High-Level Robot Control with ArmarX

Nikolaus Vahrenkamp, Mirko Wächter, Manfred Kröhnert,
Peter Kaiser, Kai Welke, Tamim Asfour

High Performance Humanoid Technologies (H^2T)
Karlsruhe Institute for Technology (KIT)
Adenauerring 2
76131 Karlsruhe
{vahrenkamp,waechter,kroehnert,kaiser,welke,asfour}@kit.edu

Abstract: The robot development environment (RDE) ArmarX aims at providing an infrastructure for developing a customized robot framework that allows realizing distributed robot software components. This includes communication properties, start-up and error handling, mechanisms for state implementations, interface definitions and concepts for the structured development of robot programs. In addition to this core functionality, we will show in this paper that ArmarX provides customizable building blocks for high level robot control and how these components can be used to build a generic backbone of the robot software architecture. ArmarX provides standard interfaces and ready-to-use implementations of several core components which are needed to setup a distributed robot software framework.

1 Introduction

In the last decade, service and humanoid robot technologies have made enormous progress in terms of integrating sensori-motor capabilities (see Figure 1). The trend towards dynamic and interactive areas of application leads to complex, distributed and asynchronously operating software systems where the robot state is distributed over multiple components. These dedicated distributed components can be developed and tested independent of the whole system, for example in a simulation environment. Nevertheless, software development complexity increases due to synchronizing aspects, concurrent access, data distribution and all challenges that arise in parallel software development.

To support development and integration of all required capabilities of such robots is the goal of robot development environments (RDEs). Different functionalities and capabilities of the robot software should be glued together by such environments in terms of system architecture and communication. In addition, RDEs should provide programming interfaces, allowing programmers to include new capabilities at appropriate levels of abstraction with minimal training effort.

As stated in [WVW+13] several RDEs have been presented during the last decades accompanying the development of robotic platforms. Different levels of abstraction are realized by RDEs, depending on the robot platform and its intended application.

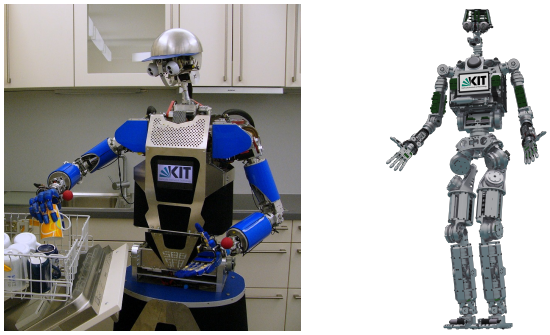


Figure 1: The humanoid robots ARMAR-IIIa [AAV+08] and ARMAR-IV [ASP+13].

Several RDE frameworks put their focus on the control level, such as OpenRTM [ASK08], MatLab/Simulink®, MCA[1], whereas others focus on the implementation of higher level capabilities of the system (e.g. ROS [QCG+09], Yarp [MFN06] and Orocos [BSK03]).

In [WVW+13] we presented the ArmarX RDE framework and its disclosure capabilities. Compared to earlier work, we will focus on the layered architecture of ArmarX in this paper. In addition, we will discuss the generic features of ArmarX that allow creating tailored robot frameworks on top of which application specific robot programs can be implemented.

2 The Robot Development Environment ArmarX

The Robot Development Environment ArmarX provides several components, functionalities, interfaces and generic ready-to-use components for building distributed robot applications. In contrast to traditional message passing protocols, well-defined interface definitions and flexible communication mechanisms are used. ArmarX is organized in several layers as shown in Figure 2. The *Middleware Layer* provides all facilities for implementing distributed applications. It abstracts the communication mechanisms, provides basic building blocks of distributed applications, and facilities for visualization and debugging. Bindings to a wide area of programming languages covering Python, C++, C# and Java are provided. While the *Middleware Layer* comprises a predefined set of components, interfaces and methods, the *Robot Framework Layer* serves several extendable and exchangeable components which can be assembled to an appropriate robot software framework. Currently provided are interfaces and corresponding components for memory, robot and world model, perception, and execution. Since no framework is suitable for all robot systems, this layer can be customized and adapted to the demands of the specific robot platform in order to tailor the robot framework to the needs of the application of the robot. ArmarX comprises several customizable components like memory structures

[1] http://www.mca2.org/

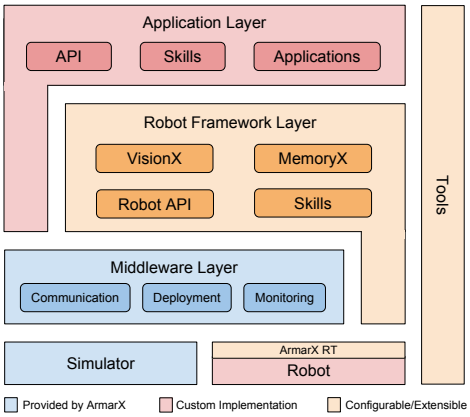


Figure 2: The Middleware Layer of the ArmarX framework provides core functionalities like distributed communication, deployment, start up mechanisms, and network transparent statecharts. The Robot Framework Layer consists of several ready-to-use parameterizable and extendable components regarding perception, memory and generic robot API methods (e.g. kinematics and visualization). Additionally, a set of basic skills such as visual servoing, grasping or navigation are included. Robot programs are developed in the Application Layer supported by the ArmarX toolset. The hardware abstraction layer (bottom of the figure) implements the interfaces of the Robot Framework layer for a specific hardware type or the simulator. Different colors indicate at which points extensions can be realized. Blue layers provide ready-to-use concepts without the possibility for extension. Red layers require user specific implementations. Within orange layers, several ready-to-use components are available which can be parametrized and/or exchanged by user specific implementations.

with interfaces to corresponding update processes compatible with the perception components. Additionally, a generic robot API is provided which includes a network transparent representation of the robot's sensor state, generic methods related to kinematics, collision detection, motion and grasp planning and scene representation. A set of basic skills is provided as configurable statecharts and can be adapted to the needs of the robot framework. The robot application is realized as a network transparent statechart within the *Application Layer*. ArmarX supports developing application specific extensions to APIs and the skill library. Tool support comprises GUI elements, a statechart editor, a physics-based simulation environment and state disclosure mechanisms for online inspection and debugging.

As shown in [WVW+13], the main design principles of ArmarX are distributed processing, interoperability, and disclosure of system state. In this work we will focus on the interoperability features of the framework provided by the ArmarX Middleware Layer, allowing the implementation of ArmarX components in different programming languages and their execution on different operating systems. Another aspect of interoperability is covered by the Robot Framework Layer. The proposed structures allow tailoring the software framework according to application-specific requirements and the capabilities of the robot. To this end, ArmarX provides generic building blocks which can be customized, parameterized, and extended to ease the possibility of embedding ready-to-use components on different robot platforms.

3 ArmarX Middleware: The Core Layer

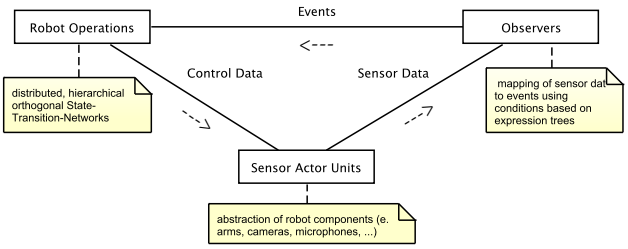


Figure 3: The application programming interface comprises four different elements. *Sensor-Actor Units* serve as abstraction of robot components, *Observers* generate events from continuous sensory data streams resulting in transitions between *Operations*. Operations are organized as hierarchical state-transitions networks. All elements are connected via the *communication* mechanisms (arrows in the figure).

The middleware layer of ArmarX comprises four main building blocks: *inter-object communication*, *Sensor-Actor Units*, *Observers*, and *Operations*. *Inter-object communication* provides convenient mechanisms for communication between objects in the system. These objects can be any of the other buildings blocks. The *Sensor-Actor Units* offer a generic interface on a low level of abstraction for robot components like motor controllers or sensors. ArmarX includes a default set of sensor-actor units for common robot components, e.g. joints, force/torque-sensors and haptic sensors. Implementing additional sensor-actor units for unsupported components is encouraged and straightforward. *Observers* monitor sensor values from sensor-actor units and send application specific events to configured *Operations*. These operations process events and send resulting control commands to the sensor-actor units controling the robot. More details can be found in [WVW+13]. In addition to these building blocks, monitoring facilities for data flow and system state are incorporated into the ArmarX framework.

3.1 Inter-Object Communication

As shown in [WVW+13], ArmarX uses well established communication mechanisms (e.g. Ethernet, TCP/IP, UDP) to provide convenient ways to create distributed applications for robot architectures. To this end, we employ the Internet Communication Engine (Ice) by ZeroC as fundamental communication middleware [Hen04].

Ice implements type-safe distributed communication over Ethernet with a variety of supported platforms and programming languages. It manages the connections between objects and provides convenient communication modes such as synchronous and asynchronous remote procedure calls and publisher-subscriber mechanisms. Communication interfaces are defined in the platform and programming language independent Interface Definition Language (IDL) *Slice*, which is then compiled to the chosen programming language.

3.2 Monitoring

Monitoring is an important concept for any robotic system. ArmarX provides means to retrieve CPU load, network utilization, IO operations, and data update frequency information. Additionally, generated sensor measurements, control flow information as well as more abstract information like condition histories can be retrieved.

Monitoring information can be gathered within a distributed ArmarX robot program for profiling running robot programs and identifying bottlenecks in the distributed application. This information can be used to improve the deployment of ArmarX components and therefore the load balancing in order to provide seamless program execution. Furthermore, resource usage profiles of ArmarX components can be generated by performing offline analysis on monitoring data.

Using the disclosure mechanisms of ArmarX [WVW+13] it is even possible to monitor the execution of robot program statecharts to build an execution model of the robot software components. Combining such execution models with generated resource profiles allows for creating control-flow dependent resource usage diagrams.

4 ArmarX RT: Real-Time Hardware Abstraction

Sensor-actor units provide a sufficient and thin hardware abstraction when real-time requirements do not apply, e.g. when working in simulation or when using inherently stable robots. However, for many robots, especially bipeds, the low-level hardware control software needs to satisfy real-time constraints in order to maintain the robot's stability and safety. To address this issue, ArmarX supports a second layer of hardware abstraction that is intended to run in a real-time context on a Xenomai[2] real-time OS.

The purpose of this hardware backend is to control the attached hardware entirely within a real-time context. Communication between hardware backends and sensor-actor units happens via Xenomai's real-time pipes. Figure 4 depicts the complete hardware abstraction layer including a generic real-time hardware backend for CANopen[3]-based robots that is included in ArmarX. Additionally, adding real-time hardware backends for other hardware as well as operating systems other than Xenomai is supported.

5 ArmarX Robot Framework: Generic Building Blocks for High-Level Robot Control

ArmarX provides a generic robot framework layer that offers standard components and interfaces to ease development of tailored robot frameworks. Several generic and ready-to-use API components can be parameterized and customized in order to build robot specific

[2] http://www.xenomai.org
[3] http://www.can-cia.org

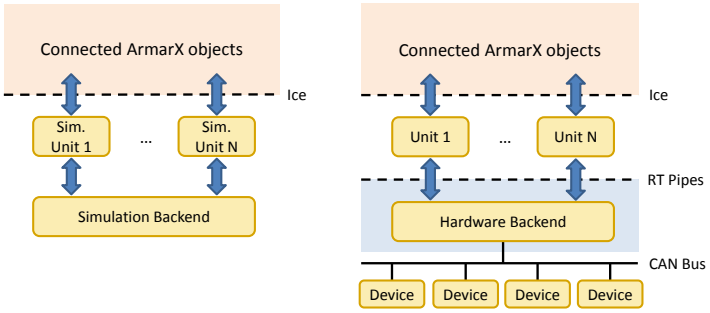


Figure 4: ArmarX hardware abstraction for a simulated robot (left) compared to the real-time capable hardware abstraction for a CANopen based robot (right). Both setups are possible with ArmarX. Switching between a simulation backend and the real hardware is completely transparent to higher-level ArmarX objects.

APIs. Robot model, world model, perception and memory components are available, each providing entry points for custom robot program implementations. These components will be briefly discussed in the following section.

5.1 Robot API and Skills

The robot API comprises several components which can be used off-the-shelf by specifying required parameter settings. If needed, the API can be extended in order to reflect specialized capabilities of the robot.

5.1.1 Kinematics, Robot and World Models

The internal robot model consists of the kinematic structure, sensors, model files and additional parameters such as mass or inertial tensors. The data is consistently specified via XML or COLLADA files compatible to the robot simulation toolbox *Simox* [VKU+12]. This allows making use of all Simox related functionalities, such as inverse kinematics, collision detection, motion and grasp planning or defining reference frames for coordinate transformations. Further, ArmarX provides network transparent access to a shared robot model data structure. This allows all distributed components to query robot internal data such as joint values or to perform coordinate transformations.

5.1.2 Kinematic Units and Robot Skills

The robot API provides reference implementations for simulating kinematic units. Several control modes such as position, velocity or torque can be used out of the box. The robot skill components cover basic functionality like Cartesian manipulator control, zero force control, platform navigation, unified end effector access, and visual servoing. This skill

library can be used by parameterizing related components, by extending them and/or by re-implementing the corresponding interfaces.

5.2 MemoryX

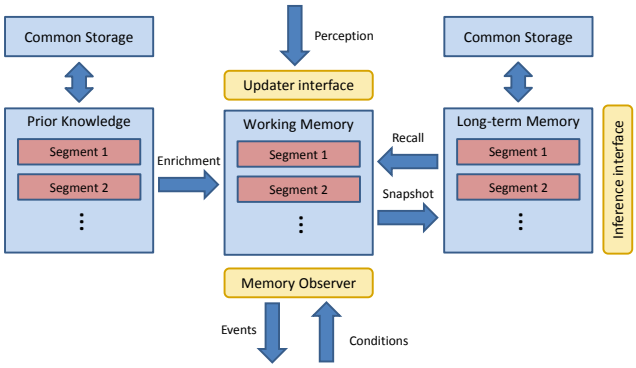


Figure 5: The offered MemoryX architecture consisting of a working memory and a long-term memory. Both memories can be accessed within the distributed application. Appropriate interfaces allow attaching processes to the memory for updating and inference.

MemoryX, the memory layer of ArmarX, includes basic building blocks for memory structures which can either be held in the system's memory or made persistent in a non-relational database [WVW+13, WKK+13]. Based on these building blocks, the memory architecture illustrated in Figure 5 is realized. A key element of MemoryX are network transparent access facilities which allows consistently updating or querying the memory within the distributed application. The architecture consists of different memory types such as working memory (WM), long-term memory (LTM), and prior knowledge (PK). Each memory type is organized in segments which can be individually addressed and used to store arbitrary data types or classes. Each segment may cover data related to classes of known objects, perceived locations of object instances or positions and status of agents in the world.

- The *Working Memory* represents the current context of the robot's internal state. It can be updated by perceptual processes or prior knowledge via an updater interface.
- *Prior Knowledge* contains information which is already known to the robot and which has been predefined by the user or operator. Entities in the PK can be enriched with known data such as 3D models or features for object recognition.
- The *Long-Term Memory* provides long-term storage capabilities and offers an inference interface which allows attaching learning and inference processes. The LTM allows creating snapshots of the current WM state to be used for later re-loading.

Although PK and LTM provide capabilities for long-term data storage, both memories differ in their conceptual role within MemoryX. While the PK holds user generated data and provides enrichment methods for entities of the WM, the LTM is used to create snapshots of the current content of the WM in order to recall them later and/or use them for inference processes.

As depicted in Figure 5, the different memories can be organized in segments. MemoryX provides several types of segments which are listed below:

- The *Object Classes* segment holds information on classes of objects and their relation. Enrichment with 3D models and or localization methods are provided.
- In the *Object Instances* segment information is stored about the positions of actual object instances which have for example been recognized by the perception system. These instance entities link to the corresponding object types within the Object Classes segment.
- The *Agent* segment stores information related to agents (e.g. robots, humans) in the world. The robot itself and any other operators, users, or robots relate to entries in this memory segment.

This memory structure allows the realization of powerful update mechanisms. For example, a perception component localizes a known object and updates the WM. This update creates a new entity of the object with the current location and automatically enriches it with the 3D model from the prior knowledge database. Besides the possibility of directly addressing the WM, an observer exists which allows installing conditions based on the memory content. If the associated content changes the matching events will get generated and processed by the according states of the robot program.

Note, that the memory components are expandable in order to customize the memory structure. However, when a lightweight framework is sufficient, it is possible to use only a subset of the components in order to avoid bloated setups.

5.3 VisionX

With VisionX we provide a perception building block in the robot framework layer which offers facilities for sensor processing in the distributed robot application. Processing currently covers stereo camera images, RGB-D data and offline video streams coming from persistent data files. VisionX consists of interfaces for image providers and image processors as illustrated in Figure 6. The image provider abstracts the imaging hardware and provides the sensor data as a stream either via shared memory or over Ethernet. Different image processors can be implemented that realize tasks related to object perception, human perception, and scene perception. The MemoryX update interface is used to write the processing results to the working memory. Optionally, perception confidence and probabilistic measures can be stored additionally in order to support quality queries and to allow for probabilistic reasoning.

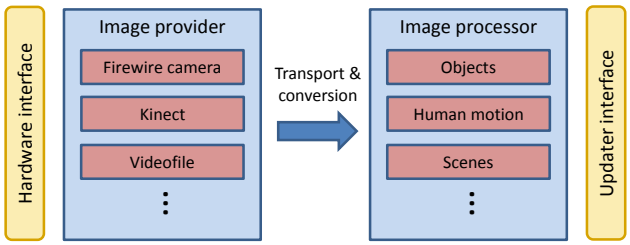


Figure 6: Image processing in the VisionX robot API. The image provider abstracts the hardware and streams the data via shared memory or Ethernet to the image processor. The processing result is written to the working memory.

6 The ArmarX Toolset

ArmarX is equipped with a set of tools designed to ease the development process in the creation and debugging phase. ArmarXPackage, a command line tool to easily create and manage own projects, several GUI viewer tools to inspect the system state on different levels, and a simulator are the key applications of the ArmarX toolset.

6.1 ArmarXPackage

Creating new C++ projects by hand is a tedious and often repetitive task. It is common practice to recycle an existing project structure and adapt it to new demands, often differing in details only. On this account, ArmarX offers the command line tool ArmarXPackage to easily create and manage user projects that depend on ArmarX. With this tool, standard ArmarX components like network accessible objects, statecharts or gui-plugins are created from templates and integrated into the project structure.

6.2 ArmarXGui

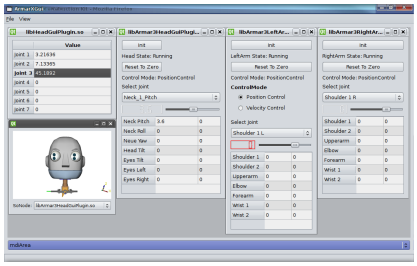
Disclosure of the system state is an important aspect of the ArmarX RDE and allows programmers to access data of many parts of the system they are working on. This data is required for debugging, monitoring and profiling purposes. Since, the amount of available data increases with the size of the developed system an abstraction of the data flow into easy to grasp visualizations is required.

To allow creating visualizations easily, we provide an extensible graphical user interface (GUI) named *ArmarXGui* which is based on the well-established Qt framework. An ArmarXGui instance provides extension points for loading and displaying custom user sup-

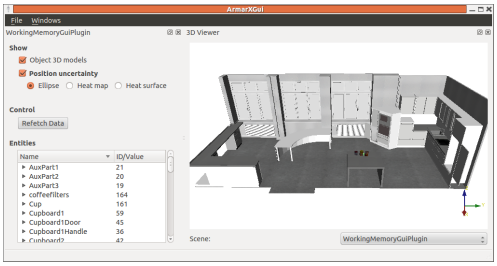
plied plugins. Each plugin has the ability to communicate with the ArmarX framework and offers a visualization which is displayed in the GUI's window. A variety of ready-to-use GUI plugins is already available within ArmarX, such as LogViewer, data plotter, 3D visualization widgets or MemoryX inspection tools.

An exemplary GUI layout is presented in Figure 7(a) containing a 3D visualization and plugins related to several sensor-actor units of the robot. These sensor-actor plugins can be used for emitting control commands as well as for visualizing internal data such as joint values. Below, an exemplary selection of GUI plugins are listed:

- **LogViewer** This plugin allows displaying, grouping, filtering, and searching of all gathered log messages emmitted by ArmarXObjects anywhere in the distributed system. Furthermore, it is possible to inspect extended information such as component state, application IDs and back tracing information.
- **Working Memory** The content of the robot's working memory can be visualized in a 3D view of the current environment. Figure 7(b) depicts an exemplary snapshot of the current content of a robot's working memory.
- **StatechartViewer** Since one statechart design principle has been state disclosure, it is possible to extract and visualize the program logic of an ArmarX robot program at runtime. With the StatechartViewer plugin it is possible to visualize each hierarchy level in the statechart of a robot program. All substates within the hierarchy level, connecting transitions as well as input- and output parameters of each state can be inspected. Both data flow and the currently active state at each hierarchy level are continuously updated.
- **Prior Memory Object Editor** This plugin offers a convenient way to add, inspect and update the object data that is stored in the *Prior Memory*. Properties such as the 3D object model, the recognition method, the motion model or parent object types can be viewed and changed.



(a) Gui with 3D visualization and sensor-actor unit plugins.



(b) Gui plugin for inspecting the current content of the robot's Working Memory.

Figure 7: ArmarXGui provides a variety of ready-to-use widgets and can be extended by a plugin mechanism in order to customize graphical user interaction

6.3 ArmarX Simulator

In order to ease development of robot applications it is of high interest that an RDE provides mechanisms to simulate program flow and robot behavior. Although a simulation environment cannot calculate the fully correct physical interaction, sensor information or execution timing, the structural setup of a robot program, in particular when distributed components are used, and an approximated physical robot behavior can be tested before the application is executed on the real robot. The *ArmarX Simulator* provides such a simulation environment within the ArmarX framework. The simulator comprises simulations of motors, sensors and dynamics and communicates with the ArmarX framework by implementing the robot's *KinematicUnit* interfaces in order to receive motor commands and to serve sensor feedback. This enables developers to run a robot program completely in simulation in order to test and inspect the program logic. An exemplary scene is shown in Figure 8.

Figure 8: The KIT kitchen scene in the ArmarX Simulator.

7 Conclusion

The robot development environment ArmarX provides infrastructure for building customized robot frameworks to ease the development of distributed robot software. The layered architecture of ArmarX supports lower level functionalities like access to real time components (*ArmarX RT*) and core features like communication, deployment, start-up, and error handling (*ArmarX Middleware Layer*). Additionally, ArmarX offers ready-to-use framework components for perception, memory, and control that can be used with little programming effort but remain open for modification and extension (*ArmarX Robot Framework Layer*). These customizable building blocks for high level robot control can be used to build a robot software framework and to tailor it according to robot capabilities and demands of potential robot applications. Finally, the ArmarX toolset offers a variety of components to support the implementation and debugging of complex robot applications.

8 Acknowledgement

This work was partially supported by the European Union Seventh Framework Programme under grant agreement no. 270273 (Xperience) and the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre Invasive Computing (SFB/TR 89).

The authors would like to thank all members and students of the Humanoids group at KIT for their various contributions to this work.

References

[AAV+08] T. Asfour, P. Azad, N. Vahrenkamp, K. Regenstein, A. Bierbaum, K. Welke, J. Schröder, and R. Dillmann. Toward Humanoid Manipulation in Human-Centred Environments. *Robotics and Autonomous Systems*, 56:54–65, January 2008. 1

[ASK08] Noriaki Ando, Takashi Suehiro, and Tetsuo Kotoku. 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, pages 87–98, Berlin, Heidelberg, 2008. Springer-Verlag. 1

[ASP+13] T. Asfour, J. Schill, H. Peters, C. Klas, J. Bücker, C. Sander, S. Schulz, A. Kargov, T. Werner, and V. Bartenbach. ARMAR-4: A 63 DOF Torque Controlled Humanoid Robot. In *IEEE/RAS International Conference on Humanoid Robots (Humanoids)*, 2013. 1

[BSK03] Herman Bruyninckx, Peter Soetens, and Bob Koninckx. The Real-Time Motion Control Core of the Orocos Project. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 2766–2771, 2003. 1

[Hen04] M. Henning. A new approach to object-oriented middleware. *Internet Computing, IEEE*, 8(1):66–75, 2004. 3.1

[MFN06] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. YARP: Yet Another Robot Platform. *International Journal on Advanced Robotics Systems*, 2006. 43–48. 1

[QCG+09] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009. 1

[VKU+12] N. Vahrenkamp, M. Kröhnert, S. Ulbrich, T. Asfour, G. Metta, R. Dillmann, and G. Sandini. Simox: A Robotics Toolbox for Simulation, Motion and Grasp Planning. In *International Conference on Intelligent Autonomous Systems (IAS)*, pages 585–594, 2012. 5.1.1

[WKK+13] K. Welke, P. Kaiser, A. Kozlov, N. Adermann, T. Asfour, M. Lewis, and M. Steedman. Grounded Spatial Symbols for Task Planning Based on Experience. In *IEEE/RAS International Conference on Humanoid Robots (Humanoids)*, pages 474–491, 2013. 5.2

[WVW+13] K. Welke, N. Vahrenkamp, M. Wächter, M. Kroehnert, and T. Asfour. The ArmarX Framework - Supporting high level robot programming through state disclosure. In *INFORMATIK Workshop*, 2013. 1, 2, 3, 3.1, 3.2, 5.2