   }

virtual bool custom_resume(){
     return true;
   }

   /**
   * @brief loop function, called at the desired
   * frequency read from configuration file
   */
   virtual void run(){}

};
```

Notice that the user can override the default (empty) implementation of *pause* and *resume* functions so that he can take the required actions in order to save and resume the state of his own control module. Instead, to keep different modules organized in a similar structure, the *init* function was required to be implemented by the user and to contain all the initialization code. Moreover, with this approach, executables could be started in any moment, while the pilot kept the possibility of choosing when a module was going to be initialized and connected to the rest of the running software.

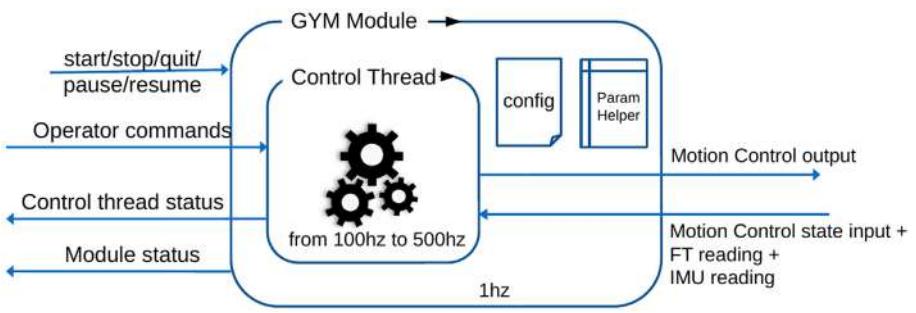


FIGURE 7 | Structure of the Generic YARP Module, with inputs and outputs from/to the pilot and the EthercatMaster.

In order to show some of the GYM library functions, we report a simple *run* implementation:

```
virtual void run()
{
   //get the command from the pilot
   command_interface.getCommand(cmd);

   //evolve the state machine accordingly to the received
     command
   current_state=sm.evolve_state_machine
     (current_state,cmd);

   //get updated joint values
   vector q_sensed,q_des;
   robot.sensePosition(q_sensed);

   //compute desidered joint values in a control law function
   q_des=control_law(q_sensed,current_state);

   //move the joints
   robot.move(q_des);

   //set the status to be streamed back to the pilot
   status_interface.setStatus(current_state);
}
```

The variable *robot* is provided by GYM and is used to interact with the hardware with simple functions such as *sensePosition* and *move*.[1]

Examples of what a complex implementation may do is to use multiple state machines depending on the cmd values, to read or ignore commands from the user, to selectively avoid sensing or moving the robot while planning a complex movement, or even to evolve a state machine automatically without requiring user commands.[2]

2.4.1. Communication Interfaces

One of the features implemented in GYM code is a set of communication interfaces between the module and the pilot: Command, Status, Warning, and Switch. These interfaces in their default implementation send through the network an array of characters; the Command and Status interfaces support the addition of a custom data serializer that can be implemented by the user in order to send any type of data.

The *Command Interface* is used to send commands to the robot related to the precise task being executed, such as "go_straight 10" to make the robot walk for 10 meters or "set_valve 0.5 0 0.1 0 0 0 1 Waist" to set the valve data for the turning valve task with respect to the *Waist* robot reference frame.

The *Status Interface* is used to send back to the pilot any information the developer considers necessary to understand the internal state of the control module, such as "turning valve," "walking," "ready."

The *Warning Interface* is an advanced interface that can be used in dangerous situations (e.g., when the balancing is compromised) to raise warning states in which the robot can assume a particular behavior (e.g., blocking every movement), from which specific actions can be performed to restore a safe state. The main differences between this interface and the Status are the priority of the data in the communication between control pc and pilot pc, and the different visualization in the pilotInterface, where Warning messages are red (see Section 2.7).

The *Switch Interface* is used to send the following commands to each module: start, pause, resume, stop, and quit. Since some of these commands are critical, they cannot be overridden with different implementations: modules are allowed to re-implement only pause and resume functions. This approach guarantees that any bug or misbehavior of the code running inside a GYM does not propagate to the whole system, since a module can always be forced to stop by the pilot with a stop command. Note that, differently from pause, the stop command does not activate any soft exiting procedure. For example, trying to stop the walking module while the robot is dynamically walking may result in a fall: if the pilot wants to stop the robot from walking and avoid falling, he should send the pause command to the current walking module, which in turn, depending on the robot status, should either stop immediately (double stance phase) or finish the current step phase and put both feet on the ground. Manipulation modules are safer in this sense since the robot is usually stable when moving its arms, nevertheless, a pause procedure should still be implemented as it allows the module to save its internal state and resume it later. Thus, the stop is used to quit a module when it is no longer needed, or to force-quit a module that is not controlling the robot but could be stuck in a loop due to bugs.

2.4.2. State Machine

The behavior of the GYM state machine is reported in **Figure 8**. Except for the special states *Constructor* and *Destructor*, there are three available states. The unique state accessible from the *Constructor* is *Running* through the **start** command. From this state, the module can be put into *Paused* state using the **pause** command or stopped (i.e., put into *Stopped* state using the command **stop**). From the *Paused* state, the module can be switched

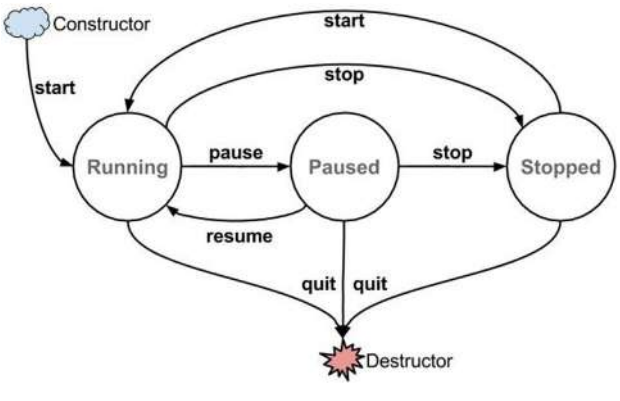


FIGURE 8 | State machine of the GYM.

[1]For the complete list of the helper functions, see https://github.com/robotology-playground/idynutils/blob/whole_robot_wrapper/src/RobotUtils.cpp

[2]For some GYM real modules, please see https://gitlab.robotology.eu/walkman-drc/drc_drive/blob/master/src/drc_drive_thread.cpp or https://gitlab.robotology.eu/walkman-drc/gaze_control/blob/whole_robot/src/gaze_control_thread.cpp

to the *Running* one by the **resume** command or can be stopped. Once the module is in the *Stopped* state, it can be only started (i.e., put into *Running* state through the command **start**). The *Destructor* state is accessible from every state sending the **quit** command. Changes of state are triggered by the watchdog thread in response to a message from the Switch Interface.

In the *Running* state, the internal control module loop is executed, the robot can receive the commands and send the state. The *Paused* state is used to freeze the internal control module loop so that, once resumed, the last command is executed. In the *Stopped* state, the internal control module loop is exited, as the program is closed, but it can be restarted again using the Switch Interface. To develop this state machine, we have been inspired by the OROCOS (Bruyninckx, 2001) *Component Lifecycle StateMachine*.

A generic representation of a control module using the GYM template together with the related widget is depicted in **Figure 9**.

2.5. Control Modules

Thanks to the GYM classes and functions, our team managed to focus on the core development of each DRC task in very short time (e.g., the module used to drive was developed in 10 working days by one single developer). It is worth noting that, although the perception module is not a proper control module, since it does not send references to the robot joints, it has been developed using the GYM template. This module uses ROS drivers to acquire data from the Multisense SL head and the standard command/status/switch interfaces to interact with the pilot. We will now describe the main components of a GYM Module, using the module designed for the driving task as an example.

The underlying structure of every control module is composed by:

- an Inverse Kinematics solver;
- a Finite State Machine (FSM); and
- a trajectory generation library,

and resembles the structure of a hybrid control architecture with discrete states associated with continuous control laws. For example, the state machine for the driving module is shown in **Figure 10**. The principle of the module is the following: a message arrives through the command interface and depending on the message information, a different transition event is triggered, which may result in a change of state. After a new state transition, a new trajectory is created for one or more end-effectors. During the control loop, a portion of the trajectory is sent to the Inverse Kinematics solver, which computes the correspondent portion of joint displacement to be sent to the robot. Modules related to manipulation tasks uses a WholeBody Inverse Kinematics library by Rocchi et al. (2015), while the module related to walking uses a different strategy and Inverse Kinematics inspired by Kryczka et al. (2015). Indeed, we decide to give freedom to the control module developers, so that they could use the control laws and IK approaches that they were more familiar with. Two control modules, based on the proposed architecture, are described in detail in Ajoudani et al. (2014) (for the valve task) and Lee et al. (2014) (for the door task).

2.5.1. Finite State Machine

In order to cope with complex tasks, a Finite State Machine is used to switch between different actions of the robot. Once the operator receives the new status from the status interface, he

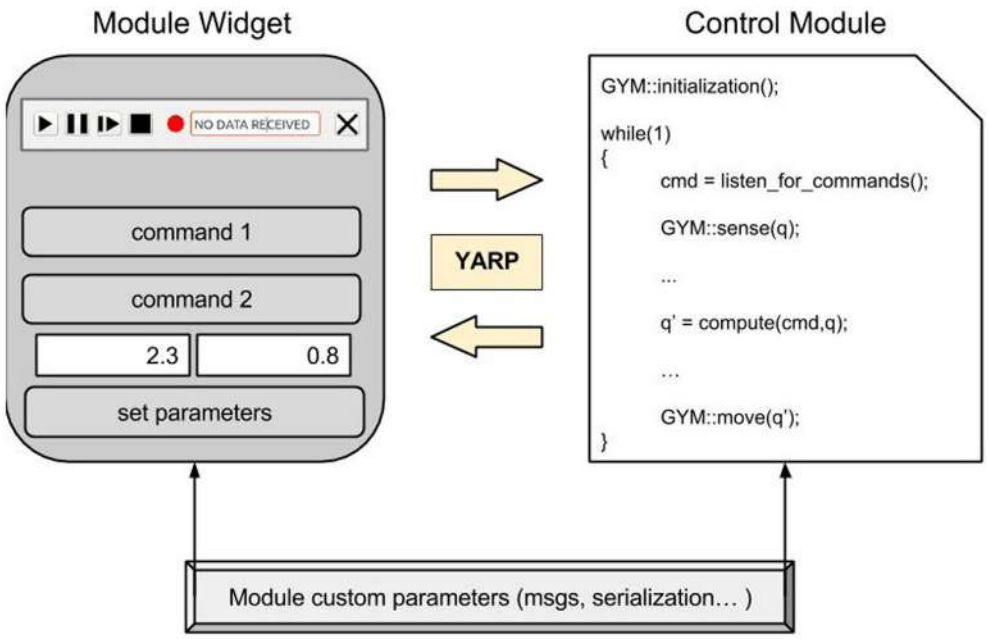


FIGURE 9 | **A control module generic infrastructure**.

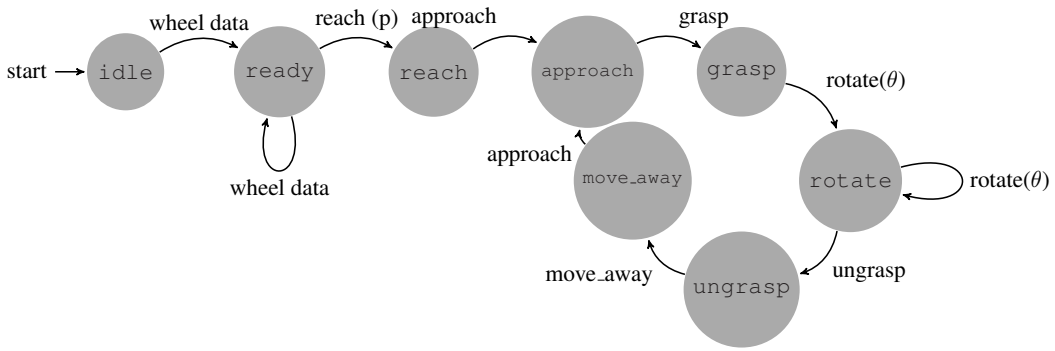


FIGURE 10 | **Finite State Machine of the drive task (wheel management only)**.

can send a message through the command interface to change the module state accordingly to the structure of the FSM. As an example, referring to the driving task and the FSM reported in **Figure 10**, once the pilot receives the information that the status "reach" has been achieved, he can send the "approach" command. Many transitions are not possible because they would result in an incoherent behavior of the robot, such as moving a hand away from the wheel while still grasping it. Every state corresponds to a specific action or to a waiting state.

2.5.2. Trajectory Generator Library

The trajectory generator library consists of a set of trajectories of two types: linear and circular. The linear trajectories are created via fifth-order polynomials, interpolating from the initial and final positions. On the other hand, the circular trajectories are parameterized on the angle of rotation: the polynomial interpolates from the initial to the final angular displacement of the trajectory. The library provides a C++ class that can be initialized with the desired type of trajectory. The API methods allow to set the trajectory parameters, get an arbitrary point of the trajectory, and reset the generator to start a new trajectory.

2.6. Unreliable Channel Management

Our robot is used with two common types of network configuration between the pilot pc and the robot. The first setup is similar to a lab environment, where the network is fully operational and the bandwidth is at least 100 Mb/s. The second one is inspired by a realistic disaster scenario, where a wireless network is discontinuously working and the average bandwidth is less than 1 Mb/s. It is desirable to have most of the software architecture independent from the network capabilities, in particular the code running in control modules and in the pilot interface should not require any changes depending on the network. Both YARP and ROS use centralized servers for naming look-up (respectively called yarpserver and roscore).

When working in the first configuration, we used a single yarpserver and roscore so that modules can communicate directly with each other; there are no networking issues from pilot to robot.

In the real-world scenario, a direct communication may result in frequent disconnections and the centralized YARP/ROS servers may not be able to recover from such disconnections. Thus, a strong division between pilot pc and the robot has been proposed, with two pairs of roscore/yarpserver running, respectively, on the pilot pc and the control pc, splitting modules into a robot subsystem and a pilot subsystem. The two subsystems are bridged using a network manager that transparently interconnects modules between the two. The developed network manager behaves as a two-way bridge between the pilot pc and the robot, it is completely transparent to the processes it connects, meaning that there is no way for the processes to understand if they are communicating through a bridge or directly. Our bridge is developed as a pair of processes, running on two different computers, called BridgeSink (in the sender pc) and BridgeSource (in the receiver pc). The Boost Asio library (Kohlhoff, 2003) was used to abstract UNIX sockets and obtain an asynchronous behavior in the communications.

For the sake of clarity, we introduce an example of the bridge transparency capabilities. Consider two PCs (PC1 and PC2) with one module each (Module Alice and Module Bob, respectively). In the first scenario, Module Alice on PC1 is sending info to Module Bob on PC2 using YARP through a direct connection (i.e., disabled bridge), Alice will try to connect to Bob and will find a YARP port P_B in the remote PC2, while Bob will listen from Alice's remote YARP port P_A in PC1.

In a second scenario the bridge is enabled, and it reproduces the port P_B in PC1 and the port P_A in PC2 so that Alice will actually connect to a local (in PC1) YARP port faking P_B that is provided by the BridgeSink process running on PC1. On the other hand, Bob will listen from a local(PC2) YARP port faking P_A provided by the BridgeSource running on PC2. Finally, BridgeSink and BridgeSource will internally transfer information from PC1 to PC2.

For network management purposes, the proposed bridge uses heuristics whose most important options are the bridge channel protocol (UPD or TCP) and the middleware (YARP or ROS). It is worth noting that the only unsupported combination is a TCP-ROS bridge, since ROS data would saturate the channel.

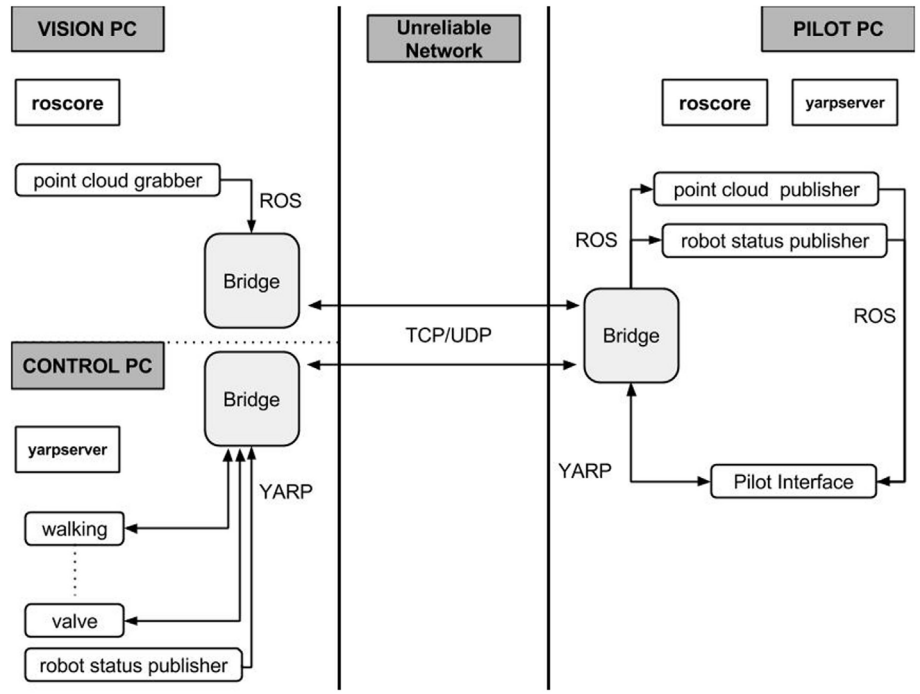


FIGURE 11 | System PCs connections and interactions.

TABLE 1 | Bandwidth usage from the robot to the Pilot, not including TCP overhead, assuming $T = 1$.

From robot	Number	Dimension (bit)	Total (bit/s)	Info
Joint IDs	33	8	264	Robot state
Joint encoders	33	16	528	Robot state
Joint toques	33	8	264	Robot state
Board temperatures	7	16	112	Seven joints per second
Module statuses	5	16	80	Dictionary-based compression
Overhead fixed	1	512	512	Serialization overhead
Overhead variable	10	64	640	Serialization overhead
Total	–	–	2400	

In **Figure 11,** we report the location (motor PC, vision PC, pilot PC) where the various programs are executed, focusing on the TCP/UDP bridge role.

2.6.1. TCP Bridge

Recall that YARP is used for all the communications between pilot pc and control pc, i.e., starting and stopping modules, modules status, modules commands. Those data are relatively small (see **Table 1**) and have high priority; thus, they are usually transmitted through a TCP channel. The bridge is heavily optimized to reduce the data overhead, such as TCP or YARP headers. It uses a configuration file to know which module should be redirected through the bridge, and associates with each module port an 8-bit identifier that is used as a header. An example of the configuration file is shown below:

```
<modules>
   <module name="walking" id="0"/>
   <module name="drc_valve" id="2"/>
   <module name="drc_drive" id="4"/>
   <module name="drc_wall" id="5"/>
   <module name="drc_door" id="6"/>
   <module name="gaze_control" id="7"/>
   <module name="temperature" id="8"/>
   <module name="drc_plug" id="11"/>
</modules>

   <!- -IDs are unique and shared between modules and custom
      modules, do not overlap!!- ->

<custom_modules>
   <module name="encoder_bridge" id="9">
      <connection port_to_open="/command:i"
      source_port="/command:o" target_port="/command:i"
      location="robot" direction="robot_to_pilot"/>
      <connection port_to_open="/switch:i"
      source_port="/switch:o" target_port="/switch:i"
      location="pilot" direction="pilot_to_robot"/>
   </module>
      <module name="walking_publish" id="12">
      <connection port_to_open="/command:i"
      source_port="/command:o" target_port="/command:i"
      location="robot" direction="robot_to_pilot"/>
   </module>
</custom_modules>
```

Note that standard GYM modules are handled automatically, while custom modules require some more information. Indeed, they offer more configurability and allow for port renaming.

All the communications requested during T seconds are packed in a single TCP packet using the 8-bit identifier to keep the original header information. In the case where a port produces multiple packets, they are all dropped except the last one. This effectively reduces the frequency of streaming port, while maintains intact pilot commands.

T is chosen depending on the network bandwidth and delay, in the DRC it was set to 0.5 s. This leaves almost 50% of the channel free to be used, e.g., to send commands from the PilotInterface to the robot modules or to start a ssh shell in the control pc.

2.6.2. UDP Bridge

ROS perception-related data and other streaming information from the robot require low latency. For this type of information, it makes little sense to implement a reliable transport that requires retransmission when packets are loss. Lost data become obsolete and it is much better to read new messages than require re-transmission. For this reason, it is preferable to use UDP protocol.

Since PointClouds and RGB Images are usually larger than the UDP packet size, they need to be split and reconstructed. This is usually done automatically by the UDP protocol implementation, but if a single packet is lost, the whole data are dropped.

Our bridge avoids this problem by splitting point clouds and images into smaller ones, each representing a 3D or 2D sub-region of the original data, so that each one is a standalone pointcloud/image contained into a UDP packet (1500 bytes). By using timestamps, the original data are reconstructed in the pilot pc. This choice results in a delay in the visualization, since BridgeSource waits to receive as many data pieces as possible in an amount of time δt. In the DRC, the parameter δt was set to 0.4 s, which ensured receiving more than 90% of the original point cloud with a delay that was visible by the pilot but not critical since there was no teleoperation involved.

All the module statuses, the robot temperatures, and encoder readings (YARP based) are also sent in the UDP channel at a different (higher) frequency than the TCP one.

2.7. Pilot Interface

To remotely control Walk-Man, a GUI, called Pilot Interface (PI), has been developed. We followed a modular approach, using Qt libraries and ROS libRViz for 3D rendering (Kam et al., 2015). Every DRC task has a dedicated widget and can be used standalone. Moreover, we also developed widgets that allow interaction with the 3D representation of the environment and widgets for monitoring the robot state.

Using our approach, the operator could monitor the environment and the robot status and could make correct decisions to perform the tasks. **Figure 12** shows a screenshot of the PI during the driving task.

Following the approach adopted for the GYM, we developed the *Generic Widget* (GW) so that every control module widget

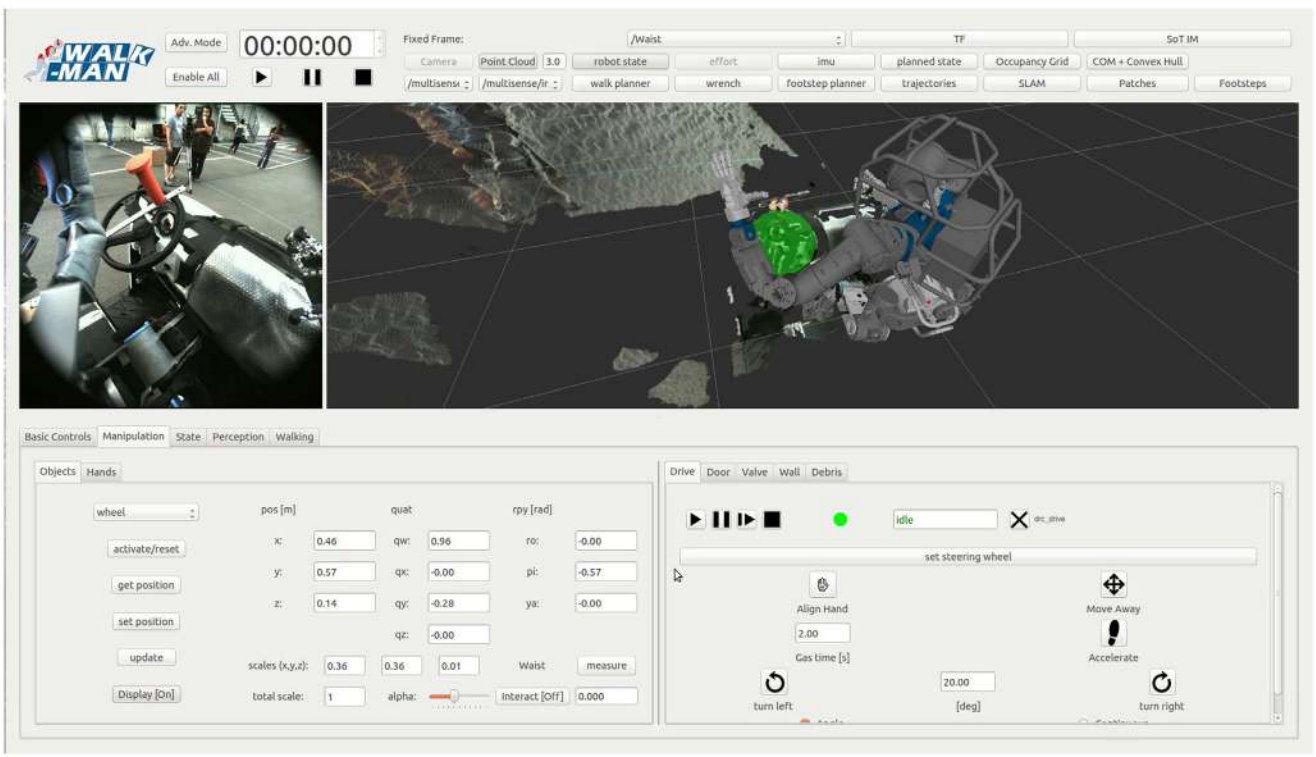


FIGURE 12 | **The pilot interface**.

FIGURE 13 | **Generic Widget Switch and Status Interface**. The red led turns green if the module is running, displaying the relative status description.

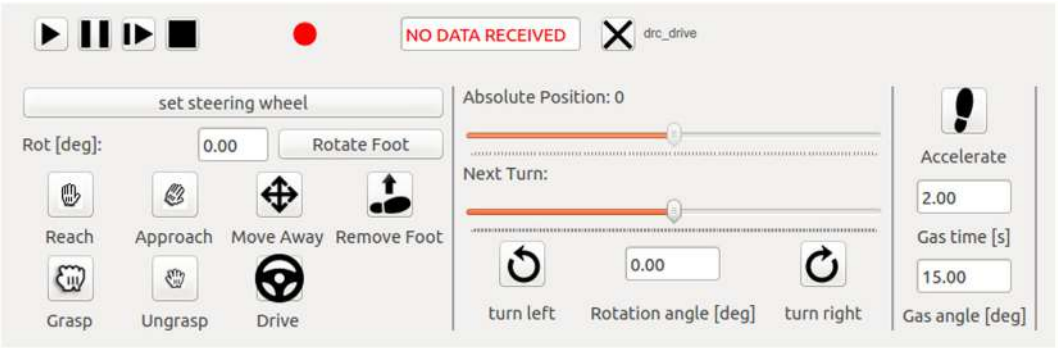


FIGURE 14 | **Driving task dedicated widget**.

has the same basic features. In particular, the GW has already the capability to send messages to the Switch Interface and receive information from the Status Interface (see **Figure 13**).

For the sake of clarity, the driving task widget is reported in **Figure 14**. On the top of the widget, there is the Switch/Status Interface-related buttons, the rest is divided into three parts. On the left, we put the buttons to set the steering wheel position and place the hand on it, together with the buttons to adjust the position of the foot dedicated to the throttle. In the center, there are the buttons to rotate the steering wheel, while sliders are used to help the pilot understanding the current steering wheel position. Finally, on the right, we placed the button for the throttle. In this case, the operator can specify the duration and the amount of throttle.

In contrast to the other DRC teams, we managed the interaction between the robot and the pilot from the motion planning perspective. In fact, our pilots did not explicitly ask and check for a motion plan before the execution started. Instead, the pilots completely relied on the correct on-board open-loop Cartesian generation and kinematic inversion, and checked only the resulting robot position at the end of the execution. This approach was a viable choice thanks to the structural compliance of the robot joints, which handles small unexpected forces from outside, such as the effect of pushing a door with the arm. Moreover, with its soft and adaptable design, the robot hand can grasp an object with a large position/orientation error, it can even hit a surface with its finger without breaking them, and finally it can keep its grasping capabilities even with some broken fingers. With these premises, it is clear that a collision with the environment or a wrong placement of the end-effector with respect to the object do not affect the result of the task. If the robot hand misses the grasp or hits a surface, the operator will simply move the arm back and try again. The use of the Warning Interface to inform the operator of external forces or robot instability further improved our strategy.

An early work describing the initial design of the Pilot Interface is in Settimi et al. (2014). In the months between this preliminary work and the DRC, many features such as the *Generic Widget* have been added. The pilot was given the possibility to activate advanced modes, where commands, usually hidden, are shown and all the buttons are enabled (the pilot knew that this mode was risky, but it might be needed to override safety behavior in an unexpected situation).

Based on the *forgiveness* design principle an implementation of the Qt:QPushButton named *QtTimedButton* has been provided: after the click, a countdown of 3 s is displayed on the button before sending the command; the command can be stopped by re-clicking on it (this is used for dangerous commands to undo erroneous or undesired clicks).

To improve the pilot awareness of the robot state, we introduced a tab dedicated to the status, showing temperatures of the boards, torques of the motors, and battery level (see **Figure 15**) together with the modules statuses and warning messages. A logging utility for commands sent to the robot and statues received has been added, the visual data from the robot is logged as well in order to be able to completely reproduce and analyze the events.

Configuration files give the user the possibility to customize the displayed widgets. In the Darpa Robotic Challenge, three pilots with three different PCs were in the pilot station, each one being focused on different critical aspects: execution of the tasks, perception of the environment, and robot status.

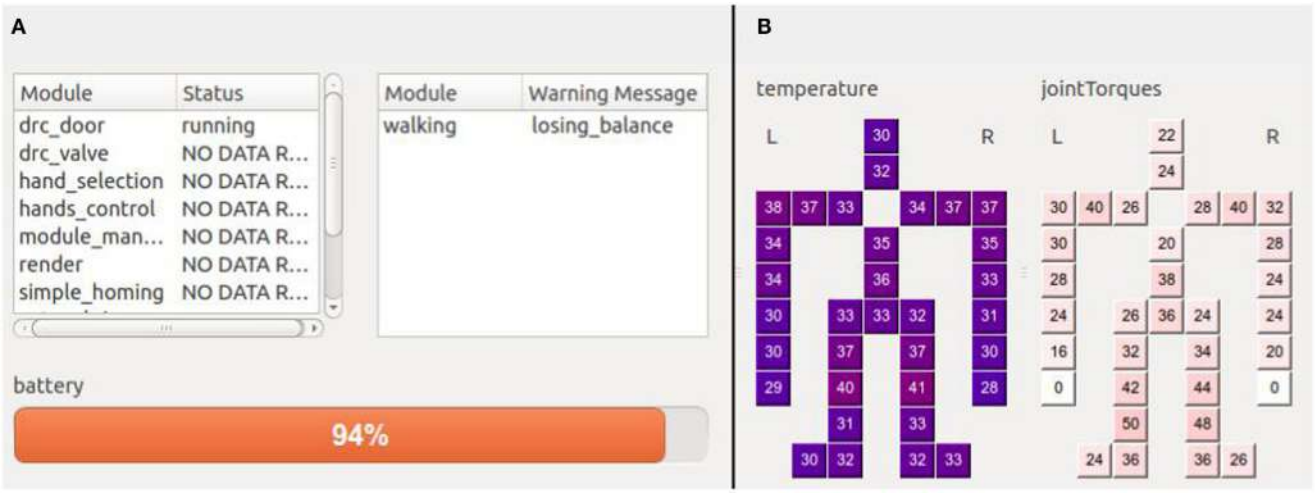


FIGURE 15 | **Status widget**. **(A)** On the left, the different modules status is reported, and the warning message are reported next. On the bottom, the battery percentage is represented by a horizontal bar. **(B)** Two human-like displays show the temperature and torque of every board.

3. RESULTS: THE DRC EXPERIENCE

The first important test of the proposed architecture has been the DRC. Later, other 4 official occasions have occurred between September and November 2015 during which both the hardware and software Walk-Man platforms have been tested. Regarding the DRC, the team got 2 out of 8 points in the competition (consisting in 2 runs) for the accomplishment of the drive and door tasks. Team strategy was to get a penalty in time and avoid the egress of the vehicle task. The dimension of the DRC door has obliged the pilot to enter the indoor scenario walking sideways. The cameras on the Walk-Man head could not provide an accurate vision feedback to compute footstep poses and the irregularity of the terrain made the robot falling after the door was crossed in one of the two runs. During another run issues with the battery and the electronic power management forced the team to accept another time penalty to reset the robot and unfortunately the time left for the run was over.

3.1. Software Components Analysis

The components running on the robot were a set of control modules, the network bridge, a point cloud grabber, multiple webcam grabbers, and the hardware abstraction layer, with ROS and YARP nameservers. In particular, the control modules were paused and resumed when needed, in order to avoid multiple modules controlling the same joints at the same time. On the other hand, on the pilot computers, multiple pilot interfaces and the network bridge were running, along with ROS and YARP nameservers. The data flows inside the robot computers were very simple: all the control modules were connected to the bridge (and consequently to the pilot) and to the hardware abstraction layer. The perception modules were only sending data to the pilot, while the hardware abstraction layer was connected to the ethercat network and received data from the control modules. Finally, all the pilot GUIs were connected to the bridge (and consequently to the control modules on the robot) and to each other.

During the drive task, the driving control module was activated along with the previous listed modules. After the reset, the driving module was stopped, while the walking module was enabled; and the latter was paused and resumed multiple times during the door task in order to allow the door control module to open the door. Indeed, as mentioned, the ethercat master is able to receive inputs from different modules at the same time, and since walking and door modules operate on the same joints, they could not be run together, although they were both needed to execute the task.

The gaze control module was instead active all the time, this way the perception pilot could watch around and place virtual markers in the 3D visualization window. The window was seen and used by all the pilots on their respective computers thanks to the distributed structure of the pilot interface.

The software components used for networking were the first to be tested during the rehersal of the DRC, and performed in a stable and deterministic way. The setup of the bridge was straightforward; during the simulated DRC outdoor mode, the pilot interface received all the information published by the robot, each data at its own designed frequency. Instead, during the simulated indoor mode the TCP channel kept providing critical data, and UDP started to provide pieces of point clouds and images at random times, as expected.

In the competition, we did not have the chance to test the indoor mode, but in the outdoor part, the pilots faced multiple resets, including complete power-offs of the on-board computer. Once restarted, the bridge automatically re-established all the YARP and ROS connections, showing the power of its transparent behavior.

During the days before the competition, multiple pilot GUI configurations were used to test the robot components.

For example, the developers of the walking control modules used a single computer with a GUI configuration having few status widgets and only one control widget (the

walking one) to tune their controller parameters, while the driving test required two pilots, one controlling the gaze and the other using the driving module to steer the wheel and accelerate.

Finally, the pilot checking the robot and network status could add another computer and another GUI during the tests whenever he needed to.

In the 5 days of tests inside the DRC garage, we experienced a single crash of one GUI, probably due to a graphic driver error. After the crash, it was sufficient to start again the GUI with the same configuration and both ROS and YARP middleware allowed to reconnect the GUI and the robot with no issues.

The GUI design helped pilots avoiding errors and parallelizing tasks. The *QtTimedButton* safety feature, which was never needed during trainings, has been exploited for the first and only time during the DRC. During a locomotion phase, the robot was positioning itself sideways with respect to the door; the pilot sent a command to the Walk-Man robot to rotate on the spot. The locomotion expert in the pilot room suddenly figured out that rotating on the spot in that particular inclined terrain could lead to a fall, if an extra stabilization procedure was not used, and he alerted the main pilot. Since the button related to the *rotate on the spot* command is a *QtTimedButton*, and the 3-s safety time window was not expired, the pilot was able to re-click the button and stop the sending of the command. This prevented the robot from falling in that situation.

The start/stop feature of GYM and the capability of modules to initialize in any robot configuration was used by the pilots a couple of times when they were no longer sure about the module status, e.g., after an unexpected network problem that disconnected the TCP safe channel (an issue of the DRC network).

As we already pointed out, the use of multiple pilots and a distributed interconnected architecture between their computers represented a remarkable choice. The advantages were demonstrated during various moments of the challenge, especially in the cooperation between the main pilot and the perception one. Indeed, the main pilot delegated to the perception pilot, among other duties, the superimposition of 3D objects to the scene in the manipulation tasks (e.g., grabbing the steering wheel or the door handle) and the continuous checking of the robot surroundings to decide how to avoid collisions and what to do during the driving. Thus, the main pilot could just focus on the correct execution of the various control sub-tasks required by each DRC task, reducing the amount of stress and consequently the error probability.

3.2. Beyond DRC

As mentioned, the Walk-Man platform has been used in several occasions after the DRC verifying its simple usage and longevity. A first example of a lab experiment is the development of a visual servoing manipulation task to improve robot autonomy. This work uses both a perception ROS module and a manipulation GYM module, which was successfully developed in few days thanks to the code and tasks already available.

During *Eurathlon 2015* and IROS15, the Walk-Man robot performed various exhibitions. The executed tasks were walking, door opening, and valve turning. The walking and door task performed as during the DRC, in a stable and repeatable fashion. It was the first time that the valve task was publicly shown outside the lab and outdoor. The task performed very well and multiple times, demonstrating its reliability and robustness to positioning errors.

The last exhibition of the robot has been in Rome for the Maker Faire Rome 2015: in this occasion the robot had to break a band to inaugurate the event and then greet the audience. We were enough confident in the behavior of the hardware abstraction layer that a colleague located in another city developed the band breaking task in the Gazebo simulator and then sent the code to the Istituto Italiano di Tecnologia labs in order to have it tested on the hardware. The code worked on the real robot out of the box.

Another relevant aspect of this demo has been the use of a single pilot. This was required due to limited space on the stage and the necessity of a quick setup. By using a reconfigured lighter pilot interface, the pilot, who was the one responsible for the status of the robot during the DRC, was able to manage every aspect, from the communication to the successful execution of the task.

4. DISCUSSION

The Walk-Man architecture has proven to be functional and robust in several different occasions and environments (indoor/outdoor challenges, labs experiments for research). Even during the architecture development, no particular criticality has been encountered to make us deviate from the original design. Three main factors have contributed to the chosen architecture design: limited time for implementation, heterogeneity of expertise of code users, and no prior availability of the hardware and, hence, lack of tested control laws.

Solutions adopted to cope with those factors, and discussed in this paper, have worked properly in any occasion the platform has been used. Even though not all the choices were *a priori* optimal, they have proven to work properly in our particular case. We will now discuss the outcomes of some of those choices starting from those made to overcome the strict time deadlines.

The most striking example of the effort done in avoiding the boilerplate code, together with the use of GYM, is the DRC driving module. Indeed, it was developed in a very small amount of time by a master student (i.e., non-expert code developer), which managed to control the gas pedal and to steer the wheel in less than 2 weeks. The module was then refined and tested for a week by two developers of the team and eventually used in the challenge.

It is well known that the design of a modular architecture does not always come for free, requiring significant time effort. Indeed, each software layer requires its own API to interface with others, and dedicated maintenance and update. Nevertheless, our team could have never been able to develop and change the modules without such APIs: the few main issues (e.g., multi-threading issues, network bridge incompatibility with custom YARP ports) encountered during the few months before the competition have been solved in a small amount of time without compromising or delaying the work of other software users.

An unintuitive and apparently wrong practice, in case of complex hardware and software platforms, such as Walk-Man, is the arbitrary choice in critical components implementations as we did for the network bridge. Indeed, non-architecture developers were not informed at all of the inner structure of the bridge. Although this in principle may lead to errors or integration issues, the alternative approach of discussing the design of the bridge among all the team members was prohibitive and required too much time. After the bridge implementation, each of the few issues emerged was solved jointly with the involved people.

Usually, in large companies and in organized open-source projects, coding quality standards, style, and procedures are mandatory and adopted by the whole team. Such approach requires dedicated advanced training and hence time. In our case, the team was formed on purpose for the DRC by including researchers of different groups with different expertise and standards. In similar situations, we strongly suggest to let every programmer choose his programing style and control approaches designing a flexible architecture to support the different users. Our architecture reflects this need by not enforcing any specific control algorithm in the modules implementation, so that developers were free to read just the sensors and to control the joints they required to achieve their specific tasks. As an example, control approaches could range from open-loop joint-space trajectories to inverse dynamics using a combination of force-torque, joint torques, and IMU measurements.

Another solution to cope with short time, which should not be underestimated, is the human pilot capabilities and improvements thanks to training. In particular, there was a trade-off between the effort required from the pilots during the challenge and the software development effort required to offload them from some tasks. As an example, we decided to skip the development of an artificial vision system for automatic object detection and recognition, and trained the perception pilot in order to be very fast and accurate in those tasks. We also noticed that, in the short time, accustom the pilot to each module's behavior pays off as much as an improvement in the module code or control law. Note that this solution cannot be used successfully in every situation. For example, in case on untrained pilots or in high complex tasks (e.g., teleoperated balancing), the only possible approach is the use of a dedicated control software.

For example, our architecture requires tens of modules to be running at the same time across multiple computers, and the modules starting order may become complex to maintain. After the first tests with the whole architecture running, we noticed that lot of pilot effort had to be put in starting the modules in the right order. We decided to reduce such requirements as much as possible, and finally ended up with only the ROS and YARP nameservers to be started before all the other modules. We believe that the effort to provide asynchronous starting order is compensated whenever the architecture complexity increases up to the point where the pilots can no longer manage the order.

While the whole architecture has demonstrated to work properly, some useful utilities were not integrated and left to each developer preferences. In particular, multiple different logging utilities in each module were storing information useful for debugging purposes both on the robot and on the pilot PCs. Some pieces of the stored information were sent commands, status of the robot, point clouds, failures, and warnings from the control modules. Although these logging utilities were custom designed and simple in their capabilities, they provided enough information to speed up the unavoidable debugging process. Their helpfulness prompted us to include, in future architecture updates, a generic logging class integrated in each module with the same style of GYM and GW.

To conclude, the architecture structure and implementation did not affect any task during the DRC, and did not impose any constraint on the control strategies implemented in each task module. Few main issues (e.g., multi-threading issues, network bridge incompatibility with custom YARP ports) were detected during the months before the competition, and they were solved in a small amount of time without affecting or compromising the software developers work. Indeed, the future progressive improvements planned by all the team members mostly relate to the perception modules providing artificial vision and object tracking, a walking module capable of reflex-style reactions to terrain irregularities and an increased automatic error handling in manipulation modules in order to provide single-click complete task execution improving robot autonomy. On the other hand, the architecture general structure is widely accepted by team members and will require very few changes. The main features to be added are hard real-time support and a Matlab-EthercatMaster interface.

AUTHOR CONTRIBUTIONS

MF, LM, AS, AR, EMH, AC, DK, CP, and LN worked on the design of the global architecture and on the network. LM, MF, AR, EMH, AC, and NT designed and developed GYM and the HAL. AS and CP developed the operator control station GUI and many control modules. LP and NT coordinated and advised other authors on all the aspects of this work.

ACKNOWLEDGMENTS

This work is supported by the European commission project Walk-Man EU FP7-ICT no. 611832. The authors would like to thank Stefano Cordasco and Alessio Margan for their work on the design and implementation of electronic boards and firmware.

REFERENCES

Ajoudani, A., Lee, J., Rocchi, A., Ferrati, M., Hoffman, E. M., Settimi, A., et al. (2014). "A manipulation framework for compliant humanoid coman: application to a valve turning task," in *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on* (Madrid: IEEE), 664–670.

Brugali, D., and Shakhimardanov, A. (2010). Component-based robotic engineering (part II): systems and models. *IEEE Robot. Autom. Mag.* 17, 100–112. doi:10.1109/MRA.2010.935798

Bruyninckx, H. (2001). "Open robot control software: the orocos project," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, Vol. 3 (Seoul: IEEE), 2523–2528.

Coleman, DT., Sucan, IA., Chitta, S., and Correll, N. (2014). Reducing the barrier to entry of complex robotic software: a moveit! case study. *J. Software Eng. Robot.* 5, 3–16.

Hammer, T., and Bäuml, B. (2013). "The highly performant and realtime deterministic communication layer of the aRDx software framework," in *16th International Conference on Advanced Robotics, ICAR 2013* (Montevideo).

Hebert, P., Bajracharya, M., Ma, J., Hudson, N., Aydemir, A., Reid, J., et al. (2015). Mobile manipulation and mobility as manipulation design and algorithms of RoboSimian. *J. Field Robot.* 32, 255–274. doi:10.1002/rob.21566

Hoffman, E. M., Traversaro, S., Rocchi, A., Ferrati, M., Settimi, A., Romano, F., et al. (2014). "Yarp based plugins for gazebo simulator," in *Modelling and Simulation for Autonomous Systems: First International Workshop, MESAS 2014*, Vol. 8906 (Rome: Springer), 333.

Johnson, M., Shrewsbury, B., Bertrand, S., Wu, T., Duran, D., Floyd, M., et al. (2015). Team IHMC's lessons learned from the DARPA robotics challenge trials. *J. Field Robot.* 32, 192–208. doi:10.1002/rob.21571

Kam, H. R., Lee, S.-H., Park, T., and Kim, C.-H. (2015). Rviz: a toolkit for real domain data visualization. *Telecommun. Syst.* 60, 337–345. doi:10.1007/s11235-015-0034-5

Kohlhoff, C. (2003). *Boost. Asio*. Available at: http://www.boost.org/doc/libs/1

Kryczka, P., Kormushev, P., Tsagarakis, N., and Caldwell, D. G. (2015). "Online regeneration of bipedal walking gait optimizing footstep placement and timing," in *Proc. IEEE/RSJ Intl Conf. on Intelligent Robots and Systems (IROS 2015)* (Hamburg).

Lee, J., Ajoudani, A., Hoffman, E. M., Rocchi, A., Settimi, A., Ferrati, M., et al. (2014). "Upper-body impedance control with variable stiffness for a door opening task," in *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on* (Madrid: IEEE), 713–719.

Metta, G., Fitzpatrick, P., and Natale, L. (2006). Yarp: yet another robot platform. *Int. J. Adv. Robot. Syst.* 3, 43–48.

Negrello, F., Garabini, M., Catalano, M. G., Malzahn, J., Caldwell, D. G., Bicchi, A., et al. (2015). "A modular compliant actuator for emerging high performance and fall-resilient humanoids," in *2015 IEEE-RAS 15th International Conference on Humanoid Robots* (Seoul: IEEE), 414–420.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., et al. (2009). "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, Vol. 3 (Kobe: IEEE-RAS).

Rocchi, A., Hoffman, E. M., Caldwell, D. G., and Tsagarakis, N. G. (2015). "Opensot: a whole-body control library for the compliant humanoid robot coman," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on* (Seattle: IEEE), 6248–6253.

Settimi, A., Pavan, C., Varricchio, V., Ferrati, M., Hoffman, E. M., Rocchi, A., et al. (2014). "A modular approach for remote operation of humanoid robots in search and rescue scenarios," in *Modelling and Simulation for Autonomous Systems: First International Workshop, MESAS 2014*, Vol. 8906 (Rome: Springer), 192.

Walck, G., Cupcic, U., Duran, T. O., and Perdereau, V. (2014). A case study of ROS software re-usability for dexterous in-hand manipulation. *J. Software Eng. Robot.* 5, 36–47.

Yi, S.-J., McGill, S. G., Vadakedathu, L., He, Q., Ha, I., Han, J., et al. (2015). Team THOR's entry in the DARPA robotics challenge trials 2013. *J. Field Robot.* 32, 315–335. doi:10.1002/rob.21555

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

frontiers in Robotics and AI

ORIGINAL RESEARCH
published: 26 April 2016
doi: 10.3389/frobt.2016.00024

The iCub Software Architecture: Evolution and Lessons Learned

Lorenzo Natale, Ali Paikan, Marco Randazzo and Daniele E. Domenichelli*

iCub Facility, Istituto Italiano di Tecnologia, Genova, Italy

The complexity of humanoid robots is increasing with the availability of new sensors, embedded CPUs, and actuators. This wealth of technologies allows researchers to investigate new problems like multi-modal sensory fusion, whole-body control and multi-modal human-robot interaction. Under the hood of these robots, the software architecture has an important role: it allows researchers to get access to the robot functionalities focusing primarily on their research problems and supports code reuse to minimize development and debugging, especially when new hardware becomes available. But more importantly, it allows increasing the complexity of the experiments that can be carried out before system integration becomes unmanageable, and debugging draws more resources than research itself. In this paper, we illustrate the software architecture of the iCub humanoid robot and the software engineering best practices that have emerged driven by the needs of our research community. We describe the latest development of the middleware supporting interface definition and automatic code generation, logging, ROS compatibility, and channel prioritization. We show the robot abstraction layer and how it has been modified to better address the requirements of the users and to support new hardware as it became available. We also describe the testing framework, and we have recently adopted for developing code using a test-driven methodology. We conclude the paper discussing the lessons we learned during the past 11 years of software development on the iCub humanoid robot.

Keywords: humanoid robotics, software engineering, software middleware, Quality of Service, test-driven development

OPEN ACCESS

Edited by:
Samer Alfayad,
Université de Versailles Saint-Quentin-en-Yvelines, France

Reviewed by:
Arnaud Blanchard,
University of Cergy-Pontoise, France
Vincent Hugel,
Université de Toulon, France

****Correspondence:***
Lorenzo Natale
lorenzo.natale@iit.it

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *23 November 2015*
Accepted: *04 April 2016*
Published: *26 April 2016*

Citation:
Natale L, Paikan A, Randazzo M and Domenichelli DE (2016) The iCub Software Architecture: Evolution and Lessons Learned. Front. Robot. AI 3:24. doi: 10.3389/frobt.2016.00024

1. INTRODUCTION

The rapid evolution of humanoid robots is pushing the requirements on their software infrastructure. The availability of low-cost, off-the-shelf sensors for depth perception, IMUs, tactile and force sensing allows robots to be equipped with richer and redundant sensory systems. New actuators give joints higher maximum torque, allow designers to increase the dexterity of the robots and to implement force or impedance control. New technology for optical or magnetic encoders allows measuring movement in various points in the kinematic chain providing redundancy, fault tolerance or, in presence of elastic elements, accurate torque measurement. This evolution opened up new research problems, such as multi-modal sensory fusion, whole-body force control, and multi-modal human–robot interaction to mention just a few. However, exploring these research directions comes at a high cost in terms of software development. When existing hardware is replaced with new one, it yields to software obsolescence, new development, debugging and consequent changes in low-level software layers that trigger the redesign of higher layers. Experiments that build on top of simpler capability are possible only if the software architecture supplies researchers with appropriate

tools that allow them to focus on the goals of their research. Also, the software itself evolves. Languages, operating systems, and libraries get upgraded and change, sometimes without maintaining backward compatibility. The appearance of Robot Operating System (ROS) and its rapid adoption and growing community (Quigley et al., 2009) have changed how people develop software and pushed many robot developers to provide ROS compatible interfaces for their software or to adopt it altogether.

These problems have been present since the beginning of the development of the iCub platform and through the past 11 years of its evolution. The iCub is a humanoid robot platform that was designed for research in cognitive system. Its main goal is to support experimental research and, for this reason, it is not designed with a specific application in mind. The hardware development of the robot was also driven by research goals. The software infrastructure of the robot was designed and adapted following these constraints. At the lowest level, it had to support new hardware as soon as it was released and reduce the impact of hardware changes to the user code. At the higher level, the software architecture was designed to support rapid prototyping of experiments that required integration of many capabilities: visual and speech perception, control of attention, learning, reaching, grasping, and, more recently, balancing and walking.

In this paper, we provide a review of the software architecture of the iCub robot including recent developments aimed to better support the evolution of the robot and the needs of the research community. We describe the Yet Another Robot Platform (YARP) middleware and how it has been extended with facilities to increase determinism in time-critical loops. We present new tools and best practices that have been adopted for logging and to aid developers in the task of developing component interfaces, defining new data types, and interoperate with software developed for other robots (including software from the ROS eco-system). We describe the robot abstraction layer, which allows the same code to control the real robot or a simulation, on-board, or through a network link. This abstraction layer separates high-level components from the hardware implementation including the communication infrastructure. The core of the robot abstraction layer has not changed much in the years preserving backward compatibility, but it has been extended to better support new control modes and sensors. We describe the component that provides access to the robot, i.e., *robotinterface*, showing how it can be configured depending on the available hardware and to run time-critical control loops directly on the robot.

Robotic software applications quickly grow in the number of components and are therefore difficult to engineer and develop. Software engineering best practices suggest to divide such systems in simple units that are independently developed, tested, and integrated at a later stage. In the second part of the paper, we describe the tools that we have developed to support deployment and monitoring of components (i.e., the *yarpmanager*) and, more recently, testing. Test-driven development (Beck, 2003) was used to develop the YARP middleware but was adopted only recently to validate the robot software interface and the control algorithms. We developed the *Robot Testing Framework* (RTF) that allows testing robot software using the real robot as well as simulations. We describe the design choices that have driven the development of the RTF and how it has been adopted for testing software components and interfaces of the iCub robot. We conclude the paper with a discussion on the lessons we have learned in the past years of software development on the iCub robot drawing the conclusions of this work.

2. A BRIEF OVERVIEW OF THE iCUB AND ITS EVOLUTION

The iCub is an open humanoid robot platform that was developed for research in cognitive systems (Metta et al., 2010; Parmiggiani et al., 2012). It has 53 joints actuated with brushless and DC motors. Motion generation is carried out in dedicated boards embedded on the robot and interconnected through a local bus (initially we used CAN bus, but shifted recently to Ethernet to increase the available bandwidth). These boards host programmable CPUs that can perform position control with trajectory interpolation, velocity, and torque control. The iCub was initially equipped with cameras for vision, microphones an IMU on the head, and motor encoders for measuring motion. This initial set of sensors grew with time, by introducing 6 axis F/T sensors in various points of the kinematic chains (roughly located at the shoulders and hip, and eventually at the ankles), and a system of tactile sensors[1] that, starting from the hands and forearms, has been extended to cover a large part of the whole robot (for a total of 4000 sensing units located on the arms, torso, legs, and feet soles). At the same time, inertial units and gyroscopes became inexpensive and easy to integrate in the electronics that control the tactile systems and the motors. The robot mounts on the head a PC104 computer equipped with an Intel CPU that runs Linux. This computer works as a bridge interconnecting the CPUs on the local bus with the external cluster of computers that performs heavy computation outside the robot. Connection with the external cluster is achieved using either gigabit Ethernet or wireless.

The software architecture of the robot can be broadly separated in two layers. The firmware consists in the first layer and runs on the embedded CPUs. Communication between boards and PC104 computer uses a custom networking protocol (over CAN or Ethernet). The second layer consists in all the software components that run on the head computer and on the external cluster. These components communicate using a peer-to-peer publish-subscribe architecture implemented using the YARP middleware (Fitzpatrick et al., 2008; Metta et al., 2010).

3. MIDDLEWARE

Best practices in robotics advocate adoption of component-based development (Brugali and Scandurra, 2009) and the so-called separation of concerns between computation, communication, coordination, and configuration (Bruyninckx et al., 2013). Following this approach, components encapsulate robot functionality in a way that promotes interoperability, composability, and reuse, irrespective of the robot, programing language,

[1]Tactile sensors on the iCub are based on capacitive technology (Maiolino et al., 2013).

operating system, computing architecture, and communication protocol being used.

Computation is the core of the components and includes functionalities and algorithms. Communication allows modules to exchange data in a way that is agnostic with respect to the underlying operating system, medium or protocol. The separation of concerns is implemented by the middleware in the form of read/write primitives to receive and transmit data. Coordination is the code required to orchestrate modules: it determines how modules interact to achieve a certain task (Lütkebohle et al., 2011; Klotzbücher and Bruyninckx, 2012; Paikan et al., 2014a). Configuration allows reconfiguration of modules to better adapt them to specific domains, it allows controlling all parameters that affect the functioning of the system, including dependencies across modules and protocols.

Software middleware supports some or all the functionalities described above. Generic middleware like CORBA,[2] Ice,[3] D-Bus,[4] or ∅MQ,[5] provide complete communication backbones. They are rarely employed in robotics because they lack specific components and have a steep learning curve. Robotics middleware [OROCOS (Bruyninckx, 2001), Player (Collett et al., 2005), YARP (Fitzpatrick et al., 2008), Orca (Brooks et al., 2005), ROS (Quigley et al., 2009), OpenRDK (Calisi et al., 2012), Mira (Einhorn et al., 2012), LCM (Huang et al., 2010) to mention just a few] provide a subset of communication paradigms (Remote Procedure Call and/or publish-subscribe). Some middleware defines interfaces for families of devices (Collett et al., 2005) for better modularity and portability.

The iCub software architecture is similar to the port-based software abstraction (Stewart et al., 1997) and is built on top of the facilities provided by the YARP middleware. To better illustrate the communication patterns supported by the YARP middleware, we will adopt the terminology introduced in (Eugster et al., 2003) for publish-subscribe architectures. Eugster et al. (2003) introduce three levels of decoupling to characterize various flavors of communications, namely: *space decoupling*, *time decoupling*, and *synchronization decoupling*. *Space decoupling* is achieved when components produce messages without being explicitly aware of the number and location of the receivers. *Time decoupling* guarantees message delivery even if senders and receivers are not active or connected at the same time. Finally, *synchronization decoupling* requires that messages are sent and received asynchronously by the communicating entities. When communication is asynchronous, it is sometimes important to guarantee that messages are correctly received by slow recipients [this is called *persistence* and is another key property of publish-subscribers architectures Eugster et al. (2003)]. In this respect, robotic middleware often implements policies that aim at reducing latency in real-time control loops, even at the cost of dropping messages [e.g., Fitzpatrick et al. (2008) and Dantam et al. (2015)].

YARP (Fitzpatrick et al., 2008) implements a variant of the publish-subscribe paradigm, i.e., the *observer pattern* (Gamma et al., 1995), which is a type of distributed publish-subscribe providing space and synchronization decoupling. In addition it is multi-platform, in that it provides a portable abstraction for the operating system, the communication protocol and the robot hardware. In YARP *Port* objects deliver messages of any size and type across a network, using various underlying protocols – including shared memory. *Ports* can be configured to implement publish-subscribe with different levels of decoupling and dynamically reconfiguration of connections and protocols. YARP *Ports* have read and write primitives that can be blocking or non-blocking for synchronous or asynchronous communication. A component that uses *Port* objects to perform a synchronous write, waits until all receivers confirm reception of the message. Similarly, a component that performs a synchronous read waits until a new message is received by the *Port*. By default, *Port* objects in YARP are configured for both synchronous read and write: this guarantees correct delivery of messages without extra code. The *BufferedPort* object is a specialization of a *Port*, which provides synchronization decoupling. *BufferedPorts* are active objects able to store and handle messages internally either for transmitting or receiving them using dedicated threads. Possible buffering policies are: First In First Out (FIFO) and Oldest Packet Drop (ODP). In the first case, messages are queued in a list that grows and guarantees that no messages are dropped. In the second case, the size of the queue is fixed, and new messages overwrite old ones to guarantee minimum latency. Read operations in a *BufferedPort* can be blocking in case we want execution to wait for incoming messages. Publish subscribe is convenient for one-way communication. In some cases, however, communication requires replies. In YARP, this is called RPC and is supported *via* two specialization of the *Port class*: *RPCServer* and *RPCClient*, respectively, for managing the server and client side of the communication.

In the iCub software, architecture components are runnable pieces of software (usually implemented as executables, but sometimes also as software drivers) that export a certain interface using one or more *Port* (or *BufferedPort*) objects. The component sends and receives data through its *Port* objects; depending on the type of service provided by the components the port can be configured for synchronous, asynchronous or RPC operations.

YARP is multi-platform and implements all its functionalities on Linux, Windows, and MacOS. With respect to other robotics middleware-like OROCOS (Bruyninckx, 2001), Player (Collett et al., 2005), and ROS (Quigley et al., 2009), the communication layer of YARP supports a richer set of protocols. A notable example is the multicast protocol, which reduces bandwidth and transmission overhead in case of one-to-many communication [which to the best of our knowledge is implemented only by LCM (Huang et al., 2010)]. More importantly, YARP implements a plug-in system that allows users to write custom protocols and interconnect it with other systems (like cameras providing images in mjpeg format or a web server). This mechanism was used to add ROS compatible protocols [i.e., *tcpros*, *xmlrpc*, see Fitzpatrick et al. (2014) for more details], port monitor (Paikan et al., 2014a) and to add QoS and channel prioritization, as described in Section 3. The Thrift IDL provides a language for defining component interfaces that is more flexible than the one that can

[2] http://www.corba.org/
[3] https://zeroc.com/products/ice
[4] https://www.freedesktop.org/wiki/IntroductionToDBus/
[5] http://zeromq.org/

be implemented using ROS's services. Similarly to Player (Collett et al., 2005), YARP provides interfaces for hardware devices, and, in particular, a sophisticated hardware abstraction layer for motor control, as described in Section 4. These interfaces simplify the control of robots hiding complexity due to the underlying communication layer and vendor-specific APIs.

Communication performance of various middleware has been compared experimentally in Hammer and Bäuml (2014). In this study, YARP demonstrated comparable – and sometimes even faster – performance than ROS. Interestingly, round-trip time of both middleware was consistently better than OROCOS. It is worth noticing that YARP was not designed to support hard real time. The recent QoS and channel prioritization extensions partially cope with this, providing significantly performance improvement in terms of communication and scheduling jitter, especially when adopted together with the RT-PREEMPT Linux kernel. For applications that require lower latency and higher determinism OROCOS (Bruyninckx, 2001) and aRDx (Hammer and Bäuml, 2014) may be a preferable choice. It is worth noticing that OROCOS offers a YARP compatible transport that can be used for interoperability with YARP applications.

YARP is written in C++, but similarly to ROS, it can be used from other languages. Through SWIG,[6] YARP provides language bindings for many languages, namely: Perl, Python, Ruby, Lua, TCL, C#, Java, Octave, Chicken, and Alegro. Java also provides seamless integration with the Matlab environment, although, to achieve better performance, we have recently started developing support for Matlab and Simulink *via* mex functions. YARP was interfaced with Android and iOS devices.

Compilation of YARP is simple. YARP can be compiled under various operating systems, including major distribution of Linux (Debian and Ubuntu), MacOS X, and Windows.[7] Dependencies have been voluntarily kept at minimum. The main dependency is ACE[8] which is – by design – a portable library that can be compiled on a plethora of systems. The build system automatically detects optional dependencies and disables features or plug-ins accordingly. To simplify compilation with Linux, we decided to support a set of distributions and to use features available only on the libraries provided therein. This greatly simplified compilation and distributions of binaries for the Linux system. Compilation on Windows was made more complicated by the lack of a proper packet management system. For this reason we decided to build and distribute precompiled binaries of the required dependencies for all the supported compilers. For compilation on MacOS X, we rely on *homebrew*[9] and maintain appropriate *recipes* scripts for both YARP and the iCub main software. To ensure correct compilation of the software on all the supported platforms, a compile farm performs compilation tests, periodically and upon any commit to the YARP and iCub main repositories.

[6]http://www.swig.org/

[7]At the time of writing YARP (and the iCub main software) were supported on Debian 7-9 and Ubuntu 14.04-16.04. On Windows supported compilers were Visual Studio 10-12. Support for MacOS included the latest release 10.11, codename *El Captain*.

[8]http://www.cs.wustl.edu/~schmidt/ACE.html

[9]http://brew.sh/

TABLE 1 | YARP libraries footprint.

Library	Footprint (KB)
libACE	1673
libYARP_OS	2088
libYARP_sig	244
libYARP_dev	94
libYARP_math	1530

See text for details.

To conclude this section, we report additional information about YARP. **Table 1** reports the footprint of the YARP libraries. The table reports the value obtained running the Linux command *size*. These values corresponds to YARP 2.3.64 compiled in "Release" mode, gcc version 5.3.1, libc 2.21, CMake 3.4.1, libACE 6.3.3, and libGSL 2.1 (the latter is optional and it provide signal processing and linear algebra routines to YARP). Notice that libACE in the Linux environment is optional and is required only for compilation on Windows. YARP is adopted by a large number of people. The iCub community consists in approximately 30 teams. YARP is also adopted on the COMAN (Tsagarakis et al., 2013) and in the projects Walkman,[10] DREAM (Vernon et al., 2015), and Fireswarm,[11] while OROCOS includes a YARP compatible transport protocol. YARP received contributions from 75 developers in total and from 29 developers in the past 12 months (source: Open HUB[12]).

3.1 Logging

YARP provides macros that allow users to log messages with increasing levels of importance and severity (i.e., trace, debug, info, warning, error, and fatal). These macro print on the standard output using the facilities provided by the host operating system. To implement the logging system we follow these guidelines: (i) in a distributed architecture messages should be collected from different machines, (ii) logging should be optional to avoid using unnecessary resources and finally, (iii) it should be possible to collect output from components that have been written without YARP. These features have been achieved by relying on the *yarprun* service. The latter is a software service that is used to execute components remotely using a GUI (the *yarpmanager*, as described in Section 8). *yarprun* spawns processes and can therefore manipulate and redirect their output. This offers a simple way to log the output of all components without modifying their code or forcing adoption of YARP specific macros for logging. *yarprun* prefixes all messages from a component with the name of the machine on which it is running and the process identifier of the component itself. All messages are redirected to common recipient using YARP *Ports*; the recipient can either be a command line tool or a graphical interface that allow logging, filtering, and visualization of these messages. This solution allows capturing and logging the output of components even when they do not use YARP's logging functions (although with limited functionalities).

[10]https://www.walk-man.eu

[11]http://fireswarm.nl/

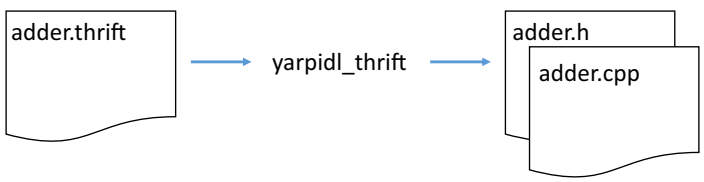
[12]https://www.openhub.net

3.2. The Thrift IDL

A *Port* object can transmit a data type only if proper serialization and deserialization functions are defined. YARP supports a few basic types that already include serialization functions: *Vector*, *Matrix*, *Image*, and *Bottle*. The first three types are self-explaining, they define containers for double precision floating point vectors, matrices, and images with various pixel types. The *Bottle* object is a list of mixed type values: it can store arbitrarily integers, strings, doubles, lists, or binary blobs of memory. This type is quite flexible and can be used to send virtually anything, provided the sender and receiver agree on the data they exchange. The compiler in fact cannot determine if the data sent through a *Bottle* is correctly parsed by the receiver. This can be acceptable for small applications but become soon a limitation, especially when modules are developed asynchronously by different developers. Services are also a concept that is not natively supported by YARP. Services can indeed be implemented with YARP, but the programmer has to manually write all the code required to parse incoming messages and prepare replies. Writing this code is boring and error prone, its maintenance becomes soon complex and difficult.

All these problems have been solved in YARP with the adoption of an Interface Definition Language (IDL) based on the Thrift language. The Thrift IDL can define services. From this definition, the *yarpidl_thrift* compiler generates all the code that implements the communication between the clients and the service across the YARP network. This process is exemplified in **Figure 1**. A new network type can be implemented in a similar way. In this case, the *yarpidl_thrift* compiler generates the .h and .cpp files for the C++ class that implements the type in YARP. The compiler automatically generates serialization and deserialization routines. See **Figure 2** for details.

The Thrift IDL is currently adopted as best practice for defining new types and interfaces for components in the iCub software architecture. An important feature of the Thrift YARP compiler is that it copies verbatim all comments from the Thrift file to the C++ implementation class. This feature is used to document the interfaces using Doxygen. All Doxygen comments are added to the Thrift file(s) that defines the interface of components, when code is generated, these comments are copied in the resulting C++ header files and parsed to produce the documentation. In this way, the documentation of the interfaces is stored with the




```
adder.thrift
service Adder {
      /** Documentation for get_answer*/
      i32  get_answer();
      /** Documentation for get_answer*/
      bool set_answer(1:i32 val)
      /** Documentation for add*/
      i32  add (1:i32 x);
}
```

```
adder.h:
class Adder: public yarp::os::Wire {
public:
       /** Documentation for get_answer */
       virtual int32_t get_answer();
       /** Documentation for get_answer */
       virtual bool set_answer(const int32_t val);
       /** Documentation for add */
       virtual int32_t add(const int32_t x);
};
```

Cient code

```
client.cpp:
#include "adder.h"
int main() {
    Port port;
    Adder client;
    port.open("/client");

    client.yarp().attachAsClient(port);

    client.set_answer(10);
    client.add(1);
    cout<<client.get_answer();
}
```

Server code

```
adderserver.h:
#include "adder.h"
class AdderServer : public Adder {
public:
    int32 _val
    int32_t get_answer()
       { return _val; }
    bool set_answer(const int32_t val)
       {_val=val; }
    int32_t add(const int32_t x)
       { _val+=x; return _val}
};
```

```
server.cpp:
#include "adderserver.h"
int main() {
    Port port;
    AdderServer server;
    server_port.open("/server");

    client.yarp().attachAsServer(port);

    while(true)
      {...}
}
```

FIGURE 1 | Service implementation using Thrift. The definition of the service *Adder* is written using the Thrift IDL (*adder.thrift*); from this file, the *yarpidl_thrift* compiler generates the corresponding C++ interface (*adder.h*) and the full implementation of the client (*adder.cpp*). At the client side, function calls take care of marshaling the parameters and shipping them across the YARP network. At the server side, a *Port* object receives these messages, performs de-marshaling, and invokes the corresponding C++ function. Notice that comments in the Thrift file are preserved in the C++ class. Bottom: the client main function instantiates the YARP *Port* that will be used for the communication, it then instantiates the client object, attaches the *Port* to it and invokes the service functions as if they were local calls. At the server side, the developers provides implementations for the service functions in an new class (*AdderServer*) that overrides the virtual functions defined in *Adder* (*adder.h*); it then instantiates the *Port* that handles the communication and attaches it to an instance of *AdderServer*. At this point, all messages received by the *Port* at the server side are dispatched to *AdderServer*.

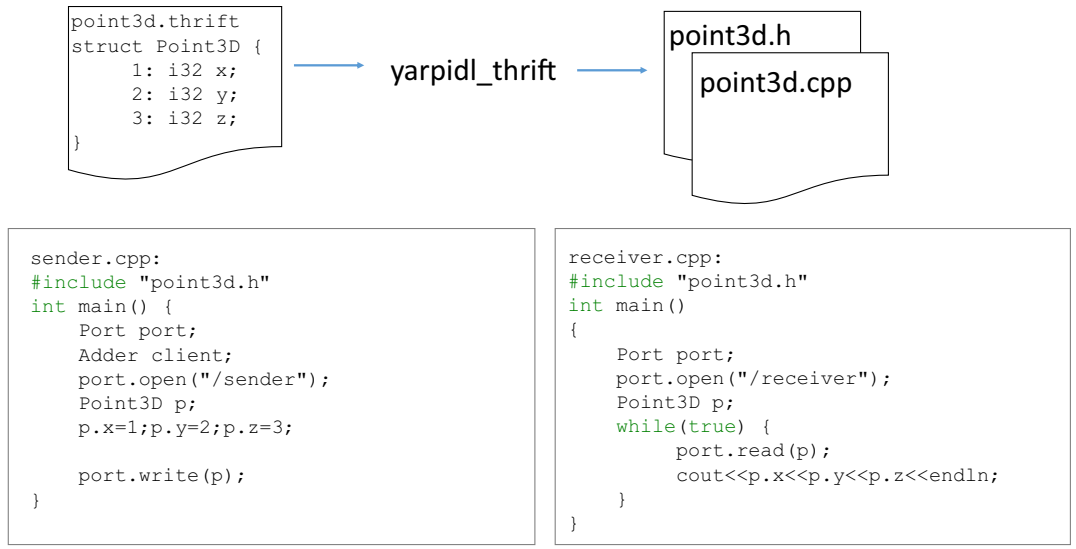


FIGURE 2 | Specification of a new type using Thrift. The *yarpidl_thrift* compiler generates the C++ object that implements the type *Point3D* starting from its specification in the Thrift IDL. Bottom: this object contains serialization and deserialization functions and can be used directly for read/write operations in a YARP *Port* or *BufferedPort* (not shown).

code that generates them; this results in better documentation and easier maintenance.

A recurring pattern in the development of software on the robot is the following: A module defines a set of inner parameters that modify its behavior and exports them through a *Port*, together with a set of functions for manipulating them. Thrift allows defining a structure for grouping parameters so that YARP can provide support for reading and writing this structure through a *Port*. In addition to this, YARP generates an object called *Editor*, which provides methods for setting and getting individual values within the structure as well as callbacks that can be customized to execute code before and after the value is modified. This feature reduces the amount of code that is manually written and maintained when writing interfaces for modules, which, in robotics applications usually consists in several parameters. The *Editor* has been introduced only recently in YARP but is currently adopted as best practice when writing new modules. **Figure 3** illustrates this concept in more details.

3.3. Increasing Determinism in Distributed Applications

Many robotic applications require real-time functionalities, especially when timing constraints on task execution, data processing, and synchronization are crucial. In distributed architectures, extra care must be taken to avoid mutual interference between components in the communication layer. The YARP middleware deals with this problem by providing functionalities at the level of the connections (in the *Port* and *BufferedPort* objects) that allow assigning different priorities to individual communication channels (we call this approach "channel prioritization") (Paikan et al., 2015b). This approach simply leverages the operating system functionalities to prioritize specific communication channels between publishers and subscribers.[13] The properties of individual connection channels are extended to specify (i) the priority level and scheduling policy of the threads that handle the communication and (ii) the priority of the packets on the network (i.e,. the network Quality of Service parameters). This approach does not require specific components for message prioritization and it does not add any overhead to the communication. In addition, and, more importantly, it allows for remote configuration of Quality of Service (QoS) and for run time, dynamic prioritization of communication channels. Configuring real-time properties such as priority or scheduling policy of the user thread can be done either programmatically from the user code or automatically using component middleware functionalities and dedicated tools (Mastrogiovanni et al., 2013).

When configured for asynchronous communication *Port* objects send and receive user data in separate threads. A conceptual example is depicted in **Figure 4**, where a publisher (Publisher 1) pushes data to two subscribers. When a publisher writes data to a *Port*, it passes it to the corresponding thread. At this point, execution is determined by the operating system and the thread real-time properties decide with which priority the thread will manage to write data to the socket. Similarly at the receiver side, real-time properties of a thread affect the chances that it will deliver data to the user (i.e., high priority will reduce jitter). This is useful when a subscriber receives data from multiple senders, and the application requires to assign higher priority to one of them.

Each connection in YARP has a state that can be manipulated by external (administrative) commands, which in turn manage

[13]Notice that in this section we refer to "best-effort" or "soft" real-time, as opposed to "hard" real-time performance. Because the implementation of YARP has not been developed to support hard real-time constraint, we rely on the RT-PREEMPT Linux to reduce scheduling latency.

```
receiver.cpp:
#include "MyPoint3d.h"
int main() {
     Port port;
     Point3D p;
     Point3D::Editor editor(p);
     port.setReader(editor);
     port.open("/receiver");
     while (true) {
      printft("%d %d %d\n",p.x p.y p.z);
      /* wait some time */
     }
}
```

```
MyPoint3d.h
#include "Point3d.h"
class MyPoint3d: public MyPoint3d  {
public:
      bool will_set_x()
       { prinf "about to set x"; }

      bool did_set_x()
       { printf "finished setting x"; }
};
```

```
$ ./receiver
yarp: Port /receiver active at tcp://127.0.0.1:10001
yarp: Receiving input from /writer to /receiver
0 0 0
about to set x
10 0 0
finished setting x
```

```
$ yarp write /writer /receiver
yarp:Port /writer active at tpc://127.0.0.1:10002
set x 10
...
```

FIGURE 3 | The *Editor.* For each new type, the Thrift compiler generates an Editor object, which provides methods for manipulating only a subset of the type. The listing on the top-left shows how to use the *Editor*. An instance of the *Editor* is connected to *Point3D* (see **Figure 2**) and to the module's *Port* object (*i.e. port*). Messages received *by port* are parsed by the *Editor*, which decodes and handles them appropriately. Top-right: optionally, the user can define call-back functions that are invoked before and after data is modified. Bottom: we show how the value of the field *x* in *Point3D* can be modified by sending text messages to a *Port* (left: output of *receiver*, right: text commands sent to the *receiver*'s *Port* using the standard tool *yarp write*).

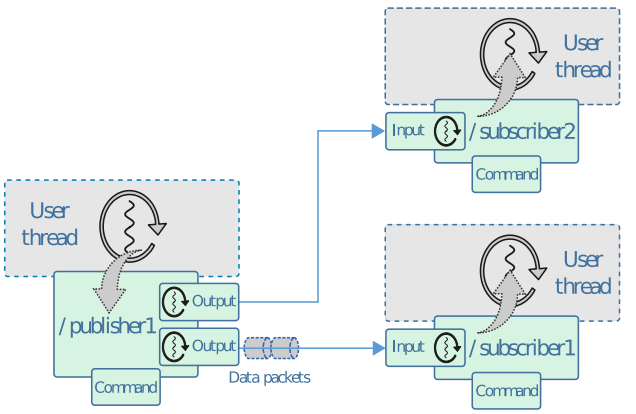


FIGURE 4 | An example of components asynchronous communication in YARP. A publisher is pushing messages to two different subscribers using separate dedicated threads.

the connection and/or obtain state information from it. Using *Port* administrative commands, QoS and real-time properties of *Port* objects can be configured with the granularity of individual connections. In the current implementation, the *Port* administrator provides two set of commands that affect the priority of a communication channel: setting the scheduling policy/priority of a communication thread and configuring network QoS parameters (i.e., the TOS/DSCP bits) for the data packets it delivers.

In the example from **Figure 4**, we can configure the real-time properties of the channel that links/*publisher1* to/*subscriber1*:

```
$ yarp admin rpc/publisher1
> > prop set/subscriber1 (sched
((policy SCHED_FIFO)
(priority 30)))
```

The first line "*yarp admin rpc*" simply opens an administrative session with the *Port* object of /*publisher1*. The second line is the real command to the administrative *Port*. It adjusts the scheduling policy and priority of the thread in/*publisher1*, which handles the connection to /*subscriber1* respectively to SCHED_FIFO and 30 on Linux machines.[14]

Ip networks define four classes of services (Almesberger et al., 1999). These classes are selected so that packets can be treated similarly by the OS queuing policy (if available) and in the network switch. For example, a packet with priority class `Low` will be in the lowest priority band (Band 2) of the Linux queuing policy and will have the lowest priority in the network switch. Analogously, data packet priority can be configured *via* administrative commands by setting one of the predefined priority classes:

```
$ yarp admin rpc/publisher1
> > prop set/subscriber1 (qos ((priority HIGH)))
```

This simply sets the outbound packets priority to HIGH for the connection from /*publisher1* to /*subscriber1*.

These parameters can be set for every channel in the same way and jointly define the actual priority of a communication channel

[14]The thread scheduling properties and policies are highly OS dependent and a proper combination of priority and policy should be used.

in our publish/subscribe architecture. Alternatively, real-time properties of the communication channels can be configured from the user code using the YARP API:

```
QosStyle style;
style.setThreadPolicy(SCHED_FIFO);
style.setThreadPriority(30);
style.setPacketPriorityByLevel
(QosStyle:PacketPriorityHigh);
NetworkBase:setConnectionQos("/publisher1", "/subscriber1",
style);
```

We have experimentally demonstrated that channel prioritization significantly reduces latency and increases communication determinism in presence of conflicting connections in robotic applications (Paikan et al., 2015a,b).

4. ROBOT INTERFACE

The interface to the sensors and motors of the robot is implemented by a set of user-level drivers that access the hardware using vendor-specific API. On the iCub, cameras use a IEEE1394 Firewire bus whereas the majority of the other sensors use custom electronics connected *via* can bus or, in recent versions, Ethernet. These devices use custom protocols, whose details are not important for this paper. For such devices YARP defines a set of C++ interfaces that provide an abstraction layer that is independent of the specificities of the hardware components that implement them. Communication with the hardware is achieved by instantiating user-level devices, which directly send messages to the hardware. These devices implement a set of interfaces that allow reading sensor values and controlling the motors at the joint level. A second layer of devices can be instantiated and connected to these (*via* a function called *attach*) to implement functionalities like robot calibration, on-board control loops, and network remotization (see **Figure 5** for more details).

The life-cycle of all objects is decoupled: all instances are created independently and references to drivers are passed to higher layers using the function *attach*. The opposite operation is performed by calling a function called *detach*, which removes all references held by a device before shutting down and releasing memory. In the development of the iCub interfaces have played an important role because they have preserved the user code in face of deep changes in the hardware. The code required to perform basic functionalities like reading images or controlling the motors in position or velocity modes has not changed significantly while

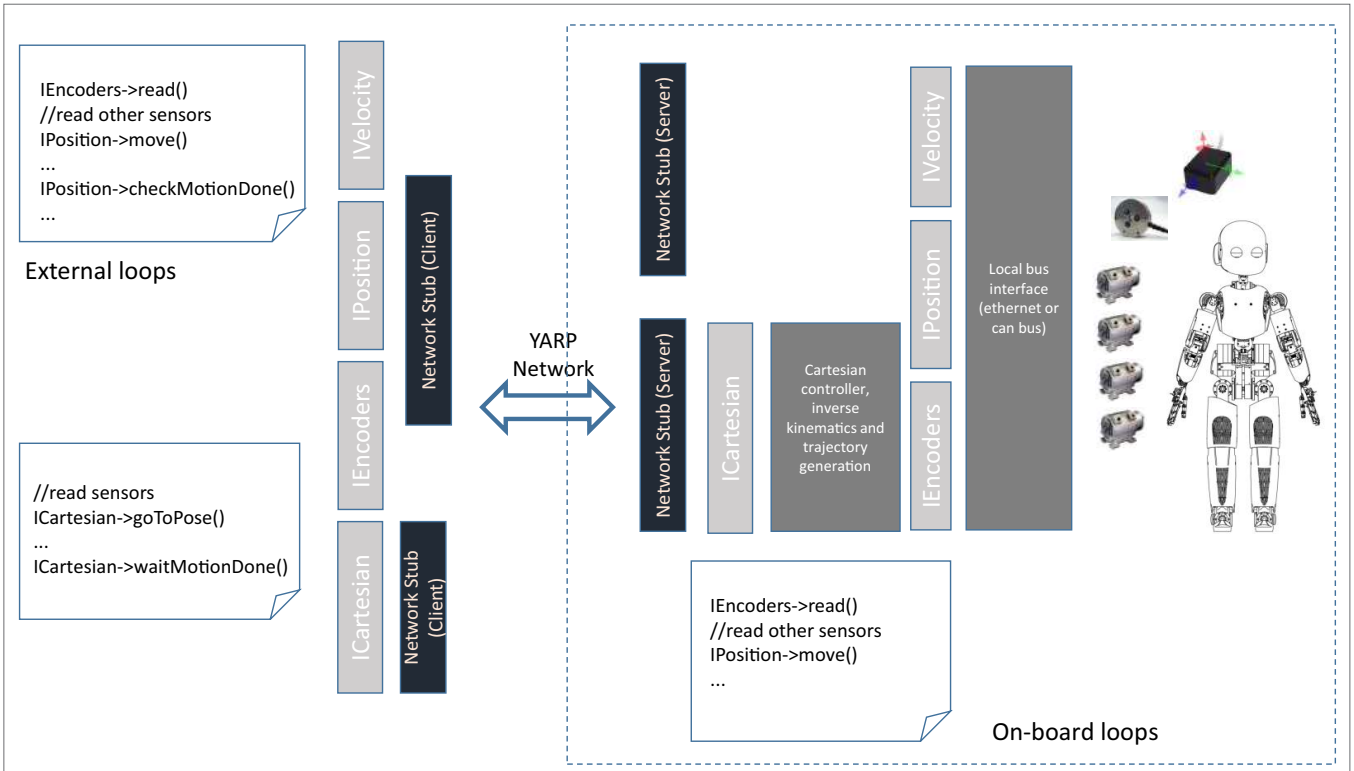


FIGURE 5 | **Interfaces to the robot**. User-level drivers communicate to the hardware using device drivers. These objects (dark gray) implement a set of standard interfaces (light gray) for reading sensors and controlling the motors at the joint level. Other controllers can be connected (*via* an action called *attach*) to these drivers to implement control loops (like for example Cartesian control of the limbs). Special network stubs (black) export the functionalities of the drivers through the YARP network and allow users to execute control loops externally to the robot. To do so, the user instantiates clients network stubs that implement the same interfaces by dispatching requests through the network using YARP *Ports*. These messages are received by the remote servers, which translate messages *via* function call to the hardware device using the user drivers. If needed, the server also takes care of preparing the reply and sending it to the client, which, finally, returns control to the user code.

the robot underwent subsequent revisions of the electronics and improved with the addition of new sensors and capabilities like impedance or force control. The set of robot interfaces are generic and have been implemented various simulators, thus allowing switching control of simulators and the real robot at no cost (Tikhanoff et al., 2008; Mingo et al., 2014; Habra et al., 2015).

The startup of the robot consists in running a single executable called *robotinterface*. This executable reads a configuration file that specifies the list of drivers to be instantiated and their parameters. In addition, the configuration file may specify a list of special objects that perform specific operations using the drivers. The first example is the network stubs objects that provide remote communication. Other objects are calibrators and controllers. Calibrators are objects that implement routines for calibrating the joints and bringing them in a home position (parking). Controllers are objects that implement control functionalities using low-level drivers; an example of this is the Cartesian Controller, which implements on top of joint-level controllers the functionalities required to control the arm (or the head) in Cartesian space. These controllers are time-critical and safety-critical and therefore must communicate with the low-level drivers with minimal latency using function calls (and avoiding network communication).

To allow robot configuration and shutdown *robotinterface* implements, a state machine that can be configured to execute custom activities with a predefined order. **Figure 6** describes the finite state machines. Immediately after execution, *robotinterface* instantiates all objects passing the required parameters (usually specified in a XML file); it then enters the *startup* phase. The latter is further configured to execute custom actions with a specific order. These actions include calibrating the robot and invoking *attach* functions to configure high-level objects. These operations are executed in reverse order to park the robot and uninitialize the objects during *shutdown*. *robotinterface* also defines states that are triggered upon reception of termination signals to interrupt or abort on-going operations. The left side of **Figure 6** illustrates the finite state machine and provides further details. An example of a typical execution of *robotinterface* in a typical run is reported on the right in **Figure 6**. *robotinterface* opens a YARP *Port* that can be queried to know the its state of execution of *robotinterface*. This functionality was added recently to allow modules to make sure that the robot is functional and synchronize their execution with the termination of the *startup* phase.

As we discussed above, the robot interfaces are generic and can be implemented for robots other than iCub. All objects that

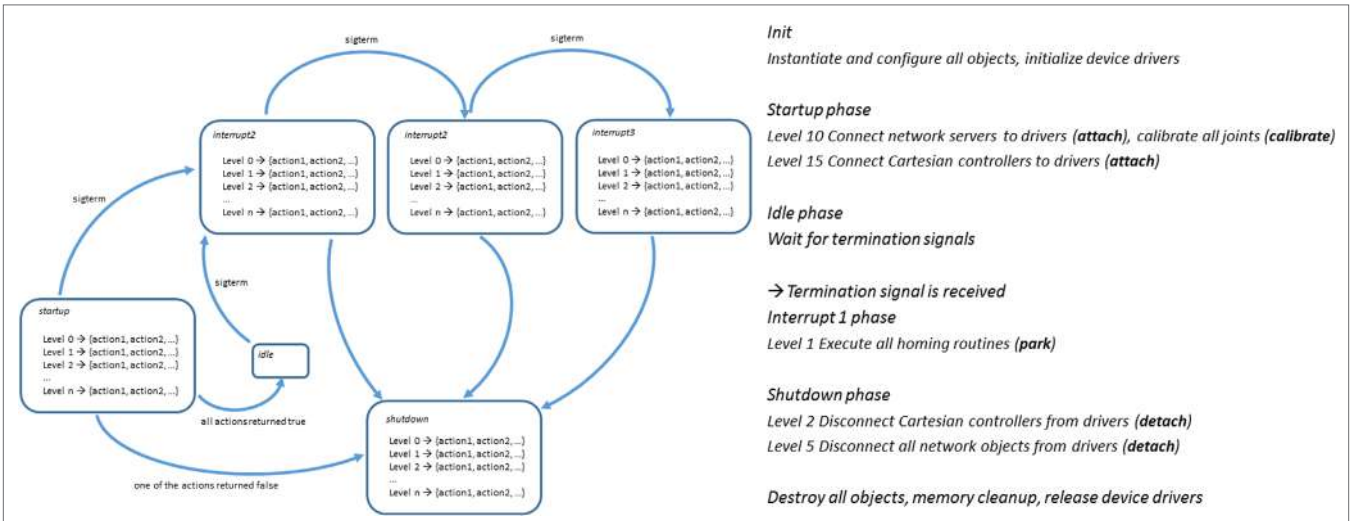


FIGURE 6 | State machine of *robotinterface*. Left: We defined a set of *phases* that are executed upon activation of certain events, namely: *startup*, *interrupt1*, *interrupt2*, *interrupt3*, and *shutdown*. Each phase is further separated in a set of numbered *levels* each linked to a specific set of *actions* to be executed. When entering a given phase *robotinterface* executes all the actions that correspond to the levels that have been configured. Actions at the same levels are executed in parallel in different threads; execution waits for the termination of all threads before proceeding to the next level. In the configuration files, each object can specify one or more actions to be executed, when to execute them (i.e., the *phase*) and in which order (the *level*). The *startup* and *shutdown* phases are executed, respectively, at the beginning and before *robotinterface* terminates execution. The phases *interrupt1, interrupt2, interrupt3* are entered upon subsequent reception of the termination signal (in Linux it is the *SIGTERM* signal) and are followed by the *shutdown* phase. During *startup robotinterface* configures the devices. All network stub objects and controllers register an *action* called *attach* specifying the device they will use to perform their tasks. This is usually done at level 1, i.e., just after all devices have been created. They also register the action *detach* in the phase *shutdown* (level 5, i.e., after the robot is parked). The calibrators objects register actions *calibrate* in the *startup* phase. This is done at level 10, after all *attach* are executed. Calibrators also register the action *park* corresponding to phase *interrupt1* (level 1) so that the robot is brought to the homing position upon reception of the first termination signal. The calibrators also support an *abort* action that is executed to abruptly stop any on-going parking or calibration routines after the third termination signal is received. This action is registered for execution at level 1 in the *interrupt3* phase. The objects that perform Cartesian control are instantiated during the *startup* phase and execute *attach* to get an instance to the joint-level controllers. Proper operation of the Cartesian controllers requires that calibration is performed and that all network objects are instantiated; the *attach* action for these objects is therefore scheduled at level 15. Symmetrically, their disconnection (i.e., *detach* action) is scheduled early on during *shutdown*, i.e., at level 2. Right: Main steps of execution of a typical run of the iCub robot.

are instantiated by *robotinterface* have been implemented using the YARP plug-in system. This means that *robotinterface* is not statically linked against any of the libraries or device driver's API that are required to operate the hardware devices. All the objects instantiated by *robotinterface* are contained in dynamic libraries that are loaded at runtime and can therefore be compiled, maintained, and distributed separately for each robot.

4.1. Motor Control Interfaces

During the development of the robot the motor interfaces underwent several revisions. To the original position and velocity control interfaces, we have added interfaces for open-loop, torque, and impedance control. In this paper, we revise the control modes implemented on the iCub and the corresponding interfaces. Full specifications can be found in the iCub control modes specifications document (Randazzo, 2004) and online in the YARP documentation of control board interfaces.[15]

An important concept in iCub is the fact that high-level components need to be aware only whether the robot can receive position, velocity commands, or open-loop commands controlling the reference for the controller directly. However, they do not need to know how the low-level controllers implement these functionalities. This is achieved by separating control modalities in groups and by defining a special control mode called *interaction mode*. The interaction control mode determines whether the low-level controls PWM with a PID (*stiff interaction*) or the torque in closed-loop (*compliant interaction*). Stiff interaction is the conventional control mode of industrial robots, which are required to execute accurate position/velocity trajectories in controlled environments. In compliant interaction mode, instead, it is possible to control the joint impedance (i.e., stiffness and damping) during the execution of position or velocity commands. To simplify usage many of the interfaces provide functions for controlling individual, all or only a subset of the joints in a kinematic chain.

IPositionControl and *IVelocityControl* define the simplest form of control for the robot. With *IPositionControl*, the controller receives a new reference position, and it generates a trajectory that smoothly interpolates the current and desired state of the robot (position and velocity). On the iCub, the trajectory generator produces velocity profiles that follow the minimum jerk principle. In addition, the interface allows setting the acceleration and velocity values that will be used to generate the trajectory. *IVelocityControl* requires joints to move with a certain speed. The controller accelerates the joints (using a user-defined value) until it reaches the required speed. This control paradigm is suitable for visual servoing, typically for controlling the end-effector or the robot gaze using vision.

IPositionDirect has been introduced mode has been introduced lately to accommodate specific research requirements. It allows skipping trajectory generation and immediately setting the reference value of the position controller. This modality allows generating custom trajectories in small incremental steps.

ITorqueControl sets the torque exerted by a motor. This control mode requires that force feedback is available and that a proper torque loop is implemented. Finally, the *IOpenLoopControl* interface allows to by-pass all controllers and set the PWM reference of the motors directly. This interface is used mostly for fast prototyping control algorithms or for identification. An example is the implementation of control algorithms that use and estimate the parameters of the motors (i.e., back-emf, friction etc.).

The interfaces described above provide the first layer of control at the joint level. Higher level interfaces have been defined for controlling kinematic chains in Cartesian space either in position or orientation (Pattacini et al., 2010). These interfaces separate the robot in different kinematic chains that are controlled independently. Research is today progressing further and new work is currently being done to coordinate whole-body movement for balancing and locomotion (Nori et al., 2015). To support this research, a specific interface for whole-body control is currently being developed.

The state of the robot is available through interfaces that expose joint encoder values (*IEncoders*), motor currents (*IAmplifierControl*), and allow setting and getting control modes (*IControlMode*). An important improvement in the latest revisions of the iCub has been the introduction of additional high resolution encoders that measure the position of the motor shaft (rotor). To give the user access to this additional information, we introduced a new interface (*IMotorEncoders*). Other sensors like F/T, tactile sensors, and IMU are mapped into a generic interface for analog sensors (*IAnalogSensor*), which gives methods for reading the most recent sensor values and perform calibration by setting the zero.

Interfaces are a powerful abstraction. However, getting access to the hardware solely through generic interfaces may be restrictive because a developer often needs access to functionalities that are hardware specific. This usually happens for testing and debugging especially when new functionalities are added to a robot. In this case, there is a big pressure to extend interfaces to make them accessible to the higher levels, and this forces premature design choices and unnecessary code refactoring. For this purpose, we defined a new interface (*IRemoteVariables*) that gives access to generic variables identified with a string of text. This interface defines methods to get the list of available variables as well as methods for setting and getting them individually. This interface can be used to read and manipulate quantities inside the memory space of the control boards. It can be conveniently used for monitoring or changing the internal state of a board for testing, debugging, and code fast prototyping.

4.2. Remotization

As explained in the previous section, YARP provides special objects that remotize interfaces across the network. This is achieved using network objects that come in client-server pairs (identified as network stub in **Figure 5**). The client is loaded locally in the user code; it converts function calls into messages that contain all the parameters and dispatches them across the network to the remote server. Communication is done using three *Port* objects: one for RPC and two for unidirectional communication to and from the server with reduced latency.

[15] http://www.yarp.it/namespaceyarp_1_1dev.html

The RPC *Port* dispatches all the function calls that require a reply and that are not time-critical. Examples of such functions are: getting or setting the PID and changing control modes. Notably RPC also handles commands for moving the joints in position mode with trajectory interpolation. This is because when sending requests to the server, the client always waits for an acknowledgment message. This prevents flooding the server or the network with requests and ensures that no commands are lost. This is not a problem because the additional delay is negligible with respect to the typical time requested by a joint to complete a trajectory.

The other interfaces for joint control (*IVelocityControl*, *ITorqueControl*, *IOpenLoopControl*, *IPositionDirect*) send data to the server using *BufferedPort*. In this case, buffering policy ODP avoids that latency is accumulated in the control loops. Finally, the state of the robot is collected by the server and broadcast periodically using another *BufferedPort*. The client stores this message as soon as it is received and propagates its most recent content upon request by the user (thus avoiding explicit requests). This strategy is convenient because it avoids the need for the client to perform remote requests, and it reduces latency; for this reason the state of the robot has been extended to include not only motor encoders (as it was initially) but other variables like motor currents, speed, acceleration, torque, and status flags. Using the functionalities described in Section 5, the *BufferedPort* that broadcasts the state of the robot can be configured as a ROS topic that publishes the "common" ROS joint state message (*sensor_msgs/jointState.msg*[16]). This allows better interoperability with ROS, for example, using ROS visualization tools like the popular *rviz* GUI.[17]

5. USING THE iCUB SOFTWARE WITH OTHER ROBOTS

Recent efforts have been devoted to provide functionalities at the level of the middleware to interoperate the iCub software with other robots. This was achieved in two ways: (1) by extending YARP so that it provides compatibility with the ROS middleware and (2) by extending YARP *Ports* so that they can be dynamically configured to execute code that manipulates input and output data. The second mechanism was adopted to interoperate iCub with the ARMARX software system. As the latter approach has been already described elsewhere (Paikan et al., 2015c), we provide here more details on the extensions that allow YARP to interoperate with the ROS middleware.

ROS is today the most popular middleware for robotics. Similarly to YARP, it supports developing software architectures based on the publish-subscribe paradigm. A distinguishing feature of ROS is that it requires the user to define all types that are transferred on the network with an IDL. Thanks to this ROS can statically and dynamically check that all the parties involved in a communication expect the same type. Like YARP, ROS uses a central name server which, optionally, can store parameters. To communicate with ROS, YARP had to be extended with protocols that allow communication between the name servers and with the ROS topics. More importantly, the type system of YARP had to be extended to support translating ROS data types in YARP compatible structures.

YARP can be configured to use the ROS name server (*roscore*). This is the simplest solution for ROS users, although it implies that some functionalities are not available in YARP (for example the multicast protocol). Alternatively, the YARP name server (*yarpserver*) can be configured to talk to *roscore* and to propagate queries and topic registrations from/to the ROS. We decided to implement both solutions because each addresses the needs of the YARP and ROS communities.

YARP can register nodes. This can be done in code using dedicated functions in the API or dynamically by using a special syntax when registering a YARP *Port* (for example, the *Port* name */reader@/chatter* creates and associates a topic called */reader* to a node called */chatter*).

YARP *Ports* can understand messages coming from ROS topics or generate valid ROS messages. To do so, YARP needs to be aware of ROS types. We identified two application scenarios. In one case, the user has both YARP and ROS installed. This is the easier case because YARP can read types directly from the ROS installation, and using a compiler (i.e., *yarpidl_ros*), it can generate appropriate data structures. In the second scenario, ROS is not installed on all machines that are running YARP. For this situation YARP provides a type server that can answer queries at run time and provide to YARP programs the information they need to interpret ROS messages.

As an example, **Figure 7** shows the code required to control ROS's *turtlesim* from YARP. Similarly, **Figure 8** shows how to dynamically connect an existing YARP *Port* to a ROS topic without recompiling the code.

6. COMPONENT REUSABILITY AND COORDINATION

Developing high-quality reusable software components requires careful design that strikes a good balance between potential reuse, functionalities, and ease of implementation (Sametinger, 1997). Coordination of software components in a distributed architecture usually adds considerable overhead to the robotic application design process and, often, pulls the development of software components in a specific direction. This can have a negative impact on the development process and can reduce reusability of software components.

Software should be extensible enough to be adapted to possibly unanticipated changes (Zenger, 2004). One direction to extend a module is *via* its interfaces. In YARP, interfaces are implemented by exchanging messages through the middleware connection points (*Ports*). To enhance the reusability of the iCub software, we extended the *Port*'s functionality so that it can dynamically load and execute a run-time script. In our framework, this port extension is called *Port Monitor*: in brief, it allows accessing data passing though a connection from/to the *Port* for monitoring, filtering, and transforming it. Multiple instances of *Port Monitor* can interact to allow an input *Port* to select data from multiple

[16]http://wiki.ros.org/common_msgs
[17]http://wiki.ros.org/rviz

```
#include "Pose.h"
#include "Twist.h"
/* create ROS Node /controller */
yarp::os::Node node("/controller");
/* create a subsriber for Pose.msg */
yarp::os::Subscriber<Pose> pose;
/* subscribe to /turtle1/pose  */
pose.topic("/turtle1/pose");
/* create a publisher for Twist.msg */
yarp::os::Publisher<Twist> cmd;
/* publish to /turtle1/cmd_vel */
cmd.topic("/turtle1/cmd_vel");
while(true) {
      Pose p;
      /* read a new value from the topic */
      pose.read(p);
      cout<<p.x<<" "<<p.y<<" "<<p.theta<<"\n";

      /* prepare a command for the turtle_sim */
      Twist t;
      t.linear.x=1.0;
      t.angular.z=1.0;

      /* publish the command */
      cmd.write(t);
}
```

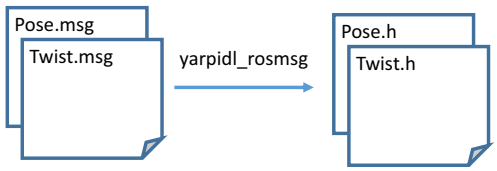


FIGURE 7 | Interfacing YARP with ROS's topics. The figure illustrates the code required to interface YARP with the *turtlesim*. Starting from the ROS messages definition in *Pose.mgs* and *Twist.msg* the *yarpidl_rosmsg* tool produces compatible YARP types, *Pose.h* and *Twist.h*. The code creates a ROS compatible node, a subscriber and a publisher, reads data that contains the state of the *turtlesim* from the topic */turtle1/pose*, and sends velocity commands to the topic */turtle1/cmd_vel* to control it and sends velocity commands to control it.

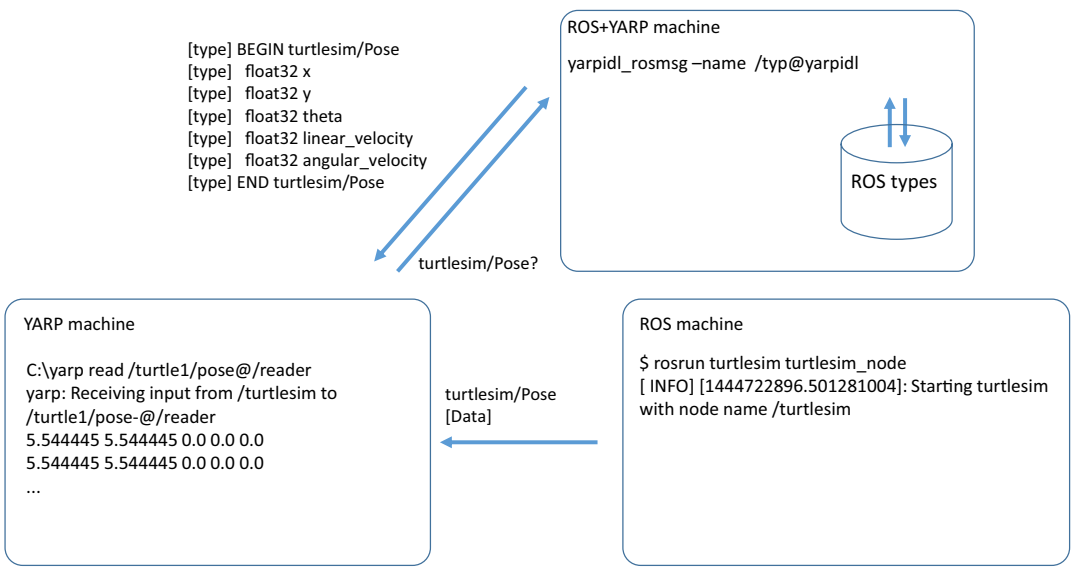


FIGURE 8 | Reading a ROS topic from an existing YARP *Port*. This figure shows how to read the output of the topic */turtle1/pose* from an existing YARP *Port*. On the left, we run the *yarp read* tool, which opens a *Port* and prints to console the data it receives. To make ROS aware of this *Port*, we decorate its name using a special syntax that instructs YARP to create a ROS node called */reader* and subscribes it to the topic */turtle1/pose*. To properly interpret the type *turtlesim/Pose* YARP queries the typeserver. Notice that all processes are running on different machines and that, in this case, the machine that runs *yarp read* knows nothing about ROS. To simplify the figure, we omit all the handshakes required for naming look up and establishing the connections. The opposite operation (writing from a YARP *Port* to a ROS topic) can be done in a similar fashion.

sources in an exclusive way. We call this mechanism *Port arbitration*: it allows coordinating components by specifying arbitration rules in the input port of a component (Paikan et al., 2014b).

6.1. Port Monitoring and Arbitration

Figure 9 (left) represents represents the situation in which the *Port Monitor* (shown as a box with M) is attached to the output

of the Face-Detector module and the input *Port* of the Head-Control module. The *Port Monitor* can load a script file [written using a standard scripting language such as, in our case, Lua (Ierusalimschy et al., 1996)] and can access and modify the data traveling through the *Port* using a simple API. This idea allows adding extra functionalities to a component like data filtering, transformation, and monitoring without modifying or rebuilding it (Paikan et al., 2014a).

As an example, the code below illustrates the pseudo-script in Lua that filters messages from Face-Detector when its confidence level drops below a defined threshold (in this case 0.8):

```
PortMonitor.accept = function(data)
  -- read face_pos from 'data'
  if face_pos.certainty <0.8 then
     return false
  end
  return true
end
```

Using the *Port Monitor*, an *input Port* can be configured to arbitrate data from multiple sources, based on user-defined constraints. **Figure 9** (right) represents a simple application where a humanoid robot looks around in search of a person's face and then tracks it. This is a common coordination problem, which can be solved in different ways (e.g., using a separate coordinator or by extending modules to interact with each other). One way to achieve this is to use a selector in the input *Port* of the Head-Control module and constrain it to receive data from each module under specific conditions. The concept is shown in **Figure 9** (right) where the *Port Arbitrator* is used in the input *Port* of the Head-Control (shown as box labeled with two "M"). The arbitration logic can be written using the Lua scripting language and is loaded by the *Port Arbitrator*.

In our approach, a *Port Monitor* is attached to each connection that delivers data to a *Port*. The *Port Monitor* analyzes the data it receives and produces (or removes) events from a container. A set of constraints in boolean logic determines from the events and for each connection if data is allowed to be delivered to the component (otherwise it is discarded). Arbitration is achieved because only one connection at the time is granted the permission to deliver data to the component. This type of arbitration mechanism can be effectively used to implement complex tasks without resorting to centralized coordinators (Paikan et al., 2013).

7. CONFIGURATION OF COMPONENTS

Components support re-use by exposing a set of parameters that affect their behavior. Parameters that are specified at the command line configure the component when it is executed.

Components choose the name of the *Ports* that they use. Because different *Ports* cannot have the same name, component reuse can be achieved only if components provide a way to change the name of the *Ports*. This is so fundamental that YARP enforces it *via* the environment variable *YARP_PORT_PREFIX*: if specified this variable defines a prefix that is added to all the *Ports* within a component (similarly ROS provides a way to remap topic names at the command line: in YARP, this solution was not viable because YARP does not monitor command line parameters).

All other configuration parameters should be specified at the command line, either explicitly or using configuration files. YARP uses a directory hierarchy to organize these files. One problem we had to solve was how to allow users to add configuration files to the ones already existing and how to package them in binary distributions. To support packaging, YARP defines a set of OS-dependent default locations for files and a set of rules that define the search priorities. To provide users with the freedom to install the software on custom directories, these default locations can be overridden or extended by modifying certain environment variables. YARP provides an helper class that allows organizing and locating configuration files, this class is called *ResourceFinder* because it allows managing all types of configuration files (called *resources* in YARP terminology) for a component.

The design of the *ResourceFinder* follows the rationale adopted in the Linux OS, i.e.,

- The software installation should be able to provide reasonable defaults for configuration files so that applications can run out-of-the-box;
- Installation directories are generally non-writable, users without root privileges cannot edit installed configuration files unless they first copy them inside their own private directories, the latter must take precedence and hide the others;
- Therefore it is normal that configuration files can be in multiple places, inside user-specific, private or shared installation directories;
- External packages can install files so that YARP can find them;
- Files are organized in families that are placed in specific sub-directories.

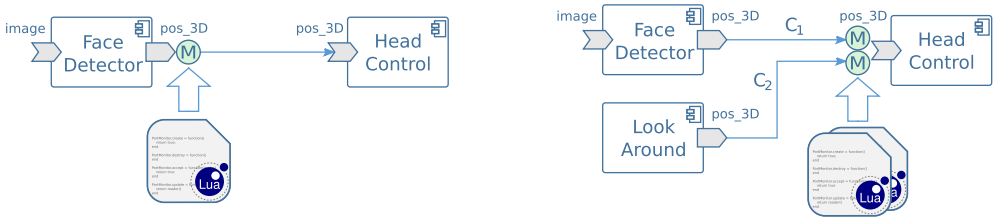


FIGURE 9 | Conceptual representation of *Port Monitor* (left) and its use for arbitration (right). Left: the output *Port* of Face-Detector is extended with a plug-in, which provides access to the outgoing data through scripting language (e.g., Lua). Right: at the input *Port*, the *Port Monitor* can arbitrate data from multiple connections.

Resource files belong to the following families: *modules, applications*, and *plugins*. *Modules* and *plugins* are text files describing modules and plugins (i.e., *manifest* files). *Applications* are files required to instantiate, configure, and connect components usually to achieve a certain task (called *application*). Configuration parameters for components are organized as key-value pairs and stored in one ore more configuration files. Different files, therefore, configure a component depending on the application. To easily switch configuration, YARP components support the parameter *--from* which allows reading configuration files from a well-defined directory (the *context* of execution). For example, the following commands:

```
myModule --from experiment1
myModule --from experiment2
```

execute *myModule* in two different ways, using files in the contexts *experiment1* and *experiment2*, respectively.

The directories *modules, applications, plugins*, and *contexts* are installed in the system directory *<prefix>/share/yarp*.[18] Users have read-only access to system directories, and they need to copy the files they want to edit in private directories in which they have write access.[19] To install and remove resource files, YARP provides the *yarp-config* tool.[20]

The *ResourceFinder* searches for configuration files in the following order of precedence:

- First, it looks in the current working directory;
- Then, it searches within *contexts* in the user private directory;
- Finally, it searches within *contexts* in the shared, installation directory(ies).

When searching for files and directories, the *ResourceFinder* follows the above order, so that files in the working directory or those modified by the user take precedence over installed ones. Search of files proceeds from the user private directories to shared installation directories. The same *context* directory can appear in multiple places and is likely to contain files with the same name. In this case, files that found first take precedence and hide those in other locations. This shadowing or masking mechanism is useful when the user needs to customize only a subset of the files for a specific context.

To allow users to modify where files are stored or to add other *contexts* to the existing ones, the *ResourceFinder* search path can be extended in two ways. Through the *YARP_DATA_DIRS* environment variables, a user can specify a list of locations, each used by the *ResourceFinder* when looking for shared installation directories. Third party package developers can add a text file that contains additional search directories in the directory *<prefix>/yarp/path.d*. This solution allows an installation package to extend YARP search path without requiring changes to the environment (to simplify this task YARP offers a set of CMake functions). For example, a user can install the YARP middleware and the iCub additional software without changing the environment. This is a useful feature for packaging and because we have found that modifying the environment is confusing and error prone for most users.

It is worth mentioning that the *ResourceFinder* follows the XDG Base Directory Specification for Linux systems.[21]

8. APPLICATION MANAGEMENT

To facilitate the application development and execution for the iCub robot, we developed a few graphical tools. The *yarpbuilder* (see **Figure 10**) enables users to easily develop an application by configuring and interconnecting the available modules. It makes use of a YARP module description in XML format and represents them as graphical entities. To build a new application, a developer can drag and drop modules, configure, and interconnect them. This tool also performs some simple model checking to ensure that some of the constraints, such as required input connections or parameters for a module, are satisfied for that specific application.

Using a resource description of the available machines in a cluster, the deployment information can be manually set for the execution of the modules or they can be configured to be deployed using the automated load balancer. Eventually, the application can be created and launched using the *yarpmanager*[22] deployment tool (see **Figure 11**). It has been developed using a multilayered software architecture, which abstracts the representation of modules, resources, and applications from their execution. The latter can employ different deployment methods (e.g., *yarprun* or *SSH*), which potentially allows executing components from different robotic middleware. The *yarpmanager* provides a rich set of functionalities such as module configuration, execution and monitoring, cluster resource discovery, load balancing, as well as establishing and checking connections. We are currently working toward integrating the *yarpbuilder* and *yarpmanager* in a single tool in which applications are developed, executed, and monitored using the same graphical representation.

9. SOFTWARE TESTING ON THE iCUB

Testing is an important topic in software engineering that has, however, received little attention in robotic research. Because robots in the future are expected to work in close interaction with humans, safety of robotic systems is going to become a fundamental issue. In this context, it is likely that software testing will play an important role. To test the iCub software we have adopted strategies for static as well as dynamic testing.

Static testing consists in checking the code without executing it. This approach includes code inspection, peer-to-peer reviews, and code verification using formal techniques. Code reviews allows increasing software quality by removing common problems and enforcing coding styles to improve readability, especially when development happen in a in distributed, open-source

[18] The actual value of *<prefix>* is system dependent, on Linux it is usually equal to */usr/share*, while on Windows it maps to *%PROGRAMFILES%/YARP,* i.e. usually *C:/Program Files/YARP*.

[19] On Linux systems this is *$HOME/.local/share/yarp* while on Windows it is *%APPDATA%/yarp*.

[20] Online documentation is available at: http://www.yarp.it/yarp_yarp-config.html

[21] http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html

[22] http://www.yarp.it/yarpmanager.html

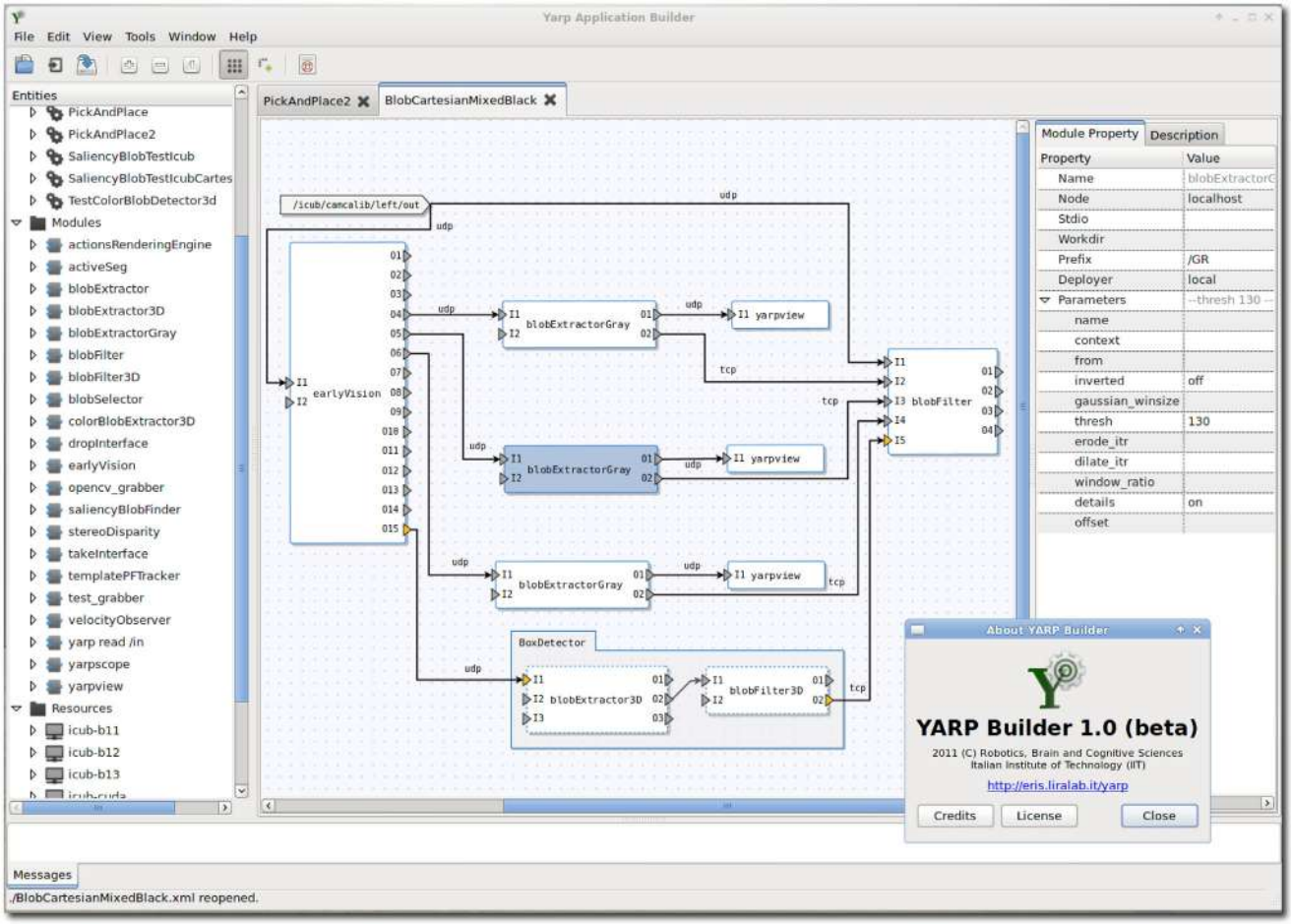


FIGURE 10 | A screenshot of yarpbuilder. The GUI allows to build *applications* by dragging and dropping *components* and by wiring the connections. The *applications* designed in the *yarpbuilder* can be executed by the *yarpmanager*.

community. On the iCub code reviews have been adopted only recently and so far mostly in the context of the development of the YARP middleware. At this aim, we rely on the functionalities offered by github,[23] the web service that hosts the majority of the repositories we manage. Non-trivial modifications to YARP are developed on a separate branch and are integrated in the master (the main development branch) only after other developers have revised and approved it. Common problems that have been identified through code reviews are: race conditions, memory leaks, and buffer overflows to mention a few. An important feature provided by github is the possibility to test software patches *before* they are integrated in the main branch (using Travis[24]). However, Travis supports only a specific distribution of Linux (at the time of writing Linux Ubuntu 12.04 LTS). Therefore, compilation tests are also executed for all the supported platforms on our compile farm. This happens periodically (nightly builds) and upon any commit (continuous builds) to the YARP and iCub main repositories.

Static code analysis can also be performed automatically using model checking techniques [see Baier and Katoen (2008) for a review]. These techniques allow ensuring that a piece of code satisfies given requirements (usually expressed in temporal logic). Model checking is preferable to other techniques because it explores systematically the behavior of a program and is completely automated. However, it requires that a model of the software is available – for example in the form of a finite state automata – and hardly scales to large systems. It has had practical applications in the verification of circuits and protocols (Kaufmann et al., 2000). In recent work, we have investigated the use of automatic techniques to derive a model of some of the components of the YARP middleware (namely *Ports* and *BufferedPorts*) and successfully applied model checking to verify properties of code that uses YARP (Khalili et al., 2014). Although promising, these techniques still require a certain degree of manual intervention from an expert and do not scale well to large programs. Further research is still

[23]https://github.com
[24]https://travis-ci.org/

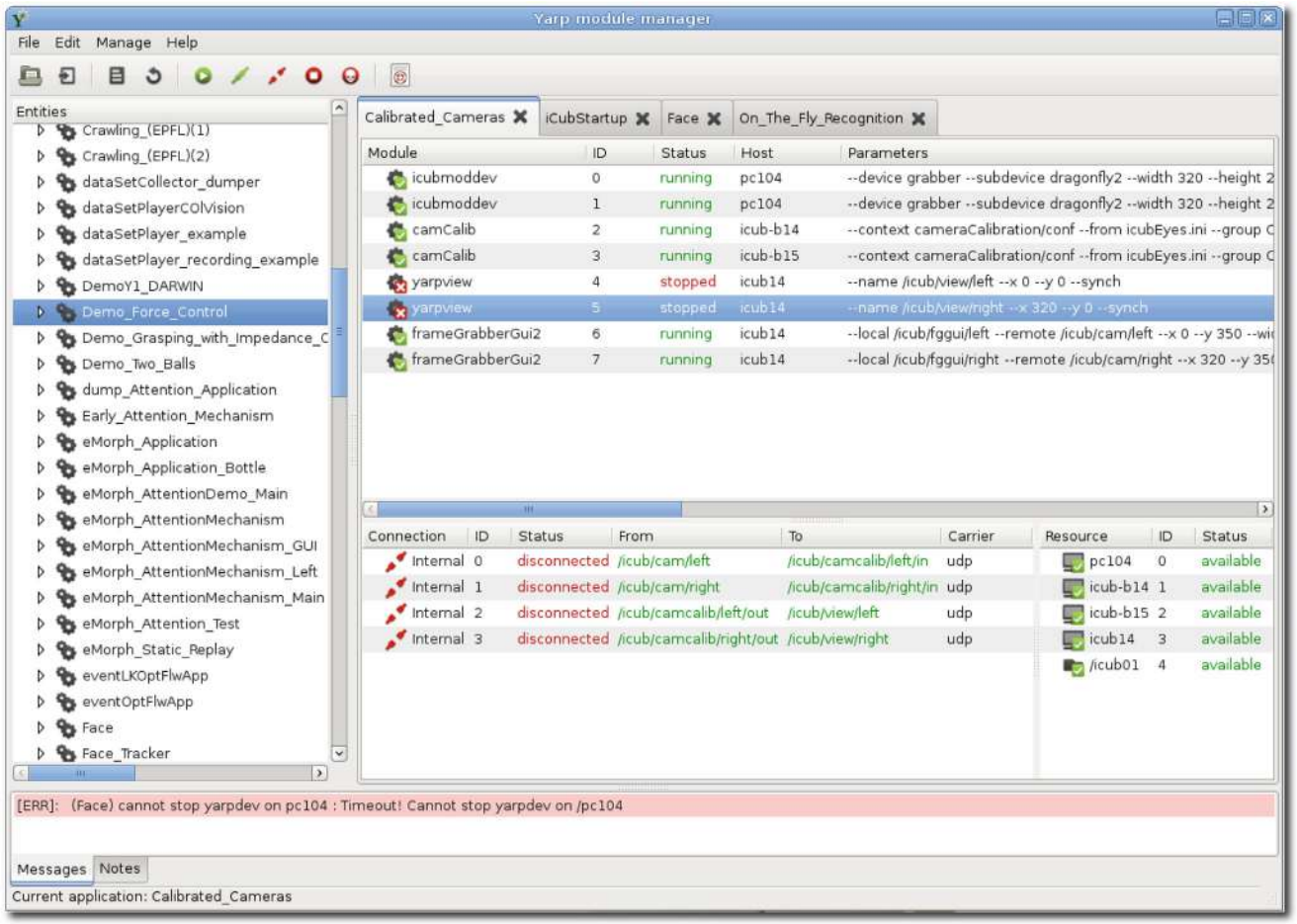


FIGURE 11 | A screenshot of *yarpmanager*. On the left, the GUI shows the list of *applications* that are available and can be loaded. Each tab in the center window contains one of the four *applications* that have been loaded. For each *application*, *the GUI shows the status of the components, the host in which it should be ran and the parameters. The bottom windows show the status of the* Port *and connections that are used by the current* application*. The bottom-right window shows the status of the nodes in the network.*

required before they can be integrated in the software development workflow.

Dynamic testing on the other hand consists of running and testing the dynamic behavior of the code. This involves writing specific pieces of code, specifically devised to exercise the functionalities of the software and verify that it complies with the specifications. Test-driven development has gained increasing attention in software engineering (Beck, 2003). Proper application of this technique requires (i) alternating writing tests and developing functional code in small and rapid iterations and (ii) executing tests automatically to ensure that modifications to existing code (new components, bug fixes, or new features) do not break existing functionalities. In the remainder of this section, we describe how unit-testing has been adopted for testing the software on the iCub robot.

9.1. The Robot Testing Framework

Unit-testing was adopted for the development of YARP since the beginning, whereas systematic testing of the software that controls the iCub was started only recently. This is because robot software cannot be tested in isolation, and it requires running other components, device drivers, or just the robot simulator (these resources are called *fixture*). Automating testing requires therefore that the testing framework is able to setup the required resources and monitor them to ensure that they remain functional during the execution of the tests, offering hooks to handle failure appropriately (i.e., restarting the robot, performing parking routines, etc.). To comply with these requirements, we developed the Robot Testing Framework (RTF) (Paikan et al., 2015d).

The RTF[25] is a generic testing framework for test-driven development of robotic systems. Its architecture is detailed in **Figure 12**. It is based on the well-known xUnit test patterns, which includes a test runner, test result formatters, and a test fixtures manager. In addition, it provides functionalities for

[25]The source code and documentation of the RTF can be accessed on-line at http://robotology.github.io/robot-testing/index.html.

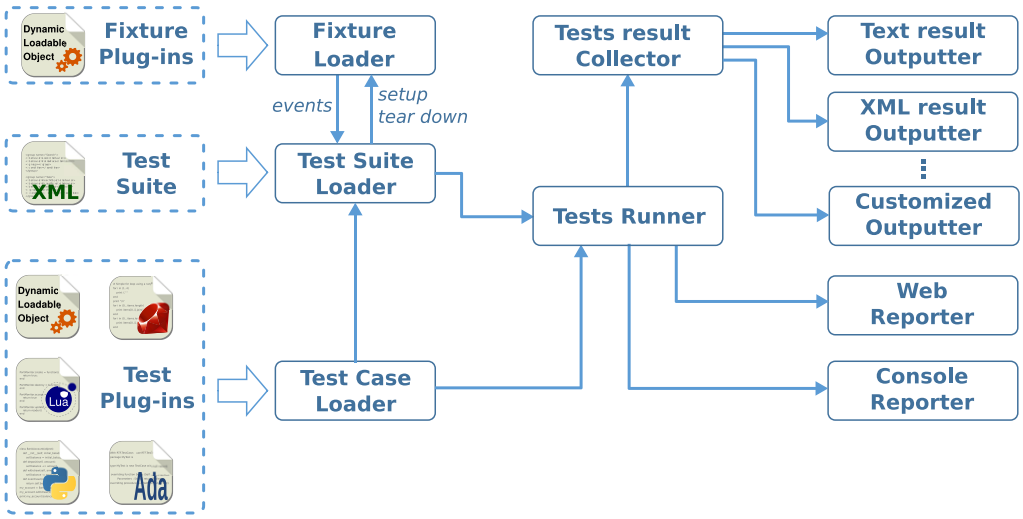


FIGURE 12 | The architecture of the *Robot Testing Framework*. Test cases can be developed as independent plug-ins using scripting languages or can be built as dynamically loadable libraries. The plug-ins are loaded by the *Test Case Loader* and are executed by the *Test Runner*. Test cases can also be grouped in different test suites, which are represented using XML. In the latter case, the *Test Suite Loader* parses the XML file and, using the *Test Case Loaders*, it loads the corresponding test plug-ins. Each test suit can optionally have a fixture manager, which is implemented as a separate plug-in (which is loaded by the *Fixture Loader*). This fixture plug-in is responsible for setting up the fixture and informing the *Test Suite Loader* when the fixture fails (e.g., crashes). In this case, the *Test Suite Loader* restarts the fixture and resumes execution of the remaining test cases. The result of the tests can be monitored from the console (through the *Console Reporter*) or remotely from a Web browser (through the *Web Reporter*). The *Test Result Collector* allows storing data in different formats. For example, this figure shows two components for storing output in text format or XML, (*Text result Outputter* and *XML Result Outputter*, respectively).

defining test cases (i.e., unit tests), suits, and assertions. RTF supports multiple middleware, languages, and operating systems. This is achieved by providing abstraction layers for the platform (i.e., operating system), the middleware, and the programing language. Moreover, RTF provides functionalities for managing complex fixtures, which support stress testing at the level of individual components (robot hardware like sensors or actuators) as well as integrated (sub) systems. It is worth pointing out that the RTF is not a tool for static analysis of code. As such it does not perform an exhaustive analysis of the code to ensure given properties of absence of deadlocks. The code is evaluated by running tests that call individual functions and verify the value of the return parameters or observe the internal status of the system under test by calling specific functions. Deadlocks can be detected indirectly by setting a timeout on the execution of individual tests.

A test suite is a set of test cases, which share the same test fixture (Meszaros, 2007). In RTF, a set of test cases (plug-ins) can be grouped as a test suite using an XML file and executed using the test runner. This allows the unit tests to be easily organized in different test suites, which are easy to maintain and extend.

The fixture manager is implemented as a separate plug-in so that it can be implemented using the deployment tools and policy of the middleware of choice. (for example OROCOS uses the component deployer, *deployer-corba*, ROS components use *roslaunch* toolset). For YARP, it is implemented using the *yarpmanager*.

Notably, RTF provides test results in different formats including Junit XML file. This allows the test results to be published and monitored using standard integration tools such as Jenkins. The next section describes the tests that are currently implemented for the iCub. The test cases can be run directly on the iCub robot or using a simulator. Some of the tests are executed automatically on the simulator using Jenkins upon any change in YARP middleware or the iCub software. These tests check that any new update in the software is compatible with the robot interfaces. There are also some tests that check the robot hardware; these tests are executed directly on the robot and under human supervision.

9.2. Testing on the iCub

We conclude this section with a description of the tests that have been implemented so far on the iCub robot. We can distinguish four categories of tests: (i) tests on the correctness of the robot configuration files, (ii) tests for specific hardware devices, (iii) tests for the compliance of low-level software and firmware with system specifications, and finally (iv), tests that are specifically written after a specific bug is identified and fixed.

The first category of tests may seem unusual. There exists about 30 iCub units, and, over the years, many of them underwent hardware customizations, revisions, or upgrades. Therefore, each unit has a specific set of configuration XML files, typically manually written and therefore subject to errors. Automatic tests attempt to minimize such errors by verifying the correct behavior of the robot.

JointLimits, for example, is a test which belongs to the first category. This test checks that range of motion written in the configuration files is achievable by the system by moving each joint to the maximum and minimum position. If a hardware

limit is unexpectedly encountered, the test fails, informing of a possible mistake in the robot configuration files. This kind of tests can be considered rather static over time: additional tests may be added if new configuration parameters are introduced, however, individual tests do not typically require maintenance.

The second category includes all the tests which validate the correct operation of a hardware device. For example, the *OpticalEncodersDrift* test moves a joint generating a sinusoidal trajectory. The test checks that, after a repeated number of cycles, the measurements of the optical encoder do not drift. A drift suggests a hardware defect such as the presence of dust on the encoder disk (for optical encoders) or electrical interference. This category of tests is dynamic: new devices are continuously introduced and new tests have to be designed for the new hardware, while existing tests have to be periodically reviewed as a result of a hardware revision of an existing device.

The third category includes all the tests which verify that the robot behavior complies with the specifications and requirements. The simplest test of this category is *MotorEncodersSignChecks*, which tests that individual encoders increase their value when the corresponding motor rotates with positive input. This is an important convention that determines the sign of the PID controllers but may be affected by incorrect mounting of the encoders, motor wiring, or firmware configuration. As previously mentioned, iCub control boards implements the concept of control mode. If a joint is controlled in position mode, for example, only position commands sent through the *IPosition* interface are accepted, while other commands (e.g., velocity, torque) must be rejected. *ControlModes* extensively verifies that all the possible states described in the specifications of control mode state machine are reachable and that the corresponding transitions are correct. Also prohibited transitions are tested. For example, if a hardware fault occurs (the test intentionally causes a hardware fault by sending an invalid command) the user has to set the joint in idle mode before switching to any other control mode. *MotorTest* exercises the interfaces for reading encoders and moving the joints in position mode. This category of tests typically evolves over time: for example, when a new control mode or interface is implemented, these tests may be extended in order to check the compatibility of the new features with existing ones.

The fourth category of tests is written when individual software defects are discovered. In this case, it is good practice to first write a test that triggers the specific defects and then ensure that a candidate software patch effectively passes the test. These tests remain in the system to ensure that future changes will not cause the same defect to reappear. An example of this test is *SensorsDuplicateRd*, which verifies that sensors values are not broadcast multiple times on a *Port*.

Table 2 lists all the tests that are currently implemented on the robot. These tests are executed periodically on the simulator and manually on the real robot. As the development is currently in progress, the list covers only partially the functionalities that could and should be tested.[26]

[26]The source code and description of the available tests for iCub can be accessed on-line at https://github.com/robotology/icub-tests.

TABLE 2 | Some of the test cases which has been developed for the iCub robot.

Test	Description
`CameraTest`	Checks the robot camera's frame rate and size
`ControlModes`	Checks control mode specifications, validates allowed, and forbidden transitions
`FtSensorTest`	Checks the robot force sensors against a predefined, known value
`JointLimits`	Checks the software joint limits configuration
`MotorTest`	Checks the IPositionControl and IEncoders interfaces, moves the motors individually or in groups, and verifies that the required position is actually obtained within a predefined amount of time
`MotorEncodersSignChecks`	Check that motor encoder readings increase when positive PWM is applied to a motor
`OpticalEncodersConst`	Checks the consistency between encoders at the motors and at the joints
`OpticalEncodersDrift`	Performs repetitive movements to verify that encoders do not drift
`PortFrequencyTest`	Checks the rate at which state information is published by the robot interfaces
`PositionDirect`	Checks the *IPositionDirect* control mode
`SensorsDuplicateRd`	Checks that a YARP *Port* publishes unique values at each update

10. LESSONS LEARNED

We report here a list of lessons we learned during the development of the YARP middleware and the iCub software architecture.

10.1. Freedom of Choice versus Freedom from Choice

The iCub is a research platform developed by and for researchers. We tried to accommodate as much as possible the need for the users of the robot giving maximum freedom in terms of development environment and tools. We support MacOS X, different flavors of Linux, and Windows. In several occasions, this turned out to be a good choice that allowed us to run the code on legacy hardware and to reach a wider group of users. It, however, required to keep support for various versions of compilers and libraries and resulted in considerable maintenance cost, sometimes slowing down the introduction of new features made available in new compilers or libraries. On the software development side, we also gave users freedom to code their components in the way the liked; for this reason, we did not introduce a rigid component model but only provided helper classes and best practices through documentation and on-line tutorials. This choice in the short term was beneficial because it reduced the learning curve, but in the long term may hinder standardization and actually slow down development. A somewhat opposite approach would be to rely on design tools and code generation to leverage the user from the task of implementing infrastructure code [this approach is followed for example by Smartsoft (Schlegel and Worz, 1999)]. Striking a good balance between the two approaches is a difficult design decision, which depends on background and expectations of the target users.

10.2. In-House Middleware

When we started the development of iCub back in 2004, only a few middleware existed in the robotics community and their use was quite fragmented. The first version of YARP was already mature and the consortium that designed the iCub included YARP's core developers in the team. This was one of the main reasons that has motivated the adoption and consequent development of YARP. Having in-house control of the middleware code-base has given us great flexibility to address the needs of the iCub community. Such examples have been described in this paper and include: the definition of the motor control interfaces and corresponding communication paradigm for remotization, the functionalities for remote execution and logging, and organization of parameters. Not less importantly it allowed us to come up with novel extensions like the *Port Monitor* and channel prioritization. YARP can be easily compiled on many Linux distributions, MacOS, and Windows. All these reasons still prevent us today from adopting ROS altogether and motivated us to add support for ROS interoperability, instead. Developing, debugging and maintaining the communication back-bone of the robot, however, has quite a high cost and should not be underestimated when developing a new robot. Our experience has shown that user code become quite entangled with the middleware data structures and build system (*middleware lock-in*). Considered that middleware technology is in constant evolution it may be a good idea to design the software architecture of the robot to reduce such dependency, so that changing middleware is possible and inexpensive.

10.3. Packet Management System

The build system of the iCub software is based on CMake. CMake offered a great degree of flexibility, allowing to customize and to automate the compilation process, including finding and configuring dependencies. However, the software ecosystem suffered from the lack of a sophisticated packet management system. The main stumbling block in this respect was the decision to support Windows, for which there are no mature systems for packet management and distribution. To partly cope with this problem, we developed custom scripts for packaging dependencies in binary distributions for Windows, Linux, and, hombrew recipes for MacOS X. Yet, the development of an iCub software ecosystem was slowed down from the lack of a powerful packet management system like the one that is available in many Linux distributions.

10.4. Lack of IDL

YARP was not born with an IDL language supporting the definition of data type and services. The reason for this choice was to facilitate adoption by reducing complexity and learning curve. Interfaces were developed using self-describing data types (i.e., the YARP *Bottle*), which allowed code to parse messages dynamically by inspecting their content. With the growth of the community, this became a limitation because it made it difficult to document modules interfaces, perform versioning, and verify compatibility between modules. In retrospect it would have been preferable to introduce the IDL much earlier in the development, enforcing its use like, for example, ROS. Finally, another advantage of the IDL is that it allows documenting service interfaces using Doxygen keeping services documentation and their code in the same files.

10.5. External Configuration of Ports

YARP provides different communication policies through the *Port* API (like buffering, streaming versus RPC). However, understanding how components communicate requires looking at the code; this increases the probability of introducing subtle bugs. It would be beneficial if the available polices were visible and configurable at run time. In Section 3, we showed initial steps in the direction for providing policies for channel prioritization; however, further development is required to give users access to the other configuration parameters.

10.6. Robot Interface Abstraction Layer

The robot interface abstraction layer had a positive impact during the development. It allowed to introduce new functionalities *via* new interfaces without affecting existing code and to execute code on-board, remotely on the real robot or simulators. These features gave a useful level of flexibility that facilitated debugging, code re-use, and fast-prototyping. Robot interfaces have evolved with time to accommodate the research requirements and to support new hardware as it became available; we tried to limit as much as possible the impact of these changes on the user code and introducing new functionalities in optional interfaces. The robot interface abstraction layer hides the user code from the details of the communication and the specific communication middleware. This allowed to extend the representation of the data structure that broadcasts the robot state without changes outside library code. In the future, it may even allow us to change the transport layer (i.e., the communication middleware) altogether with minimal impacts. Another improvement in the robot interface followed the refactoring of the plugin system, which was modified to load plug-ins at runtime using *dynamic libraries* (it was initially based on static linking). This decoupled all dependencies and allowed us to separate device drivers contributed by users in separate repositories, resulting in simpler packaging and maintenance.

10.7. Test-Driven Development

The development cycle of the YARP middleware adopted unit testing since the beginning. For practical reasons, this approach was extended only recently to the other layers of the iCub software architecture. The *Robot Testing Framework* is being used to test the low-level code that is developed for the new version of the iCub and although tests still cover a relatively small portion of the code, we already see its benefits: besides detection of bugs due to programmers errors, it allowed us to detect problems that were due to lacking or imprecise specifications. The latter is a problem that is particularly frequent in research environments.

11. CONCLUSION

In this paper, we described the software architecture of the iCub humanoid robot, including some of its recent development. We illustrated the design choices that have guided and constrained its development, including the lessons that we learned during

this endeavor. We do not claim these choices to be optimal and equally good in all cases. However, we hope that in the future, this paper may provide a useful guide for the design of other humanoid robots.

AUTHOR CONTRIBUTIONS

All authors have contributed to the conceptual design or development of the work described in this paper and have participated to drafting and revising its content. They also approve publication of the paper and agree to be accountable for all aspects of the work described therein.

ACKNOWLEDGMENTS

We acknowledge the contribution of the software development team and researchers in the iCub Facilty and Robotics Brain and Cognitive Sciences departments, as well as the contributions of the whole iCub community.

FUNDING

This project has received funding from the European Union's Seventh Framework Programme for research technological development and demonstration under grant agreement No. 270273 (Xperience), project No. 611832 (WALK-MAN).

REFERENCES

Almesberger, W., Salim, J. H., and Kuznetsov, A. (1999). "Differentiated services on linux," in *Proceedings of Globecom '99*, Vol. 1, (Rio de Janeiro: IEEE), 831–836.

Baier, C., and Katoen, J.-P. (2008). *Principles of Model Checking*, Vol. 26202649. Cambridge: MIT Press.

Beck, K. (2003). *Test-Driven Development: By Example*. Boston, MA: Addison-Wesley Professional.

Brooks, A., and Kaupp, T. Makarenko, A., Williams, S., and Oreback, A. (2005). "Towards component-based robotics," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, (Alberta: IEEE), 163–168. doi:10.1109/IROS.2005.1545523

Brugali, D., and Scandurra, P. (2009). Component-based robotic engineering. Part I: reusable building blocks. *IEEE Robot. Autom. Mag.* 16, 84–96. doi:10.1109/MRA.2009.934837

Bruyninckx, H. (2001). "Open robot control software: the orocos project," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, Vol. 3, (Seoul: IEEE), 2523–2528.

Bruyninckx, H., Klotzbücher, M., Hochgeschwender, N., Kraetzschmar, G., Gherardi, L., and Brugali, D. (2013). "The BRICS Component Model: A Model-Based Development Paradigm for Complex Robotics Software Systems," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, (Coimbra: ACM), 1758–1764.

Calisi, D., Censi, A., Iocchi, L., and Nardi, D. (2012). Design choices for modular and flexible robotic sofware development: the openrdk viewpoint. *J. Softw. Eng. Robot.* 3, 13–27.

Collett, T. H., MacDonald, B. A., and Gerkey, B. (2005). "Player 2.0: toward a practical robot programming framework," in *Proceedings of the Australian Conference on Robotics and Automation*, Sydney.

Dantam, N., Lofaro, D., Hereid, A., Oh, P., Ames, A., and Stilman, M. (2015). The ach library: a new framework for real-time communication. *IEEE Robot. Autom. Mag.* 22, 76–85. doi:10.1109/MRA.2014.2356937

Einhorn, E., Stricker, R., Gross, H., Langner, T., and Martin, C. (2012). "MIRA – Middleware for Robotic Applications," in *IEEE/RSJ International Conference on Intelligent Robots and Systems* (Vilamoura: IEEE), 2591–2598.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.* 35, 114–131. doi:10.1145/857076.857078

Fitzpatrick, P., Ceseracciu, E., Domenichelli, D., Paikan, A., Metta, G., and Natale, L. (2014). A middle way for robotics middleware. *J. Softw. Eng. Robot.* 5, 42–49.

Fitzpatrick, P., Metta, G., and Natale, L. (2008). Towards long-lived robot genes. *Rob. Auton. Syst.* 56, 29–45. doi:10.1016/j.robot.2007.09.014

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Habra, T., Dallali, H., Cardellino, A., Natale, L., Tsagarakis, N., Fisette, P., et al. (2015). "Robotran-YARP interface: a framework for real-time controller development based on multibody dynamics simulation," in *ECCOMAS Thematic Conference on Multibody Dynamics* (Barcelona).

Hammer, T., and Bäuml, B. (2014). The communication layer of the ardx software framework: highly performant and realtime deterministic. *J. Intell. Robot. Syst.* 77, 171–185. doi:10.1007/s10846-014-0095-9

Hoffman, E. M., Traversaro, S., Rocchi, A., Ferrati, M., Settimi, A., Romano, F., et al. (2014). "YARP based plugins for gazebo simulator," in *Modelling and Simulation for Autonomous Systems Workshop (MESAS)*, ed. J. Hodicky (Roma: Springer), 333–346.

Huang, A. S., Olson, E., and David, M. (2010). "LCM: Lightweight Communications and Marshalling," in *IEEE/RSJ International Conference on Intelligent Robots and Systems* (Taipei: IEEE), 4057–4062.

Ierusalimschy, R., De Figueiredo, L. H., and Celes Filho, W. (1996). Lua an extensible extension language. *Softw. Pract. Exp.* 26, 635–652.

Kaufmann, M., Moore, J. S., and Manolios, P. (2000). *Computer-Aided Reasoning: An Approach*. Norwell, MA: Kluwer Academic Publishers.

Khalili, A., Natale, L., and Tacchella, A. (2014). *Reverse Engineering of Middleware for Verification of Robot Control Architectures, Volume 8810 of Lecture Notes in Computer Science, Book Section 27* (Cham: Springer International Publishing), 315–326.

Klotzbücher, M., and Bruyninckx, H. (2012). Coordinating robotic tasks and systems with rFSM statecharts. *Int. J. Softw. Eng.* 1, 28–56.

Lütkebohle, I., Philippsen, R., Pradeep, V., Marder-Eppstein, E., and Wachsmuth, S. (2011). Generic middleware support for coordinating robot software components: the task-state-pattern. *J. Softw. Eng. Robot.* 2, 20–39.

Maiolino, P., Maggiali, M., Cannata, G., Metta, G., and Natale, L. (2013). A flexible and robust large scale capacitive tactile system for robots. *IEEE Sens. J.* 13, 3910–3917. doi:10.1109/JSEN.2013.2258149

Mastrogiovanni, F., Paikan, A., and Sgorbissa, A. (2013). Semantic-aware real-time scheduling in robotics. *IEEE Trans. Robot.* 29, 118–135. doi:10.1109/TRO.2012.2222273

Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Boston, MA: Addison-Wesley.

Metta, G., Natale, L., Nori, F., Sandini, G., Vernon, D., Fadiga, L., et al. (2010). The iCub humanoid robot: an open-systems platform for research in cognitive development. *Neural Netw.* 23, 1125–1134. doi:10.1016/j.neunet.2010.08.010

Nori, F., Traversaro, S., Eljaik, J., Romano, F., Del Prete, A., and Pucci, D. (2015). iCub whole-body control through force regulation on rigid noncoplanar contacts. *Front. Robot. AI* 2:6. doi:10.3389/frobt.2015.00006

Paikan, A., Domenichelli, D., and Natale, L. (2015a). "Communication channel prioritization in a publish-subscribe architecture," in *Software Engineering and Architectures for Realtime Interactive Systems*, Arles.

Paikan, A., Pattacini, U., Domenichelli, D., Randazzo, M., Metta, G., and Natale, L. (2015b). "A best-effort approach for run-time channel prioritization in real-time robotic application," in *International Conference on Intelligent Robots and Systems (IROS)* (Hamburg: IEEE), 1799–1805.

Paikan, A., Schiebener, D., Wacther, M., Asfour, T., Metta, G., and Natale, L. (2015c). "Transferring object grasping knowledge and skill across different robotic platforms," in *IEEE International Conference on Advanced Robotics* (Instanbul: IEEE), 498–503.

Paikan, A., Traversaro, S., Nori, F., and Natale, L. (2015d). "A generic testing framework for test driven development of robotic systems," in *Modelling and*

Simulation for Autonomous Systems: Second International Workshop, MESAS 2015, Prague, Czech Republic, April 29-30, 2015, Revised Selected Papers (Cham: Springer International Publishing), 216–225.

Paikan, A., Fitzpatrick, P., Metta, G., and Natale, L. (2014a). Data flow port's monitoring and arbitration. *J. Softw. Eng. Robot* (Chicago: IEEE), 5, 80–88.

Paikan, A., Tikhanoff, V., Metta, G., and Natale, L. (2014b). "Enhancing software module reusability using port plug-ins: an experiment with the iCub robot," in *International Conference on Intelligent Robots and Systems (IROS)* (Montevideo: IEEE), 1555–1562.

Paikan, A., Metta, G., and Natale, L. (2013). "A port-arbitrated mechanism for behavior selection in humanoid robotics," in *16th International Conference on Advanced Robotics (ICAR), 2013*, Montevideo, 1–7.

Parmiggiani, A., Maggiali, M., Natale, L., Nori, F., Schmitz, A., Tsagarakis, N., et al. (2012). The design of the iCub humanoid robot. *Int. J. HR* 9, 1250027. doi:10.1142/S0219843612500272

Pattacini, U., Nori, F., Natale, L., Metta, G., and Sandini, G. (2010). "An experimental evaluation of a novel minimum-jerk cartesian controller for humanoid robots," in *IEEE/RSJ International Conference on Intelligent Robots and Systems* (Taipei: IEEE) 1668–1674.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., et al. (2009). *ROS: An Open-Source Robot Operating System*.

Randazzo, M. (2004). *iCub Control Modes Specifications*. Technical Report. Genova: iCub Facility, Istituto Italiano di Tecnologia.

Sametinger, J. (1997). *Software Engineering with Reusable Components*. Berlin: Springer.

Schlegel, C., and Worz, R. (1999). "The software framework smartsoft for implementing sensorimotor systems," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vol. 3 (Kyongju: IEEE), 1610–1616.

Stewart, D., Volpe, R., and Khosla, P. (1997). Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. Softw. Eng.* 23, 759–776. doi:10.1109/32.637390

Tikhanoff, V., Cangelosi, A., Fitzpatrick, P., Metta, G., Natale, L., and Nori, F. (2008). "An open-source simulator for cognitive robotics research: the prototype of the iCub humanoid robot simulator," in *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems* (New York, NY: ACM), 57–61.

Tsagarakis, N. G., Morfey, S., Cerda, G. M., Zhibin, L., and Caldwell, D. G. (2013). "Compliant humanoid coman: optimal joint stiffness tuning for modal frequency control," in *IEEE International Conference on Robotics and Automation* (Karlsruhe: IEEE), 673–678.

Vernon, D., Billing, E., Hemeren, P., Thill, S., and Ziemke, T. (2015). An architecture-oriented approach to system integration in collaborative robotics research projects – an experience report. *J. Softw. Eng. Robot.* 6, 15–32.

Zenger, M. (2004). *Programming Language Abstractions for Extensible Software Components*. Ph.D. thesis, École Polytechnique Fédérale de Lausanne, Lausanne.

Conflict of Interest Statement: The work described in this paper was conducted in the absence of any commercial or financial relationship that can be construed as a potential conflict of interest.

frontiers in Robotics and AI

TECHNOLOGY REPORT
published: 25 April 2016
doi: 10.3389/frobt.2016.00020

NUClear: A Loosely Coupled Software Architecture for Humanoid Robot Systems

Trent Houliston, Jake Fountain, Yuqing Lin, Alexandre Mendes, Mitchell Metcalfe, Josiah Walker and Stephan K. Chalup*

Newcastle Robotics Laboratory, School of Electrical Engineering and Computer Science, The University of Newcastle, Callaghan, NSW, Australia

This paper discusses the design and interface of NUClear, a new hybrid message-passing architecture for embodied humanoid robotics. NUClear is modular, has low latency, and promotes functional and expandable software design. It greatly reduces the latency for messages passed between modules as the message routes are established at compile time. It also reduces the number of functions that must be written using a system called co-messages, which aids in dealing with multiple simultaneous data. NUClear has primarily been evaluated on a humanoid robotic soccer platform and on a robotic boat platform. Evaluations show that NUClear requires fewer callbacks and cache variables over existing message-passing architectures. NUClear does have limitations when applying these techniques on multi-processed systems. It performs best in lower power systems where computational resources are limited. This article aims at readers with interest in modern software engineering concepts and development of systems in areas such as robotics, smart devices and virtual reality.

Keywords: humanoid robot, robot vision, robot learning, software architecture, message passing, blackboard, co-messages, compile time message routing

OPEN ACCESS

Edited by:
Lorenzo Natale,
Istituto Italiano di Tecnologia, Italy

Reviewed by:
Neil Thomas Dantam,
Rice University, USA
Torbjorn Semb Dahl,
Plymouth University, UK

****Correspondence:***
Trent Houliston
trent.houliston@newcastle.edu.au

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *01 November 2015*
Accepted: *29 March 2016*
Published: *25 April 2016*

Citation:
Houliston T, Fountain J, Lin Y, Mendes A, Metcalfe M, Walker J and Chalup SK (2016) NUClear: A Loosely Coupled Software Architecture for Humanoid Robot Systems. Front. Robot. AI 3:20. doi: 10.3389/frobt.2016.00020

1. INTRODUCTION

A system's software architecture is the arrangement of its high level structures based on the layout of its functional code elements. As the software components of modern humanoid robotic systems become more capable, their code becomes larger and more complex. This can increase the cost of maintaining and enhancing the system. Software architecture design improves high level structures encouraging better maintainability and re-usability of components, and supports the reduction in effort in understanding and modifying software systems. However, this is difficult in embodied humanoid robots where computational hardware is limited in power, as architectural decisions to improve the maintainability of the system impact the performance of the robot. Therefore, architectures for this domain should aim to improve the efficiency for systems with limited performance.

Latency is also a key concern in robotic systems. Information must flow quickly between components for real-time control. Latency between sensing and acting reduces the robot's ability to correct issues in time. Architectural decisions that increase modularity also increase the latency between components, as their interfaces become more general. This impacts on the robot's ability to process and function as required.

The NUClear framework has been designed to address these concerns using novel techniques to improve the communication between modules. It utilizes C++ template meta-programing and smarter interfaces to reduce or eliminate costs while maintaining an expressive and easy to use interface.

2. ROBOTIC SOFTWARE ARCHITECTURES

When designing or improving software architecture for autonomous robots, it is important to analyze the techniques of existing systems. The software architectures used in the majority of autonomous humanoid robots are within a spectrum between two primary architectural paradigms. These systems can be defined by the way in which data is communicated between modules in the system. On one end of the spectrum are global data store systems, in which all data are stored centrally and accessed by expert system modules. At the other end of the spectrum are message-passing systems, where systems receive messages and perform actions on the data received. Systems also exist within this spectrum that have a degree of hybridization (**Figure 1**). These hybrids utilize message passing and also include features from the global store. This section of the paper explores various properties that contribute to the architecture quality of existing systems. Also discussed are techniques available to improve the performance of the system.

2.1. Blackboard

A blackboard architecture is a form of global store architecture (**Figure 2**). Modules within a blackboard system communicate with each other through the manipulation of data elements stored on a central data store called the blackboard. This manipulation is achieved through expert systems (Hayes-Roth, 1985) that are responsible for performing particular tasks. For example, an expert system responsible for vision may read images from the blackboard. When it has analyzed them, it writes the observed visual features to the blackboard. This design ensures that the other components are able to gather the information required by accessing it from the global data store without having to communicate directly with each other.

Blackboard architectures were originally developed from the blackboard concept that was used as a theoretical tool in the field of AI research. They were developed into a software architecture and effectively utilized in the HEARSAY-II system (Erman et al., 1980) to implement a speech processing artificial intelligence. Blackboard systems were then enhanced by Hayes-Roth (1985) to provide not just a communal data store but to also provide the control elements required for developing a robotic system. This enhancement formed the foundation of the blackboard architecture used in autonomous robotics.

Blackboard architectures are also frequently used in development projects where limited computation resources are a concern. The RoboCup contest, an annual contest involving teams from research institutions around the world, provides an excellent case study to compare robotic systems that are required to run using limited on-board computational resources (Kitano et al., 1997). Each team competing in RoboCup is required to construct and program robots to perform in a modified soccer contest with the goal of defeating the FIFA world champions by the year 2050. As all of the robots are performing the same task, the differences in programing and construction provide an excellent test bed to compare design choices.

Within RoboCup, there are numerous teams who have independently implemented software architectures. These robotic systems are all designed to achieve the same task. Therefore, the primary differences in the systems are either algorithmic or architectural. Three of the more successful teams using blackboard architectures have released their codebase, allowing a comparison of their implementations. Teams from B-Human (Röfer et al., 2011), UT-Austin (Barrett et al., 2013), and the University of Newcastle's NUbots (Kulk and Welsh, 2012) all have achieved success in the contest and have released code and technical reports on their architectures. Each of these teams use a slightly different implementation of the blackboard system.

Barrett et al. (2013) base their system around a pure blackboard architecture. This system uses a single storage location labeled as "memory" where all information generated by the expert systems is stored. This flat implementation is the original design of a blackboard system with a single, large data store.

Kulk and Welsh (2012) also use a single, centralized blackboard for communication of data between modules. However, several of the sub components have their own private blackboards that

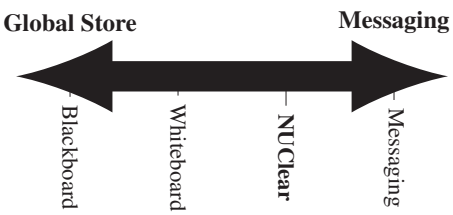


FIGURE 1 | The spectrum of robot software architectures. Most of the architectures used in robotics are some combination of global store and messaging.

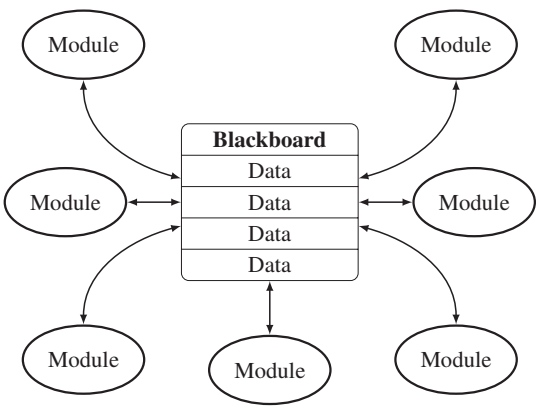


FIGURE 2 | In a blackboard architecture, all modules communicate by reading from and writing to a single global data store. Due to the central data store, all components are coupled through common coupling.

are used for internal communication and memory. This results in a hierarchy of blackboards where each successive blackboard becomes more specialized to an individual task.

The architecture of Röfer et al. (2011) involves numerous individual blackboard elements that are held within separate processes for a particular task. Each of the individual components may have overlapping access to a blackboard. However, each component will only access the blackboards that are relevant to their data requirements. This results in a system where individual blackboards are able to be modified without significantly impacting modules that do not use them.

The B-Human, UT-Austin, and NUbots teams have used blackboard architectures for many years due to the ease of implementation and low computational overhead. A blackboard system has a single location for all data and is easily understood. New states are easily added to a blackboard, and the addition of new data does not require modification to the other systems. This is valuable when performing tests for research purposes as well as being able to add debugging data to the system temporarily.

The system developed by Team KAIST for the 2015 DARPA Robotics challenge (Lim et al., 2015) utilizes a global store system as its primary method of communication. The code executes on multiple processes spread across seven machines in different physical locations. It uses a shared memory system called MPC to communicate data among its processes. The system also uses a message-passing architecture for its vision system based on ZeroMQ. Its control system uses message passing based on POSIX IPC and shared memory. This heterogeneous system has the potential to cause confusion in future development, as it is unclear which architectural style must be used to communicate with a specific component.

In relation to the level of coupling in a system, blackboard architectures perform poorly. In the best case, they exhibit common coupling. All the elements depend on a single data store [Page-Jones (1988), p. 73]. At worst, there can be pathological coupling. This is where modules interact and operate by modifying each others' internal state [Page-Jones (1988), p. 77]. These forms of coupling make it difficult to perform modifications to the codebase due to the flow on effects of modifying the blackboard [Page-Jones (1988), p. 80]. Additionally, despite providing a successful platform for playing soccer, these systems are unable to easily adapt to other roles due to the necessary specialization of the blackboard itself.

Blackboards also present challenges in multithreading situations as individual components are not aware when new and relevant data becomes available. This is of concern in modern multi-core embedded platforms, as modules may miss data updates, read the same data twice, or even read a partially written data element. This makes it difficult to use time-series data effectively, as it requires modifications to both the provider of the data and the blackboard to ensure elements are not missed. Finally, as use cases change and data formats are updated, it becomes necessary to make modifications to all the modules accessing the updated items in the blackboard. This leads to a situation where it is easier to add new data types to the blackboard, rather than refactor the existing data types to improve flexibility in the system.

2.2. Messaging

Message-passing systems are a generalization of a pipeline system where the output from one system is used as the input into the next system (**Figure 3**). Each module creates information and then publishes this information to the rest of the system. Other modules subscribe to these information updates and on receipt of the information, perform their own operations using it.

There have been several successful message-passing architectures developed for robotic systems. Most of these are designed for robots with significant computational resources. However, there are more recent frameworks designed that have considered performance and latency.

OROCOS (Bruyninckx, 2001) is an early open source robotics framework based on a CORBA object broker system. This system allows it to work across multiple languages and processes. It provides a consistent interface for robotics use and is designed to keep code modular for code reuse among systems. However, compared to modern methods, OROCOS is slow (Hammer and Bäuml, 2014) and does not support systems that are not based on object-oriented design patterns.

Dynamic Data eXchange (DDX) (Corke et al., 2004) is a message-passing robotic software architecture developed at the Commonwealth Scientific and Industrial Research Organization (CSIRO). DDX was designed to improve on the slow speed of previous message-based robotic software architectures. These previous systems required messages to pass through a significant amount of the network stack, limiting performance. DDX uses interprocess communication (IPC) to provide a publish/subscribe architecture. As a message-passing system, it suffers from the requirements of serialization of data. Serialization reduces performance significantly as large amounts of data

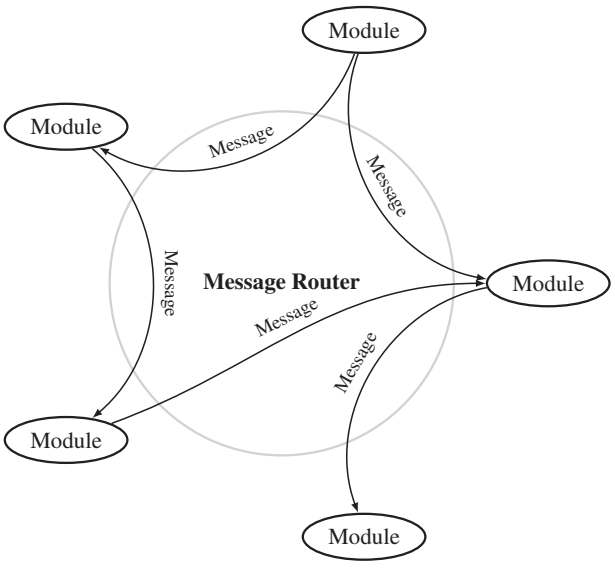


FIGURE 3 | Message-passing systems treat each module as a producer/consumer. Data are sent through a message routing system to subscribers to that type. This message router may be a single entity, or the task may be distributed among individual modules.

must be processed and copied. To resolve this, DDX provides direct shared memory access between modules. However, if used incorrectly, this functionality transforms DDX into a blackboard system and takes on its associated architectural issues. Problems within DDX are also compounded due to the lack of type safety in a system that communicates interprocess. This makes it difficult to ensure that the data received are the anticipated format.

YARP (Metta et al., 2006) is a recent message-passing system for robotics based on the observer pattern. It provides observable entities called ports that can be provided over network and local protocols. YARP utilizes C++ templates to enable customizable serialization of types for transfer. It is able to use shared memory models to reduce the number of data transfers that occur and uses networked methods to distribute information to other systems. YARP modules listen to the observable data either by waiting on a port or by scheduling a callback function to execute when data come in. YARP is an excellent example of a flexible message-passing system.

A common research-oriented robotic software architecture is the Robot Operating System (ROS) (Quigley et al., 2009). It has a centralized node named the ROS Master that is responsible for establishing message routes within the system. Messages do not pass through the ROS Master. Instead, the ROS Master enables nodes to form direct connections. These routes are established using IP networking protocols (TCP or UDP).

Each individual component of the robotic system runs in its own environment without direct knowledge of other components. These components are expert systems responsible for individual tasks, such as localization or visual processing. Communication is handled through predefined messages that are able to be serialized into a transferrable byte representation. Messages that only travel within a single node or systems built to run as a single node, such as ROS nodelets, are able to transfer without copying or serialization.

ROS was designed to provide a platform capable of running code in multiple programing languages on a number of distinct computers to control a robotic system. The flexibility of its programing language and the large number of researchers using this system has made it standard for modern robotics research. This has seen it implemented on numerous robotic platforms ranging from robotic vacuum cleaners to large humanoid robots. The use of ROS has also gained interest in industrial robotics through the ROS-Industrial project (Edwards and Lewis, 2012).

Ach (Dantam et al., 2015) is a messaging system inspired by POSIX message queues. POSIX message queues are suboptimal for robotic systems, as queues create a preference for older data. It caters to multi-processed systems that run on a POSIX operating system and is implemented using a circular array of shared memory that can be accessed by any process that is a part of the system. The shared memory for a particular message is described as a channel. Messages are distributed by adding them to the circular buffer for a channel. Subscribers request new data from the channels they need and then wait for it to become available. Each subscriber has an individual pointer that tracks which messages in the queue have been read. The circular buffer will wrap around and overwrite the old data, potentially causing issues for a subscriber if every data element must be processed. Ach has the advantage as it operates at a low latency due to its use of shared memory. A limitation of Ach is that its use of shared memory restricts its use to a single machine. Additionally, the pull style interface adds complexity for the developer when managing multiple simultaneous requests.

Message-passing systems solve many of the issues encountered with a blackboard system. In relation to the tight coupling of a blackboard, there is no longer any single object that is required to know the entire state of the system. This makes it easier to adapt existing modules for use in a new system. Listening to changes in data allows the system to catch every event as it is modified, removing the need for polling a data store. The ability to distribute the program across multiple threads of execution or even multiple systems makes message-passing excellent for systems with ample computational power. Time-series operations are also possible as every event is accessible without requiring modification to other modules.

Message-passing modules are also able to act as a service. A common pattern is to send a request message and another module will reply with a response message.

Message-passing interfaces will either have a pull interface, where a function must be called to wait for new messages, or a push interface, where a function is provided and executed when data becomes available. In a pull interface, multiple message reads must be multiplexed or have multiple threads listening to receive data. This adds extra complexity to the interface, as there is often increased work required to wait on multiple messages. Push interfaces are simpler to develop, as there is no additional complexity to wait on multiple messages simultaneously. Additionally, push interfaces make multithreading easier as each callback function can be executed on separate thread.

However, there are several disadvantages that exist in these systems that are not present in a blackboard system. A message-passing system must either provide a copy of the data for each subscriber of a message or make all access read only. This results in a performance penalty in the system. Messaging also means that there is no longer a central data store that can be used. Therefore, if a module requires information from more than one message, it must handle the storage of this data itself to access it. This adds significant extra load on the modules, which makes development harder and reduces its performance.

2.3. Whiteboard

Thórisson et al. (2005b) have developed an enhanced version of blackboards in order to utilize some advantages of a message-passing system. Whiteboard architectures have publish/subscribe extensions added to the blackboard in addition to the statically stored data (**Figure 4**). This hybrid system of global storage and message-passing solves several concerns associated with a traditional blackboard system. It is able to use the publish/subscribe features to remove the need for polling in the system, allowing the system to react when data has changed. It also allows the system to perform computations only on new data, rather than repeatedly calling the old data.

The whiteboard design has been used in robotics development. Reykjavik University (Thórisson et al., 2005a) integrates

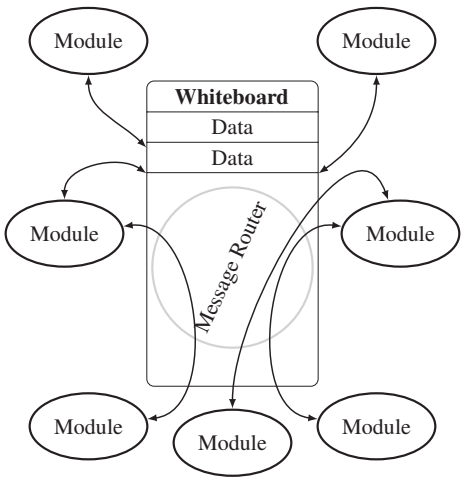


FIGURE 4 | A whiteboard is a blackboard that is able to have messages passed through it to other modules. It gains some of the advantages of message passing but maintains the tight coupling of a blackboard.

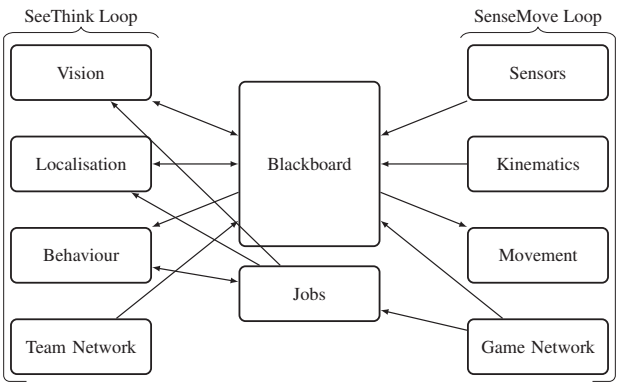


FIGURE 5 | The information flow from the NUbots' software architecture of 2012.

whiteboard architecture in robotic software, and the Mi-Pal team from Griffith University utilizes a whiteboard architecture to solve issues associated with blackboard architectures (Coleman et al., 2013). Mi-Pal is able to use the publish/subscribe architectural feature to ensure that a module will receive every data update posted. They have recently improved this architecture to provide compile time checking of data types, easing the debugging of the system. Systems that hybridize message passing and global store architectures are a midpoint between the fields of AI that use blackboards extensively and modern software architecture that focus on decoupled design.

Despite these advantages, whiteboard architecture experiences several drawbacks. In a whiteboard system, the weakness of requiring a single blackboard at the center results in common coupling. Even when a stream is used, the stream is stored on the blackboard. This means that modifications to a stream alter the components that use them. The other major issue that arises from using a whiteboard architecture is the use of an implementation that is not a true hybrid of global storage and message passing, but instead two architectures deployed in parallel. In this case, the user either takes on the benefits and weaknesses of a message-passing system or those of a blackboard system.

The performance of the robotic platforms these architectures are deployed on provides insight into why they are utilized. Robots that execute on hardware with tighter performance constraints, such as those with on-board computation, are designed using the blackboard model and occasionally using the whiteboard model. In robotic systems where performance is not restricted, message-passing systems are used to allow greater collaboration between developers. Although message-passing systems provide excellent maintainability, testability, and modifiability, the performance of these systems in relation to latency and overhead is a significant concern when there is limited processing power.

2.4. 2012 NUbots' Architecture

The 2012 NUbots' software architecture was designed as an easy to understand and extend object-oriented system (Kulk and Welsh, 2012). However, over time, design-limited module communication and workarounds limited the overall value of the system. This resulted in the system fragmenting and impacted on its quality as the use cases for the system diversified. It resulted in a system that used different assumptions and design patterns. The architecture also caused duplication of effort and the use of similar, redundant classes in various modules due to the lack of a clear architecture. A simplified diagram of the architecture modules and flow of the existing system can be seen in **Figure 5**.

Common with many embodied robot architectures, the 2012 NUbots' architecture revolved around a central blackboard data store. Several modules utilized a global queue known as the jobs system, with most communicating through direct function calls, implicit or global state such as singletons. This diversity in communication resulted in a technical debt and reduced productivity as the system grew. As in order to make modifications to a component, an understanding of the interface, including anything it interacted with or any of the interface's new components, was required. This was problematic as the NUbots' architecture had components that needed to communicate to three or more other components. This required an understanding of a minimum of four different systems before a developer could make any change.

The vision system provides a good example of this unnecessary complexity. The vision system communicated utilizing a two step process. It accessed the sensor system directly and asked it to process a new frame. It then waited on the sensor system to place the frame information on blackboard. Once the frame information was placed on blackboard, the vision system read the information from blackboard and continued its processing. Although the vision system waited for a new frame, the entire robot was blocked and could not make any decisions. It was also important to ensure that no other systems intended to request the latest frame, as doing so would break the robots functionality.

Another unresolved architectural challenge was experienced in the Movement module. The 2012 NUbots' system defined

a number of movement handlers, each responsible for a set of movements. This could include kicking a ball or walking. The movement system periodically retrieved all of the jobs in the queue and sent them to the appropriate movement handler. The movement handlers then communicated to the action system to execute the selected motions. This base case did not have any issues. However, it was not possible for any other component to talk to the movement handlers directly *via* the blackboard system. Therefore, at any point in time, any class in the system was a potential candidate for triggering a movement. This made it very easy to create hard to maintain access paths and added to the complexity of the system.

Additionally, each movement handler is indirectly dependent on each other. Movement handlers can lock specific motors and if they attempt to use a motor in use by another system, the action failed. Forgetting to check the ownership of a motor broke the currently executing motion, causing the robot to fall down and possibly injure itself. In order to add a new movement, the code was written for the movement, and the developer was required to understand how all the existing motion modules worked and interacted. This included cases where any of the managers might be triggered, in order to avoid interrupting a critical movement when the motors were locked. Additionally, the developer needed to understand the locking model to prevent accidentally trying to move a motor that should not be moved. These are only two examples of a number of pitfalls within the 2012 NUbots' architecture.

2.5. Comparisons

Previous studies have compared the techniques used in traditional blackboard systems and messaging systems (Orebäck and Christensen, 2003; Magyar et al., 2015; Matamoros et al., 2015). This research concludes that blackboard architectures are easier to maintain and provide greater performance than peer-to-peer systems. However, a significant body of research (Page-Jones, 1988; Pressman, 2005; Beck and Diehl, 2011) has found that loosely coupled messaging systems should provide much greater maintainability and extensibility than globally coupled blackboard systems.

A possible explanation for this discrepancy is that a blackboard system is much easier to comprehend than a message-passing system. This makes it easier for a small blackboard system to be worked on by a developer. However, as the system grows and contains more components, increased complexity should reveal message-passing systems as having the advantage.

3. THE NUClear FRAMEWORK

The NUClear framework was designed to utilize the advantages and address the problems that exist with the architectural styles of message passing and blackboards. The primary advantages of the two competing architectures are the high data availability in a blackboard system and the excellent decoupling properties of a message-passing system. A message-passing system must be used as the primary architectural paradigm to achieve loose coupling. The challenge is to incorporate the advantages that a global store provides into the message-passing system. An ideal system should maintain the loose coupling that message-passing affords, without suffering from data management issues. In order to address these issues and provide a more effective software architecture, a solution called co-messages has been implemented. It more effectively hybridizes the two paradigms in NUClear.

NUClear introduces a modification to how messages are dispatched to the subscribers of the data. In robotic systems, modules will typically have a primary information type that they will perform their operations on. However, they often require additional supplementary data provided from other sources. These additional data must be available when the operations are performed, and they may be created at a different rate to that of the primary data source. Accessing these data at any time is impossible in a pure message-passing system, as the data are only available transiently. If a system requires information created by another module, it must subscribe and then store this information itself to ensure it is available. In NUClear, one of the subscribed types will be designated the primary message type and messages will only be delivered when this message is created by a publisher. The most recent messages of the remaining types are then bound to the subscription, allowing access to the messages without requiring that the modules store the information (**Figure 6**). These additional messages are co-messages. By controlling when messages are received based on the arrival of other messages, the modules in the system are no longer required to manage their own cache of variables.

This method of accessing data is distinct when compared to multiplexing multiple message channels into a single input. In comparison to using select or poll on multiple channels, co-messages do not introduce a cost for messages that are not used. This is often the case when the rate of the messages do not match. When one message occurs more frequently than another, a system using select or poll system must inspect and discard each of these updates when they are not relevant. Instead, in NUClear, those messages are fetched on demand when the primary data

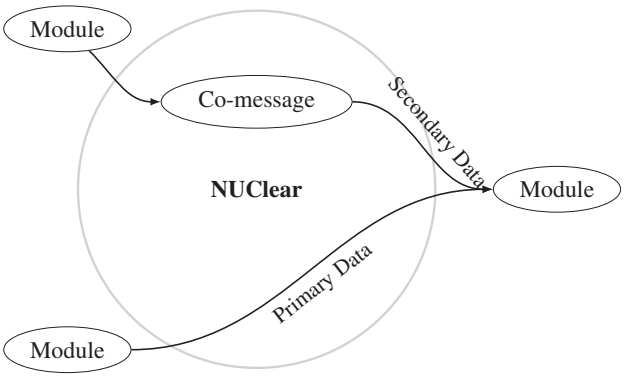


FIGURE 6 | The NUClear co-messaging system. The latest version of each message is stored in NUClear. A module requests a primary data type, along with the most recent version of a secondary data type. When these primary data are generated, the stored most recent copy of this co-message is bound into the callback. This affords the module a more expressive interface for gathering messages.

type is received, reducing the overhead both computationally and for the developer.

Additionally, as the latest version of each message must be stored in order to be bound when needed, a virtual global store is created from these messages. These globally stored data are not an explicit component, but a byproduct of the architecture. It does not increase the coupling between modules. Therefore, the developer is not required to write code for data elements in this global store. This resulting architecture has the advantages of loose coupling from the message-based system, while gaining the advantages of high data availability provided by a centralized store.

The virtual data store is also able to be explicitly accessed, providing direct access that is similar to a blackboard. As the messages are persistent, they are able to be accessed without setting up a listener for the message. This form of access is discouraged, as it increases the coupling in the system. However, this provides an advantage to NUClear in its compatibility with code written for other architectures. NUClear is able to accommodate the access patterns of any of the three other architectures with only minimal modification to the components themselves.

A key requirement of a multi-platform robotic system is the replacement of components with functional equivalents. For example, it should be possible to replace the system that reads camera frames from the hardware with one that reads frames from a file without changing the rest of the system. However, another important aspect of robotics is the requirement for real-time interaction with the world. This requires optimized code paths. As identified in the background research, the architectures excel in one of these categories, but often trade speed for decoupling.

The majority of mature robotic systems are developed with a message-driven architecture. ROS has a serialization penalty associated with message passing, as communication between modules is achieved using sockets. This is unacceptable for low-power embedded platforms, such as the Darwin-OP. Other robot systems, such as Dynamic Data eXchange and Pack Service Robotic Architecture, also experience similar issues. These contemporary robotic systems have traded performance for message passing and cross-language compatibility, anticipating that future hardware performance will accommodate them.

3.1. Simple API

NUClear is designed to have a simple and intuitive interface that requires minimal training to use. It is designed for use by second year software engineering students who have a basic understanding of C++. There must be a well-defined function for the two key features of the architecture: sending and receiving messages. Sending is accomplished using the *emit* function, whereas receiving is accomplished using the *on* function. NUClear provides a small domain specific language (DSL) for easy access to common message pattern use cases. This DSL is designed to match an English description of the task if possible. Having a small API improves understandability and reduces the learning curve for using the NUClear system. This makes it easier for new programmers to learn and use NUClear.

In addition to the DSL words that are already in the system, NUClear is designed to be able to extend its vocabulary with new words developed by the user of the framework. This allows common functionality to be provided across modules specific to the needs of a system. Internally, all NUClear DSL words are implemented using the same extension system.

3.1.1. Domain-Specific Language

The following is a list of the DSL words that are included and most commonly used in NUClear along with a description and an example of their use.

on `on<...>(runtime...).then(function);`
The on DSL word is the wrapper for every subscription in NUClear. NUClear uses this to wrap the template descriptions of the subscription's purpose. The other DSL words are entered as template arguments to this function, with any runtime arguments passed as function arguments.

emit `emit(message)`
Emit is the function that handles the publish part of the messaging system. It takes data and forwards it to the functions that have subscribed directly (routed at compile time).

Trigger `on<Trigger<Type>>()`
Trigger statements set up the callbacks and execute when the type is emitted. It flags the used data type as a triggering (primary) data type. When this callback is executed, it will pass the data that were emitted.

With `on<Trigger<TypeA>, With<TypeB>>()`
With statements describe additional information that is used by the callback. The provided function will not be executed when these data are emitted. However, when this function is executed, the latest copy of this data will be provided.

Every `on<Every<10, milliseconds>>()`
Every statement fills the role of periodic callbacks in the system. When an every statement is used, the function will execute at that rate.

Always `on<Always>().then(function);`
Always is used in the rare case that functions must continually execute as fast as possible. It allows the system to terminate as a whole when it shuts down by ending the execution of this function.

Single `on<Trigger<Type>, Single>()`
Single is a DSL keyword that ensures that only a single instance of a message will be executed at one time. When additional messages of the same type are given to this function, they will be dropped.

Buffer `on<Trigger<Type>, Buffer<3>>()`
Buffer is the general case of the Single keyword. It ensures that only the requested number of messages will execute simultaneously. When additional messages beyond the requested number are given to this function, they will be dropped.

Sync `on<Trigger<Type>, Sync<Group>>()`
Sync is used to ensure mutual exclusion between several functions at a scheduling level. Rather than blocking a thread on a mutex, it will delay execution until it has exclusive access among a group of functions. Additional messages of the same type are queued for execution unless combined with single.

Priority `on<Trigger<Type>, Priority::HIGH>()`
Priority can control the response of the callback. It will control both the priority used to determine the scheduling order in the thread pool and also the priority of the thread it will execute on.

Startup `on<Startup>()`
Functions with this word will execute at startup.

Shutdown `on<Shutdown>()`
Functions with this word will execute at shutdown.

Configuration `on<Configuration>("File.yaml")`
This allows a program to watch a file in a configuration directory and be provided with the latest version of the file when it changes. It is used to keep configuration up to date.

Optional `on<Trigger<TypeA>, Optional<With<TypeB>>>()`
Optional allows statements to signify certain requested types as optional. In a traditional co-message call, if both messages are not available, the function will not run. If optional is used, these functions can run with an empty second argument.

Last `on<Last<10, Trigger<Type>>>()`
Last instructs NUClear to cache the last messages that were emitted. When the function is called, it will receive all of the collected messages.

IO `on<IO>(file_descriptor)`
Used to interact with file descriptors and execute when they are read/writeable. This is used for communicating with serial devices as well as network ports.

Network `on<Network<Type>>()`
NUClear provides a networking protocol to send messages to other devices on the network. This can be used to make a multi-processed NUClear instance, or communicate with other programs running NUClear. The serialization and deserialization is handled by NUClear.

TCP `on<TCP>(port)`
TCP allows a program to make a callback on TCP activity, listening on a port.

UDP `on<UDP>(port)`
UDP allows a program to make a callback on UDP activity, listening on a port. It also supports listening to UDP broadcast and UDP multicast sockets.

3.2. Low Performance Penalty

In the context of embedded or low performance hardware, it is essential that message routing is performed as quickly as possible. Other message-passing systems use technologies that incur performance penalties from message serialization and copying. Instead, NUClear uses shared memory for messages passed within a single process. This removes the cost of serializing a message and can greatly improve the performance of large messages.

It is also important to minimize the time it takes to dispatch a message. To achieve this, NUClear uses template meta-programing to establish message routes at compile time. Generally, when a message is sent in a message-passing system, there is a message broker that executes code to find subscribers who are interested in the message. Alternatively, in simpler systems, there is a message bus that all subscribers listen to and gather messages they are interested in. NUClear eliminates this cost by evaluating the route messages take at compile time. The result is that when messages are dispatched from a module they are directly sent to the modules that require them. This reduces the cost of routing the message to acquiring the required data and directly calling the subscribers callback function. The downside to this technique is that it is only applicable when running within a single process.

NUClear also has an additional latency improvement for modules arranged in a pipeline structure. This improvement is used when a module emits a message at the end of its callback, and that message is only consumed by only one other module. In this case, NUClear is able to continue and execute the following code directly rather than returning to the thread pool. This greatly reduces the latency between the two modules, as when combined with compile time routing, there is little that occurs between the executions.

Additionally, NUClear is able to perform compile time message memory allocation using template meta-programing. It uses the information to preallocate the space before the program runs. This allows it to scale to any number of messages while maintaining $\mathcal{O}(1)$ dispatch look-up time. The mechanism used to allocate messages also enables optimizing compilers to use the knowledge of message memory allocation to apply a number of powerful optimizations to how messages are sent.

3.3. Simple Utilization of System Resources

Another requirement derived from resource-constrained environments is the need to easily utilize the full power of the hardware. This is primarily achieved by introducing transparent multithreading that automatically uses every CPU core available on the system. Transparent multithreading in NUClear utilizes a thread pool that has enough threads to saturate every CPU core. Using this thread pool, NUClear is able to schedule each execution of a callback function to a different thread. This allows the system to utilize all of the cores without the developer needing to interact with threads directly.

When a message is sent in the system, the central coordination object, known as the PowerPlant, takes ownership of it. From this point forward, no modules can modify the message. It is provided to other modules with read-only access. The PowerPlant then executes callbacks that, known as reactions, are subscribed to this message. Each reaction receives an immutable reference to the original message and can perform any read operations on it.

The immutability of the data makes transparent multithreading in the system easier. If multiple reactions want to read the data, and if it can be guaranteed that they do not modify it, then the reactions can be run in parallel without concern for race conditions. By forcing immutability, all threading logic can be moved directly into NUClear, allowing developers to concentrate on their modules instead of threading problems. This technique of using immutable data to allow easy multithreading systems has been proven in programming languages, such as Erlang and Elixir.

In most cases, multithreading will be completely transparent. However, it is still important that developers understand that they are working in a multi-threaded environment. If a single module shares data between two reactions and those reactions run in parallel, then the shared data will need to be secured with a thread-safe mechanism such as a mutex. NUClear also provides functionality in its DSL to let the user specify that certain reactions should not be run in parallel. Specifically, it provides the word *Single* to ensure that only a single instance of a reaction is running and to drop any future messages. It also provides the word *Sync* to ensure that only one of a group of reactions is running and to queue the remainder. These can be used to solve threading concerns.

Differences exist between a multi-threaded system and a multi-processed system. In robotic systems, it is common to run in a multi-processed mode rather than multi-threaded. This allows the system to be distributed across multiple physical hardwares, making more efficient use of available resources. It can provide a level of crash safety. If a single process in the cluster crashes, it does not result in an entire system crash provided the remaining modules can run independently. Multi-threaded

systems also have this property if the threads the code executes on are managed correctly. This requires additional work by designer of the multi-threaded system.

Multi-processed systems also have disadvantages. Multi-processed systems do not always run in a shared memory environment, as they may be distributed across machines. They must serialize and copy data to the destination nodes. This adds an increase in latency between modules and when there is a large amount of data to be transferred, this can have significant implications.

Multi-processed systems also suffer from an increase in the context switching time. When the operating system in a multi-processed system switches from one process to another, it changes its virtual memory space. This requires the translation lookaside buffer to be dumped. Multi-threaded processes share their virtual memory space, allowing the buffer to be retained.

NUClear is able to run in a multi-threaded mode, a multi-processed mode, or a hybrid of the two by grouping modules into individual processes. However, when it runs in a multi-processed mode, it loses many of its advantages from compile time message routing. This is a necessary side effect, as the compiler can no longer optimize code paths and it must pass messages by serializing them over a socket.

3.4. Time-Series Data

A common need when interfacing with real-world electrical systems is to keep a record of recent data for validation and decision-making. The simplest example of this is de-bouncing the electrical noise in a button press. To perform this task, the on/off state is monitored over a time-series, with an internal stateful check deciding whether the switch is deemed to have closed or not. These types of tasks commonly access the last n instances of created data when performing an evaluation.

NUClear handles time-series data at an architectural level by increasing the number of previous messages that are stored latent in the system. This functionality is provided through the Last keyword.

A blackboard system is unable to receive a history of elements, as there is no notification of when data are updated. This may result in duplicate and missed data depending on the rate of polling. Rather, the functionality must be added by the producer of the data. They must store the additional data on the blackboard unaware of when it is not needed.

Message-passing systems are able to provide time-series data, as the arrival of messages allows the subscriber to maintain a history of the last few messages. This requires the subscriber of the messages to maintain their own data store of the most recent messages.

In a whiteboard system, it is possible to obtain time-series data using the same mechanism as message passing, or blackboard, depending on the stored data. However, they are also able to use the publish/subscribe channel in order to remain informed of changes to static data, allowing the data to be copied.

3.5. Soft Real-Time

Using the functionality provided by Every and Priority, NUClear can operate as a soft real-time system. One reason NUClear is successful at operating at soft real-time is its compile time dispatch of messages. When the binaries are compiled, the periodic functions are compiled with them. This allows the periodic functions to operate with the jitter and accuracy that the operating system providing the timing is capable of. The jitter in NUClear's periodic Every function was measured at 80 μs when triggering at a rate of 1 kHz.

Additionally, as NUClear routes messages at compile time, it can achieve lower end-to-end latency between modules. This is a reduction in time between dispatching and receiving a message. This lower latency can assist systems that have strict timing requirements such as hardware feedback loops.

A simple test system was constructed in NUClear and ROS that timed and transferred an empty message from end to end. This should eliminate any performance differences due to serialization and copying of information. The tests were completed with and without the CPU being loaded to 100%. All tests were completed on an Intel i7 1.8 GHz with 16-GB RAM running Ubuntu 14.04 Desktop (Linux kernel version 3.13.0.74.80). This test generated 100,000 data points for each of the sets. The results are shown in **Figure 7**. A TCPROS node was included in the test, as this is the default communication method within ROS. To use ROS inter-node communication requires special setup using ROS nodelets, which may not always be possible. The inclusion of TCPROS also provides a reference point for network communication speeds.

These results show that NUClear is faster at routing messages than ROS. In fact, the latency between modules in NUClear is faster than routing within a node in ROS. Interestingly, NUClear's performance improved when using a thread pool under system load. This is believed to have been caused by the delay in waking up a sleeping thread. When the system is already under load, these threads do not go to sleep, which reduces latency. More rigorous testing of the compile time message routing system is planned for future research.

3.6. Statistics, Logging, and Traceability

In complex systems, it can be difficult to determine a system's operational state. NUClear is designed to support powerful statistics and logging tools. NUClear stores runtime statistics about each module and provides mechanisms to receive the information

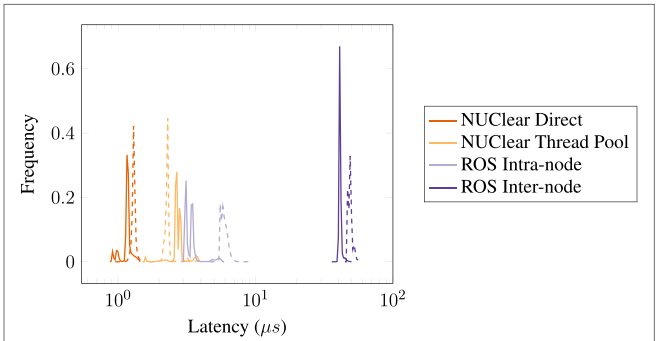


FIGURE 7 | Frequency distribution of end-to-end latency for NUClear *via* a thread pool, NUClear directly sending messages, ROS within a single node, and ROS between nodes on a single machine (TCPROS). Dashed lines represent the test done with the system running at 100% CPU load.

logs on a per-module or per-event basis. These features provide useful information that assist with debugging and understanding the robot's system.

If an error occurs, it is possible to capture the input that caused the error and replay it on the module. This is possible as the architecture itself is aware of the information used by each callback. Each callback requests the list of all messages required to complete its task. This represents the current state of the system. Since these messages can be captured at the point of callback, data that cause errors can be captured and examined more closely. This recording functionality can be used to develop a powerful array of tests that accurately reproduce real-world scenarios. These features are compiled on demand, with unused features not impacting on the performance of the system.

A networked visual debugger, NUsight, is also built into the NUbots' codebase. This system supports streaming operational data in real time to a web-based visualizer. The visualizer can perform real-time charting of time-series and 3D visualization of the robot's believed state. These features are easy to use in modules in the system due to NUClear's extensibility. Additional DSL words are added making code to view internal state only to exist when needed and easy to use.

4. NUClear EVALUATION

The NUbots' codebase is used as an example to quantitatively assess the improvements provided by the NUClear framework. This codebase was chosen as it is a large (~80,000 LOC) codebase that previously implemented a blackboard architecture that can be used for comparison. Additionally, this codebase has multiple distinct binaries with each performing a distinct functional or testing role. Each of these roles includes a different set of modules. Due to the compile time message passing of NUClear and the fact that it uses co-messages, it is possible to extract the graph of message relationships between modules from each compiled binary. Once extracted, it is possible to transform this graph into the equivalent graph for a more traditional message-passing system, such as ROS or YARP, by taking all non-triggering data and creating a separate subscription event for it. This adds a new subscription for this type, as well as a local cache to store the message for when it is used. Only one cache is needed per module, so it is only counted for the first instance of a type.

4.1. Interface Size

Using co-messages in NUClear reduces listener code compared to other message and event-based systems. Rather than having a separate subscriber and cache for each data type, there can be a single subscriber for multiple data types without needing a separate cache. This directly reduces the number of functions that must be written to handle these cases. The difference in the number of functions that must be written can be seen in **Figure 8**. In this graph, the number of subscription handlers is shown for a NUClear system, compared to the theoretical equivalent message-passing system for each module.

Figure 8 shows that while some modules have the same number of callbacks in both systems, there are many cases where up to double the number of subscription handlers must be written for a traditional message-passing system. A large number of these handlers will also be caching the variable for use by the primary function. When refactoring these, caching handlers can be forgotten and contribute to dead or poorly documented code.

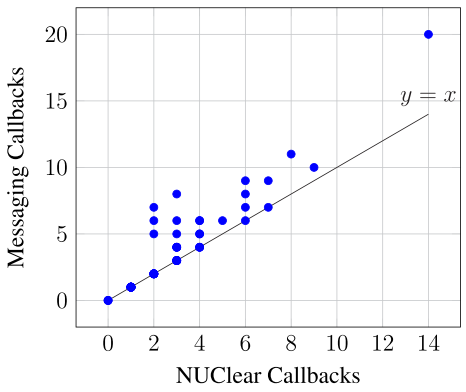


FIGURE 8 | The number of functions needed by a module in NUClear compared to a messaging system. Each point is a module in the NUbots' codebase. The height above the $y = x$ line indicates how many additional callback functions must be written in a message passing system.

Direct comparison with a blackboard system is difficult, as it does not use callbacks and the data must instead be polled. However, variables can be read at any time in a blackboard system. This results in various problems, including thread safety and synchronization.

Whiteboard-based systems allow a similar level of performance to the NUClear system in relation to the code efficiency of each module, as data are accessible at any time. However, whiteboards move the burden of dead and poorly documented data from the individual modules to a large, central data store. Over time, this can become difficult to maintain.

In comparison with these systems, the callbacks in the NUClear system specify which data are needed for a specific operation. This removes the responsibility of implementing cached data storage from the system and results in a system with significantly less implementation overhead, while retaining the speed benefits of universally accessible data structures, such as blackboard architectures. It also maintains most of the modularity and flexibility of traditional message-passing systems, such as ROS.

4.2. Memory Usage

NUClear provides a reduced memory footprint in comparison to message-passing or blackboard-based systems. This is because it is able to determine and store only the data that are live at any time. **Figure 9** shows the number of additional cache variables required for a messaging implementation of each of the binaries present in the NUbots' codebase. These additional variables are required as each module must cache any data type that arises as a result of a non-triggering message ("With" messages in the NUClear framework). As a result of NUClear managing all of these variables centrally, additional caching variables are not required in a NUClear system. This can be useful for memory constrained systems.

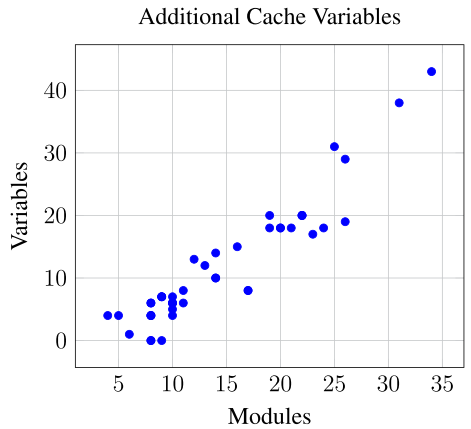


FIGURE 9 | The number of additional cache variables that would be required if the NUbots' codebase was implemented in a message-passing system. These additional variables would be required to store data that were requested as a non-triggering type. Each point represents a single binary from the NUbots system.

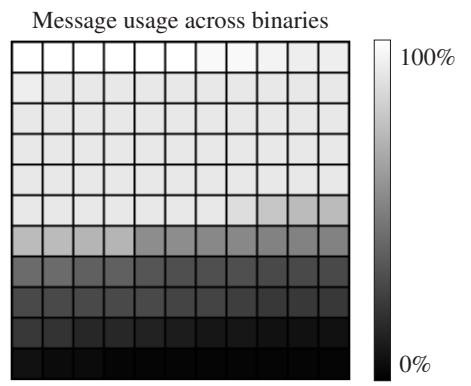


FIGURE 10 | The distribution of usage for the 121 message types across binaries in the NUbots' system. This serves as an indication as to how much memory an equivalent blackboard system would use.

An indication of memory usage in a theoretical blackboard system can be seen in **Figure 10**. In the NUbots' system, there are currently 121 message types displayed as an 11 × 11 grid. However, only a subset of these messages is used in any binary. The binary that uses the largest number of messages only uses 110. If a blackboard were to be used, all of these messages would be stored for every binary. This can also be seen by comparing the system to the previous blackboard based NUbots' system. By comparison, NUClear is able to determine which messages are needed for listeners at compile time and does not store unused data types.

4.2.1. Cache Computational Overhead

In a message-passing system, there is an additional cost incurred. When a message is received in a traditional messaging system and is used as additional data, it must be cached. Writing this cache costs performance from copying the data to a local variable for storage. Large messages can have a significant cost if they must be repeatedly copied to local variables. Additionally, when the data rates are not matched, much of the copied data are never used. For example, in the NUbots' codebase, the sensors are read at a rate of 120 Hz and images at 30 Hz. Ninety sensor messages per second would not be used, but in a message-passing system would still be cached. When the code is running on a system with limited CPU power, these costs reduce the available computational resources for other tasks.

FIGURE 11 | Modified Darwin-OPs of the 2015 NUbots team equipped with new 3D-printed heads using a higher resolution camera, padded jackets to soften falls, and soccer studs to improve grip on artificial grass.

5. TARGET PLATFORMS

The current primary target for the NUClear architecture is the Darwin-OP platform (Ha et al., 2011). Modified versions of the Darwin-OP that have slightly different kinematics and camera sensors are also supported, pictured in **Figure 11**. These platforms have twenty degrees of freedom provided by ROBOTIS serial controlled servomotors, a six degree-of-freedom IMU and a webcam for sensing. On-board processing is provided by an embedded Atom z530 processor running at 1.6 GHz.

The NUClear architecture has also been used in the RobotX Maritime challenge on an autonomous marine platform. This platform involved the use of an embedded control system and a set of four global shutter cameras with wide field of view lenses. Due to the system modularity that NUClear allows, large parts of the vision system were taken from the NUbots' soccer codebase with minimal changes.

Future plans include deploying NUClear to the NimbRo-OP platform (Allgeuer et al., 2015) and autonomous quadcopters. These will be interesting platforms to test the flexibility of the NUClear architecture. The larger Nimbro-OP platform also provides opportunities for direct comparisons with a ROS-based framework. This future work will support a comparison of maintainability and efficiency across a spectrum of robot software architectures with the factor of hardware variation removed.

6. CASE STUDY – NUbots' CODEBASE

The architecture implemented in NUClear allows a humanoid robot to perform a variety of tasks within structured and semi-structured environments. Taking advantage of the message passing and storing capabilities of the system and building on these in each of the subsystems has allowed a flexible, but principled approach to robot control. A simplified flow of information through the NUbots' system is given in **Figure 12**.

The core of the NUbots' system is a see-think-do type information flow. This is common to previous architectures. It allows for information to be taken in from the surroundings and for planning and movement to take place. In addition to innovations and improvements within the subsystems of this loop, the architecture also allows a generic fast-path that enables all motor skills to react to changes in the environment quickly. This can provide balance and reflexive protection skills more easily than in previous systems, with reaction speeds that are faster than current message-passing systems.

6.1. Vision Pipeline

Many challenges for mobile and embedded robotics involve the use of computer vision to sense and navigate environments. In recent years, environments for humanoid robotics challenges have transitioned from a well-defined color and pattern-based structure toward more realistic environments. As these challenges increasingly reflect the real-world, semi-structured environments where shape and context give vital additional information about the world must be considered. The machine vision community has developed many approaches to structure-based detection. However, most of these cannot yet be computed in real-time on lower power embedded systems. The vision pipeline of the NUClear system represents an efficient compromise between structure-based and color-based systems by using color classification to find regions of interest, followed by edge-based shape fitting to find particular objects (Quinlan et al., 2004; Henderson et al., 2008; Houliston et al., 2015).

The vision system used in the NUbots' system shows one significant difference to embodied vision systems, popular in the ROS message-passing system. The source code provided by Allgeuer et al. (2015) uses a monolithic vision module in ROS, as the performance penalty for message-passing (particularly for large data types such as images) for ROS communications is considered quite high. This can make message-passing vision systems in ROS slow to process and report changes in the environment. Due to the much faster message dispatch, the NUClear architecture is able to provide both better modularity and performance by parallelizing module execution where possible. Additionally, the multithreading controls provided by the NUClear architecture remove the need for detecting when the system is overloaded and lagging. This removes the need for implementing queues and high-watermark throttling.

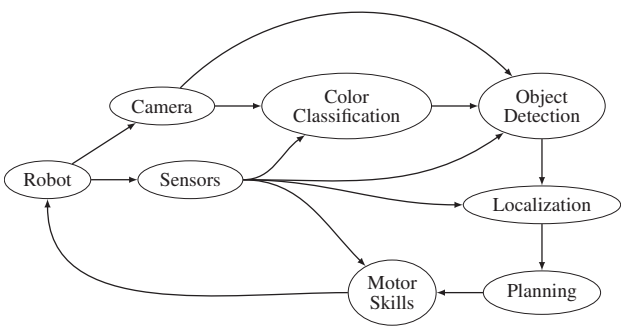


FIGURE 12 | High level overview of the NUbots' information flow loop. Each of the elements represents a collection of modules that work together to achieve the goal of that element.

The first stage in the vision pipeline involves reducing an image from raw pixel data to color and edge information. Using the thread-aware features of the NUClear architecture, the vision pipeline ignores incoming images when the compute load is too high. It is also possible to define multiple input camera streams, each of which processes and drops frames independently and fairly. This allows smooth operation for multi-camera robots, as well as avoids loading down the system with heavy image processing. The image color classification itself is performed using a look-up table that converts pixel values into symbolic colors.

Several investigations have been made into automating and improving color classification within these systems (Henderson et al., 2008; Röfer et al., 2011). The current system furthers this work by dynamically adapting to light and color conditions based on detected objects and regions. Using the NUClear framework's Priority keyword, it is possible to run the dynamic color adaptation as a low priority process that does not interfere with the system's normal running or process outdated data.

Objects are detected using a collection of independent color and shape-based detector modules (Murch and Chalup, 2004). As the inputs and outputs for detector modules are defined as messages and the NUClear system is compiled as a collection of modules, it is very simple to include or swap out detectors for various testing purposes. This strength is similar to the utility that ROS provides; however, message dispatch times are significantly reduced. This allows real-time tracking of relatively fast-moving objects such as rolling balls. NUClear also provides a functionality to disable and enable the execution of modules at run time, allowing detectors that are not needed for the current task to be switched off.

Versions of image color classification have been used within the context of robot competition environments for some time. However, improvements in camera sensors and noise filtering have reached a level of robustness suitable for deployment in more difficult and dynamic environments. As such, this system was also used in the RobotX Maritime Challenge with minimal modification.

6.2. Localization and Mapping

With the modularity of the rest of the NUClear architecture, it is important when developing localization components to allow for a range of inputs and drop-in replacements. Localization modules take visual detector observations and orientation sensor updates as inputs. The structure of observations provides an angle

and distance from the robot in three dimensions. This supports inputs to be as diverse as users' faces to marker symbols on the ground, while using the same interface. Multiple estimations with differing confidences are allowed for each detection, as there may be more than one hint in the image about the distance of the object. The significant difference between NUClear and other message-passing architectures for localization and mapping is that NUClear removes the need for maintaining caches and matching sensor timestamps when performing data fusion updates reducing the amount of code required. This improves the clarity of the code and algorithms used, as well as improving efficiency slightly.

6.3. Planning and Actuation

The NUbots' system utilizes an extended form of subsumption logic (Brooks, 1986) to arbitrate access between skill modules and the robot's hardware. The system operates through the registration of each module at system start-up (which can be specified by the NUClear Startup event), along with their current subsumption priority and the hardware components they wish to control. The subsumption controller then allocates resources to the registered modules based on which module has the highest priority for a subset of the robot's limbs. The subsumption priority of a module can be changed in real time by request from that module, allowing the system to dynamically allocate control as priorities change.

The use of a flexible, thread-aware message-passing system with very fast dispatch times supports the implementation of modular reflexive behaviors and failsafes without impacting on the implementation of the rest of the system. This has been used in the NUbots' system to implement universal reflexive responses. An obvious candidate for reflexive behavior on a bipedal platform is the implementation of a protective reflex to reduce damage from falls. This reflex acts on the filtered gyroscope and accelerometer data produced from the sensors without going through any other processing or delays. If the robot is falling, the reflex takes highest priority on all limbs until the accelerometer indicates that the robot has come to rest.

The fall reflex was originally designed to mimic humans, putting arms out to soften the impact. It was found that throwing the arms forward with elbows bent and then deactivating the motors to become fully compliant was effective at reducing the impact to the robot's body. These types of reflexes are only effective if action is taken almost at the instant the robot realizes it will fall; otherwise, there will not be enough time to effectively reposition the body. For this reason, slower message-passing systems may not be able to effectively implement reflexes as independent modules.

Head behavior is one of the unique challenges involved with humanoid robotics. If a robot is to be truly humanoid, it should not use a visual system that has a field of view greater than that of a human. This limits the amount of the environment that can be perceived at any one time. Humans meet this challenge with two key behavioral systems: head movement and eye movement. The movement of the head adjusts the coarse field of view, while the much faster movement of the eye defines where high detail is perceived (Duchowski, 2007).

The NUbots' head behavior system has been developed to mimic the blur-reduction of human eye behavior, in particular, the vestibulo-ocular reflex (Fetter, 2007). The vestibulo-ocular reflex is the mechanism by which stationary objects in the world are stabilized on the retina of the eye during head movement. This works through a tight feedback loop with both vestibular information from the balance system and visual information. Humanoid robots have a similar problem with image stabilization. When the robot's body rotates, typically the head rotates with it. This can blur the camera images. The NUbots' system takes the most recent orientation information, in the form of a rotation matrix, and uses this to set a constant look direction in the global reference frame regardless of the robot's motion. As with the fall-protection reflex, this type of reflexive behavior requires very fast end-to-end response times to be effective.

6.4. Robot Learning

Learning is a fundamental human skill. As online planning and machine learning algorithms advance, it is important to incorporate features that simplify their implementation on any robot software architecture. Distinctions must be made between online learning, offline learning, on-board learning, and learning processed on an more powerful external system. Most message-passing systems support network communication and data output for offline learning. The NUClear architecture also provides a high efficiency framework for implementing embodied and online learning systems, as well as methods for sharing, storing, and visualizing the data produced by these systems.

Of particular use in the context of learning is the NUClear Last keyword. This provides a cached stream of events to be processed during learning updates. This simplifies the implementation and maintenance of embodied movement-based optimization and learning algorithms, which often operate in batch mode on a stream of evaluation data at the end of a movement (Kalakrishnan et al., 2011; Budden et al., 2013). As such, NUClear simplifies the development process for many embodied machine learning algorithms when compared to previous architectures such as the platform of Kulk and Welsh (2012). Due to lower system overhead and faster inter-module communications, NUClear is also more efficient than traditional message-passing architectures such as ROS when fulfilling this role. NUClear has key advantages over other systems due to its co-messaging and built-in keywords that allow simpler implementation of machine learning algorithms.

7. CONCLUSION

Message-passing systems are an ideal solution to robotic architectures. This is supported by the popularity of the ROS research architecture and other message-passing architectures. Systems based on message passing have been at the forefront of software architectures for high functioning robots for the last decade. The isolated components operate independently and when coupled with a message-passing system, they can be altered and replaced with much greater ease.

The limitations that prevented these architectures from being deployed on robots without sufficient performance have been

removed through the development of the NUClear framework. The NUClear framework is able to operate at speeds that approach native function calls, while also providing transparent multi-threading and type safe data storage. The ability to access multiple message types simultaneously greatly speeds the development of new modules, as this access pattern mirrors humanoid robots' sensor systems. In contrast to other message-passing systems, it allows programmers to specify the global data requirements as a part of the listener declaration, rather than constructing a separate listener for each piece of data required. These data are guaranteed by the NUClear framework to be accurate and act as a clear dependency definition in one place, which improves the knowledge of the system.

These advantages make the NUClear system a valuable proposition for research robotics. The advantages also suggest that further investigation should be taken into message based systems that contain a global message store. These systems have properties that greatly simplify the fusion and processing of data streams that are required for modern humanoid robotics, which could lead to more maintainable and flexible systems in the future.

AUTHOR CONTRIBUTIONS

The system has been developed over several years where the students of the team (TH, JW, JF, and MM) contributed code development with the software architecture work performed by TH. The academics in the team (SC, AM, and YL) contributed management and supervision. The lead authors of the current manuscript are TH with assistance from JW. Information for sections of the case study was provided by JF and MM.

ACKNOWLEDGMENTS

The authors are grateful to all previous members of the NUbots team who contributed to development of the software base since 2002. Additional key developers and contributors at various stages of this process were Brendan Annable, Monica Olejniczak, Peter Turner, Aaron Wong, and Jake Woods. The authors thank Alex Biddulph for his assistance running the timing comparisons with ROS. The authors are also grateful to Ellie-Mae Simpson for assistance proofreading this manuscript.

REFERENCES

Allgeuer, P., Farazi, H., Schreiber, M., and Behnke, S. (2015). "Child-sized 3d printed igus humanoid open platform," in *IEEE-RAS International Conference on Humanoid Robots (Humanoids)* (Seoul: IEEE).

Barrett, S., Genter, K., He, Y., Hester, T., Khandelwal, P., Menashe, J., et al. (2013). "UT austin villa 2012: standard platform league world champions," in *RoboCup 2012: Robot Soccer World Cup XVI, Volume 7500 of Lecture Notes in Artificial Intelligence (LNAI)*, eds X. Chen, P. Stone, L. E. Sucar, and T. V. D. Zant (Berlin; Heidelberg: Springer), 36–47.

Beck, F., and Diehl, S. (2011). "On the congruence of modularity and code coupling," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11* (New York: ACM), 354–364.

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE J. Robot. Autom.* 2, 14–23. doi:10.1109/JRA.1986.1087032

Bruyninckx, H. (2001). "Open robot control software: the OROCOS project," in *IEEE International Conference on Robotics and Automation, 2001. Proceedings 2001 ICRA*, Vol. 3. (Piscataway, NJ: IEEE), 2523–2528.

Budden, D., Walker, J., Flannery, M., and Mendes, A. (2013). "Probabilistic gradient ascent with applications to bipedal robot locomotion," in *Australasian Conference on Robotics and Automation 2013 (ACRA 2013)*, eds J. Katupitiya, J. Guivant, and R. Eaton (Australian Robotics & Automation Association).

Coleman, R., Estivill-Castro, V., Fernandez, E., Geffner, H., Gilmore, E., Ferrer, J., et al. (2013). *Mi-Pal Team Description 2013*. Available at: http://www.informatik.uni-bremen.de/spl/pub/Website/Teams2013/MiPAL.pdf

Corke, P., Sikka, P., Roberts, J. M., and Duff, E. (2004). "DDX: a distributed software architecture for robotic systems," in *Proceedings of the 2004 Australasian Conference on Robotics and Automation (ACRA 2004)*, eds N. Barnes and D. Austin (Australian Robotics & Automation Association).

Dantam, N., Lofaro, D., Hereid, A., Oh, P., Ames, A., and Stilman, M. (2015). The Ach library: a new framework for real-time communication. *IEEE Robot. Autom. Mag.* 22, 76–85. doi:10.1109/MRA.2014.2356937

Duchowski, A. (2007). *Eye Tracking Methodology: Theory and Practice, Second Edition*. Springer-Verlag London Limited.

Edwards, S., and Lewis, C. (2012). "Ros-industrial: applying the robot operating system (ROS) to industrial applications," in *IEEE Int. Conference on Robotics and Automation, ECHORD Workshop*. St. Paul.

Erman, L. D., Hayes-Roth, F., Lesser, V. R., and Reddy, D. R. (1980). The hearsay-II speech-understanding system: integrating knowledge to resolve uncertainty. *ACM Comput. Surv.* 12, 213–253. doi:10.1145/356810.356816

Fetter, M. (2007). Vestibulo-ocular reflex. *Dev. Ophthalmol.* 40, 35–51. doi:10.1159/000100348

Ha, I., Tamura, Y., Asama, H., Han, J., and Hong, D. (2011). "Development of open humanoid platform DARwIn-OP," in *Proceedings of SICE Annual Conference 2011 (SICE2011)* (Tokyo: The Society of Instrument and Control Engineers (SICE) and IEEE), 2178–2181.

Hammer, T., and Bäuml, B. (2014). The communication layer of the aRDx software framework: highly performant and realtime deterministic. *J. Intell. Robot. Syst.* 77, 171–185. doi:10.1007/s10846-014-0095-9

Hayes-Roth, B. (1985). A blackboard architecture for control. *Artif. Intell.* 26, 251–321. doi:10.1016/0004-3702(85)90063-3

Henderson, N., King, R., and Chalup, S. (2008). "An automated colour calibration system using multivariate Gaussian mixtures to segment HSI colour space," in *Proceedings of the 2008 Australasian Conference on Robotics and Automation (ACRA 2008)*, eds J. Kim and R. Mahony (Australian Robotics & Automation Association).

Houliston, T., Metcalfe, M., and Chalup, S. K. (2015). "A fast method for adapting lookup tables applied to changes in lighting colour," in *RoboCup 2015: Robot World Cup XIX, Volume (9513) of Lecture Notes in Artificial Intelligence (LNAI)* (Berlin; Heidelberg: Springer), 190–201.

Kalakrishnan, M., Chitta, S., Theodorou, E., Pastor, P., and Schaal, S. (2011). "Stomp: stochastic trajectory optimization for motion planning," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on* (IEEE), 4569–4574.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., and Osawa, E. (1997). "RoboCup: the robot world cup initiative," in *Proceedings of the First International Conference on Autonomous Agents, AGENTS '97* (New York: ACM), 340–347.

Kulk, J., and Welsh, J. S. (2012). "A NUPlatform for software on articulated mobile robots," in *Leveraging Applications of Formal Methods, Verification, and Validation, Communications in Computer and Information Science*, eds R. Hähnle, J. Knoop, T. Margaria, D. Schreiner, and B. Steffen (Berlin; Heidelberg: Springer), 31–45.

Lim, J., Shim, I., Sim, O., Kim, I., Lee, J., and Oh, J.-H. (2015). "Robotic software system for the disaster circumstances: system of team KAIST in the DARPA robotics challenge finals," in *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)* (Danvers: IEEE), 1161–1166. doi:10.1109/HUMANOIDS.2015.7363509

Magyar, G., Sinčák, P., and Krizsán, Z. (2015). "Comparison study of robotic middleware for robotic applications," in *Emergent Trends in Robotics and Intelligent Systems, Volume 316 of Advances in Intelligent Systems and Computing*, eds P. Sinčák, P. Hartono, M. Virčíková, J. Vaščák, and R. Jakša (Cham: Springer International Publishing AG), 121–128.

Matamoros, J. M., Savage-Carmona, J., and Ortega-Arjona, J. L. (2015). "A comparison of two software architectures for general purpose mobile service robots," in *2015 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC 2015)*, eds A. Valente, R. Morais, L. Almeida, and L. Marques (Los Alamitos: IEEE Computer Society), 131–136.

Metta, G., Fitzpatrick, P., and Natale, L. (2006). YARP: yet another robot platform. *Int. J. Adv. Robot. Syst.* 3, 43–48. doi:10.5772/5761

Murch, C. L., and Chalup, S. K. (2004). "Combining edge detection and colour segmentation in the four-legged league," in *Proceedings of the 2004 Australasian Conference on Robotics & Automation (ACRA'2004)*, eds N. Barnes and D. Austin (Australian Robotics & Automation Association).

Orebäck, A., and Christensen, H. I. (2003). Evaluation of architectures for mobile robotics. *Auton. Robots* 14, 33–49. doi:10.1023/A:1020975419546

Page-Jones, M. (1988). *The Practical Guide to Structured Systems Design*, 2nd Edn. (Upper Saddle River, NJ: Yourdon Press).

Pressman, R. S. (2005). *Software Engineering: A Practitioner's Approach*, 6th Edn. (Columbus, OH: McGraw-Hill Education).

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., et al. (2009). "ROS: an open-source robot operating system," in *2009 IEEE ICRA Workshop on Open Source Software*, Vol. 3.

Quinlan, M. J., Chalup, S. K., and Middleton, R. H. (2004). "Application of SVMs for colour classification and collision detection with AIBO robots," in *Advances of Neural Information Processing Systems (NIPS'2003)*, Vol. 16, eds S. Thrun, L. Saul, and B. Schölkopf (Cambridge, MA: The MIT Press), 635–642.

Röfer, T., Laue, T., Müller, J., Burchardt, A., Damrose, E., Fabisch, A., et al. (2011). *B-Human Team Report and Code Release 2011*. Available at: http://www.b-human.de/downloads/coderelease2012.pdf

Thórisson, K. R., List, T., DiPirro, J., and Pennock, C. (2005a). *A Framework for AI Integration*. Technical Report, RUTR-CS05001. Reykjavik University Department of Computer Science.

Thórisson, K. R., List, T., Pennock, C. C., Dipirro, J., Magnusson, F., Thórisson, K. R., et al. (2005b). "Whiteboards: scheduling blackboards for interactive robots," in *AAAI-05 Workshop on Modular Construction of Human-Like Intelligence, Twentieth Annual Conference on Artificial Intelligence, Pittsburgh, PA, July 9-13, 2015* (Palo Alto, CA: AAAI).

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

frontiers in Robotics and AI

TECHNOLOGY REPORT
published: 08 April 2016
doi: 10.3389/frobt.2016.00017

A Cross-Platform Tactile Capabilities Interface for Humanoid Robots

Jie Ma[1] *and Torbjørn S. Dahl*[2]*

[1] China Research Laboratory, IBM, Beijing, China, [2] Centre for Robotics and Neural Systems, School of Computing, Electronics and Mathematics, Plymouth University, Plymouth, UK

This article presents the core elements of a cross-platform tactile capabilities interface (TCI) for humanoid arms. The aim of the interface is to reduce the cost of developing humanoid robot capabilities by supporting reuse through cross-platform deployment. The article presents a comparative analysis of existing robot middleware frameworks as well as the technical details of the TCI framework that builds on the existing YARP platform. Currently, the TCI framework includes robot arm actuators with robot skin sensors. It presents such hardware in a platform-independent manner, making it possible to write robot control software that can be executed on different robots through the TCI frameworks. The TCI framework supports multiple humanoid platforms, and this article also presents a case study of a cross-platform implementation of a set of tactile protective withdrawal reflexes that have been realized on both the NAO and iCub humanoid robot platforms using the same high-level source code.

Keywords: tactile capabilities interface, humanoid robotics, robot skin, protective reflexes, robot software engineering

OPEN ACCESS

Edited by:
Nikolaus Vahrenkamp, Karlsruhe Institute of Technology, Germany

Reviewed by:
Fulvio Mastrogiovanni, University of Genoa, Italy
Kensuke Harada, National Institute of Advanced Industrial Science and Technology, Japan

***Correspondence:**
Torbjørn S. Dahl
torbjorn.dahl@plymouth.ac.uk

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *07 December 2015*
Accepted: *24 March 2016*
Published: *08 April 2016*

Citation:
Ma J and Dahl TS (2016) A Cross-Platform Tactile Capabilities Interface for Humanoid Robots. Front. Robot. AI 3:17. doi: 10.3389/frobt.2016.00017

1. INTRODUCTION

During the last few decades, robots have been used with success in various domains ranging from manufacturing (Merzouki et al., 2010), space exploration (Ambrose et al., 2010), and surgery (McMahan et al., 2011) to mining (Bednarz et al., 2011) and military assistance (Wooden et al., 2010). Developing robotic software is difficult and time-consuming, especially when the same functionality must be developed separately for robots with different physical dimensions, hardware control protocols, mechanical configurations, or actuators and sensors. Even on a single robot, it is common for low-level components, such as dynamics and servos, to vary due to upgrades during the robot's lifetime. The cost of robot software development can be reduced significantly if the software can be reused across different models and platforms. In general, the main challenges of developing humanoid robot software are *modeling complexity*, *modularity*, and *repeatability*.

Humanoid robots are often equipped with a large number of actuators and sensors. The first difficulty a developer may encounter is to learn these specifications. Even when building a simple robotic behavior with just handful devices, nevertheless, it may be time consuming for the developer to work out the correct mappings from the platform-related infrastructure to build an appropriate behavioral model. For example, a humanoid robot NAO that our research employed has 21 DoF servos and 648 tactile sensors (see **Figure 1**), but a withdrawal reflex behavior studied in §4 is only interested in 5 servos and about 30 taxels. To identify and configure the correct taxels may become a challenge for the robotic behavior engineer.

From the perspective of the software engineering, modularity is also important in humanoid robot. Currently, humanoid robot projects are usually requiring intensive collaboration among

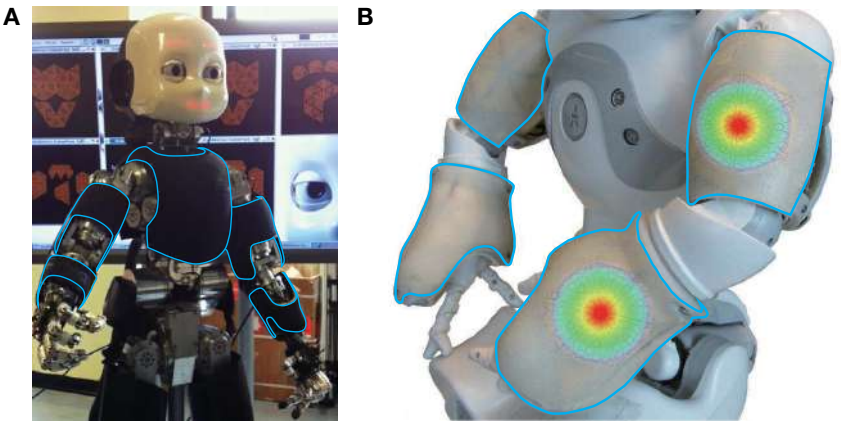


FIGURE 1 | (A,B) illustrate iCub and NAO robots, respectively. Their equipped robotic skin is highlighted. The TCI framework presented in this article was evaluated on both platforms and both on the real robots and in simulations.

different specialists. Developers may use diverse programming languages, operating systems, or even computing hardware. Thus, the need to separate functions into reusable modules has been increasingly growing, so people can just focus on smaller-scaled problems. Without a cross-platform interface, however, it makes communication and integration difficult as it typically triggers extra work to facilitate the interaction between the separate components and thus makes modularity difficult to achieve. Shared generic interfaces can make robotic software more extensible and reduce the couplings among modules. They can also facilitate the development by supporting multiple operating systems and programming languages.

After the development of a behavior for a specific robot, software engineers are commonly interested in transferring the same behavior to other types of robot. As a matter of fact, behavioral transfer is not easy to achieve and the repeatability of humanoid robotics is sometimes criticized for being difficult to reproduce outside of their original laboratories (Anderson and Thomaz, 2010). The main reason making repeatability difficult is the hardware differences among the humanoid robots. The cost of directly migrating a platform-specific solution to a new type of robot is high without a decent generic interface, because a developer needs to figure out the geometric transformations and adjust high-level behavioral parameters correspondingly. By doing this, the repeatability of the behavior is broken and so it becomes obscure to verify and compare the effectiveness of the same behavior on the new robot.

The work presented in this article goes beyond traditional robot middleware platform by attempting to hide all platform-specific details from developers and, thus, allow them to produce reusable cross-platform behaviors via an abstract interface. The interface focuses on interpreting abstract information for different native different robot arms with different physiologies. This article presents results in developing a cross-platform tactile capabilities interface (TCI) that aims to improve the reusability of humanoid robot software and hardware. The results presented are limited to humanoid arms but includes a standardization of both actuators and tactile sensors that covers a large area of a robotic arm.

The research presented is directly motivated by our experiences from developing cross-platform software during the FP7 ROBOSKIN project (Cannata et al., 2012). Our research involved different humanoid robots, including the NAO and iCub robots (see **Figure 1**). During the research, the main challenges were to develop generic robot capabilities. In particular, we developed prototype algorithms for one robot platform and later re-implemented them on others. The objective of TCI is to facilitate such transfers of robot capabilities by providing a generic interface that is practical for a range of different humanoid robots. Our approach aims to enable the developers working on cross-platform capabilities, in particular protective withdrawal reflexes, to focus on controlling an abstract, platform-independent robotic component through a set of abstracted interfaces. TCI consists of generic actuator interfaces and generic robot skin interfaces. It acts as an interpreter translating the messages between the cross-platform algorithms and the platform-specific layers of the actual robots. High-level algorithms then become reusable and extensible, and new robots can be supported by providing them with support for TCI.

In §2, related concepts, systems, and literature are reviewed. The theories and methods for promoting robot software reusability are discussed, and a selection of robot middleware platforms compared. §3 presents the architecture implementing our interface and discusses its design. The TCI specifics of the arm actuators and the arm skin sensors are presented in §3.1 and §3.2, respectively. This is followed by a case study of how TCI was used to support a cross-platform implementation of a humanoid robot protective arm withdrawal reflexes in §4. Conclusions are made in § 5.

2. RELATED WORK

A humanoid robot system commonly consisted of a set of layered modules. Low-level modules focus on solving hardware-related preprocessing and reasoning problems, such as localization and sensory data fusion, whereas high-level ones are making cognitive behavioral decisions based on the high-level states produced by

mother modules. To develop a generic software for controlling multiple type of humanoid robots is usually difficult, and the key problem is to design a high-level robot behavior that can be easily deployed on different robots.

Machine learning (ML) algorithms can be one possible solution, which provides a generic way of giving different robots the same capabilities, even in the absence of a detailed understanding of the underlying specifications or kinematics model. The core philosophy of ML is data driven; it focuses on the mappings from a robotic action to the corresponding feedback. Instead of directly solving transformation problems, ML undertakes a training process to understand the outcomes of actions. During the training, it continuingly changes action with an aim of achieving a certain behavior on a robot. Such approaches include learning the kinematics and dynamics of a generic robotic arm (Atkeson, 1989; Caligiore et al., 2010), generic trajectory tracking using neural control (Martins et al., 2008), and high-level humanoid behaviors such as learning biped locomotion (Huang et al., 2001; Ma and Cameron, 2009a,b, 2011).

The training processes of the ML approaches can be time consuming both in terms of acquiring and processing training data. The learning results may also be unreliable without performance guarantees, as sometimes the results are represented in the form of an implicit neural network, making users difficult to verify behavioral effectiveness. However, in reality, robot developers are not completely accessible to the specifications of a new robot. Instead, they are commonly interested in model-driven approaches that directly translate actuator and sensor data to a new platform. More importantly, the translation should be better processed in real time without waiting for the learning phase to complete. ML approaches, therefore, have disadvantages in real-time data processing. Compared with ML approaches, even hard-coded solutions can be implemented quicker and can sometimes provide reliable performance across a problem space.

In order to improve the reusability of hard-coded transformations, we have proposed the cross-platform tactile capabilities interface (TCI). This is a model-driven approach consists of a set of standardized representations and functions for humanoid actuators and tactile sensors. We have realized the interface for the humanoid arms of iCub and NAO robots.

This section reviews related approaches to increase the reusability of robot software and hardware. Currently, there are several popular robot middleware platforms, and their features are discussed and compared in §2.2. Since TCI provides a generic platform not only for tactile sensors but also for the actuators using them, this section also includes the kinematics features of the middleware. TCI uses YARP (Yet Another Robot Platform) (Metta et al., 2006) as its middleware platform because it was already been supported on several humanoid robots. YARP also supports a wide range of networking protocols that can be flexibly deployed in diverse circumstances. Technical details of YARP will be further discussed in this section.

2.1. The Reusability of Robotic Software

Robot software reusability can be achieved by decomposing robot functions into modules connected to each other within a shared middleware framework. A middleware framework typically provides a fundamental infrastructure for module interaction including abstractions for sensors and actuators, so that different abstract modules are not restricted to specific robot hardware or operating systems, but can be instantiated by multiple specific solutions. With the help of such middleware, supplemental tools, such as behavioral abstraction composition applications, can also be provided in order to promote reusability further.

2.1.1. Modularity

In software engineering, modularity implies that software is broken down into a number of simpler modules. High modularity means more modules and less coupling, i.e., dependency among them. At the other end of the spectrum, monolithic approaches implement all the required tasks within a single program. Designing reusable software involves finding the best trade-off between too much modularity, which can be wasteful, and too little, which limits reusability (Brugali and Scandurra, 2009). A monolithic approach must include all the aspects of robot control, from low-level messaging to high-level algorithms. This approach includes solving platform-specific issues and, as a consequence, reusability is difficult to achieve. On the other hand, too much modularity can also reduce reusability. Practically, it is time consuming to integrate a large number of modules, and maintaining and documenting many modules also demands a large amount of resources. With the balanced level of modularity, the high-level modules can focus on cross-platform capabilities using abstract interfaces. The low-level modules are then responsible for translating the abstract representations and algorithms to platform-specific data and instructions.

2.1.2. Middleware and Toolkits

In order to promote modularity, much effort has been made to create shared robot middleware for different robots. Middleware platforms typically provide a communications framework for robot software modules, supporting both low-level control and high-level algorithms. Middleware typically also aims to reduce infrastructure-level programming and promote the development of reusable robot modules. This can be done through standardized interfaces, drivers for diverse hardware, and supporting multiple programming languages. The last point is often achieved by basing module communication on cross-platform communication technologies, such as TCP/IP sockets, supported by most programming languages.

Standardized data representations and control and communication interfaces allow middleware platforms to also provide a number of development toolkits for speeding up development and debugging. These toolkits can comprise visualization tools to display data, such as *playercam* (from *Player*) that remotely monitors camera images, a graphical user interface (GUI) that manages the modular processes and their connections, such as *yarpmanager* (from *YARP*), and a utility to query and inspect code trees and find dependencies, such as *rospack* (from *ROS*). Abstraction mechanisms, such as *Rosbridge* (Crick et al., 2012), can further increase reusability by allowing third-party tools, e.g., the image processing library (IPL) and the OpenCV library to be accessed from within reusable robot modules. A detailed analysis

of the specific features of a selection of popular middleware platforms is presented in §2.2.

2.1.3. Behavior Abstraction and API Standardization

Another mechanism that increases the reusability of robot software is behavior abstraction. Complex robotic tasks are easier to achieve if they can be decomposed into a hierarchy of capabilities or behaviors. In cases where such decomposition is possible, sub-behaviors constitute potential reusable modules, leaving high-level deliberative decision-making mechanism, such as planning to focus on scheduling abstract behaviors, rather than handling a higher number of low-level actions directly. Such behavioral decomposition has been widely used in various AI systems (Maes, 1991; Stone and McAllester, 2001; Nesnas et al., 2006), especially multi-agent systems (MAS) where cooperative behaviors are needed (Stone, 2000; Ma, 2011). It has also been successful in learning scenarios.

Abstract robot behaviors need to present standard data structures and application programming interfaces (APIs) in order to be used by high-level decision processes. The task description language (TDL)[1] is a C++ library that provides syntactic support for behavioral decomposition, synchronization, and execution monitoring. TDL reduces the difficulties of maintaining task-level behaviors, and it has been successfully used in several mobile robot projects, including CLARAty (Nesnas et al., 2006), a reusable platform for NASA's robots. Another similar example is the humanoid motion planner that is designed for humanoid robots in the Joint French-Japanese Robotics Lab (JRL) (Yoshida et al., 2005). The motion planner uses hierarchical architecture to control multiple reusable dynamic tasks such as path planning, gait generation, and collision checking.

2.2. Robot Middleware

Robot middleware platforms provide abstract platforms for robot software. They promote software modularity by providing tools that support flexible communication between different robot components, including communication between distributed processes. Most robot middleware uses networking packages to connect modules, making modules platform independent. Allowing distributed modules also means that modules can be executed on different processors, potentially under different operating systems and stored on diverse media.

YARP (Yet Another Robot Platform)[2] is a robot middleware platform designed for humanoid robots. It is a lightweight open-source platform derived from University of Genova and MIT, and it supports many mainstream programming languages, such as C++, Matlab, Python, JAVA, Perl, and L. It connects modules using various protocols, such as TCP, UDP, UDP multicast, and HTTP. A YARP-based system is a peer-to-peer network of *port* objects, where each object has *read* and *write* ports to receive and send data streams, respectively. One of the advantages of YARP is its synchronization mechanism. A write port can choose whether to wait for all its read ports before or after each update step. From the perspective of a developer of high-level behaviors, a kinematic chain, such as a robotic arm or leg, is implemented using a *PolyDriver* class. Modules based on this class can be read from and written to using an ordered vector of values based on the kinematic joint angles. The low-level limits of an actuator, however, are not effectively managed by the framework, e.g., YARP does not check the velocity limits of a servo before sending a command. It may even break the servo.

ROS (Robot Operating System)[3] takes slightly different messaging approach. Instead of using *read* and *write* ports it employs a publish–subscribe mechanism (Quigley et al., 2009). *Nodes* are computational processes, which communicate with each other by passing messages. A node sends messages by publishing it to a given *topic*, and nodes subscribe to selected topics to receive messages. ROS is also written in C++ but supports other programming languages as well, including Python, Octave, and LISP. In terms of communication, ROS supports the TCP and UDP protocols. In ROS, a kinematic chain is presented as an actuator array, where actuator properties are also defined such as velocity and torque limits. Actuator channels can be subscribed separately or manipulated at the same time by sending vectors of joint angles in a particular order.

The OROCOS (Open Robot Control Software) project (Bruyninckx, 2001; Bruyninckx et al., 2003) is different from YARP and ROS in that it does not emphasize communication between robot components. Instead, it focuses on toolkit libraries for solving common problems encountered by industrial robots. OROCOS is designed for a single robot, and it is not suitable for multi-robot systems where different robots need to cooperate with each other (Namoshe et al., 2008). Based on the GPL license, OROCOS is composed of four C++ libraries: a Kinematics and Dynamics Library (KDL) that solves real-time kinematics and dynamics problems, a Bayesian Filtering Library (BFL) that provides generic filtering functions such as Kalman Filter and Particle Filter, and two supporting libraries that couple robotic components with each other using debugging tools, i.e., Component Library (OCL) and Real-Time Toolkit (RTT). Originally, OROCOS only supported C++, but libraries have been integrated to connect OROCOS to other robotic middleware such as YARP and ROS. Through this mechanism, other languages are indirectly supported. The advantage of the OROCOS libraries is that they implement dynamics algorithms independently of platforms using a predefined abstraction and formalization of the underlying platform hardware, such as the kinematics.

Other robot middleware platforms include the Player/Stage Project[4] and LCM (Lightweight Communications and Marshaling).[5] The Player/Stage Project (Gerkey et al., 2003) consists of *Player*, a robot device server with standardized interface to sensors and actuators, and *Stage*, a 2D multi-robot simulator. Player provides a network interface to a variety of robot and sensor hardware, and many Player-based applications have been adapted for use under ROS. Player can be regarded as a cut-down version of YARP in that it only supports reliable TCP protocols without advanced synchronizing mechanism. In particular, in Player,

[1] TDL is available at http://www.cs.cmu.edu/~tdl/
[2] YARP is available at http://eris.liralab.it/yarp
[3] ROS is available at http://www.ros.org
[4] Player/Stage project is available at http://playerstage.sourceforge.net
[5] LCM is available at http://lcm-proj.github.io/

device interfaces are not buffered. Compared to Player, LCM is a relatively new library that aims to provide a low-latency message passing system for real-time robotics applications. Comparing to ROS, LCM showed notably better transmission efficiency by leveraging its UDP multicast infrastructure (Huang et al., 2010). With regards to representing a kinematic chain, the original *Player* does not support chained actuators. Instead, each actuator has to be used on its own, and it is up to the client application to keep track of any kinematic relations. Similarly, LCM is a lightweight platform for sharing information efficiently. It also does not natively support the management of kinematic chains.

Table 1 compares the properties of different robot middleware platforms in terms of network protocols, communication mechanisms, and open-source licenses. Although the number of the robotic devices in YARP is not as great as in ROS, YARP supports a greater number of humanoid robots. Player is a popular platform for wheeled robots, especially in the navigation domain. However, it is rarely used for humanoid robot control. Therefore, the work presented in this article has adopted YARP as the middleware platform due to its better support for humanoid robots and network communication.

The aforementioned frameworks provide general-purpose mechanisms to design and implement concurrent and distributed software robot components. However, in the particular domain of robot skin area, these frameworks may have problem in processing tactile sensory data. Robot skin often contains a large scale of taxels, which will cause latency for the frameworks to handle a huge volume of data. To solve this problem, Skinware, a framework designed for the large-scale skin, was proposed recently (Youssefi et al., 2014, 2015a,b). Skinware especially emphasizes on real-time tactile data processing and minimizes the latency for concurrent queries. It provides a unified interface to access information originating from heterogeneous robot skin systems and assures portability among different robot skin solutions.

Different from the Skinware, TCI proposed in this article focuses on solving geometrical transformations of the taxels and providing online actuator translations for tactile-based behaviors.

TABLE 1 | A comparison of popular robot middleware platforms, including the supported humanoid robot platforms, communication protocols, and interaction models.

Platform	Humanoid robots	Protocols	Model	License
YARP	NAO, iCub, Babybot, Obrero, Domo, COG, Kismet, and BERT2 KASPAR	TCP, UDP, UDP multicast HTTP, and QNet	R/W	LGPL
ROS	NAO, Romeo, Reddy, and Kondo KHR	TCP and UDP	P/S	BSD
OROCOS	Robonaut	TCP	C/S ORB SRB	GPL
Player	–	TCP	R/W	GPL
LCM	–	UDP multicast	P/S	LGPL

The license under which they are published is also included.
R/W, read/write; P/S, publish/subscribe; C/S, client/server; ORB, object request broker; SRB, service request broker.

Real-time data processing and latency analysis is not the focus of our work. Currently, even with the help of the aforementioned robotic middleware, it is still difficult to directly represent a generic kinematic chain of an arm for multiple humanoid robots. This is due to the fact that different robots use different reference systems for angles and speeds and also due to their different kinematics. **Listing 1** demonstrates the different code implementing the same reflex motion (see **Figures 6C,D**) on the iCub (Gamez et al., 2012) and NAO humanoid robots through YARP. For the right arm, NAO has 6 DoF while iCub has 16. The iCub robot also has more low-level limits, e.g., for damping and speed. The two extracts of code introduce difficulties for maintainability and reusability because each robot arm has a unique initial position tuple and unique coordinate systems. These differences introduce extra costs when it comes to supporting multiple humanoid platforms, even for simple generic motions. This problem is addressed by the TCI, which forces arms to be represented in the form of abstracted five DoF, disregarding the DoF it actually has. The interface is presented in detail in this article.

3. THE CROSS-PLATFORM TACTILE CAPABILITIES INTERFACE

One way of reducing the cost of developing cross-platform humanoid behaviors is to provide a generic interface that allows developers to reuse the same code for different robots. Since the robot control process is bidirectional, a generic robotic interface at least consists of two functions: incoming *sensor abstraction* and outgoing *actuator abstraction*. The differences in robot control software for various robots platforms stem from hardware issues, such as different physical dimension and mechanical configuration, as well as from software issues, such as platform-specific speed and angle units and servo-indexing mechanism.

Different humanoid platforms also have different schemas for joint indexing, as can be seen from the code provided in **Listing 1**. When migrating high-level code, it is up to the developer to figure out how to transfer abstract motions to the target robot. The problem is that this transfer is robot specific and, as a result, a different transfer is required for each robot that is to be supported, accommodating their unique joint indexing or naming. TCI provides a generic representation of humanoid robot arms, including joint control, joint position data, and kinematic chain information and data from robot skin sensors covering large areas of the arm. Such standardization promotes robotic reusability by hiding the low-level differences between specific robot platforms. In order to implement the interface, other platforms must be represented in a way that conforms to the specified data and command formats, e.g., a robot using log-encoded joints must provide a translation layer to convert the generic interface commands into platform-specific commands. Similarly, it must provide a translation layer converting the platform-specific data format to the format used by our interface.

In TCI, the DoF of a generic arm is fixed. All arms are represented by 5 DoF with indices from 0 to 4, always referring to the shoulder pitch/roll/yaw and the elbow pitch/yaw. Our interface enforces such a referencing standard even when the underlying

LISTING 1 | YARP-based code for NAO and iCub implementations of a single reflex motion.

```
/**** NAO robot ****/
const int DOF = 6;
double StiffnessArr[DOF]={0.8, 0.8, 0.8, 0.8, 0.8, 0.8};
double ReflexArr[DOF]={1.19, −1.10, 2.07, 0.04, 0.0, 0.0};
double MIN[DOF]={−2.09, −0.31, −2.09, −1.54, −1.82, 0.0};
double MAX[DOF]={2.09, 1.33, 2.09, −0.03, 1.82, 1.00};
pos− >SetStiffness(StiffnessArr);
for (int i=0; i<DOF; i++){
  StiffnessArr[i]=CheckLimits(MIN[DOF], ReflexArr[i], MAX[DOF]);
}
pos− >positionMove(ReflexArr);
/**** iCub robot ****/
const int DOF = 16;
double StiffnessArr[DOF]={0.4, 0.4, 1, 0.2, 0.2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
double DampingArr[DOF]={0.03, 0.03, 0, 0.01, 0.2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
double SpeedArr[DOF]={20, 20, 0, 20, 40, 0, 0, 50, 50, 50, 50, 50, 50, 50, 50, 50};
double ReflexArr[DOF]={−16.1, 83.3, 0, 15.5, −59, −0.45, 0, 0.3, 50, 24, 90, 5, 90, 16, 90, 115};
double MIN[DOF]={−90, 15, −30, 15, −90, −90, −20, 0, 10, 0, 0, 0, 0, 0, 0, 0};
double MAX[DOF]={10, 95, 50, 105, 90, 0, 40, 60, 90, 90, 180, 90, 180, 90, 180, 270};
SetStiffness(StiffnessArr);
SetDamping(DampingArr);
SetSpeed(SpeedArr);
for (int i=0; i<DOF; i++){
  StiffnessArr[i]=CheckLimits(MIN[DOF], ReflexArr[i], MAX[DOF]);
}
pos− >positionMove(ReflexArr);
```

platform is missing a particular DoF, e.g., no elbow yaw, or has extra DoF. Our interface also standardized the kinematic chain information, using a fixed grounding point, fixed units, ranges, and signs to represent angles and distances.

For the skin sensor data, rather than providing information related to indexed touch-sensitive transistors known as "taxels," our interface uses a spatial reference framework where a taxel is represented as a point in space, defined relatively to the underlying kinematics. Our implementations of the TCI translate the messages between the abstract high-level representations and the specifics of the individual robot representations in real time. This frees a developer from the tedious task of repeatedly resolving the low-level configuration differences.

In practice, developers need an interface to be flexible. Generic capabilities that can be potentially migrated to different robots can be implemented using our abstract interface. However, it is common to mix the generic capabilities with platform-specific capabilities making use of platform-specific sensors and actuators. In our layered architecture, presented in **Figure 2**, the platform-specific API is available through YARP. The complete architecture consists of four layers. Layer L1 is the low-level robotic controller layer. This is the lowest layer that a module can access and contains the hardware specific APIs. Layer L2 is the YARP middleware layer, which provides the communicative platform for modules to send commands and transfer data. This layer also includes the YARP converters that connect specific platforms to the YARP framework, e.g., NaoYarp[6] is a YARP interface for the NAO humanoid platform that wraps up the official NaoQi interface using YARP ports. Our abstract humanoid arm interface forms layer L3, which contains two core interfaces, the actuator interface, i.e., the arm kinematic chain, and the skin interface. These interfaces are presented in detail in §3.1 and §3.2, respectively. Our architecture also provides platform-specific information and calculations through modules, such as forward kinematics (FK) and inverse kinematics (IK), and technologies that have been extensively discussed by Diaz-Calderon et al. (2006). The run-time availability of platform-specific data, such as the kinematics, would allow the integration of our abstract interface and generic libraries such as OROCOS. We aim to include a querying mechanism in future versions of

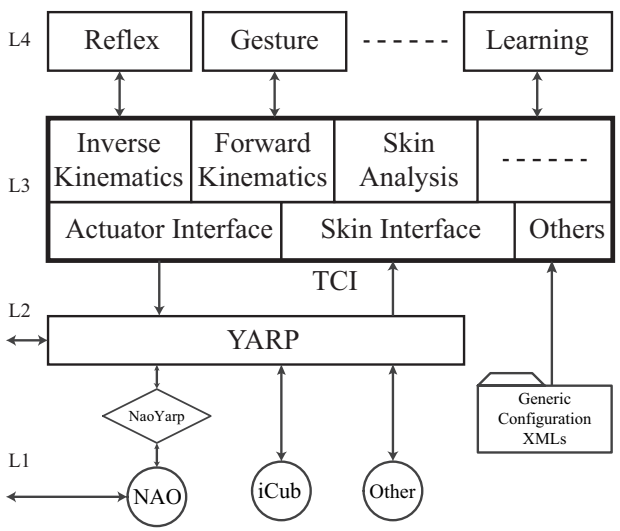


FIGURE 2 | The layered architecture of TCI. L1 and L2 are the low-level robotic controller layers and the YARP middleware layer, respectively. The tactile capabilities interface is defined in L3, which is followed by L4 that consists of reusable cross-platform capabilities.

[6]The NaoYarp software can be found at https://github.com/cbm/NaoYARP

our interface in order to enable the use of third-party libraries in the implementation of platform-independent capabilities. In the top layer, L4, high-level capabilities use our abstract interface to achieve generic tasks, such as withdrawal reflexes, gesture reproduction, and other robotic learning activities.

3.1. The Generic Actuator Interface

The configuration of the actuators of humanoid robots varies dramatically even when they are located on the same kinematic chain of a humanoid part. Practically, each actuator has some unique properties such as speed limits, position limits, initial position, axes, and stiffness settings.

A humanoid robot is usually designed with a specific domain of research in mind, e.g., the iCub robot supports research on cognitive representations for control of the robot head, arms, and hands. On the other hand, NAO robots are designed for locomotive behaviors in research domains such as robot football. As a result, corresponding robot parts on different platforms often have different features, including different degrees of freedom (DoF) and different sequencing of the individual DOF on a kinematic chain. The robot *head* for example has six DoF on the iCub but only two on the NAO. Similarly, the robot *arm* has sixteen DoF on the iCub but only six on the NAO.

Currently, our implementation relies on XML to specify the platform-specific elements of the actuator and skin interfaces. The generic actuator interface consists of a vector of abstract robot parts. Each part is accessible to the high-level components at runtime, providing platform-specific information in a standardized generic format. This unified interface reduces the development complexity for high-level modules by allowing them to focus on the key actuators and ignore any redundant ones that are not needed within a module. In the XML configuration file, the platform-specific information must be presented in terms of the defined number of actuators, axes, and initial positions. As our first attempt, each generic actuator is considered as a linear transformation, e.g., a specific abstract actuator, *GS* maps to a specific destination servo *DS*. The transformation can be defined as in equation (1). Nevertheless, our interface is not limited to this transformation. Other controls, such as transformation in Cartesian space, can also be configured using XML. However, the implementation of other kinematics models will be further evaluated in our future work.

$$GS = \gamma DS + \sigma \tag{1}$$

In equation (1), γ is the transform factor and σ is the transform displacement. Thus, a complex robot part can be generically defined as a vector of generic actuators, each of which is a transformation to a specific destination servo. This representation is formalized in equation (2).

$$GenericPart = \left\langle GS_i|_{i=1}^{n} \right\rangle = \left\langle \left(\gamma_i DS_{mapping(i)} + \sigma_i\right)\Big|_{i=1}^{n} \right\rangle \tag{2}$$

This is the configurations of a vector of servos, and the other properties of a kinematic chain, such as length information, are defined elsewhere. The difference of the limb lengths also affects the position of the end-effector position for given angles.

LISTING 2 | An example of the XML configuration file of a generic robot part. It defines an abstracted *head* part with 2 generic servos mapping from the robot part called *icub_head*. The generic servo 1 is transformed from the destination servo 2 with predefined transform properties and servo limits. The initial position of each generic servo vector is also configured.

```
<Generic_Robot_Part>icub_head</Generic_Robot_Part>
        <generic_servo_number>2</generic_servo_number>
        <generic_servo id="1">
          <destination_servo_id>2</destination_servo_id>
          <transform_factor>1.00</transform_factor>
          <transform_displacement>0.00</transform_displacement>
          <stiffness>1.00</stiffness>
          <damping>0.00</damping>
          <speed>20.00</speed>
        </generic_servo>
          ......
<initial_positions>
          <servo type="deg" destination_servo_id>="0" pos="−2.00"/>
          <servo type="deg" destination_servo_id>="1" pos="15.00"/>
</initial_postions>
```

For position-based behaviors, inverse kinematics and forward kinematics modules (see **Figure 2**) are used to calculate the low-level angles. The servos that make up one generic robot part are not necessarily from the same destination robot part; this feature gives users more flexibility to design robot parts properly. An example of the XML configuration file of a generic robot can be found in **Listing 2**. The generic servo *1* is transformed from the destination servo *2* with $\gamma = 1$ and $\sigma = 0$.

The XML configuration file contains not only the underlying data needed for controlling the real servos but also the information necessary to support a run-time reconstruction of the kinematic chain, e.g., TCI can provide a YARP device that gives run-time access to the Denavit–Hartenberg parameters as well as servo-specific information such as angle and velocity limits. Such a device facilitates the use of generic code that uses third-party libraries for generic computations such as inverse kinematics (IK) calculations (Diaz-Calderon et al., 2006).

3.2. The Generic Robot Skin Interface

A robotic skin sensor is a tactile sensor that gives a robot the "sense of touch" over large areas of its surface. Recently, Dahiya et al. (2010) have extensively compared more than 30 robotic tactile sensors in terms of their transduction method, number of taxels, range of force, and force sensitivity. Though there is no standardized representation of robot skin sensors, they can be regarded as a set of taxels (tactile pixels), where each taxel is located on the same continuous surface and each taxel is able to report, in real time, the force of any contacts. Our interface assumes that the different skin sensors represented have the same level of sensitivity. If this is the case, their readings may be normalized in interface implementation layer, L3, in **Figure 2**. Our interface, as presented here, is not yet sophisticated enough to cover skin sensors with different sensitivity.

One of the main difficulties in creating a generic interface for robot skin sensors has been the identification of a generic representation of a geometric model of the surface on which the taxels are located. Without such a model, it is difficult to

reconstruct accurately and reliably the relative position and proximity of taxels across the surface of the robot. Although other information, such as the number of taxels, may also be useful, in practice, we have found that many high-level behaviors normally do not require such information and so we can achieve an acceptable performance level using only spatial taxel coordinates. An example of a generic high-level behavior based on two skins with different number of taxels and different taxel distributions are illustrated in § 4.1.

To make practical use of the taxel data, it is commonly necessary to relate this spatial information to the underlying kinematic chain, making it obvious where a taxel is located in relation to the robot's body, e.g., on the upper left arm. The generic robotic skin interface maps individual taxels to points in space relative to elements in the kinematic chain, abstracting away any underlying platform-specific taxel-indexing mechanisms as well as any related connectivity information. The abstract spatial taxel information removes any platform-specific representations and provides a skin representation that can support the implementation of cross-platform capabilities. An example of the forearm skin of the iCub robot represented using the TCI 3-dimensional spatial skin model is presented in **Figure 3**, where each small blue dot is a taxel on the skin. Our work leveraged the existing work of spatial calibration to generate the skin model, in the joint research (Prete et al., 2011; Denei et al., 2015).

In some cases, the three-dimensional position information of each taxel of the robot skin sensors is not available to support the calculations that are needed to present the taxels using the spatial coordinates required by the generic skin interface. In this case, it is time consuming and inaccurate to manually measure the precise 3D positions for all the taxels and instead they have to be approximated.

In our experience, for a taxel t, the displacement along the limb and the transverse angle of the taxel location relative to the limb orientation are typically more important than its distance from the central axis of the limb. As a consequence, we have approximated the spatial distribution of the taxels on the NAO robot skin using truncated cone. Given the estimated dimensions of the truncated cone, our 3D skin model can approximate the real taxel distribution. As a consequence, in order to support a TCI representation of the NAO skin sensors, the position P_t of a taxel t is modeled as a vector $<x_i, \theta, r_t>$ formalized in equation (3).

$$SKIN = \left\langle P_t|_{t=1}^{n} \right\rangle = \left\langle \langle x_t, \theta_t, r_t \rangle|_{t=1}^{n} \right\rangle \tag{3}$$

In equation (3), the value x_t represents the displacement of the taxel along the central axis of the element of the kinematic chain on which it is located. The value θ_t represents the displacement angle within the transverse plane relative to the orientation of the element, along which the taxel is located. The value r_t represents the distance (radius) from the central axis at which the taxel is located. With this taxel model, a generic robot skin sensor can be approximated and represented within the TCI framework. The truncated cone model is presented graphically in **Figure 4**.

Our truncated cone representation has the added benefit that the radius r_t need not be explicitly represented but can be calculated from the displacement x_t. As a consequence, we end up with the approximated skin representation $\overline{SKIN}$ given in equation (4). Admittedly, this model has its limitations on a robot with complex tactile surface where multiple curvatures are present. More investigations are needed as our future work.

$$\overline{SKIN} = \left\langle P_t|_{t=1}^{n} \right\rangle = \left\langle \langle x_t, \theta_t \rangle|_{t=1}^{n} \right\rangle \tag{4}$$

4. A TCI CASE STUDY: HUMANOID TACTILE WITHDRAWAL REFLEXES

In the previous sections we presented the TCI architecture and its two core interfaces, the *Generic Actuator Interface* and the *Generic Robot Skin Interface*. This section presents the application of the TCI framework to the design and the implementation of a cross-platform tactile withdrawal reflex for humanoid robots. The behavior was realized and demonstrated on two physical robots, the iCub and the NAO.

Robotic tactile withdrawal reflexes aim to improve the safety of human–robot interaction by reducing the potential for harm to humans and robots. Safe-path planning, padding, compliant limbs, and withdrawal reflexes all contribute at strategic point in

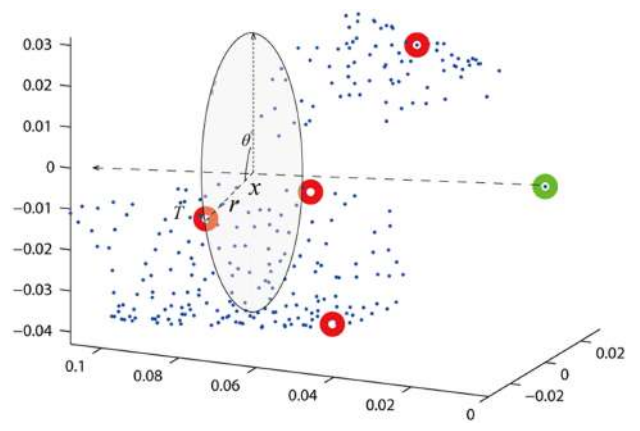


FIGURE 3 | An example of 3-dimensional generic robotic skin model: the forearm skin of iCub.

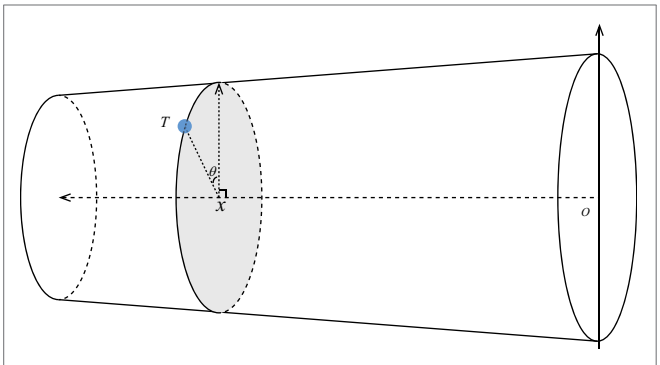


FIGURE 4 | The simplified model of generic robotic skin. The shape of the skin is arbitrarily approximated as a truncated cone.

time to improve safety and increase the scope for the application of human robots in unstructured human environments.

Based on a new robot skin sensor that covers large areas of a robot (Schmitz et al., 2011) and information about the mechanisms supporting human withdrawal reflexes, Dahl and Paraschos (2012) proposed a force–distance reflex model for humanoid robots. This model represents the reflex motion as a base motions moderated by two discount factors: the force of the impact and the distance between the stimulus and the center of the closest *reflex receptive field* (RRF). The RRFs will be discussed in details in §4.1. The base motions for the robot withdrawal reflexes were established through a set of experiments capturing reflex motions from humans using a motion capture suit (Dahl and Palmer, 2010). In our previous research, withdrawal reflex data was obtained using five stimulation locations on the upper and lower arm (four on the lower arm and one on the upper arm).

4.1. The Force–Distance Reflex Model

The force–distance (FD) reflex model is inspired by the concept of reflex receptive fields (RRFs), where each reflex has a trigger in the form of a continuous area on the surface of the skin, within which stimulation will provoke a reflex motion. The strongest response, i.e., the largest motion, is produced when the stimulus is in the center of the field. The strength of the response is gradually reduced as the distance between the stimulation point and the center of the field increases. The edge of the field acts as a threshold, beyond which no motion is triggered. In addition to being sensitive to the location of the stimulus, the size of the response under the FD model is also sensitive to the intensity of the stimulus, i.e., its force. The force–distance model is formalized in equation (5).

$$\theta_{i,j} = \begin{cases} \phi F_i \psi d_i \Theta_{i,j} & \text{if } d_i < r_i \text{ and } F_i > \delta_f \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

The FD reflex model is essentially a set of mappings from tactile stimulus to robotic reflex motions. Each mapping i defines the radius r_i of a circular receptive field centered at location C_i. The actual reflex motion $\theta_{i,j}$ is a vector of angle displacements, which is a moderation of a vector of base angle displacements $\Theta_{i,j}$. The base angle displacements were obtained by analyzing reflex motions captures from the human subjects. They are discounted by two factors ϕ and ψ, respectively, corresponding to the stimulus force F and the stimulus distance d to the center C_i. Simulation of a location, not covered by a receptive field, will not trigger a response.

Using the tactile capabilities interface, the FD reflex model can be made generic by representing the values involved using values and structures available in the generic robotic skin model discussed in §3.2 and the Generic Actuator Interface presented in § 3.1.

Reproducing the reflex motions $\theta_{i,j}$ using the tactile capabilities interface required us to calculate the stimulation distance d using the generic spatial taxel model. The FD reflex model, rewritten using this representation, is presented in equation (6). § 4.2 discusses, in detail, the implementation of the robotic reflexes under TCI.

$$\theta_{i,j} = \begin{cases} \phi \bar{F}_i \psi \left| \bar{P}_i - C_i \right| \Theta_{i,j} & \text{if } \left| \bar{P}_i - C_i \right| < r_i \text{ and } \bar{F}_i > \delta_f \\ 0 & \text{otherwise} \end{cases}$$

$$\left| \bar{P}_i - C_i \right| = \left| \langle x, r\sin\theta, r\cos\theta \rangle - \langle x_i, r_i \sin\theta_i, r_i \cos\theta_i \rangle \right| \tag{6}$$

In equation (6), $\bar{P}_i$ denotes the position of the center of the pressure and $\bar{F}_i$ is the average force of the triggered taxels C_i of the receptive field and the stimulus point $\bar{P}_i$. In practice, as taxels are close to each other, a stimulation typically triggers multiple taxels simultaneously. For a stimulation with n triggered taxels, F_t and P_t are the force and the position of a taxel t, respectively. $\bar{P}_i$ is the center of the stimulation weighted by the force of the triggered taxels. Similarly, $\bar{F}_i$ is the average taxel force weighted by the distance to $\bar{P}_i$. The position of the center of the pressure $\bar{P}_i$ and the average force $\bar{F}_i$ are formalized in equation (7).

$$\bar{P}_i = \frac{\sum_{t=1}^{n} F_t \cdot P_t}{\sum_{t=1}^{n} F_t}, \bar{F}_i = \frac{\sum_{t=1}^{n} F_t \cdot (2r_i - |P_t - \bar{P}_i|)}{\sum_{t=1}^{n} (2r_i - |P_t - \bar{P}_i|)} \tag{7}$$

In equation (7), $2r_i$ denotes the diameter of the receptive circle, which is the maximum geometrical distance for which a tactile stimulation can produce a response.

4.2. The Implementation of the Robotic Reflexes

This section presents the implementations of the force–distance reflex model on two real robot platforms, NAO and iCub using the tactile capabilities interface. The problem of using the same reflex module on two different robots to establish the same robotic withdrawal reflex motions is discussed in detail. A NAO robot is a 4.5-kg, 58-cm tall humanoid robot designed and manufactured by Aldebaran Robotics. It has 21 DoF (for the RoboCup edition) and is equipped with a range of sensors including two cameras, sonars, touch sensors, and accelerometers. An iCub is a 1-m high humanoid robot test-bed for research on cognitive robot behaviors. The robot is open source both in terms of the hardware design and the software resources. An iCub has 53 DoF and weighs around 22 kg. Both the NAO and iCub robots used for the work presented here were equipped with robot skin sensors developed under our previous joint research (Cannata et al., 2012). The robot reflexes were implemented and evaluated on both the simulated and the real robots. The FD model implementation on the iCub is illustrated in **Figure 5**. The blue dots are taxels of the forearm skin. When stimulations occur within receptive fields, i.e., the four circular red areas, the *Reflex* module produces the generic withdrawal actions. The actions are further translated to iCub specific commands by TCI. According to the FD model (equation (6)), iCub does not response to the stimulations located outside of the receptive fields. Similarly, the FD model on the real NAO is illustrated in **Figure 1B**.

Using the layered structure presented in **Figure 2**, the FD reflex model was implemented as a *Reflex* module located in layer L4. The actual robots APIs in layer L1 send raw tactile data to the YARP interface in layer L2. Within the TCI structures in layer L3, the *Skin Interface* module translates the raw skin data into the abstract spatial representation presented in §3.2.

The high-level *Reflex* module uses the abstracted skin data to calculate the moderated reflex motion $\theta_{i,j}$ using equation (6). Correspondingly, controlling the specific actuators is the reverse process. The reflex motion $\theta_{i,j}$ found by the layer L4 module is, sent down to the generic actuator interface located in layer L3 which again translates the generic motion to platform-specific commands. The YARP layer, L2, further translates the high-level commands to the actual actuators on the specific robot in the bottom layer, L1. **Figure 6** illustrates the robot reflexes produced by the same *Reflex* module being executed on simulated NAO and iCub robots through TCI. Two stimulations are simulated on the bottom and inside parts of the forearms of the two robots, respectively.

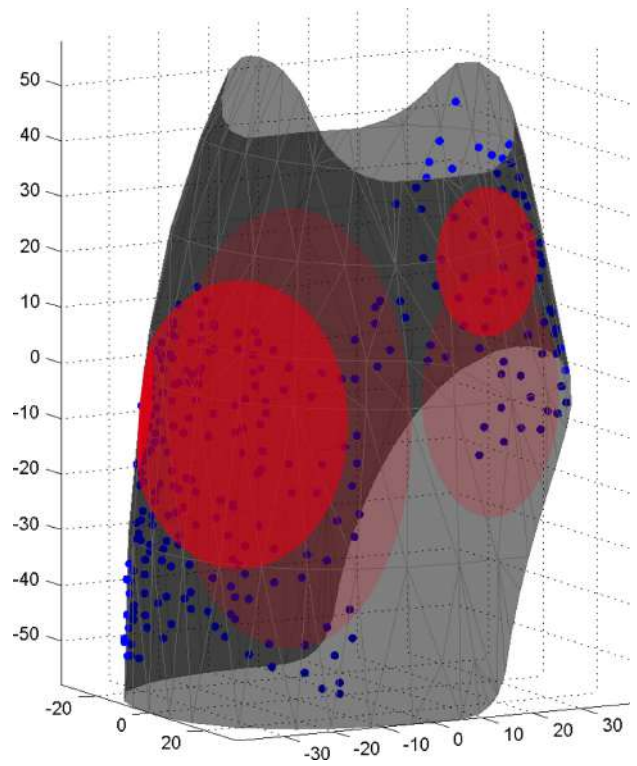


FIGURE 5 | An example of the 3D receptive fields of iCub forearm using generic robotic skin interface. The blue dots are the taxels, and the four circular areas are receptive fields.

5. CONCLUSION AND FUTURE WORK

This article presented a cross-platform tactile capabilities interface (TCI) for development of humanoid tactile capabilities. TCI promotes reuse of high-level modules by providing abstract hardware-independent representations of humanoid robot sensors and actuators. These representations allow control software to focus on the platform-independent elements of the control algorithms, delegating the translation of these abstracted representations to platform-specific commands, to the lower levels of the TCI infrastructure. The literature related to generic interfaces and current approaches to improve robot software reusability was reviewed in §2. As an important method to promote reusability, the state-of-the-art robotic middleware platforms are compared. In §3, a layered architecture for reusable generic robotic modulars was proposed, where TCI is used as an "interpreter" between high-level modules and YARP. TCI contains two core interfaces, a generic actuator interface (§3.1), which solves the configuration differences of the low-level actuators, and a generic robotic skin interface (§3.2) that abstracts the skin data. The article has presented the case study of generic humanoid tactile withdrawal reflexes. The force–distance (FD) reflex model is extended so as to be used under the TCI framework. The FD model has been implemented, and the same module was used to control two different real robots, the NAO and the iCub robots, equipped with different skin sensors. Experiments show that generic reflex motions have been successfully realized under TCI both in simulated environments and on real robots.

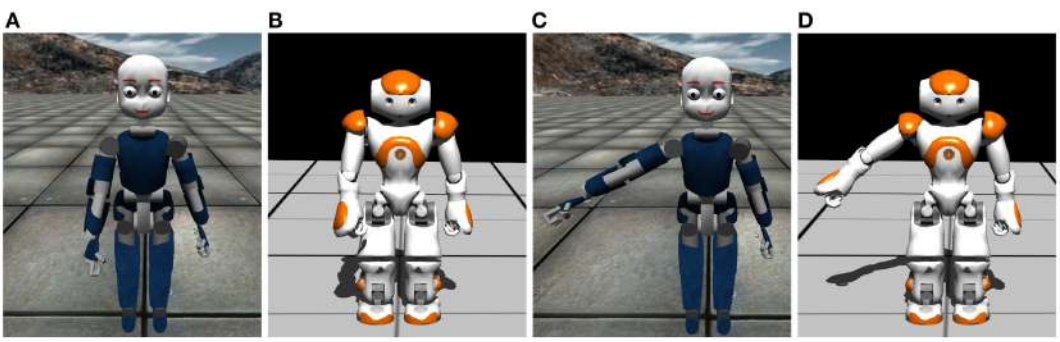


FIGURE 6 | The generic robotic reflexes produced and executed by the same *Reflex* module through TCI. Two stimulations are simulated on the NAO and iCub robots, respectively. Images **(A,B)** illustrate the reflexes from the stimulations located on the bottom of the forearms, while stimulations illustrated in images **(C,D)** are inside of the forearms.

The tactile capabilities interface is our first attempt at developing a cross-platform humanoid robot interface. Currently, only position-based actuator control is implemented and evaluated, although the framework is ready to integrate other types of actuator controls such as velocity control and torque control. Another limitation of our interface is that it does not provide an abstraction for tactile force. This is because in the study case, both the platforms used the same type of the tactile sensors and so the intensity representations are identical. Extra transformations will be needed if the comparison of the forces is required.

Our aim for the future work is to further investigate other control methods and other parts of common humanoid robots in the TCI framework. Also, the integration of forward and inverse kinematics is to be addressed. This feature will remove the requirement for platform-specific kinematics calculations if more sophisticated behaviors are needed. Recently, some research on defecting the direction of tactile force has also been proposed (Fumagalli et al., 2012; Stassi et al., 2014), and this could also become an extension to our current interface.

AUTHOR CONTRIBUTIONS

TD lead the research work presented, contributed to the development of the underlying ideas, wrote parts of the article, and reviewed it before submission.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Commission's Seventh Framework Programme (FP7/2007-2013) under grant agreement ROBOSKIN ICTFP7-231500. The authors would also like to thank Alexandros Paraschos for his contributions to the development of the NaoYarp software.

REFERENCES

Ambrose, R., Wilcox, B., Reed, B., Matthies, L., Lavery, D., and Korsmeyer, D. (2010). *Robotics, Tele-Robotics and Autonomous Systems Roadmap*. Technical Report. Washington, DC: National Aeronautics and Space Administration (NASA).

Anderson, M., and Thomaz, A. L. (2010). Enabling intelligence through middleware: report of the AAAI 2010 workshop. *AI Mag.* 32, 87–90. doi:10.1609/aimag.v32i1.2339

Atkeson, C. G. (1989). Learning arm kinematics and dynamics. *Annu. Rev. Neurosci.* 12, 157–183. doi:10.1146/annurev.ne.12.030189.001105

Bednarz, T., James, C., Caris, C., Haustein, K., Adcock, M., and Gunn, C. (2011). "Applications of networked virtual reality for tele-operation and tele-assistance systems in the mining industry," in *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry, VRCAI '11* (New York, NY: ACM), 459–462.

Brugali, D., and Scandurra, P. (2009). Component-based robotic engineering (part I). *Rob. Autom. Mag. IEEE* 16, 84–96. doi:10.1109/MRA.2009.934837

Bruyninckx, H. (2001). "Open robot control software: the OROCOS project," in *IEEE International Conference on Robotics and Automation (ICRA)*, Vol. 3 (Seoul: IEEE), 2523–2528.

Bruyninckx, H., Soetens, P., and Koninckx, B. (2003). "The real-time motion control core of the OROCOS project," in *IEEE International Conference on Robotics and Automation (ICRA)*, Vol. 2 (Taipei: IEEE), 2766–2771.

Caligiore, D., Guglielmelli, E., Borghi, A., Parisi, D., and Baldassarre, G. (2010). "A reinforcement learning model of reaching integrating kinematic and dynamic control in a simulated arm robot," in *IEEE 9th International Conference on Development and Learning (ICDL)* (Ann Arbor, MI: IEEE), 211–218.

Cannata, G., Mastrogiovanni, F., Metta, G., and Natale, L. (2012). "Advances in tactile sensing and touch based human-robot interaction," in *7th ACM/IEEE International Conference on Human-Robot Interaction (HRI)* (Boston, MA: IEEE), 489–490.

Crick, C., Jay, G., Osentoski, S., and Jenkins, O. C. (2012). "ROS and rosbridge: roboticists out of the loop," in *Proceedings of the Seventh Annual ACM/IEEE International Conference on Human-Robot Interaction, HRI '12* (New York, NY: ACM), 493–494.

Dahiya, R., Metta, G., Valle, M., and Sandini, G. (2010). Tactile sensing – from humans to humanoids. *IEEE Trans. Robot.* 26, 1–20. doi:10.1109/TRO.2009.2033627

Dahl, T., and Paraschos, A. (2012). "A force-distance model of humanoid arm withdrawal reflexes," in *Advances in Autonomous Robotics, Volume 7429 of Lecture Notes in Computer Science*, eds G. Herrmann, M. Studley, M. Pearson, A. Conn, C. Melhuish, M. Witkowski, J.-H. Kim, and P. Vadakkepat (Berlin, Heidelberg: Springer), 13–24.

Dahl, T. S., and Palmer, A. (2010). "Touch-triggered protective reflexes for safer robots," in *International Symposium on New Frontiers in Human-Robot Interaction (AISB'10)* (Leicester: AISB), 27–33.

Denei, S., Mastrogiovanni, F., and Cannata, G. (2015). Towards the creation of tactile maps for robots and their use in robot contact motion control. *Rob. Auton. Syst.* 63, 293–308. doi:10.1016/j.robot.2014.09.011

Diaz-Calderon, A., Nesnas, I. A. D., Nayar, H. D., and Kim, W. S. (2006). Towards a unified representation of mechanisms for robotic control software. *Int. J. Adv. Rob. Syst.* 3, 61–66. doi:10.5772/5757

Fumagalli, M., Ivaldi, S., Randazzo, M., Natale, L., Metta, G., Sandini, G., et al. (2012). Force feedback exploiting tactile and proximal force/torque sensing. *Auton. Robots* 33, 381–398. doi:10.1007/s10514-012-9291-2

Gamez, D., Fidjeland, A. K., and Lazdins, E. (2012). iSpike: a spiking neural interface for the iCub robot. *Bioinspir. Biomim.* 7, 025008. doi:10.1088/1748-3182/7/2/025008

Gerkey, B. P., Vaughan, R. T., and Howard, A. (2003). "The Player/Stage Project: tools for multi-robot and distributed sensor systems," in *The International Conference on Advanced Robotics* (Coimbra: IEEE), 317–323.

Huang, A., Olson, E., and Moore, D. (2010). "LCM: lightweight communications and marshalling," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Taipei: IEEE), 4057–4062.

Huang, Q., Yokoi, K., Kajita, S., Kaneko, K., Arai, H., Koyachi, N., et al. (2001). Planning walking patterns for a biped robot. *IEEE Trans. Rob. Autom.* 17, 280–289. doi:10.1109/70.938385

Ma, J. (2011). *Learning and Cooperation in Multi-agent Systems*. Ph.D. thesis, Department of Computer Science, University of Oxford, Oxford.

Ma, J., and Cameron, S. (2009a). "Learning robust and energy-efficient biped walking patterns using QWalking," in *The Twelfth International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines (CLAWAR 2009)*, eds O. Tosun, H. L. Akin, M. O. Tokhi, and G. S. Virk (Istanbul: World Scientific), 599–606.

Ma, J., and Cameron, S. (2009b). "Learning to walk: a model-free biped locomotion approach," in *Towards Autonomous Robotic Systems (TAROS)*, eds T. Kyriacou, U. Nehmzow, C. Melhuish, and M. Witkowski (Londonderry: University of Ulster), 229–235.

Ma, J., and Cameron, S. (2011). "Learning fast walking patterns for a NAO robot using Qwalking," in *Proceedings of the Fourteenth International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines (CLAWAR2011)* (Paris: UPMC University), 257–266.

Maes, P. (1991). "The agent network architecture," in *Special Interest Group on Artificial Intelligence (SIGART) Bulletin*, Vol. 2 (New York, NY: ACM), 115–120.

Martins, N., Bertol, D., Lombardi, W., Pieri, E., and Dias, M. (2008). "Neural control applied to the problem of trajectory tracking of mobile robots with uncertainties," in *10th Brazilian Symposium on Neural Networks, 2008 (SBRN'08)* (Salvador: IEEE), 117–122.

McMahan, W., Gewirtz, J., Standish, D., Martin, P., Kunkel, J., Lilavois, M., et al. (2011). Tool contact acceleration feedback for telerobotic surgery. *Haptics IEEE Trans.* 4, 210–220. doi:10.1109/TOH.2011.31

Merzouki, R., Fawaz, K., and Ould-Bouamama, B. (2010). Hybrid fault diagnosis for telerobotics system. *Mechatronics* 20, 729–738. doi:10.1016/j.mechatronics.2010.01.009

Metta, G., Fitzpatrick, P., and Natale, L. (2006). YARP: yet another robot platform. *Int. J. Adv. Rob. Syst.* 3, 43–48. doi:10.5772/5761

Namoshe, M., Tlale, N., Kumile, C., and Bright, G. (2008). "Open middleware for robotics," in *The International Conference on Mechatronics and Machine Vision in Practice (M2VIP)* (Auckland: IEEE), 189–194.

Nesnas, I. A., Simmons, R., Gaines, D., Kunz, C., Diaz-Calderon, A., Estlin, T., et al. (2006). CLARAty: challenges and steps toward reusable robotic software. *Int. J. Adv. Rob. Syst.* 3, 23–30. doi:10.5772/5766

Prete, A. D., Denei, S., Natale, L., Mastrogiovanni, F., Nori, F., Cannata, G., et al. (2011). "Skin spatial calibration using force/torque measurements," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on* (San Francisco, CA: IEEE), 3694–3700.

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., et al. (2009). "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software* (Kobe: IEEE), 5.

Schmitz, A., Maiolino, P., Maggiali, M., Natale, L., Cannata, G., and Metta, G. (2011). Methods and technologies for the implementation of large-scale robot tactile sensors. *IEEE Trans. Robot.* 27, 389–400. doi:10.1109/TRO.2011.2132930

Stassi, S., Cauda, V., Canavese, G., and Pirri, C. F. (2014). Flexible tactile sensing based on piezoresistive composites: a review. *Sensors* 14, 5296–5332. doi:10.3390/s140305296

Stone, P. (2000). *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. Cambridge, MA: MIT Press.

Stone, P., and McAllester, D. (2001). "An architecture for action selection in robotic soccer," in *The Fifth International Conference on Autonomous Agents*, eds E. Andre, S. Sen, C. Frasson, and J. P. Müller (New York, NY: ACM), 316–323.

Wooden, D., Malchano, M., Blankespoor, K., Howardy, A., Rizzi, A., and Raibert, M. (2010). "Autonomous navigation for BigDog," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on* (Anchorage: IEEE), 4736–4741.

Yoshida, E., Belousov, I., Esteves, C., and Laumond, J.-P. (2005). "Humanoid motion planning for dynamic tasks," in *The IEEE-RAS International Conference on Humanoid Robots* (Tsukuba: IEEE), 1–6.

Youssefi, S., Denei, S., Mastrogiovanni, F., and Cannata, G. (2014). "Skinware: a real-time middleware for acquisition of tactile data from large scale robotic skins," in *2014 IEEE International Conference on Robotics and Automation (ICRA)* (Hong Kong: IEEE), 6421–6426.

Youssefi, S., Denei, S., Mastrogiovanni, F., and Cannata, G. (2015a). A real-time data acquisition and processing framework for large-scale robot skin. *Rob. Auton. Syst.* 68, 86–103. doi:10.1016/j.robot.2015.01.009

Youssefi, S., Denei, S., Mastrogiovanni, F., and Cannata, G. (2015b). Skinware 2.0: a real-time middleware for robot skin. *SoftwareX* 3, 6–12. doi:10.1016/j.softx.2015.09.001

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

frontiers in Robotics and AI

ORIGINAL RESEARCH
published: 08 March 2016
doi: 10.3389/frobt.2016.00006

Unix Philosophy and the Real World: Control Software for Humanoid Robots

Neil T. Dantam[1], Kim Bøndergaard[2], Mattias A. Johansson[3], Tobias Furuholm[3] and Lydia E. Kavraki[1]**

[1] Department of Computer Science, Rice University, Houston, TX, USA, [2] Prevas A/S, Aarhus, Denmark, [3] Rocktec Division, Altas Copco Rock Drills AB, Örebro, Sweden

Robot software combines the challenges of general purpose and real-time software, requiring complex logic and bounded resource use. Physical safety, particularly for dynamic systems such as humanoid robots, depends on correct software. General purpose computation has converged on unix-like operating systems – standardized as POSIX, the Portable Operating System Interface – for devices from cellular phones to supercomputers. The modular, multi-process design typical of POSIX applications is effective for building complex and reliable software. Absent from POSIX, however, is an interproccess communication mechanism that prioritizes newer data as typically desired for control of physical systems. We address this need in the Ach communication library which provides suitable semantics and performance for real-time robot control. Although initially designed for humanoid robots, Ach has broader applicability to complex mechatronic devices – humanoid and otherwise – that require real-time coupling of sensors, control, planning, and actuation. The initial user space implementation of Ach was limited in the ability to receive data from multiple sources. We remove this limitation by implementing Ach as a Linux kernel module, enabling Ach's high performance and latest-message-favored semantics within conventional POSIX communication pipelines. We discuss how these POSIX interfaces and design principles apply to robot software, and we present a case study using the Ach kernel module for communication on the Baxter robot.

Keywords: real-time software, middleware, robot programing, humanoid robots, software engineering

OPEN ACCESS

Edited by:
Lorenzo Natale,
Istituto Italiano di Tecnologia, Italy

Reviewed by:
Ali Paikan,
Istituto Italiano di Tecnologia, Italy
Torbjorn Semb Dahl,
Plymouth University, UK

****Correspondence:***
Neil T. Dantam
ntd@rice.edu;
Lydia E. Kavraki
kavraki@rice.edu

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *13 October 2015*
Accepted: *12 February 2016*
Published: *08 March 2016*

Citation:
Dantam NT, Bøndergaard K, Johansson MA, Furuholm T and Kavraki LE (2016) Unix Philosophy and the Real World: Control Software for Humanoid Robots. Front. Robot. AI 3:6. doi: 10.3389/frobt.2016.00006

1. INTRODUCTION

Humanoid robot software presents a broad set of requirements. Humanoids have physical dynamics, requiring fast, real-time software. Humanoids have many sensors and actuators, requiring high performance network code. Humanoids, ideally, operate autonomously, requiring complex application logic. Satisfying these three requirements is a challenging software design and development problem. Fortunately, there are existing solutions to many of these challenges. Unix-like operating systems have served over many decades as the foundation for developing complex software. The standards and design principles learned developing these operating systems and applications provide many lessons and tools for humanoids and other robots.

Humanoid robotics would benefit by building on the significant design and engineering effort employed in unix development.[1] These operating systems are codified in POSIX, the IEEE standard for a Portable Operating System Interface (POSIX, 2008). POSIX provides a vendor neutral interface for systems programing, and there are numerous high-quality implementations that run on all major computer architectures. However, POSIX is more than just a standard for accessing filesystems and networks. Many POSIX applications follow a common design approach based on composing multiple, independent, modular processes. The multiprocess design promotes rapid development of robust and flexible software by isolating errors to a single process and enabling composition of existing tools to address new requirements (Raymond, 2003; Tanenbaum and Bos, 2014), a lesson already largely adopted by the robotics community with frameworks that often, though not always, compose applications from multiple processes (Bruyninckx et al., 2003; Brugali and Scandurra, 2009; Quigley et al., 2009). Unix-like systems are also pervasive in network-intensive applications, leading to powerful communication capabilities. Moreover, the widespread use and long history of unix has bred a variety of tools and conventions to aid system integration – a major challenge for robotics – by producing and configuring software that is flexible and portable. These features and corresponding design approaches present in Unix-like systems address many, though not all, the needs of humanoid robot software.

While Unix-like operating systems have been phenomenally successful for general purpose computing, they are less prevalent in real-time control of physical processes. Typically, a physical process such as a robot is viewed as a set of continuous, time-varying *signals*. To control this physical process with a digital computer, one must *sample* the signal at discrete time intervals and perform control calculations using the sampled value. To achieve high-performance control of a physical system, we must process the latest sample with minimum latency. This differs from the requirements of general computing which focus on throughput over latency and favor older data over newer data. While nearly all POSIX communication favors the older data, in robot control, the newest data are critical. However, some parts of the system, such as logging, may need to access older samples, so this also should be permitted at least on a best-effort basis. In this paper, we address the need for real-time communication within the larger context of POSIX programing. *We demonstrate a Linux kernel module for high performance, real-time communication, and discuss its use in the application of POSIX programing practice to humanoid robots.*

The Ach interprocess communication library provides fast communication that favors latest message data as typically desired for real-time control of physical systems. Ach is not a new framework that discards or duplicates the existing and significant tools for systems programing. Instead, Ach is a mechanism that integrates and builds upon the vast useful features of the POSIX and Linux ecosystem. In previous work, we presented an implementation of the Ach data structure in the POSIX user space (Dantam and Stilman, 2012; Dantam et al., 2015). User space Ach was limited in the ability to receive data from multiple sources. Now, we present an implementation of Ach as a Linux kernel module. Kernel space Ach enables applications to efficiently receive data from many sources, a crucial feature for mechatronic systems such as humanoid robots which contain many sensors, actuators, and software modules. The Ach Linux kernel module presents the conventional file descriptor interface used for communication in POSIX, enabling direct integration into existing communication systems and frameworks.

Though this work was initially developed for humanoid robots, it is broadly applicable to other complex mechantronic systems such as robot manipulators and intelligent vehicles. These evolving technologies all present similar requirements for complex software with real-time performance. The unix philosophy is effective for building complex software systems, and the Ach library grounds this approach to real-time, physical control.

2. LEARNING FROM UNIX

Humanoid robotics can learn from of the POSIX programing community. Important and challenging issues for humanoid robot software, such as high-performance communication, real-time memory allocation, and software integration, are largely addressed by existing techniques and standards. The humanoid robotics community would benefit by building on this work.

2.1. Communication and Scalability

Humanoid robotics can benefit from the strong communication capabilities of unix-like operating systems. Historically, Unix and the Internet developed in concert (Quarterman et al., 1985). POSIX provides a variety of communication and networking approaches, which largely address the performance and scalability needs of humanoid robot software. We summarize communication with many other nodes in Section 2.1.1 and service lookup in Section 2.1.2. Later, we address the unique needs of humanoid robots with the Ach library in Section 3, building on the capabilities offered by POSIX.

2.1.1. Multiplexing Approaches

Both general network servers and humanoid robots must communicate with a large number of other devices, be they network clients or hardware sensors and actuators. There are several techniques to communicate with multiple different nodes, each having trade-offs in implementation complexity and computational efficiency.

2.1.1.1. Fixed Interval Loop

A simple method to handle multiple connections is to service each connection at a fixed interval. The advantages of this approach are that it is simple to implement and it is similar to the fixed timestep commonly used in discrete-time control. However, there are

[1] Historically, the capitalized, trademarked "UNIX™" referred to operating systems based on the original code from AT&T Bell Laboratories, while the terms "unix-like," the lower-cased "unix," and the wildcards "un*x," "*nix," etc. conventionally refer to the larger family of similar, often independently developed operating systems (Raymond, 2008). Currently, UNIX™ is a trademark of The Open Group, which licenses the brand to certified, conforming operating systems (Gray v. Novell, 2011).

computational disadvantages. Messages may be delayed because the connections are only serviced once per step. Additional computation may be required to check connections that have no new messages. Furthermore, readers may block if attempting to service a connection with no data to read, and writers may block on full write buffers. While this approach can handle a small, fixed number of connections, it is not a practical consideration for network servers because it performs poorly with a large and varying number of connections.

2.1.1.2. Process-Per-Connection

One approach to handle varying numbers of connections is to create a separate worker process or thread for each connection. Creating worker processes was traditionally popular because it is easy to implement, process creation on unix-like systems is inexpensive, and separating connections in different processes provides isolation between them. The `inetd` superserver is based entirely on the approach of starting a handler process for each new connection. In addition, on modern multi-core machines, separate processes provide true concurrency. Separate handler processes also provide the unique feature of user-based access control; this is useful for low-volume and security-critical services such as SSH. The downside of using separate processes is the overhead to create and maintain the additional processes (Tanenbaum and Bos, 2014). Each connection requires memory for the process's function call stack, and context switching between processes introduces overhead. Consequently, this approach does not scale to very large numbers of connections.

2.1.1.3. Asynchronous I/O

Asynchronous I/O promises to allow applications to initiate operations, which are performed in the background with the application notified on completion. This would seem to address the scalability issues of the process-per-connection approach. However, current implementations of asynchronous I/O are not mature. The implementation on GNU/Linux uses threads to handle background I/O and scales poorly (Kerrisk, 2014).

2.1.1.4. Event-Driven I/O

Event-based methods allow efficient handling of many connections through a synchronous interface that notifies applications when a connection is ready for I/O. These methods use the traditional `select` call from System V UNIX and `poll` from BSD. The more recent `kqueue` call on FreeBSD and `epoll` on Linux reduce the overhead for very large numbers of connections. Though all these calls differ slightly in their semantics, the underlying premise is the same. The application provides the kernel with a list of file descriptors, and the kernel notifies the application when one of those descriptors is ready for a requested I/O operation. While this approach does require explicitly managing lists of active connections, it efficiently scales to large numbers of connections.

A rough benchmark for network servers is the ability to handle 10 thousand concurrent network connections (C10K) (Kegel, 2006). Though at one point this was a challenging problem, it is now easily handled through event-based methods such as `epoll` and `kqueue`. The popular and efficient Nginx[2] webserver uses event-based methods as does the libevent[3] library, which underlies communication in memcached and the Google Chrome web browser, among others. For handling many concurrent connections, event-based methods are widely used and scale on ordinary hardware to thousands of concurrent connections.

2.1.2. Name Resolution and Service Discovery

Another important issue in communication is name resolution and service discovery. Humanoid robots have many distinct software modules that need to locate the underlying mechanism for communication. Many middlewares provide their own form of service discovery: CORBA (CORBA, 2011) provides its naming service to locate remote objects, ONC RPC provides the port mapper (Srinivasan, 1995) to resolve the port numbers to connect to a desired program, and ROS resolves topic names in the `rosmaster` process (Quigley et al., 2009). However, name resolution and service discovery are addressed in a standard and general way via multicast DNS (mDNS) (Cheshire and Krochmal, 2013), a peer-to-peer variation of the traditional, hierarchical domain name system (DNS). DNS and mDNS are flexible protocols and can even store arbitrary information in TXT records (Rosenbaum, 1993). Of course, non-naming features such as connection monitoring are outside the scope of DNS. Multicast DNS is a standard protocol with existing implementations, so using mDNS instead of a specialized resolution method reduces the number of separate daemons which must run as well as separate code which must be maintained. Consequently, we use mDNS in Ach to locate communication channels on remote hosts.

2.1.3. Lesson Learned

Humanoid robots need communication that is both scalable and real-time. Event-based methods impose the lowest overhead of the various POSIX communication approaches and are the typical choice for scalability-critical network servers (Gammo et al., 2004). Communication for humanoid robots would benefit from the scalability of an event-based approach, and we discuss the real-time requirements next in Section 2.2. Event-based methods operate on kernel file descriptors (Stevens and Rago, 2013), which motivated the development of the Ach kernel module (see Section 3.2). To name and locate services, the standard mDNS protocol and implementations provide the necessary capabilities; there is no need to duplicate the features of mDNS.

2.2. Real-Time Software

Humanoid robot software requires not only the complex logic and efficient communication of general purpose software but also real-time response to handle physical dynamics. The software infrastructure for humanoids should address the need for real-time performance without unnecessarily sacrificing the capabilities of general purpose systems. While it is a challenge to develop real-time software on general purpose systems, acceptable performance can still be achieved.

[2] http://nginx.org/
[3] http://libevent.org/

2.2.1. Real-Time Communication

POSIX provides a variety of general purpose communication mechanisms; however, none are ideal for robot control. Robot control requires the latest data sample each control cycle. General purpose communication, however, gives priority to older data, which must be read or flushed before newer data can be received. This is the *Head of Line (HOL) Blocking* problem. The specific issues of each POSIX communication mechanism are discussed in Dantam et al. (2015). It was this HOL blocking challenge that motivated the initial development of Ach (Dantam and Stilman, 2012), which always provides access to the most recent data sample.

Though general network servers can handle thousands of concurrent connections (Gammo et al., 2004), there is a key difference from the needs of humanoid robots. Network servers are primarily concerned with maximizing throughput – serving as many clients as possible. Robot control, on the other hand, requires minimizing latency – handling each communication operation in minimal and bounded time. Throughput-focused methods often attempt to reduce copying, e.g., by eliding a copy to a kernel buffer for network socket communication or directly mapping a buffer into another process's addresses space via shared memory or relaying a file descriptor through a local socket. However for robots, individual real-time messages are typically small, e.g., a few floating point values read from a sensor, so the overhead of copying the data is minimal. Instead, overhead from system calls and process context-switching dominates. This shift in focus from throughput to latency is one aspect of the difference between general-purpose and real-time systems, and is a concern that we consider in the design of the Ach data structure (see Section 3.1).

Network communication uses Quality of Service (QoS) mechanisms to improve response for traffic with special requirements, for example, by reserving bandwidth or offering predictable delays (Huston, 2000). Linux provides *queuing disciplines* to prioritize sent traffic and reduce HOL blocking at the sending end (Siemon, 2013). However, HOL blocking or dropped packets may still occur at the receiving end if the receiver does not process messages as quickly as they are sent. The popular, real-time Controller Area Network (CAN) includes a dedicated priority field in messages to guarantee that higher priority messages are sent first, though messages of equal priority are still processed first-in-first-out, different senders must use unique message priorities to avoid collisions, and packet routing is not considered (ISO 11898-1:2015, 2015). Higher-level communication frameworks also employ QoS, the improve predictability of communication (DDS 1.2, 2007; Hammer and Bauml, 2013; Paikan et al., 2015). Appropriate use of QoS can improve real-time network performance, but the underlying queuing of network communication still presents challenges when one needs the most recent data sample.

The Ach library that we present in Section 3.1 is an interprocess communication mechanism rather than a network protocol, resulting in a distinct set of capabilities and challenges. Network communication must address issues such as limited bandwidth, packet loss, collisions, clock skew, and security. In contrast, processes on a single host can access a unified physical memory, which provides high bandwidth and assumed perfect reliability; still, care must be taken to ensure memory consistency between asynchronously executing processes. While network protocols use QoS to prioritize traffic, Ach maintains a specific data structure (see **Figure 1B**) to guarantee constant-time access to the most recent data sample. Furthermore, Ach communication is compatible with process priorities and priority inheritance, so higher priority processes gain first access to read from and write to an Ach channel. Overall, we view Ach is complementary to network communication. The low latency and fast latest-message-access of Ach make it well suited for real-time interprocess communication.

2.2.2. Real-Time Kernels

The trade-off between throughput and latency exists also at the level of the operating system kernel. General purpose kernels such as Linux and XNU (used in MacOSX) focus on maximizing throughput while real-time kernels focus on minimizing latency. QNX and VxWorks are POSIX kernels that focus on real-time performance, but both are proprietary. Open source kernels provide greater flexibility for the user, which is important for research where requirements are initially uncertain. There are two real-time variants of the open source Linux kernel. The Linux PREEMPT_RT patch (Dietrich, 2005) seamlessly runs Linux applications with significantly reduced latency compared to vanilla Linux, and work is ongoing to integrate it into the mainline kernel. However, it is far from providing formally guaranteed bounds on latency. Xenomai runs the real-time Adeos hypervisor alongside a standard Linux kernel (Gerum, 2004). It typically offers better latency than PREEMPT_RT but is less polished (Brown, 2010; Dantam et al., 2015) and its dual kernel approach complicates development. Because of the maturity, positive roadmap, and open source code base of Linux PREEMPT_RT (Fayyad-Kazan et al., 2013), we initially implement multiplexable Ach channels within this kernel.

2.2.3. Memory Allocation

Memory allocation is a particularly critical part of software development, even more so for real-time software. The ubiquitous `malloc` and `free` pose issues for real-time performance. Typical implementations are tuned for throughput over latency. The allocator in the GNU C Library (glibc) commonly used on Linux lazily batches manipulation of free lists. Calls to glibc's `free` are usually fast but sometimes take very long to complete (Lea, 2000). In contrast, the real-time focused Two-Level Segregate Fit (TLSF) allocator (Masmano et al., 2004) promises O(1) performance, though it is not tuned for multi-threaded applications. For garbage collected languages, latency is even more severe. Collection cycles introduce pause times that are unacceptable for real-time performance on humanoid robots (Johnson et al., 2015). Research on real-time garbage collection is ongoing (Yuasa, 1990; Bacon et al., 2003; Kalibera et al., 2011), while (Smith et al., 2014) use Java for real-time control by disabling garbage collection in the real-time module. If we can constrain the ordering of allocations and frees, then the situation improves. Region-based allocators impose a last-allocated, first-freed constraint, and operate in O(1) time with low overhead (Hanson, 1990). In addition, they provide the software-engineering advantage that all objects allocated from a

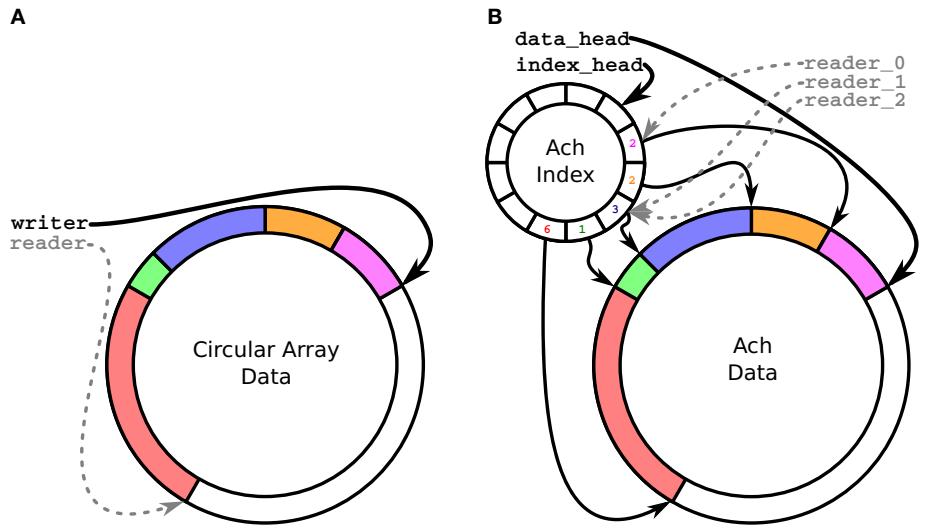


FIGURE 1 | **Logical memory structure for a conventional circular buffer (A) and an Ach channel (B)**. **(A)** shows a single writer, single reader circular buffer. **(B)** shows the Ach data structure with separate index and data buffers. The index buffer records metadata on the messages, enabling multiple readers, and detection of overwritten messages.

region can be freed with a single call, potentially reducing the bookkeeping necessary to avoid memory leaks. Though none of these memory allocation approaches are the universal solution for real-time constraints, each has advantages and is useful in the appropriate circumstances.

2.2.4. Lesson Learned

Wringing real-time performance out of a general purpose system is a careful balancing act. It requires understanding the overheads introduced by low-level calls and avoiding those with the potential to cause unacceptable resource usage. Selecting appropriate kernels and runtime support helps, but universal and guaranteed solutions are rare. A fundamental challenge is that general purpose computation considers time not in terms of correctness but only as a quality metric – faster is better – whereas real-time computation depends on timing for correctness (Lee, 2009). This is one area where continuing research is needed.

2.3. System Integration

System integration is a major challenge in robotics (Johnson et al., 2015; Zucker et al., 2015). While this is a broad issue covering algorithms, software, hardware, and operating environments, there are still lessons from the unix community that inform integration of humanoid robot software. In this regard, the humanoid robotics community mirrors the general open source world, depending on a wide variety of software packages from globally distributed authors and running on a wide variety of underlying platforms. We discuss design issues for software extensibility and compatibility in this section. In Appendix B, we discuss how build systems and package managers help integrate the many software packages required by humanoids. These general approaches for extensible design and software management are useful for similar software integration problems on humanoids.

2.3.1. Compatibility and Extensibility

Flexible and adaptable software is crucial to humanoid robots, where requirements and platforms are continually evolving. Tools and design principles from the POSIX programing community enable software that gracefully handles both the constant churn of ongoing development and the larger shifts of evolving platforms.

2.3.1.1. Mechanism vs. Policy

A key consideration in designing flexible software is the *Separation of Mechanism and Policy* (Silberschatz et al., 2009). Flexible software helps both in development by making it easier to prototype new systems and in long-term maintenance by making it easier to adapt to changing requirements. Software is more flexible when it provides mechanisms to perform some activity, but does not dictate overly restrictive policies over when or how to execute those activities. A canonical example of this approach is the X Window System (X11), which provides a mechanism for handling the display, but defers on policies for window management and "look-and-feel" (Scheifler, 2004). Compared to other, more integrated windowing systems, X11 has been extraordinarily long-lived, surviving various alternatives, e.g., Gosling et al. (1989), Linton and Price (1993), and Thomas et al. (2003), through changing graphics platforms. For research in particular, the separation of concerns is critically important to handling requirements that evolve as understanding of the project grows. We have followed this approach also in Ach by providing a communication mechanism but not dictating policies for message encoding or event handling. This separation of *policy* from *mechanism* is important for flexibility.

2.3.1.2. Binary Compatibility

When we modify a library on the robot, it is desirable to avoid the need to modify or recompile programs using that library. This

requires maintaining *binary compatibility*. Preserving binary compatibility requires the library to export a compatible Application Binary Interface (ABI). Maintaining ABI compatibility helps users by avoiding the requirement to install multiple library versions and by reducing the need to recompile applications. Drepper (2011) provides a detailed explanation of shared libraries and compatibility. In general, preserving binary compatibility requires that symbol names not be changed or removed and that client-visible structures preserve both their total size and the offsets of their fields. In C++, changes to the class hierarchy or virtual methods break binary compatibility. These requirements present challenges as new features are added to software. Adding new functions will not break compatibility; however, changes to structures may. There are some options to change structures while still preserving the ABI. One option, used in Ach, is to reserve space in structure declarations for fields to be added in the future. Reserving space maintains the total size of the structure when new fields are added. The cost is additional memory usage for the reserved space. Another option is to encapsulate all structure allocation and access within the library, exposing structures only as opaque pointers along with functions to access their fields. Encapsulating allocation permits changes to the underlying structure. The cost is the additional indirection and function call overhead to access the structure. When breaking the ABI is necessary, it is desirable to permit multiple ABI versions of the library to be installed together. Installing multiple ABI versions can be done by changing the library name, typically by including the version number in the library name; however, this may unnecessarily create different ABI versions when the new library version actually maintains binary compatibility. The alternative is to maintain a separate ABI version from the library version number. ABI versioning is handled differently on different operating systems; however, the Libtool component of Autotools provides a uniform interface for library versioning (GNU Libtool, 2015). While preserving ABI compatibility requires care and planning, it is generally possible and benefits library users.

2.3.1.3. Source Compatibility

If we cannot maintain binary compatibility when we modify a library, it is desirable to at least require only a recompilation of programs using the library rather than modifications to the programs' source code. This requires maintaining *source compatibility*. Preserving source compatibility requires a library to export a compatible Application Programing Interface (API). Maintaining API compatibility helps users by avoiding or reducing the need for them to modify their code to accommodate API changes. API compatibility is easier to maintain than ABI compatibility, generally requiring only that symbols not be removed or renamed and that argument lists remain the same. If such changes are necessary, there are some options to reduce the burden on users. One can give users time to change their code by first *deprecating* symbols before they are removed. For example, the `gets` function, vulnerable to buffer overflows, was deprecated in ISO/IEC 9899:1999 (1999) and removed in ISO/IEC 9899:2011 (2011). When structure fields must be renamed, one can preserve API compatibility by including both names within an anonymous union field. The old name can then be marked as deprecated. If it is possible that additional arguments may at some point be needed for a function, one can pass multiple arguments as fields within a structure or as items in a bitmasked integer. This allows additional arguments to be later included as fields in the structure or bits of the integer. This approach is used by the POSIX theads API (pthreads) in their various attribute structure arguments (POSIX, 2008). Several functions in Ach also take a similar attribute structure as an argument. Taking these precautions to preserve API compatibility eases the task of software maintenance for library users.

2.3.1.4. Language Selection

Programing language selection is an important, though contentious issue, and no language is universally ideal for the diverse needs of humanoid robots. Developing complex applications is easier in high-level, garbage-collected languages, while strict real-time requirements preclude garbage-collection, leaving lower-level languages such as C and C++. Though C++ has many features over C that are sometimes useful, it comes at a cost which should be considered. C is often preferred by performance-sensitive projects, e.g., the Linux kernel, because it is easier for the programmer to understand and control important, low-level details such as error handling and memory allocations which C++ abstracts through exceptions and constructors. C++ also presents compatibility issues. Because C identifiers map directly to assembly language symbols, it is generally possible to link C code built with different compilers. C++, on the other hand, uses implementation-specific name mangling on identifiers, e.g., to handle overloaded functions, so linking C++ code built with different compilers may not be possible. Changes to operating systems ABIs for C++, though still infrequent, occur more often than for C. When performance requirements permit high-level, garbage-collected languages, binding low-level libraries written in C is generally easier than for C++. C is universally supported among high-level languages for foreign function bindings, e.g., JNI for Java, CFFI for Lisp, and ctypes for Python, whereas the ability to directly interact with C++ classes is less common. Because Ach is performance sensitive and real-time, it is implemented in C. To interface with high-level, non-real-time modules, Ach provides foreign function bindings for Common Lisp, Python, and Java. Given the trade-offs among programing languages, one should be judicious in selecting languages for implementations and interfaces.

2.3.2. Lesson Learned

The unix programing tradition provides many tools and conventions to assist with system integration of humanoid robot software. Following established conventions to preserve ABI and API compatibility makes software easier to use by reducing the system administration and software maintenance task for users. Appropriate languages ease software development and maintenance while still providing acceptable performance. Though this is far from covering the full range of system integration issues for humanoid robots, it goes a long way toward addressing software-specific system integration.

3. EXTENDING LINUX COMMUNICATION

POSIX provides a rich variety of communication methods that are well suited for general purpose information processing, but

none are ideal for real-time robot control. General computation favors throughput over latency. POSIX communication favors older data over newer. In contrast, real-time control requires low latency access to the newest data. Dantam et al. (2015) discusses the challenges of POSIX communication in detail. This gap has made it difficult to develop real-time applications in the multi-process POSIX style. To address this communication need, we developed the Ach library.

3.1. The Ach IPC Library

Ach provides a message bus or publish-subscribe style of communication between multiple writers and multiple readers (Dantam et al., 2015). Robots using Ach have multiple channels across which individual data samples are published. Messages are sent as byte arrays, so arbitrary data may be transmitted such as floating point vectors, text, images, and binary control messages. The primary unique feature of Ach is that newer messages always supersede older messages whereas POSIX communication gives priority to older data and will block or drop newer messages when buffers are full. Ach's latest-message semantics are appropriate for continuous, time-varying signals such as reference velocities or position measurements. In other cases where reliable messaging is required, such as updating a PID gain value, Ach may provide sufficient reliability by using a separate channel with a large buffer; however, this is a secondary consideration. Ach's primary focus on latest-information, publish-subscribe messaging give it unique capabilities for real-time communication of physical data samples.

3.1.1. Relation to Robotics Middleware

There are many other communication systems developed for robotics; however, Ach provides a unique set of features and capabilities. First, many other systems operate as frameworks (Bruyninckx et al., 2003; Metta et al., 2006; Quigley et al., 2009) that impose specific structure on the application. Sometimes this structure is helpful when it fits the desired application, and other times such imposed structure may impede development if the requirements are outside the particular framework's model. In contrast, Ach strictly adheres to the idea of *mechanism, not policy* (see Section 2.3.1.1), providing a flexible communication method that is easily integrated with other approaches (see Appendix A). One could also view Ach as providing low-level capabilities that could serve as a useful building-block for such higher-level frameworks. Second, many other systems focus on network communication (Metta et al., 2006; Quigley et al., 2009; Huang et al., 2010; Hammer and Bauml, 2013). In contrast, Ach focuses on local, interprocess communication. This focus enables it to achieve superior performance in its domain (Dantam et al., 2015), and we view Ach as complementary to various network protocols. Finally, Ach provides unique semantics that make it especially suited to real-time communication of continuously varying data. Similar to multicast methods (Huang et al., 2010), Ach efficiently supports multiple senders and receivers. Ach implicitly supports *process priorities* whereas network-based methods use *QoS* (DDS 1.2, 2007; Hammer and Bauml, 2013). Crucially, Ach eliminates any possibility HOL blocking (see Section 2.2.1). Network-based methods can handle HOL blocking at the sending and receiving ends (Metta et al., 2006), but dealing with assumptions in intermediate infrastructure and code is a difficult challenge (Gettys and Nichols, 2012). Overall, the unique design decisions underlying Ach result in special advantages for local, real-time communication, and we consider Ach as a key component within a larger robot software system.

3.1.2. Design of Ach

The data structure for each channel, shown in **Figure 1B**, is a pair of circular buffers, (1) a data buffer with variable sized entries and (2) an index buffer with fixed-size elements indicating the offsets into the data buffer. Ach provides additional capabilities compared to a typical circular buffers, such as in **Figure 1A**:

- Ach allows multiple receivers;
- Ach always allows access to the newest data;
- Ach drops the oldest data – instead of the newest data – when the buffer is full.

Two procedures compose the core of ach: `ach_put` and `ach_get`. Detailed pseudocode is provided in Dantam and Stilman (2012), and their use is discussed in the Ach manual (Dantam, 2015b) and programing reference (Dantam, 2015a).

The procedure `ach_put` inserts new messages into the channel. It is analogous to the POSIX `write`, `sendmsg`, and `mq_send` functions. The procedure is given a pointer to the shared memory region for the channel and a byte array containing the message to post.

Algorithm 1 (ach_put). There are four broad steps to the `ach_put` procedure:

1. Get an index entry. If there is at least one free index entry, use it. Otherwise, clear the oldest index entry and its corresponding message in the data array.
2. Make room in the data array. If there is enough room already, continue. Otherwise, repeatedly free the oldest message until there is enough room.
3. Copy the message into data array.
4. Update the offset and free counts in the channel structure.

The procedure `ach_get` receives a message from the channel. It is analogous to `read`, `recvmsg`, and `mq_receive`. The procedure takes a pointer to the shared memory region, a storage buffer to copy the message to, the last message sequence number received, the next index offset to check for a message, and option flags indicating whether to block waiting for a new message and whether to return the newest message bypassing any older unseen messages.

Algorithm 2 (ach_get). There are four broad steps to the `ach_get` procedure:

1. If given the option argument to wait for a new message and there is no new message, then wait. Otherwise, if there are no new messages, return a status code indicating this fact.
2. Find the index entry to use. If given the option argument to return the newest message, use the newest entry. Otherwise, if the next entry we expected to use contains the next sequence

number, we expect to see, use that entry. Otherwise, use the oldest entry.
3. According to the offset and size from the selected index entry, copy the message from the data array into the provided storage buffer.
4. Update the sequence number count and next index entry offset for this receiver.

Ach provides unique semantics compared to traditional POSIX communication. Processes on a single host can access a unified physical memory, which provides high bandwidth and assumed perfect reliability; still, care must be taken to ensure memory consistency between asynchronously executing processes. In contrast, real-time communication across a network need not worry about memory consistency, but must address issues such as limited bandwidth, packet loss, collisions, clock skew, and security.

3.1.3. User Space Limitations

The initial implementation of Ach located the data structure shown in **Figure 1B** in POSIX shared memory and synchronized access using a mutex and a condition variable. This presented a few potential error modes and limitations: a rogue or faulty process could deadlock or corrupt a channel and each thread was limited to waiting for data on single channel at a time. We discuss these potential issues next and resolve them with the kernel space implementation described in Section 3.2.

While the formal verification of Ach (Dantam et al., 2015) guarantees that it will not deadlock with regular use of the library calls, deadlock may still occur if a reader or writer dies, e.g., with a `kill -9`, inside a library call. This is partially mitigated by the use of robust POSIX mutexes, which detect this condition and handle interrupted reads. Additional code could be added, which would rollback an interrupted write.

Because all processes accessing the channel must have read and write access to the shared memory region, a rogue process could corrupt the channel data structures. Currently, unintentional corruption is weakly detected with guard bytes. This could be improved with better sanity checks of the channel and automatic recreation of corrupted channels.

The use of POSIX threads synchronization primitives limits each thread to wait for new messages on a single channel at a time. Readers wait for new messages on a per-channel POSIX condition variable and are notified by the writer when a new message is posted. POSIX threads are limited to waiting on only a single condition variable at a time; thus, there is no way in this implementation for a thread to simultaneously wait for data on multiple channels. Alternative file-based notification, e.g., using pipes or sockets, would allow multiplexing but may cause extra context-switching and additional logic would be required to ensure that tasks run in priority order. This semantic limitation was the primary motivation for the development of the kernel space Ach implementation.

3.2. Kernel Space Ach

To address the limitations of user space Ach presented in Section 3.1.3, we develop a new kernel space implementation of the Ach data structure and procedures. This implementation runs in the Linux kernel. The channel buffers shown in **Figure 1B** are located in kernel memory, protecting channels from corruption and deadlock (see **Figure 2**). Critically, channels are accessed from user space via file descriptors, enabling efficient multiplexing through event-based `poll`/`select` style calls. This enables efficient real-time communication using established network programing idioms.

3.2.1. Kernel Module Implementation

Ach is implemented in kernel space as a Linux module that creates a device file for each channel. When the module is loaded, it creates the `/dev/achctrl` device to manage channel creation and removal. Each channel is represented with a separate virtual device, e.g., `/dev/ach-foo` for channel `foo`. These virtual devices are not accessed directly by applications but instead are accessed through the Ach library using the same API as user space channels. This provides backward source-compatibility with the user space implementation and allows applications to freely switch between user and kernel space channels. The library functions to create channels (`ach_create`) and remove channels (`ach_unlink`) operate on kernel channels through `ioctl` system calls on the `/dev/achctrl` device. The library functions to send (`ach_put`) and receive (`ach_get`) messages map to `write` and `read`, respectively. Additional parameters for receiving messages, such as timeouts or flags to retrieve the newest message, are passed to the kernel via `ioctl`s. Event-based multiplexing of ach channels – alongside sockets, pipes, and other file descriptors – is possible by passing the file descriptor for the channel device file to `poll`, `select`, etc; the kernel module performs the appropriate notification when a new message is posted to the channel. By providing this kernel-supported, file-descriptor-based interface to ach channels, we improve the ability to handle multiple data sources and to interoperate with other communication mechanisms using the standard POSIX event-based I/O functions.

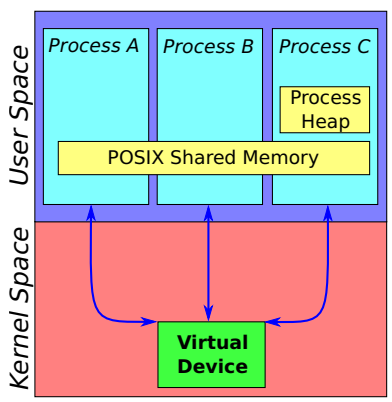


FIGURE 2 | Backing memory locations for Ach channels. Channels in the process heap are private to the process. Channels in POSIX shared memory are accessible by multiple processes. The kernel channels presented in this paper are available through per-channel virtual devices and are also accessible by multiple processes. The virtual devices are accessed from user space through file descriptors, meaning the kernel channels can be multiplexed via `select`, `poll`, etc. Because kernel channels are in protected kernel memory, they cannot be corrupted by rogue processes.

Multiplexing of Ach kernel channels is possible with the following steps:

Algorithm 3 (Ach Multiplexing).

1. Open all channels with `ach_open`.
2. Obtain the channels' file descriptors by calling `ach_channel_fd`, and record the file descriptors in the `struct pollfd`.
3. Call `poll` to wait for new data on any channel.
4. Call `ach_get` on channels with new data.

Appendix A provides a complete example of multiplexing Ach channels alongside conventional POSIX streams.

3.2.2. Advantages of Kernel Space Ach

The in-kernel Ach implementation removes the limitations of the user space implementation discussed in Section 3.1.3. Primarily, it permits multiplexing of multiple channels alongside other POSIX communication mechanisms using standard and efficient event-based I/O, e.g., `select` and `poll`. For humanoid robots, where a process may need to receive data from a large number of sources, this ability to conduct efficient I/O is a critical advantage. In addition, the potential of user space channels for corruption and deadlock from rogue processes is eliminated in the kernel implementation. Kernel channels are in kernel memory which cannot be directly accessed by user space processes. The kernel implementation eliminates the faults of user space Ach, giving it the same features and robustness as standard POSIX communication mechanisms.

3.2.3. Disadvantages of Kernel Space Ach

The in-kernel implementation does present some potential disadvantages for portability and a caveat regarding robustness.

While the user space implementation used standard POSIX calls, the in-kernel implementation is Linux-specific, running on vanilla and PREEMPT_RT kernels. This would present an issue if it is necessary to use a non-Linux kernel, requiring additional work to implement Ach within that separate kernel. However, the Ach code is modular and well-factored – the core code is largely shared between the user and Linux kernel space implementations. Adding an additional backend for another kernel should not be a major challenge, and we hope to develop a kernel module for Xenomai in the future.

Code in kernel space faces stricter correctness requirements than in user space. Software errors in the Ach kernel module – as with any kernel space code – can potentially crash the entire operating system. However, for humanoid robots, any software error, whether in user or kernel space, can potentially – and literally – crash the entire robot. Thus, moving Ach to the kernel does not significantly change the severity of potential errors. Still, it is important to understand the strict requirements on kernel space code.

3.3. Benchmarks

We provide benchmark results of message latency for Ach compared to a variety of other kernel communication methods.[4] Latency is often more critical than bandwidth for real-time control as the amount of data per sample is generally small, e.g., state and reference values for several joint axes. Consequently, the actual time to copy the data is negligible compared to other sources of overhead such as process scheduling. The benchmark application performs the following steps:

1. Initialize communication structures;
2. `fork` sending and receiving processes;
3. Sender: Post timestamped messages at the desired frequency;
4. Receivers: Receive messages and record latency of each messaged based on the timestamp.

Figure 3 shows the results of the benchmarks, run on an Intel® Core™ i7-4790 at 3.6 GHz under Linux 3.18.16-rt13 PREEMPT RT. We compare Ach with several common POSIX communication mechanisms. In contrast to Ach and these lightweight, kernel methods, heavyweight middleware such as ROS and CORBA impose several times greater communication latency (Dantam et al., 2015). All the methods shown in **Figure 3** are similar in performance, indicating that the bulk of overhead is due to the process context switch rather than the minimal time for the actual communication operation. For the single receiver case, both user and kernel space Ach provide comparable latency to POSIX communication. While the latency is similar, there are also important feature differences. Kernel space Ach can multiplex across multiple channels while user space Ach cannot, and unlike POSIX communication, Ach directly supports multiple subscribers. The results in **Figure 3** show that Ach provides strong performance, along with its ability to handle multiple subscribers and its unique latest-message-favored semantics.

3.4. Case Study: Baxter Robot

We use the new kernel space Ach implementation in our control system for the Baxter robot. The Baxter is a dual-arm manipulator. Each arm has 7 degrees of freedom and a parallel jaw gripper. The integrated electronics enable position, velocity, and torque control of the robot's axes. We implement several modes of multi-axis control using this interface.

Figure 4 shows our Ach-based control system for the Baxter robot. In this system, each driver and controller runs as a separate, isolated operating system process. Previous ach-based systems ran drivers as separate processes, but multiple controllers were combined into a single processes (Dantam et al., 2015). Because kernel space Ach channels are efficiently multiplexable, we can run the Baxter controllers in separate processes, each outputing to a distinct channel. The `ref` process receives messages from all these channels, selecting the highest priority message to communicate to the robot. This design efficiently integrates multiple control modes for the robot, each running in separate processes.

In addition to the processes shown in **Figure 4**, we also run a separate, non-real-time logging process. All the control system processes write log messages to a single event channel. In normal operation, the logger waits for new messages on the event channel, stepping through each posted message and recording it to a log file. However, if log messages are posted faster than the logger can process them, for example whether due limited CPU cycles

[4] Benchmark code available at http://github.com/golems/ach

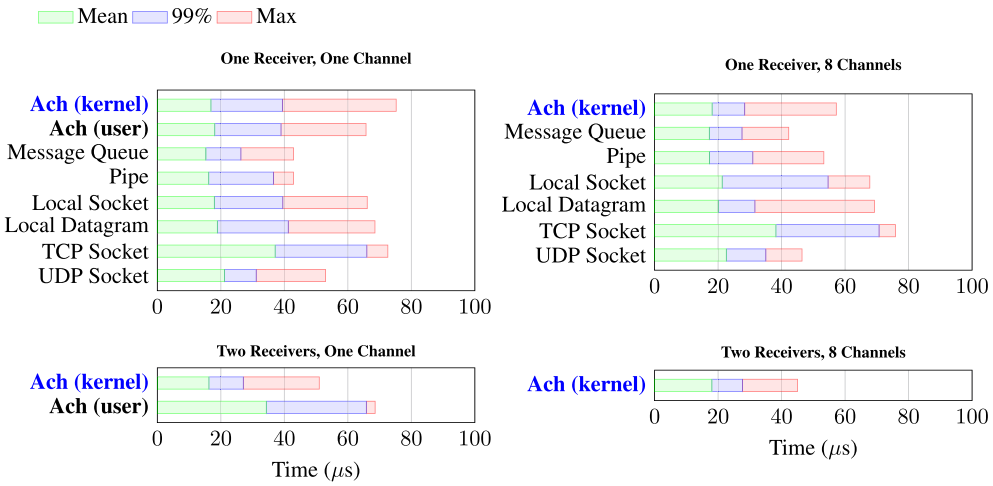


FIGURE 3 | Message Latency for Ach and POSIX communication. Ach has comparable performance to optimized POSIX communication, and unlike POSIX methods, Ach enables multiple receivers and prioritizes newer data. In the plots, "Mean" is the average latency over all messages, "99%" is the latency that 99% of messages beat, and "Max" is the maximum recorded latency. The long-tail of the worst-case behavior arises from the focus of general purpose platforms on average-case performance instead of bounded response needed by real-time systems. For example, lazy algorithms and hardware memory caches improve average performance, but still leave the worst-case cost to pay.

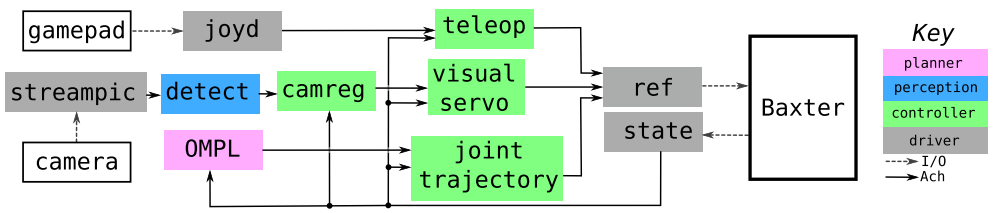


FIGURE 4 | Ach-based control system for the Baxter robot. Three control pipelines run concurrently: a teleoperation pipeline for manual control (top), a camera registration and visual servoing pipeline (middle, Dantam et al., 2016), and a joint planning (Sucan et al., 2012) and interpolation pipeline (bottom). The `ref` process multiplexes the output of these three pipelines and selects the highest priority command to send to the robot. A non-real-time logging process (not shown) also receives informational and error messages from the other control system processes over a single Ach channel and records the messages in a log file.

or a programing error, some messages will be skipped as they are overwritten in the circular buffer (see **Figure 1B**). The alternative to skipping messages would be to block the message sender until the logger can process messages or to buffer an unbounded number of unprocessed log messages. Neither blocking a real-time process nor growing a buffer without bound is desirable in a real-time control system. Instead, Ach overwrites the oldest message and the system continues, missing only the skipped log messages.

The primary advantages of this multi-process control system design are modularity and robustness to software errors. Separating drivers and controllers into different processes means they can be developed and tested independently. This separation is particularly useful to prototype new controllers, which can be developed without disturbing previously tested work. For example, we developed and tested the workspace controller described in Kingston et al. (2015) without any modification to the processes in **Figure 4**. Failures encountered while developing and testing the new controller did not affect the other running control processes. In contrast, combining multiple controllers in a single process as was necessary for the user space Ach systems presented in Dantam et al. (2015) means that errors encountered while prototyping a new controller will not interfere with existing control modes. In research applications on robot control, easing controller development and testing is a key advantage.

4. CONCLUSION

We have discussed the application of unix design principles to robot software. Among the various unix tools and conventions, the multiprocess design typical of unix applications improves modularity and robustness, critical needs for complex systems such as humanoid robots. We enable this multiprocess design for real-time control with the efficient Ach communication library. This approach is general, applying to multiple types of robots and other complex mechatronic systems that require coordination of many hardware devices and software modules. Ach fills a need in robot software unmet by POSIX, providing a communication

mechanism that supports multiple receivers and gives priority to newer messages. The kernel space implementation presented in this paper exposes a file descriptor interface, enabling multiplexing of messages from many sources and efficient integration with other communication methods. Kernel space Ach enables development of real-time robot software in the conventional modular, robust, multi-process unix style.

The approach to robot software development we have presented – advocating use of existing tools and conventions from the unix programing community – is inherently conservative. While building on unix provides a mature set of capabilities, new research in techniques to automate software development have the potential to radically improve the development process.

Formal methods for software verification and synthesis can greatly ease software development. Some tools are already in widespread use (Cimatti et al., 2002; Ball et al., 2004), and we used SPIN (Holzmann, 2004) to verify Ach in Dantam et al. (2015). Formal methods continue to be an active research area in robotics (Wang et al., 2009; Dantam and Stilman, 2013; Liu et al., 2013; Nedunuri et al., 2014; Lignos et al., 2015), bridging the fields of software engineering, automatic control, and planning and scheduling. Though limits remain on which problems admit formal reasoning, further research has the potential to broaden the scope of formal methods for humanoid robot software, changing our fundamental approach to software development.

AUTHOR CONTRIBUTIONS

ND, primary author of overall software, secondary author of additional modules described in paper. KB, primary author of additional modules described in paper. MJ, additional contributions to software. TB, supported work at Atlas Copco and Prevas, contributions to design and testing methods. LK, supported work at Rice University, contributions to testing and presentation of the work.

ACKNOWLEDGMENTS

Work at Rice University by ND and LK has been supported in part by NSF IIS 1317849, NSF CCF 1514372, and Rice University Funds. Work by KB, MJ, and TF has been supported by Atlas Copco Rock Drills AB. We thank Zachary K. Kingston for his development work and continuing maintenance of the presented Baxter software example.

REFERENCES

Bacon, D. F., Cheng, P., and Rajan, V. (2003). "A real-time garbage collector with low overhead and consistent utilization," in *Symposium on Principles of Programming Languages*, Vol. 38 (New Orleans, LA: ACM), 285–298.

Ball, T., Cook, B., Levin, V., and Rajamani, S. K. (2004). "SLAM and static driver verifier: technology transfer of formal methods inside Microsoft," in *Integrated Formal Methods, Volume 2999 of Lecture Notes in Computer Science* (Canterbury: Springer), 1–20.

Brown, J. H., and Martin, B. (2010). *How Fast Is Fast Enough? Choosing between Xenomai and Linux for Real-time Applications*. Technical Report, Rep Invariant Systems. Available at: https://www.osadl.org/fileadmin/dam/rtlws/12/Brown.pdf

Brugali, D., and Scandurra, P. (2009). Component-based robotic engineering (part I)[tutorial]. *IEEE Rob. Autom. Mag.* 16, 84–96. doi:10.1109/MRA.2009.934837

Bruyninckx, H., Soetens, P., and Koninckx, B. (2003). "The real-time motion control core of the Orocos project," in *International Conference on Robotics and Automation*, Vol. 2 (Taipei: IEEE), 2766–2771.

Catkin. (2015). *Catkin Conceptual Overview*. Available at: http://wiki.ros.org/catkin/conceptual_overview

Cheshire, S., and Krochmal, M. (2013). *Multicast DNS*. Internet Engineering Task Force. Available at: https://tools.ietf.org/html/rfc6762

Cimatti, A., Clarke, E. M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., et al. (2002). "Nusmv 2: an opensource tool for symbolic model checking," in *Computer Aided Verification, CAV '02* (London: Springer-Verlag), 359–364.

CORBA. (2011). *Common Object Request Broker Architecture (CORBA/IIOP)*, 3.1.1 Edn. The Object Management Group. Available at: http://www.omg.org/spec/CORBA/3.1.1/

Dantam, N. T. (2015a). *Ach IPC Library*. Available at: http://golems.github.io/ach/api/

Dantam, N. T. (2015b). *Ach IPC User Manual*. Available at: http://golems.github.io/ach/manual/

Dantam, N. T., Amor, H. B., Christensen, H., and Stilman, M. (2016). "Online camera registration for robot manipulation," in *Experimental Robotics*. eds M. Ani Hsieh, O. Khatib, and V. Kumar (Switzerland: Springer), 179–194.

Dantam, N. T., Lofaro, D., Hereid, A., Oh, P., Ames, A., and Stilman, M. (2015). The Ach IPC library. *Rob. Autom. Mag.* 22, 76–85. doi:10.1109/MRA.2014.2356937

Dantam, N. T., and Stilman, M. (2012). "Robust and efficient communication for real-time multi-process robot software," in *International Conference on Humanoid Robots* (Osaka: IEEE), 316–322.

Dantam, N. T., and Stilman, M. (2013). The motion grammar: analysis of a linguistic method for robot control. *Trans. Rob.* 29, 704–718. doi:10.1109/TRO.2013.2239553

DDS 1.2. (2007). *Data Distribution Service for Real-time Systems*, 1.2 Edn. The Object Management Group. Available at: http://www.omg.org/spec/DDS/1.2/

Dietrich, S.-T., and Walker, D. (2005). The evolution of real-time Linux. In *7th Real Time Linux Workshop*.

Drepper, U. (2011). *How to Write Shared Libraries*. Technical Report, Red Hat. Available at: http://www.akkadia.org/drepper/dsohowto.pdf

Fayyad-Kazan, H., Perneel, L., and Timmerman, M. (2013). Linux PREEMPT-RT vs. commercial RTOSs: how big is the performance gap? *J. Comput.* 3, 135–142. doi:10.5176/2251-3043_3.1.244

Gammo, L., Brecht, T., Shukla, A., and Pariag, D. (2004). "Comparing and evaluating epoll, select, and poll event mechanisms," in *Proceedings of the 6th Annual Ottawa Linux Symposium*. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.215.7953

Gerum, P. (2004). *Xenomai – Implementing a RTOS Emulation Framework on Gnu/Linux*. Technical Report, Xenomai.

Gettys, J., and Nichols, K. (2012). Bufferbloat: dark buffers in the internet. *Commun. ACM* 55, 57–65. doi:10.1145/2063176.2063196

GNU Libtool. (2015). *GNU Libtool – Portable Dynamic Shared Object Management*. Free Software Foundation. Available at: https://www.gnu.org/software/libtool/manual/

GNU Standards. (2015). *GNU Coding Standards*. Free Software Foundation. Available at: https://www.gnu.org/prep/standards/

Gosling, J., Rosenthal, D. S., and Arden, M. J. (1989). *The NeWS Book: An Introduction to the Network/Extensible Window System*. New York, NY: Springer Science & Business Media.

Gray v. Novell, Inc. (2011). *United States Court of Appeals, Eleventh Circuit, NO. 09-11374*. Florida.

Hammer, T., and Bauml, B. (2013). "The highly performant and realtime deterministic communication layer of the ardx software framework," in *2013 16th International Conference on Advanced Robotics (ICAR)* (Montevideo: IEEE), 1–8.

Hanson, D. R. (1990). Fast allocation and deallocation of memory based on object lifetimes. *Software Pract. Exp.* 20, 5–12.

Holzmann, G. (2004). *The Spin Model Checker*. Boston, MA: Addison Wesley.

Huang, A. S., Olson, E., and Moore, D. C. (2010). "LCM: Lightweight communications and marshalling," in *Intelligent Robots and Systems* (Taipei: IEEE), 4057–4062.

Huston, G. (2000). *Next Steps for the IP QoS Architecture*. Internet Engineering Task Force. Available at: https://tools.ietf.org/html/rfc2990

ISO 11898-1:2015. (2015). *Road Vehicles – Controller Area Network (CAN) – Part 1: Data Link Layer and Physical Signalling*, 2 Edn. ISO. Available at: http://www.iso.org/iso/catalogue_detail.htm?csnumber=63648

ISO/IEC 9899:1999. (1999). *Programming Languages – C*, 2 Edn. ISO/IEC. Available at: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=29237

ISO/IEC 9899:2011. (2011). *Programming Languages – C*, 3 Edn. ISO/IEC. Available at: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=57853

Johnson, M., Shrewsbury, B., Bertrand, S., Wu, T., Duran, D., Floyd, M., et al. (2015). Team IHMC's lessons learned from the DARPA robotics challenge trials. *J. Field Rob.* 32, 192–208. doi:10.1002/rob.21571

Kalibera, T., Pizlo, F., Hosking, A. L., and Vitek, J. (2011). Scheduling real-time garbage collection on uniprocessors. *ACM Trans. Comput. Syst.* 29, 8. doi:10.1145/2003690.2003692

Kegel, D. (2006). *The C10K Problem*. Available at: http://www.kegel.com/c10k.html

Kerrisk, M. (2014). *aio – POSIX Asynchronous I/O Overview*, 3.74 Edn. Linux Man-Pages. Available at: https://www.kernel.org/doc/man-pages/

Kingston, Z. E., Dantam, N. T., and Kavraki, L. E. (2015). "Kinematically constrained workspace control via linear optimization," in *International Conference on Humanoid Robots* (Seoul: IEEE), 758–764. doi:10.1109/HUMANOIDS.2015.7363455

Lea, D. (2000). *A Memory Allocator*. Available at: http://g.oswego.edu/dl/html/malloc.html

Lee, E. A. (2009). Computing needs time. *Commun. ACM* 52, 70–79. doi:10.1145/1506409.1506426

Lignos, C., Raman, V., Finucane, C., Marcus, M., and Kress-Gazit, H. (2015). Provably correct reactive control from natural language. *Auton. Robots* 38, 89–105. doi:10.1007/s10514-014-9418-8

Linton, M., and Price, C. (1993). "Building distributed user interfaces with fresco," in *Proceedings of the 7th X Technical Conference* (Boston, MA: O'Reilly), 77–87.

Liu, J., Ozay, N., Topcu, U., and Murray, R. M. (2013). Synthesis of reactive switching protocols from temporal logic specifications. *Trans. Autom. Control* 58, 1771–1785. doi:10.1109/TAC.2013.2246095

Masmano, M., Ripoll, I., Crespo, A., and Real, J. (2004). "TLSF: a new dynamic memory allocator for real-time systems," in *Euromicro Conference on Real-Time Systems* (Catania: IEEE), 79–88.

Metta, G., Fitzpatrick, P., and Natale, L. (2006). YARP: yet another robot platform. *Int. J. Adv. Rob. Syst.* 3, 43–48. doi:10.5772/5761

Nedunuri, S., Prabhu, S., Moll, M., Chaudhuri, S., and Kavraki, L. E. (2014). "SMT-based synthesis of integrated task and motion plans from plan outlines," in *International Conference on Robotics and Automation* (Hong Kong: IEEE), 655–662.

Paikan, A., Pattacini, U., Domenichelli, D., Randazzo, M., Metta, G., and Natale, L. (2015). "A best-effort approach for run-time channel prioritization in real-time robotic application," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Hamburg: IEEE), 1799–1805.

pkg-config. (2013). *pkg-config*. Available at: http://www.freedesktop.org/wiki/Software/pkg-config/

Poettering, L. (2014). *Revisiting How We Put Together Linux Systems*. Available at: http://0pointer.net/blog/revisiting-how-we-put-together-linux-systems.html

POSIX. (2008). *IEEE Std 1003.1-2008*. The IEEE and The Open Group. Available at: http://pubs.opengroup.org/onlinepubs/9699919799/

Quarterman, J. S., Silberschatz, A., and Peterson, J. L. (1985). 4.2 BSD and 4.3 BSD as examples of the UNIX system. *ACM Comput. Surv. (CSUR)* 17, 379–418. doi:10.1145/6041.6043

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., et al. (2009). "ROS: an open-source robot operating system," in *International Conference on Robotics and Automation, Workshop on Open Source Robotics* (Kobe: IEEE).

Raymond, E. S. (2003). *The art of Unix Programming*. Boston, MA: Addison-Wesley Professional.

Raymond, E. S. and Landley, R. (2008). *OSI Position Paper on the SCO vs. IBM Complaint*. Available at: http://www.catb.org/~esr/hackerlore/sco-vs-ibm.html

Rosenbaum, R. (1993). *Using the Domain Name System To Store Arbitrary String Attributes*. Internet Engineering Task Force. Available at: https://tools.ietf.org/html/rfc1464

Scheifler, R. W. (2004). *X Window System Protocol*. X Consortium, Inc. Available at: http://www.x.org/archive/X11R7.5/doc/x11proto/proto.pdf

Siemon, D. (2013). Queueing in the linux network stack. *Linux J.* Available at: http://www.linuxjournal.com/content/queueing-linux-network-stack

Silberschatz, A., Galvin, P. B., and Gagne, G. (2009). *Operating System Concepts*. Danvers, MA: J. Wiley & Sons.

Smith, J., Stephen, D., Lesman, A., and Pratt, J. (2014). "Real-time control of humanoid robots using OpenJDK," in *International Workshop on Java Technologies for Real-time and Embedded Systems* (Niagara Falls, NY: ACM), 29.

Srinivasan, R. (1995). *Binding Protocols for ONC RPC Version 2*. Internet Engineering Task Force. Available at: https://tools.ietf.org/html/rfc1833

Stevens, W. R., and Rago, S. A. (2013). *Advanced Programming in the UNIX Environment*. Indianapolis: Addison-Wesley.

Sucan, I., Moll, M., and Kavraki, L. E. (2012). The open motion planning library. *Rob. Autom. Mag.* 19, 72–82. doi:10.1109/MRA.2012.2205651

Tanenbaum, A. S., and Bos, H. (2014). *Modern Operating Systems*. Upper Saddle River, NJ: Prentice Hall Press.

Thomas, M., Lupu, E., and Rükert, D. (2003). *Y: A Successor to the x Window System*. Technical Report, Imperial College London. Available at: http://www3.imperial.ac.uk/pls/portallive/docs/1/18619743.PDF

Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., and Mahlke, S. (2009). "The theory of deadlock avoidance via discrete control," in *SIGPLAN Notices*, Vol. 44 (Savannah: ACM), 252–263.

Yuasa, T. (1990). Real-time garbage collection on general-purpose machines. *J. Syst. Software* 11, 181–198. doi:10.1016/0164-1212(90)90084-Y

Zucker, M., Joo, S., Grey, M. X., Rasmussen, C., Huang, E., Stilman, M., et al. (2015). A general-purpose system for teleoperation of the DRC-HUBO humanoid robot. *J. Field Rob.* 32, 336–351. doi:10.1002/rob.21570

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

The reviewer AP and the handling editor LN declared their shared affiliation, and the handling editor states that the process nevertheless met the standards of a fair and objective review.

APPENDIX

A. Ach Multiplexing Example

Ach kernel channels can be multiplexed using the conventional `select`, `poll`, etc. functions. We provide an example program that multiplexes two Ach channels along with the program's standard input and echoes all data to standard output. This example demonstrates Ach's efficient handling of multiple data sources and compatibility with other forms of POSIX communication.

```
#include <stdlib.h>
#include <pthread.h>
#include <inttypes.h>
#include <stdio.h>
#include <poll.h>
#include <unistd.h>
#include <ach.h>

int main(int argc, char **argv)
{
    const char *names[] = {"channel-0", "channel-1"};
    const size_t n_channels = sizeof(names) / sizeof(names[0]);
    const size_t n_pfd = n_channels + 1;
    struct ach_channel channel[n_channels];
    struct pollfd pfd[n_pfd];

    /*****************************/
    /* Initialize pollfd structs */
    /*****************************/
    for( size_t i = 0; i < n_channels; i ++ ) {
        /* Open Channel */
        enum ach_status r = ach_open( &channel[i], names[i], NULL );
        if( ACH_OK != r ) {
            fprintf(stderr, "could not open channel '%s': %s\n",
                    names[i], ach_result_to_string(r));
            exit(EXIT_FAILURE);
        }
        /* Get channel file descriptor */
        r = ach_channel_fd( &channel[i], &pfd[i].fd );
        if( ACH_OK != r ) {
            fprintf(stderr, "could not get file descriptor for channel '%s': %s\n",
                    names[i], ach_result_to_string(r));
            exit(EXIT_FAILURE);
        }
        /* Set events to poll for */
        pfd[i].events = POLLIN;
    }
    /* Also, poll() standard input */
    pfd[n_channels].fd = STDIN_FILENO;
    pfd[n_channels].events = POLLIN;


    /***************/
    /* poll() loop */
    /***************/
    for(;;) {
        /* poll() for new data */
        int r_poll = poll( pfd, n_pfd + 1, -1 );
        if( r_poll < 0 ) {
            perror("poll");
            exit(EXIT_FAILURE);
        }
        /* Find file descriptors with new data */
        for( size_t i = 0; i < n_pfd && r_poll > 0; i++ ) {
            if( (pfd[i].revents & POLLIN) ) {
                char buf[512];
                size_t data_size = 0;
                if( i < n_channels ) {
                    /* Get new data on an Ach channel */
                    enum ach_status r = ach_get( &channel[i], buf, sizeof(buf), &data_size,
```

```
                                                    NULL, ACH_O_NONBLOCK | ACH_O_FIRST );
                    switch(r) {
                    case ACH_OK:
                    case ACH_MISSED_FRAME:
                        break;
                    default:
                        fprintf( stderr, "Error getting data from '%s': %s\n",
                                 names[i], ach_result_to_string(r));
                        exit(EXIT_FAILURE);
                    }
                } else {
                    /* Read new data on a file descriptor */
                    ssize_t r = read(pfd[i].fd, buf, sizeof(buf));
                    if( r < 0 ) {
                        perror("read()");
                        exit(EXIT_FAILURE);
                    } else {
                        data_size = r;
                    }
                }
                /* Echo the data to standard output */
                ssize_t wr = write(STDOUT_FILENO, buf, data_size);
                if( wr < 0 ) {
                    perror("write");
                    exit(EXIT_FAILURE);
                }
                r_poll--;
            }
        }
    }
    return 0;
}
```

B. Configuration, Building, and Packaging

Software build systems and package managers are useful tools to address the system integration needs of humanoid robot software. Open source software distributions such as Debian and FreeBSD have developed approaches for integrating and maintaining enormous numbers of software packages. Their goal is to make software that is portable, that builds robustly, and that is easy to install, upgrade, and remove. These tools are general and suited to the needs of humanoids as well.

Different humanoid robots provide varying hardware capabilities and software environments, and it is important that humanoid software be adaptable across these different robots. A key step to achieving this portability is the *configuration* step of the build process, where the software adapts to conditions of the environment in which it must run. For example, configuration may determine whether to use the previously mentioned `epoll` or `kqueue` calls depending on whether it must run on Linux or FreeBSD, or it may determine how to interface with the fieldbus linking the robot's embedded electronics. In general, configuration chooses alternate implementations or optional components to build based on the available features of the host system. This adaptability is vital to building software that is portable and that builds robustly.

The two predominant build systems are the GNU Autotools and CMake. Overall, both offer similar capabilities with a number of superficial differences. There is, however, a difference in design philosophy that influences the use of these systems. Autotools assumes little about the host platform beyond a POSIX shell, testing at compile-time for essentially every other feature, an approach that is robust to new and changing platforms but requires additional time for the feature tests. In contrast, CMake maintains a database of modules for platforms and libraries, which can reduce compilation time by omitting feature tests but is unhelpful for differing platforms and dependencies. One can also maintain a build-system agnostic database of available libraries using pkg-config (pkg-config, 2013), which works with CMake, Autotools, and other build systems. Additionally, projects using Autotools generally follow a strict set of conventions such that all can be configured, built, and installed by the same procedure (GNU Standards, 2015). There are fewer established conventions for CMake so it is common for different CMake projects to require different steps in the build process. One should consider the need for adaptability, conformity, and configuration performance when selecting a build system.

We use Autotools to build Ach due to their maturity and strict conventions compared to CMake. In addition, Autotools enable more direct integration with the Make-based build system of the Linux kernel, which simplifies building and installing the Ach Linux kernel module (see Section 3.2).

To manage the large number of software packages on humanoid robots, package managers are an invaluable tool. Package managers handle the details of installation, cross-package dependencies,

and package versioning. The two main styles of package managers are binary-based and source-based. Binary-based package managers download and install pre-compiled packages. Examples include Redhat's RPM, Debian's APT, and – if viewed broadly – "App Stores" such as that of Apple's iOS and Google Play. Source-based package managers download package source code and build it on the local machine. Examples include FreeBSD ports, Gentoo Portage, and Homebrew for MacOSX. The advantage of binary packages is that no time must be spent to compile the package on the local machine. The advantage of source-based package managers is that packages can be custom-configured with optional features based on the users preferences and fewer server resources are required to store the compiled binaries. The choice of a package manager is typically dictated by the operating system distribution of the user. For handling software deployment, these package managers are mature and useful tools.

Two new data storage tools offer the potential to improve existing build systems and package managers: distributed revision control – e.g., git and mercurial – and copy-on-write (COW) filesystems – e.g., ZFS and BTRFS. Existing build systems and package managers were developed at a time when source code was typically downloaded as tarballs from a scattered collection of servers. Today, source code is often downloaded via a distributed version control system. Directly accessing the revision controlled files is a poor fit for Autotools' approach of generating a portable configuration script, and it makes some features of current package managers redundant, such as hosting and distributing multiple tarball versions and applying patches before building. Second, COW filesystems provide the capability to make cheap, writable snapshots. This could be used to maintain multiple concurrent images of the operating system, for example to install different versions of one package or two different conflicting packages. One view on how these new tools could change build systems and packages managers is given by Poettering (2014). These new developments show that while existing build systems and package managers are useful and mature, there is room for ongoing development and new innovation.

ROS (Quigley et al., 2009) provides a different view on build systems and package management, focusing on robots. ROS combines the build system and package manager into a single framework based on CMake (Catkin, 2015); however, binary packages are still distributed using the existing APT package manager. This approach eliminates some duplication of metadata – i.e., which files must be installed – necessary with the traditional distinction between package managers and build systems. However, this is unhelpful if packages are to be installed on a non-ROS system. Additionally, packages are also constrained to use the given CMake-based build system which may not always be the best fit, e.g., for non-C/C++ packages, or provide necessary capabilities, e.g., the site-based configuration feature of Autotools. Still, ROS presents an interesting take on how we build and package software.

frontiers in Robotics and AI

OPINION
published: 29 September 2015
doi: 10.3389/frobt.2015.00023

Cryptobotics: why robots need cyber safety

Santiago Morante, Juan G. Victores and Carlos Balaguer*

Robotics Lab, Automation and Engineering Systems Department, Universidad Carlos III de Madrid, Madrid, Spain

Keywords: cryptography, robotics, cyber safety, communications, cyber-physical, cryptobotics, cyber security

Introduction

With the expected introduction of robots into our daily lives, providing mechanisms to avoid undesired attacks and exploits in robot communication software is becoming increasingly required. Just as during the beginnings of the computer age (Pfleeger and Pfleeger, 2002), robotics is established in a "happy naivety," where security rules against external attacks are not adopted, assuming that robotics knowledgeable people are well intended. While this may have been true in the past, the mass adoption of robots will increase the possibilities of attacks. This fact is especially relevant in defense, medical and other critical fields involving humans, where tampering can result in serious bodily harm and/or privacy invasions. For these reasons, we consider that researchers and industry should deploy efforts in cyber safety and acquire good practices when developing and distributing robot software. We propose the term *Cryptobotics* as a unifying term for research and applications of computer and microcontrollers' security measures in robotics.

Stating the Problem

The problems that the field of robotics will face are similar to those the computer revolution faced with the widespread of the Internet 30 years ago. Among the common attacks computers may suffer, there are: denial-of-service, eavesdropping, spoofing, tampering, privilege escalation, or information disclosure for instance. To these problems, robots add the additional factor of physical interaction. While taking the control of a desktop computer or a server may result in loss of information (with its associated costs), taking the control of a robot may endanger whatever or whoever is near.

As robots become more integrated on the communications networks, it seems appropriate to reuse the tools designed for web applications in order to controls the robots. However, the authors consider there are differences between regular computers communicating through the network, and robots performing the same actions. Mohanarajah et al. (2015) states differences between web and robotic applications: "Web applications are typically stateless, single processes that use a request-response model to talk to the client. Meanwhile, robotic applications are stateful, multiprocessed, and require a bidirectional communication with the client. These fundamental differences may lead to different tradeoffs and design choices and may ultimately result in different software solutions for web and robotics applications." To these differences, we could also add the real-time constraints that characterize robotics applications. Despite other sources of issues, like software bugs or vulnerabilities [buffer overflow, command injection, etc. (Tanenbaum and Bos, 2014)], we consider that communications currently are one of the main vulnerabilities in robotics.

A number of fields in robotics where security and privacy are particularly relevant can be addressed.

- Defense and Space: The military field should be very aware of the best practices in cyber security to be followed regarding its robots. Unmanned aerial vehicles, commonly called "drones," are being destined to surveillance and also to combat missions. Common sense dictates that any communications with these vehicles should be encrypted (Javaid et al., 2012), but reality shows us

OPEN ACCESS

Edited by:
Lorenzo Natale,
Istituto Italiano di Tecnologia, Italy

Reviewed by:
Fulvio Mastrogiovanni,
University of Genoa, Italy
Emanuele Ruffaldi,
Scuola Superiore Sant'Anna, Italy
Ali Paikan,
Istituto Italiano di Tecnologia, Italy

****Correspondence:***
Santiago Morante
smorante@ing.uc3m.es

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *04 June 2015*
Accepted: *11 September 2015*
Published: *29 September 2015*

Citation:
Morante S, Victores JG and Balaguer C (2015) Cryptobotics: why robots need cyber safety. Front. Robot. AI 2:23. doi: 10.3389/frobt.2015.00023

differently. For example, in the year 2012 it was reported that only between 30 and 50 percent of America's Predators and Reapers (two of the most used drones in US) were using fully encrypted transmissions.[1]

> *Situation: a non-authorized entity eavesdrops surveillance images of drones, takes its control, exploiting a non-encrypted connection, and crashes it into a populated area.*
> *Situation: a non-authorized entity takes control of a robot inside International Space Station and sabotages an ongoing experiment.*

- Telemedicine and Remote surgery: This exciting field can make remote surgery become an everyday reality, where experts can operate patients from the other side of the world. While this is beneficial to society, we must consider the potential dangers. In 2009, the Interoperable Telesurgery Protocol (ITP) (King et al., 2009) was proposed as a preliminary specification for interoperability among robotic telesurgery systems. Recently, the fact that ITP does not use any form of encryption or authentication was discovered.[2] This is an obvious system exposure to exploits using a man-in-the-middle attack for taking control of the robot (Bonaci et al., 2015).

  > *Situation: a non-authorized entity takes control of a surgery robot during an operation, endangering the life of the patient.*

- Household robots: This market is growing both in research and commercially available robots. Robots will be used as assistants at home. For instance, one of these projects is Care-O-bot (Hans et al., 2002), a robotic assistant in homes. In one of the available versions, this robot is equipped with microphones, cameras and 3D sensors. This set of sensors can collect a huge amount of information, which must be protected (Denning et al., 2009). Service robots may one day also collect data about the health status of a person; law regulations require that this data is handled with extra care.

  > *Situation: a non-authorized entity takes control of a household robot and obtains streams of images with private data.*

- Disaster robots: Since the Fukushima Daiichi nuclear disaster in 2011, the robotics community has increased its efforts to build and deploy robots for disaster scenarios. One of the expected tasks these robots will have to face in a disaster scenario is related to accessing and repairing/disconnecting dangerous systems. Due to the potential danger that may arise in these situations (Vuong et al., 2014), robots should not be able to be externally modified by an external attack.

  > *Situation: a non-authorized entity takes control of a robot deployed to disconnect a nuclear platform that may suffer a partial meltdown, and can thwart the disconnection operation.*

Current State of Security in Mainstream Robotic Software

Robots are a combination of mechanical structures, sensors, actuators, and computer software that manages and controls these devices. Mainstream practices in robotics involve component-based software engineering. Each component is designed as an individual computer program (e.g., a motor moving program) which communicates with other components using predefined protocols. While a large quantity of software libraries for communication already exist, the robotics community has developed a number of "software architectures." Currently, one of the most popular robotics-oriented architecture is ROS (Robot Operating System) (Quigley et al., 2009). Another co-existing architecture is YARP (Yet Another Robot Platform) (Metta et al., 2006). Both systems work similarly: a system built using ROS or YARP consists of a number of programs (nodes or modules), potentially on several different hosts, connected in a peer-to-peer topology.

According to ROS documentation[3]: "Topics are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics (.) In general, nodes are not aware of who they are communicating with." From the point of view of security, this anonymous communication scheme is a welcome sign toward exploits (McClean et al., 2013). Messages are sent unencrypted through TCP/IP or UDP/IP. The default check performed is an initial MD5 sum of the message structure, a mechanism used to assure the parties agree on the layout of the message. Some researchers have developed an authentication mechanism for achieving secure authentication for remote, non-native clients in ROS (Toris et al., 2014). While it can increase the security of the overall system, without data encryption, an eavesdropper could acquire non-encrypted information.

Part of the ROS community is dedicating efforts to integrating OMG's DDS (Data Distribution Service) as a transport layer for ROS 2.0.[4] A preliminary alpha version has just been released. DDS is a standard specification followed by several vendors for a middleware providing publish-subscribed communications for real-time and embedded systems. RTI provides plugins which comply with the DDS Security specification including authentication, access control and cryptography. It would be a big step forward for securing our robots if ROS 2.0 aimed to comply with the DDS Security specification as well.

YARP states among its documentation[5]: "A [default] new connection to a YARP port is established via handshaking on a TCP port. So everyone who can access this TCP port can connect to your YARP port. So if you are not behind a firewall, you are exposing your YARP infrastructure to the world (.) And if your application is vulnerable to corrupted data, it is a security

[1] Most U.S. Drones Openly Broadcast Secret Video Feeds: http://www.wired.com/2012/10/hack-proof-drone/
[2] Interoperable Telesurgery Protocol (ITP) Plaintext Unauthenticated MitM Hijacking: http://osvdb.org/121842
[3] http://wiki.ros.org/Topics
[4] http://design.ros2.org
[5] http://wiki.icub.org/yarpdoc/yarp_port_auth.html

leak." Other YARP documentation reads clearly[6]: "If you expose machines running YARP to the Internet, expect your robot to 1 day be commanded to make a crude gesture at your funders by a script kiddie in New Zealand." However, an authentication mechanism can be activated in YARP, which adds a key exchange to the initial handshaking in order to authenticate any connection request. It has been enabled by default so it is always compiled. However, to preserve backward compatibility, the feature is skipped at runtime if the user does not configure it by providing a file that contains the authentication key.

Additionally, a new port monitoring and arbitration (Paikan et al., 2014) functionality inside YARP has been used to implement a LUA encoder/decoder of data.[7] Data are passed through a Base64 encoder before being sent, and decoded upon reception at the target port. A similar mechanism could potentially be used to encrypt and decrypt the data.

A limited amount of other works has also focused on securing robot communications. In Groza and Dragomir (2008), they implement an authentication protocol to assure the authenticity of the information when controlling a robot via TCP/IP. However, they do not implement encrypted communications. In Coble et al. (2010), they implemented a hardware system that verifies integrity and health of the system software (to avoid tampering) in telesurgical robots. Regarding the previously mention ITP protocol, some researchers are working on security enhancements (Lee and Thuraisingham, 2012). One commercially available robot that does take cyber security into account is BeamPro, a telepresence robot[8] where secure protocols, symmetric encryption, and data authentication are used, thus providing security and privacy.

Secure communications are even more important in new trends in robotics which aim at outsourcing computation, namely *Cloud Robotics*. In this paradigm, robots use their sensors to collect data, and then upload the information to a remote computation center, where the information is processed, and may be shared with other robots. Rapyuta (Mohanarajah et al., 2015) is an example of this paradigm where the technologies used (e.g., WebSockets) allow to secure the information.

Another usual way of communications between robot's devices is through communication buses (CAN, EtherCAT, etc.). Currently, none of the traditional field buses offers security features against intentional attacks (Dzung et al., 2005). However, those based on ethernet could potentially make use of the security measures included in TCP/UDP/IP. For instance, secure routers (e.g., EDR-G903), include firewalls and VPNs, and support EtherCAT.

Discussion

A big market of opportunities for research regarding cyber safety in robotics exists. Most robots are not yet prepared, from a security point of view, to be deployed in daily life. The software is not prepared to protect against attacks, because communications are usually unencrypted.

Regarding the dates of the exploits presented, and the current hype in deployment of daily robotics (vacuum cleaners, amateur drones, etc.), *Cryptobotics*, understood as a mix of cyber security and robotics, comes just in time to prepare these systems to be safely used.

An important issue to be discussed is whether the implementation of encrypted communications may affect the performance, especially in real-time systems. The question about performance is highly dependant on the hardware, the software and the network used. Encrypted communications on the Internet (https, ssh) show us that it is possible to perform secure communications and offer remote services. For instance, Adam Langley (Google Senior Staff Software Engineer) has stated: "when Google changed Gmail from http to https (.) we had to deploy no additional machines and no special hardware. On our production front-end machines, SSL/TLS accounts for less than 1 of the CPU load.[9]" From our experience in humanoid robotics, a 1% overhead (while respecting determinism in time) can be acceptable if it means our devices can be less vulnerable to cyber attacks. Could an 8 MHz microcontroller perform real-time encryption? Is it reasonable to implement authentication mechanisms along field buses in time-constrained scenarios? This article intends to raise awareness for developers to determine whether it is viable to integrate these mechanisms depending on each specific use case.

Some may ask why these problems have not been addressed previously. In recent years, intrinsically safe industrial robots, the rise of domestic robots, and the use of mobile robots in public spaces, have arisen issues that the robotics community did not have to face in its previous 60 years of existence. Researchers are now focused on developing applications to make robots useful, which may have made cyber safety a low priority.

Author Contributions

SM discovered the potential issue in mainstream robotics software and wrote part of the paper. JV found the technical support behind the security issues and wrote and improved the text. CB provided context for the topic, reviewed the text, defined the criteria to evaluate the work, and contributed to the text.

References

Bonaci, T., Herron, J., Yusuf, T., Yan, J., Kohno, T., and Chizeck, H. J. (2015). "To make a robot secure: an experimental analysis of cyber security threats against teleoperated surgical robots," in *arXiv Preprint arXiv:1504.04339*.

Coble, K., Wang, W., Chu, B., and Li, Z. (2010). Secure software attestation for military telesurgical robot systems" in *Military Communications Conference, 2010 – Milcom 2010*, 965–970. doi:10.1109/MILCOM.2010.5679580

Denning, T., Matuszek, C., Koscher, K., Smith, J. R., and Kohno, T. (2009). "A spotlight on security and privacy risks with future household robots: attacks

[6]http://wiki.icub.org/yarpdoc/what_is_yarp.html
[7]http://wiki.icub.org/yarpdoc/coder_decoder.html
[8]http://www.suitabletech.com/
[9]http://www.imperialviolet.org/2010/06/25/overclocking-ssl.html

and lessons," in *Proceedings of the 11th International Conference on Ubiquitous Computing* (New York, NY: ACM), 105–114. doi:10.1145/1620545.1620564

Dzung, D., Naedele, M., Von Hoff, T. P., and Crevatin, M. (2005). Security for industrial communication systems. *Proc IEEE* 93, 1152–1177. doi:10.1109/JPROC.2005.849714

Groza, B., and Dragomir, T.-L. (2008). "Using a cryptographic authentication protocol for the secure control of a robot over TCP/IP," in *IEEE International Conference on Automation, Quality and Testing, Robotics, 2008. AQTR 2008*, Vol. 1, 184–189. doi:10.1109/AQTR.2008.4588731

Hans, M., Graf, B., and Schraft, R. (2002). "Robotic home assistant care-o-bot: past-present-future," in *Proceedings of the 11th IEEE International Workshop on Robot and Human Interactive Communication, 2002*, 380–385. doi:10.1109/ROMAN.2002.1045652

Javaid, A. Y., Sun, W., Devabhaktuni, V. K., and Alam, M. (2012). "Cyber security threat analysis and modeling of an unmanned aerial vehicle system," in *2012 IEEE Conference on Technologies for Homeland Security (HST)*, 585–590. doi:10.1109/THS.2012.6459914

King, H. H., Tadano, K., Donlin, R., Friedman, D., Lum, M. J., Asch, V., et al. (2009). "Preliminary protocol for interoperable telesurgery," in *International Conference on Advanced Robotics, 2009. ICAR 2009*, 1–6. Available at: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5174711&isnumber=5174665

Lee, G. S., and Thuraisingham, B. (2012). Cyberphysical systems security applied to telesurgical robotics. *Comput Stand Interfaces* 34, 225–229. doi:10.1016/j.csi.2011.09.001

McClean, J., Stull, C., Farrar, C., and Mascareñas, D. (2013). "A preliminary cyber-physical security assessment of the robot operating system (ROS)," in *Proceedings of SPIE 8741, Unmanned Systems Technology XV*, 874110. doi:10.1117/12.2016189

Metta, G., Fitzpatrick, P., and Natale, L. (2006). Yarp: yet another robot platform. *Int J Adv Rob Syst* 3, 43–48. doi:10.5772/5761

Mohanarajah, G., Hunziker, D., D'Andrea, R., and Waibel, M. (2015). Rapyuta: a cloud robotics platform. *IEEE Trans Autom Sci Eng* 12, 481–493. doi:10.1109/TASE.2014.2329556

Paikan, A., Fitzpatrick, P., Metta, G., and Natale, L. (2014). Data flow ports monitoring and arbitration. *J Software Eng Rob* 5, 80–88.

Pfleeger, C. P., and Pfleeger, S. L. (2002). *Security in Computing*. Prentice Hall Professional Technical Reference.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., et al. (2009). "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 3, 5.

Tanenbaum, A. S., and Bos, H. (2014). *Modern Operating Systems*. Prentice Hall Press.

Toris, R., Shue, C., and Chernova, S. (2014). "Message authentication codes for secure remote non-native client connections to ros enabled robots," in *2014 IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*, 1–6. doi:10.1109/TePRA.2014.6869141

Vuong, T., Filippoupolitis, A., Loukas, G., and Gan, D. (2014). "Physical indicators of cyber attacks against a rescue robot," in *2014 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 338–343. doi:10.1109/PerComW.2014.6815228

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Advantages of publishing in Frontiers

OPEN ACCESS
Articles are free to read for greatest visibility and readership

FAST PUBLICATION
Around 90 days from submission to decision

HIGH QUALITY PEER-REVIEW
Rigorous, collaborative, and constructive peer-review

TRANSPARENT PEER-REVIEW
Editors and reviewers acknowledged by name on published articles

Frontiers
Avenue du Tribunal-Fédéral 34
1005 Lausanne | Switzerland

Visit us: www.frontiersin.org
Contact us: info@frontiersin.org | +41 21 510 17 00

REPRODUCIBILITY OF RESEARCH
Support open data and methods to enhance research reproducibility

DIGITAL PUBLISHING
Articles designed for optimal readership across devices

FOLLOW US
@frontiersin

IMPACT METRICS
Advanced article metrics track visibility across digital media

EXTENSIVE PROMOTION
Marketing and promotion of impactful research

LOOP RESEARCH NETWORK
Our network increases your article's readership