The Robot Software Framework ArmarX

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

Abstract: With ArmarX we introduce a robot programming environment that has been developed in order to ease the realization of higher level capabilities needed by complex robotic systems such as humanoid robots. ArmarX is built upon the idea that consistent disclosure of the system state strongly facilitates the development process of distributed robot applications. We show the applicability of ArmarX by introducing a robot architecture for a humanoid system and discuss essential aspects based on an exemplary pick and place task. With several tools that are provided by the ArmarX framework, such as graphical user interfaces (GUI) or statechart editors, the programmer is enabled to efficiently build and inspect component based robotics software systems.

ACM CCS: Software and its engineering → Software organization and properties → Contextual software domains → Software infrastructure → Middleware; Computer systems organization → Embedded and cyber-physical systems → Robotics

Keywords: Robotics Software Framework, Robot Development Environment, Component-based Software Development, Software for Humanoid Robots.

1 Introduction

Robotic platforms in the service and assistive robotics area have made tremendous progress in the last decade regarding integration of motor and sensory capabilities. Complex motor capabilities such as locomotion, two-arm and dexterous manipulation combined with rich sensory information from visual, auditory, and haptic systems allow us to push the limits towards more dynamical and interactive areas of application. Both flexibility and adaptability of these platforms constantly increases through a steady increase of onboard computing power as well as the availability of distributed and cloud computing facilities.

In order to benefit from these developments and to establish current service and assistive robots in our daily life, required algorithms and software tools need to co-develop in a similar way. To support the development and integration of all required capabilities of such robots is the goal of robot development environments (RDEs). On one side, RDEs should provide the glue between different functionalities and capabilities of the robot software in terms of system architecture and communication. On the other side, RDEs should provide a programming interface allowing roboticists to include new capabilities at an appropriate level of abstraction with a minimum amount of training effort.

Several RDEs have been presented in the past decades accompanying the development of robotic platforms.

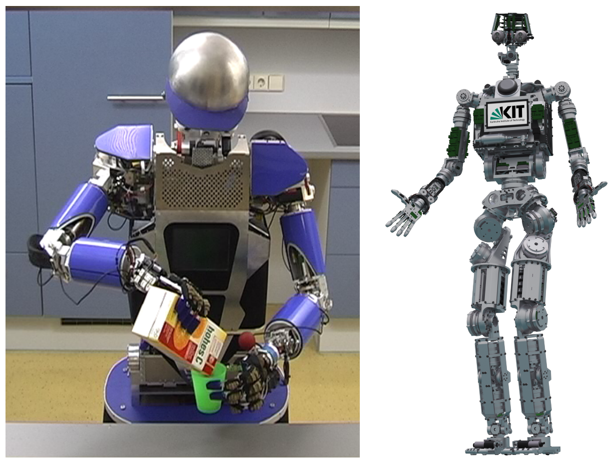


Figure 1: The humanoid robots ARMAR-IIIa [1] and ARMAR-4 [2].

Depending on the target platform and its intended application, a different level of abstraction is realized by each RDE. Several RDE frameworks such as OpenRTM [3], MatLab/Simulink®, MCA [4] focus on the control level. Others focus on the implementation of higher level system capabilities (e.g. ROS [5] and Orocos [6]). Yarp [7], the software framework of the iCub robots, provides low level communication features on top of which higher level robot capabilities are implemented. RDEs with message-based communication mechanisms, like ROS and YARP, provide simple but convenient communication methods that allow passing data between components. In addition, ROS and YARP provide mechanisms for remote procedure calls. Compared to these approaches, the RDEs Orocos and OpenRTM rely on the CORBA middleware which supports *remote method invocation*. This concept allows operating on remote objects as if they were available locally. Since CORBA is known to be complex and implementations do not cover the full feature set of the specification [8], we decided to use Ice [9] as communication middleware for ArmarX as described in the next section.

In addition to such technical design choices, we think it important to provide an initial set of structured and extensible components communicating via well-defined interfaces. Compared to ROS, which provides just basic communication mechanisms, but a huge set of components, such structuring elements support the concept of exchangeable and reusable components. Based on this idea, ArmarX provides generic framework elements that can be used to establish a robot specific software framework as well tools to support the development of robot applications.

Robot programs for complex robots, such as humanoids, are difficult to handle, in particular when the control flow is not disclosed. To cope with this problem, statecharts [10] offer a convenient and robust way of representing robot programs. ROS offers a Python-based library for specifying models of robotic behaviour called SMACH [11] that allows realizing robot programs as flowcharts. With Orocos the restricted Finite State Machine (rFSM) model can be used to coordinate the execution flow of a robot program. More details and a good survey on different statechart mechanisms in the robotics context can be found in [11].

Most state-of-the-art RDEs fulfill preconditions like platform independence and distributed processing required to support a variety of robotic platforms. Interoperability and open source availability are necessary in addition to ease software integration. Assistance in developing higher level skills is another critical aspect which essentially requires disclosure of the system state. Access to the current system state on all levels enables online inspection and debugging facilities while reducing training times of new developers significantly.

Disclosure of the system state is one of the key ideas behind the ArmarX RDE. ArmarX makes use of *well-defined interface definitions* and *flexible communication mechanisms* in contrast to traditional message passing protocols. In this work we will show how to establish disclosure of the system state as well as an application programming interface (API) at a high level of abstraction on top of the previously mentioned mechanisms. To this end, we will introduce a robot architecture for a complex humanoid system (see Figure 1) based on the ArmarX framework and discuss essential aspects based on an exemplary pick and place task.

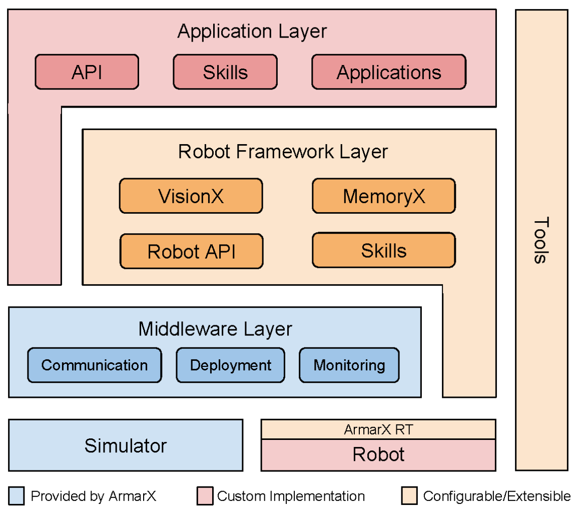


Figure 2: ArmarX is organized in three layers. The *Middleware Layer* provides all core facilities to implement distributed applications as well as basic building blocks for robot software architectures. Based on these building blocks, the *Robot Framework Layer* provides a robot API implementing more complex functionality like kinematics, memory, and perception. Robot specific APIs can be implemented by extending the provided generic robot API modules. Robot programs are realized in the *Application Layer*. They are implemented as distributed applications, making use of the generic and specific robot APIs and statecharts. The ArmarX Tools comprise a plugin-based GUI that can communicate with the components and visualize their content. Specialized components can interact with the ArmarX Simulator or the robot hardware via ArmarX RT.

2 ArmarX Overview

2.1 Design Principles

In this section we are going to provide a brief discussion of several essential prerequisites and design principles any RDE must meet in order to be applicable to today's platforms in robotics research and to match current development processes.

- **Distributed processing**
  The integrated hardware architectures of typical robotic platforms consist of several embedded PCs. Each PC usually interfaces with a set of robot subsystems and might contain specialized hardware such as CAN cards, camera interfaces, or high performance computing facilities. Therefore, robot programs are inevitably distributed by nature in order to support such heterogeneous systems. Consequently, all ArmarX applications are distributed and communication is realized using either Ethernet or shared memory to allow transparent distribution of robot program parts onto the available hardware.
- **Interoperability**
  Both Hard- and software used in today's robotic platforms vary and are far from being standardized. Hence, it is essential for RDEs to allow bridging these gaps by providing interoperability through supporting heterogeneous hardware platforms and operating systems. This enables easy integration of new hardware components without the necessity to adapt the

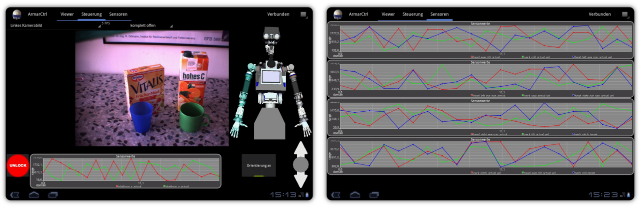


Figure 3: ARMAR-III control GUI for Android. Interfacing with ArmarX applications is based on an interface definition language (IDL) which supports a variety of hardware platforms and operating systems using different programming languages.

RDE to a new platform. To account for this requirement, the ArmarX RDE can be compiled under Linux, Windows, and Mac OS X. A distributed application can be built out of the box spanning platforms running any of the mentioned operating systems. Further, ArmarX uses an interface definition language (IDL) supporting a variety of platforms and programming languages (C++, Java, C#, Objective-C, Python, Ruby, PHP, and ActionScript). Thus, the covered hardware can be easily extended by implementing these interfaces on the target hardware using its primary programming language. These capabilities facilitate interfacing with the robot using for example mobile devices as illustrated in Figure 3.

- **Open source**
  ArmarX is available open source under the GPL license. Providing RDEs under an open source license is essential in order to achieve the most impact on robotics by allowing researchers and developers to achieve a deep insight in the underlying mechanisms. Additionally, feedback and experience from projects are easy to integrate in the RDE development process.

One other key design principle of ArmarX is disclosure of the system state. As stated in the introduction, this idea is supported by two mechanisms on the technical level: well-defined interfaces and flexible communication mechanisms. In subsequent chapters, we will further develop this idea and show how disclosure of the system state is established for the whole system, including the distributed application, the robot program, the robot kinematics, and the internal model of the world.

2.2 System Architecture

ArmarX is organized in three layers as illustrated in Figure 2: the *Middleware Layer*, the *Robot Framework Layer*, and the *Application Layer*. The Middleware Layer abstracts communication mechanisms, provides basic building blocks for distributed applications, and defines entry points for visualization and debugging. *ArmarX RT* establishes a bridge to real time components, needed for accessing low level robot control modules. The Robot Framework Layer comprises several projects providing access to sensori-motor capabilities on a higher level. These projects include memory (MemoryX), robot and world model, perception (VisionX), and execution modules. Robot specific APIs can be implemented by extending and further specializing the generic robot API building blocks. Usually, such robot specific APIs include interface implementations for accessing the lower sensori-motor levels and models dedicated to the actual robot. This layer can also include specialized memory structures, instances of perceptual functions, and control algorithms. The final robot program is implemented in the Application Layer as a distributed application, making use of the generic and specific robot APIs.

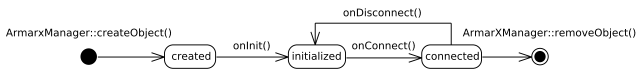


Figure 4: Each ArmarXManager can create ArmarXObjects that are able to communicate with any object in the distributed application. Explicit ArmarXObject dependencies allow monitoring its connectivity. If all dependencies are resolved, the object enters the *connected* state. Losing a dependency returns the object to the *initialized* state to ensure consistency among objects in the distributed application.

We will step through this architecture bottom-up, starting with the Middleware Layer in the subsequent section and the Robot Framework Layer in Section 4. An example for a robot specific API for the humanoid robot ARMAR-III [1] is discussed in Section 5. The application of this API is demonstrated in a pick-and-place scenario.

3 The Middleware Layer

3.1 Communication in ArmarX

The Middleware Layer builds upon the ZeroC Internet Communication Engine (Ice) [9] as distributed computing platform to provide the basic building blocks for implementing distributed robot architectures. The Ice platform implements distributed processing over Ethernet in a platform and programming language independent manner. With ArmarX, we extended this network based-communication mechanism by a shared-memory channel in order to allow efficient transfer of large data blocks in a transparent way. Further, Ice provides the Slice interface definition language (IDL) for defining network transparent interfaces and classes. Several communication modes such as synchronous and asynchronous remote procedure calls, remote method invocation, or publisher-subscriber are supported.

Two key components are offered by the ArmarX Middleware Layer: the *ArmarXObject* and the *ArmarXManager*. The ArmarXObject is the basic building block of the distributed application. It is reachable within an ArmarX distributed application via aforementioned communication mechanisms. Each process contains an ArmarXManager for creating ArmarXObjects and handling their lifecycle (see Figure 4). Explicit ArmarX-Object dependencies allow the ArmarXManager to guarantee the consistency of the distributed application. Only if all dependencies of an ArmarXObject are fulfilled, the *connected* state is entered. If a dependency is lost, the ArmarXObject returns to the *initialized* state and waits until all dependencies become available again. The lightweight ArmarXManager can easily be integrated into existing applications, thus supporting interoperability and integration of third-party software.

Additionally, the Middleware Layer provides a number of essential tools such as transparent shared-memory, Ethernet transfer of data, and thread pool based threading facilities. Further, several bridging mechanisms to real time components are provided with *ArmarX RT* allowing convenient communication with time critical modules such as the low level robot control layer.

3.2 Building Blocks of the Robot Application

An outline of the three basic elements of the application programming interface (API) is illustrated in Figure 5. The lowest level of abstraction is the *Sensor-Actor Unit* serving as abstraction of robot components. *Observers* monitor sensory data streams and generate application specific events which trigger *Operations* to issue control commands to the robot.

- **Sensor-Actor Units**
  Sensor-Actor Units provide abstractions of robot components such as kinematic structures or cameras. Within the ArmarX distributed application, sensory data is made available using a publisher-subscriber mechanism whereas the control interface is realized using remote method invocation. Sensor-Actor Units provide a request-release mechanism for handling concurrent access. Due to this unique representation of control and sensory data, other components such as Observers are enabled to communicate with these units in a standardized way.
- **Observers**
  The ArmarX API uses an event driven approach. API events originate from desired patterns in sensory data such as high motor temperatures or reaching a target pose. Observers generate these events by evaluating application defined conditions on the sensory data stream. These conditions are realized as distributed expression trees where each leaf corresponds to a check on specific sensor data. The API offers a set of basic checks and provides interfaces to support implementing more advanced and application specific checks easily.
- **Statecharts**
  One inherent design decision of ArmarX is to represent robot operations as state-transition networks which are implemented as hierarchical, distributed, and orthogonal statecharts (see [12] for more details on statecharts in robotics). Each state in the network is defined by a set of input/output parameters and can issue control commands or start asynchronous calculations. State transitions define the data flow between states by mapping output values of one state to input values of another state. These transitions are triggered by events issued from Observers. Every statechart is embedded into an ArmarXObject and can be used as a substate in other statecharts (see Figure 9).

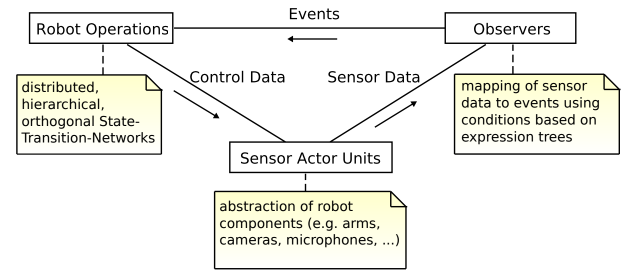

Events
Robot Operations
Observers
Control Data
Sensor Data
distributed, hierarchical, orthogonal State-Transition-Networks
mapping of sensor data to events using conditions based on expression trees
Sensor Actor Units
abstraction of robot components (e.g. arms, cameras, microphones, ...)

Figure 5: The application programming interface provided by the ArmarX Middleware Layer comprises four different elements. The *Sensor-Actor Units* serve as abstraction of robot components, the *Observers* generate events from the continuous sensory data stream resulting in transitions between *Operations.* Operations are organized as hierarchical state-transition networks. These elements are connected by the *communication* mechanisms (arrows in the figure).

4 The Robot Framework Layer

4.1 Overview

The Robot Framework Layer of the ArmarX architecture comprises robot definitions and framework components that allow interfacing uniformly with perception modules and memory structures. Several generic and ready-to-use API components can be parametrized and customized in order to adapt the API for a specific robot as shown in Section 5. In the following, we will briefly discuss these components.

4.2 Robot Kinematics and Dynamics

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 which are compatible with the robot simulation toolbox *Simox* [13]. This allows making use of all Simox related functionality, such as collision detection, motion and grasp planning or defining reference frames for transforming poses to different coordinate systems. Further, ArmarX provides mechanisms to conveniently access a shared and network transparent data structure of the robot model. This enables all distributed components to query for robot internal data such as joint values or to perform coordinate conversions.

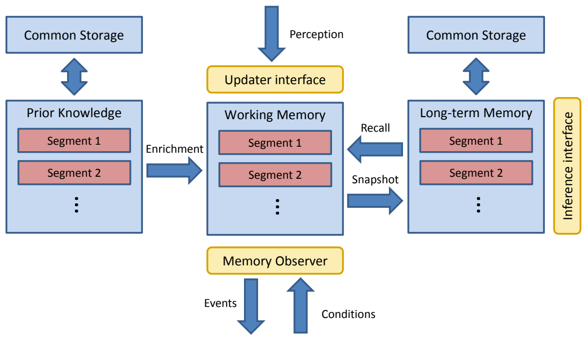


Figure 6: The Robot Framework Layer offers the MemoryX architecture consisting of working memory, long-term memory, and a prior knowledge component. All memories are accessible within the distributed application. Appropriate interfaces allow attaching processes to the memory for updating and inference.

4.3 MemoryX

MemoryX comprises all memory related components of the ArmarX Robot Framework Layer. These components include basic building blocks for memory structures which can be either held in the system's memory or made persistent in a database [14]. A memory architecture comprising a working memory (WM) and a long-term memory (LTM) is realized using these building blocks (see Figure 6). Both memory types are organized in individually addressable segments containing arbitrary types or classes which are accessible within the distributed application. The WM is updated via an updater interface either by perceptual processes or by prior knowledge. Prior knowledge is stored in a non-relational database and allows enriching entities with known data (such as models or features). Besides directly addressing the WM, the working memory observer allows generating events based on changes of the memory content. The LTM offers an inference interface which allows attaching learning and inference processes.

4.4 VisionX

The perception components of the ArmarX Robot Framework Layer provide facilities for including camera based image processing in the distributed robot application. VisionX allows implementing image providers and image processors as illustrated in Figure 7. The image provider abstracts from imaging hardware and provides a data stream via shared memory or over Ethernet. Different image processors can be implemented that fall into the classes of object perception, human perception, and scene perception. Processing results are written to the working memory of MemoryX via the updater interface. An exemplary result of a processed image stream in the VisionX GUI plugin can be seen in Figure 8.

4.5 ArmarX Simulator

In order to ease development of robot applications it is essential that an RDE provides mechanisms to simulate

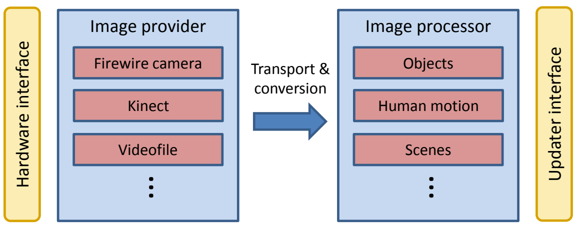


Figure 7: Image processing in VisionX. The image provider abstracts from hardware and streams data via shared memory or Ethernet to an image processor. Processing results are written to the working memory.

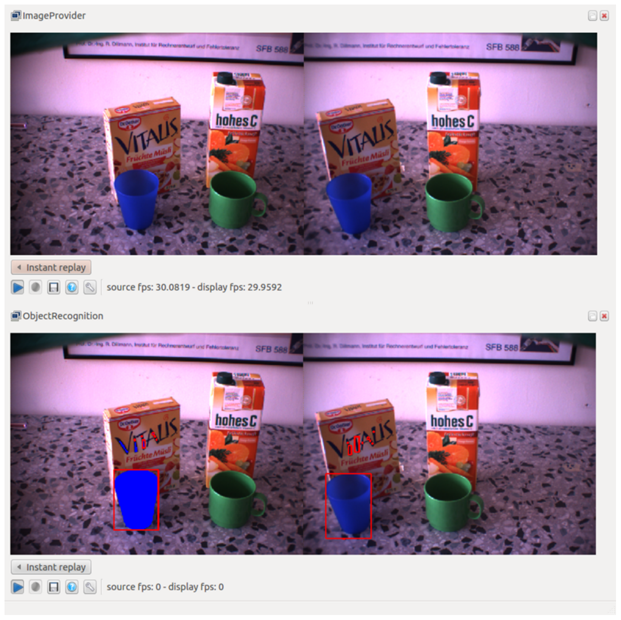

ImageProvider
Instant replay
source fps: 30.0819 - display fps: 29.9592
ObjectRecognition
Instant replay
source fps: 0 - display fps: 0

Figure 8: The VisionX image provider stream (top) and the corresponding output of an image processor related to single colored object recognition (bottom).

program flow and robot behavior. Despite being unable to provide realistic and reliable information regarding physical interaction, sensor information, or execution timing, a simulation environment helps in testing the structural setup of a robot program, in particular when distributed components are used, before executing the application on the real robot. The ArmarX Simulator component provides such a simulation environment within the ArmarX framework (see Figure 14). The simulator comprises simulations of motors, sensors, and dynamics. It communicates with the ArmarX framework by implementing the robot's *Sensor-Actor Unit* interfaces in order to receive motor commands and provide sensor feedback. This enables developers to run robot programs completely in simulation in order to test and inspect its program logic.

4.6 Robot Program

Statecharts are used to organize any robot program responsible for the overall robot behavior. The execution

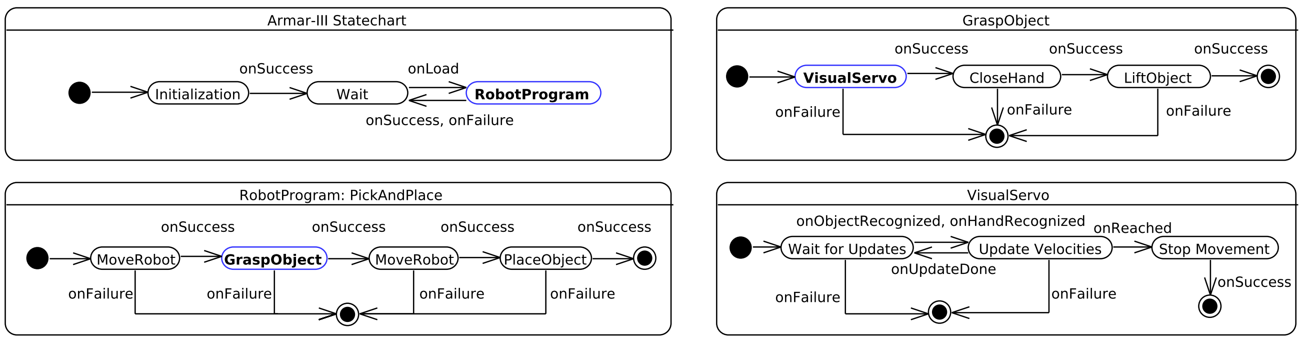


Figure 9: An excerpt of operations involved in a pick and place task on ARMAR-III. The operations are realized using the ArmarX hierarchical statechart mechanism. On each hierarchy level of this example the state highlighted in blue is further refined.

of such a robot program is embedded in a robot application defining the component setup which includes configuration settings. All program logic is represented as one comprehensive statechart consisting of many operations which might be executed on different hosts due to the distributed nature of ArmarX. The root state of each robot program contains administrative substates for starting, stopping, and initializing the robot hardware. But most importantly, the root state is responsible for dynamically loading and executing program logic. It was a deliberate design choice to not use static loading of programs to allow dynamic reconfiguration at runtime. This feature is essential to support high level planning in ArmarX which requires executing dynamically created action chains.

5 Disclosure of System State: A case study on ARMAR-III

In this section we will show the disclosure of the system state by realizing a pick and place task for the ARMAR-III robot using the ArmarX framework. Robot specific extensions to the generic ArmarX robot API are discussed in the first part. A pick and place robot program is realized using these extensions in the second part. Note that only components needed for accomplishing pick and place actions are discussed here. Last, we present several mechanisms for disclosing the system state and discuss how to apply them in the context of the proposed robot program.

5.1 Robot API for ARMAR-III

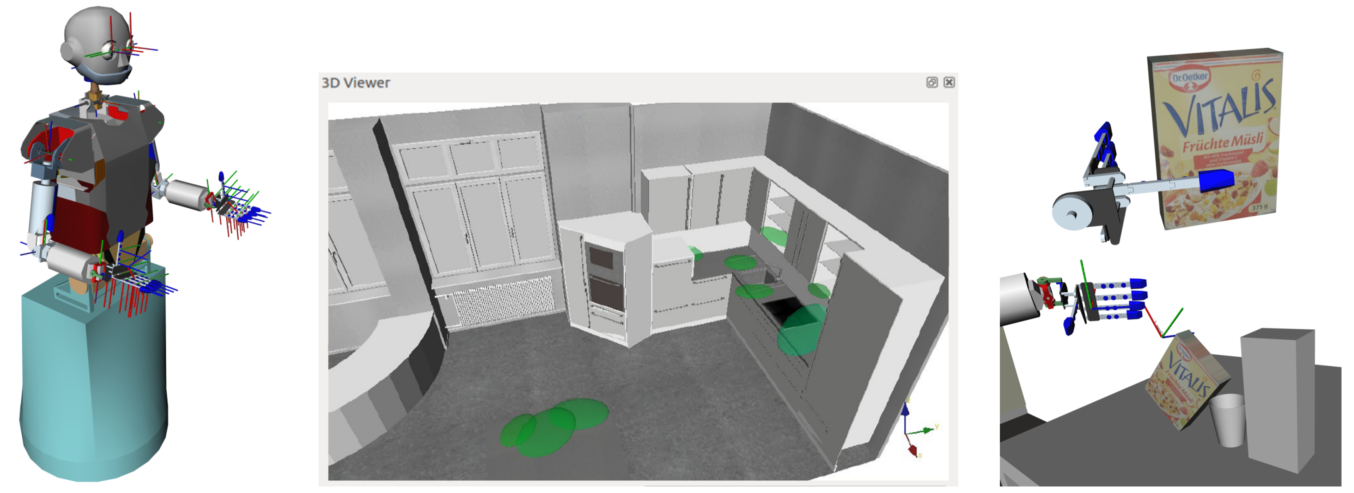
The generic ArmarX robot API of the Robot Framework Layer (see Section 4) provides several components which have to be customized for ARMAR-III. In most cases it is sufficient to provide an adequate set of parameters and configuration files which are processed by the ArmarX robot API components. Additionally, the generic robot API is extended with robot specific implementations for hardware access.

- **Kinematics and Dynamics**
  The robot kinematics and dynamics together with all relevant coordinate frames of ARMAR-III are specified via XML definition files. A visualization of the robot including all coordinate frames is shown in Figure 10(a).
- **Memory**
  The spacial segment of MemoryX contains knowledge about the robot's environment which is assumed to be known for the task. Additional known objects and grasping information are stored in the prior knowledge database.
- **Perception**
  The visual perception framework of ARMAR-III includes methods for camera based object recognition and localization for segmentable and textured objects using the methods described in [15]. Resulting object locations are stored and made available in the spatial memory segment of the working memory.
- **Hardware Access**
  Sensor-Actor Units are implemented to control arms, hands, head, and platform. These units connect to ARMAR's low level control framework for sending joint commands and receiving sensor information.
- **Simulation**
  Generic robot API components are configured in order to simulate Sensor-Actor Units for all robot components.

5.2 The Pick and Place Robot Program

A robot architecture for the humanoid robot ARMAR-III has been realized based on the ArmarX framework. Therefore, several components have been implemented based on the Middleware and Robot Framework Layers (see Section 5.1).

To realize the pick and placed robot program, 5 ARMAR-IIIa specific Sensor-Actor Units have been implemented. Beside these robot-specific implementations, several components of the generic robot API are used without modifications just by setting their parameters accordingly. Events are triggered by 7 Observers which



(a) Kinematic structure, 3D model and all relevant coordinate systems of ARMAR-III.

(b) The long term memory of the robot contains common objects locations and can be inspected with the MemoryX GUI plugin.

(c) Object with grasp definition and visualization of the left TCP coordinate system with target pose.

Figure 10: 3D visualizations of ARMAR-III and the kitchen environment in ArmarX augmented with additional information.

are associated with corresponding units or memory components. In addition, 4 memory components are required as well as 3 vision components and 3 general system components. All components can be exchanged without source code modifications since they are accessed via their generic interface descriptions located in the robot API package.

The hierarchically organized pick and place statechart contains 15 operations built from approximately 50 unique states. Some of these states are very versatile and therefore instantiated and configured multiple times. An excerpt of involved operations is illustrated in Figure 9. All operations are realized using the hierarchical ArmarX statechart mechanism. All state transitions are initiated by suitable observers which issue the required events.

5.3 Disclosure of system state

Disclosure of the system state is an important aspect of the ArmarX RDE. It allows programmers to access the data of many parts of the system 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.

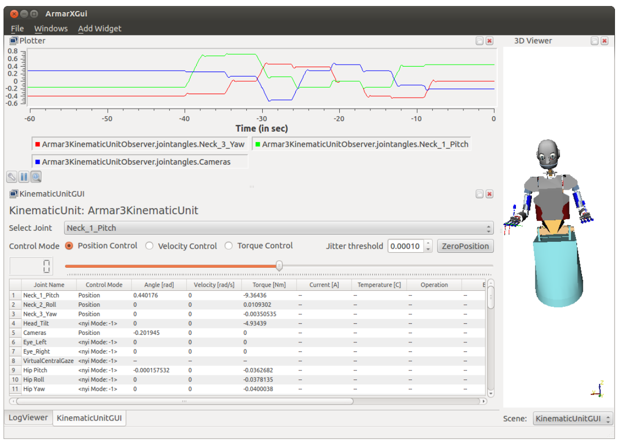
To allow creating visualizations easily, we provide an extensible graphical user interface (GUI) named *ArmarXGui* which is based on the well established Qt framework. An ArmarXGui instance provides extension points for loading and displaying custom plugins. Each plugin is able to communicate with the 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 the LogViewer, the data plotter or the 3D visualization. Mechanisms for disclosing the system state in ArmarX programs used by these plugins will be presented in the next section.

An exemplary GUI layout is presented in Figure 11 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.

5.3.1 Disclosure Mechanisms

ArmarX provides mechanisms to disclose the internal state of a robot program on different levels. These mechanisms include monitoring of sensory data, events, conditions, and the execution state of the distributed application. The following paragraphs explain a selection of available mechanisms and respective GUI plugins.

- **Logging**
  ArmarX provides a distributed logging facility, as already mentioned in Section 3.1. Each part of a distri-



KinematicUnit: Armar3KinematicUnit

Select Joint: Neck_1_Pitch

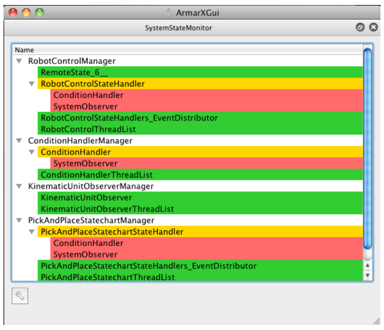
Control Mode: [✓ Position Control] Velocity Control Torque Control

Jitter threshold: 0.00010

ZeroPosition

	Joint Name	Control Mode	Angle [rad]	Velocity [rad/s]	Torque [Nm]	Current [A]	Temperature [C]	Operation	E
1	Neck_1_Pitch	Position	0.440176	0	-9.36436	--	--	--	--
2	Neck_2_Roll	Position	0	0	0.0109302	--	--	--	--
3	Neck_3_Yaw	Position	0	0	-0.00350535	--	--	--	--
4	Head_Tilt	<nyi Mode: -1>	0	0	-4.93439	--	--	--	--
5	Cameras	Position	-0.201945	0	0	--	--	--	--
6	Eye_Left	<nyi Mode: -1>	0	0	0	--	--	--	--
7	Eye_Right	<nyi Mode: -1>	0	0	0	--	--	--	--
8	VirtualCentralGaze	<nyi Mode: -1>	--	--	--	--	--	--	--
9	Hip Pitch	<nyi Mode: -1>	-0.000157532	0	-0.0362682	--	--	--	--
10	Hip Roll	<nyi Mode: -1>	0	0	-0.0378135	--	--	--	--
11	Hip Yaw	<nyi Mode: -1>	0	0	-0.0400038	--	--	--	--

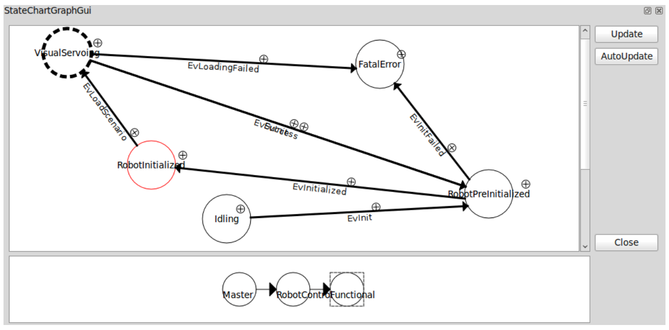
Figure 11: The ArmarXGui provides a variety of ready-to-use widgets and can be extended by user supplied plugins.


ArmarXGui

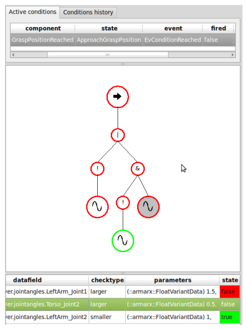
SystemStateMonitor

Name
- RobotControlManager
  - 🟩 RemoteState_6__
  - 🟨 RobotControlStateHandler
    - 🟥 ConditionHandler
    - 🟥 SystemObserver
  - 🟩 RobotControlStateHandlers_EventDistributor
  - 🟩 RobotControlThreadList
- ConditionHandlerManager
  - 🟨 ConditionHandler
    - 🟥 SystemObserver
  - 🟩 ConditionHandlerThreadList
- KinematicUnitObserverManager
  - 🟩 KinematicUnitObserver
  - 🟩 KinematicUnitObserverThreadList
- PickAndPlaceStatechartManager
  - 🟨 PickAndPlaceStatechartStateHandler
    - 🟥 ConditionHandler
    - 🟥 SystemObserver
  - 🟩 PickAndPlaceStatechartStateHandlers_EventDistributor
  - 🟩 PickAndPlaceStatechartThreadList

(a) Distributed robot program connectivity before the SystemObserver component is started.



(b) StatechartViewer for inspecting program logic and the current state.


Active conditions | Conditions history

component	state	event	fired
GraspPositionReached	ApproachGraspPosition	EvConditionReached	false

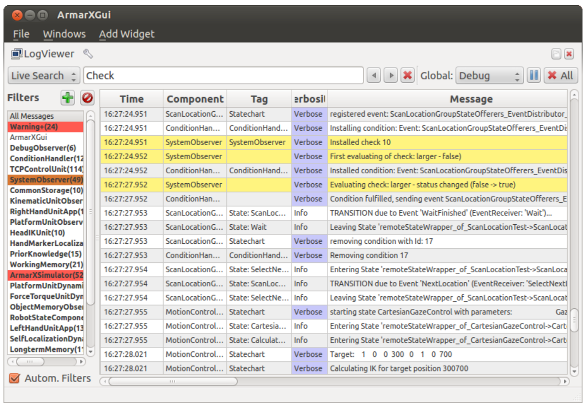
datafield	checktype	parameters	state
ver.jointangles.LeftArm_Joint1	larger	(::armarx::FloatVariantData) 1.5,	false
ver.jointangles.Torso_Joint2	larger	(::armarx::FloatVariantData) 0.5,	false
ver.jointangles.LeftArm_Joint2	smaller	(::armarx::FloatVariantData) 1,	true

(c) The ConditionViewer is used for inspecting conditions.

Figure 12: A selection of available ready-to-use GUI plugins supported by the plugin mechanism of ArmarXGui.

buted ArmarX robot program is capable of sending log messages which are accessible via the aforementioned Publish/Subscribe mechanism. A corresponding ArmarXGui LogViewer plugin is available which allows displaying, grouping, sorting, and searching of all gathered log messages. Additionally, it is possible to inspect extended information such as component state, application IDs and back tracing information (see Figure 13).

- **Application Dependencies**
  The system state of a distributed ArmarX robot program can be queried and visualized at runtime. As explained in Section 3.1, all ArmarXObject instances are accessible over Ice together with their explicit dependencies. Thus, a hierarchical view of the system's connectivity can be generated containing information about established and missing dependencies. Figure 12(a) shows a system state where all components except the SystemObserver have been started. This prevents the ConditionHandler from entering the *initialized* state which in turn blocks both RobotStateHandler and PickAndPlaceStatechartStateHandler components. Once the missing dependency can be resolved all remaining waiting connections can be established and both red and orange blocks will turn to green. Even though components are running on different hosts, their dependencies can be inspected from anywhere in the network.


ArmarXGui

File Windows Add Widget

LogViewer

Live Search: Check   Global: Debug   All

Filters
All Messages
Warning+(24)
ArmarXGui
DebugObserver(6)
ConditionHandler(12
TCPControlUnit(114)
SystemObserver(49)
CommonStorage(10)
KinematicUnitObser
RightHandUnitApp(1
PlatformUnitObserv
HeadIKUnit(10)
HandMarkerLocaliza
PriorKnowledge(15)
WorkingMemory(21)
ArmarXSimulator(52
PlatformUnitDynami
ForceTorqueUnitDyn
ObjectMemoryObse
RobotStateCompone
LeftHandUnitApp(13
SelfLocalizationDyn
LongtermMemory(1

Autom. Filters

Time	Component	Tag	erbosil	Message
16:27:24.951	ScanLocationG...	Statechart	Verbose	registered event: ScanLocationGroupStateOfferers_EventDistributor_
16:27:24.951	ConditionHan...	ConditionHand...	Verbose	Installing condition: Event: ScanLocationGroupStateOfferers_EventDi
16:27:24.951	SystemObserver	SystemObserver	Verbose	Installed check 10
16:27:24.952	SystemObserver		Verbose	First evaluating of check: larger - false)
16:27:24.952	ConditionHan...	ConditionHand...	Verbose	Installed condition: Event: ScanLocationGroupStateOfferers_EventDis
16:27:27.952	SystemObserver		Verbose	Evaluating check: larger - status changed (false -> true)
16:27:27.952	ConditionHan...		Verbose	Condition fulfilled, sending event ScanLocationGroupStateOfferers_E
16:27:27.953	ScanLocationG...	State: ScanLoc...	Info	TRANSITION due to Event 'WaitFinished' (EventReceiver: 'Wait')...
16:27:27.953	ScanLocationG...	State: Wait	Info	Leaving State 'remoteStateWrapper_of_ScanLocationTest->ScanLocat
16:27:27.953	ScanLocationG...	Statechart	Verbose	removing condition with Id: 17
16:27:27.953	ConditionHan...	ConditionHand...	Verbose	Removing condition 17
16:27:27.954	ScanLocationG...	State: SelectNe...	Info	Entering State 'remoteStateWrapper_of_ScanLocationTest->ScanLoca
16:27:27.954	ScanLocationG...	State: ScanLoc...	Info	TRANSITION due to Event 'NextLocation' (EventReceiver: 'SelectNextl
16:27:27.954	ScanLocationG...	State: SelectNe...	Info	Leaving State 'remoteStateWrapper_of_ScanLocationTest->ScanLocat
16:27:27.955	MotionControl...	Statechart	Verbose	starting state CartesianGazeControl with parameters: Gaz
16:27:27.955	MotionControl...	State: Cartesia...	Info	Entering State 'remoteStateWrapper_of_CartesianGazeControl->Cart
16:27:27.955	MotionControl...	State: Calculat...	Info	Entering State 'remoteStateWrapper_of_CartesianGazeControl->Cart
16:27:28.021	MotionControl...	Statechart	Verbose	Target: 1 0 0 300 0 1 0 700
16:27:28.021	MotionControl...	Statechart	Verbose	Calculating IK for target position 300700

Figure 13: The LogViewer GUI plugin provides a structured view of all components in the network providing convenient additional debugging information and filtering options.

- **Conditions**
  Section 3.2 talked about the observer mechanism provided by ArmarX. Distributed conditions are used to trigger events upon failure or fulfillment. The ConditionViewer plugin helps in analyzing currently active and already expired conditions. This plugin visualizes arbitrarily large, boolean conditions as a tree-graph structure (see Figure 12(c)). Colors are used to visualize the current state of a complete condition and its subterms. Green is used for fulfilled terms, while red is used to highlight unfulfilled terms. In the context of the pick and place scenario such a condition might be a check for the distance between the current end effector pose and the target grasping pose. Once this distance falls below a defined threshold, the condition is met and an event is triggered resulting in a switch to the next state (i.e. close the hand for grasping).
- **Statecharts**
  The ArmarX statecharts have been described in detail in Section 3.2. Since one statechart design principle is state disclosure, it is possible to extract and visualize the program logic of an ArmarX robot program at runtime. This can be done with the StatechartViewer plugin depicted in Figure 12(b). Each hierarchy level in the statechart of a robot program can be visualized with this plugin. All substates within the hierarchy level, connecting transitions as well as input- and output parameters of each state are shown. Both data flow and the currently active state at each hierarchy level are continuously updated. Additionally, user interactions with the statechart are provided. It is possible to trigger transitions manually or to insert breakpoints in specific states to halt the complete robot program for debugging purposes.
- **Memory**
  Besides control flow, the memory contents is another important aspect of a robot's state. It is possible to inspect all memory types and their segments. Multiple

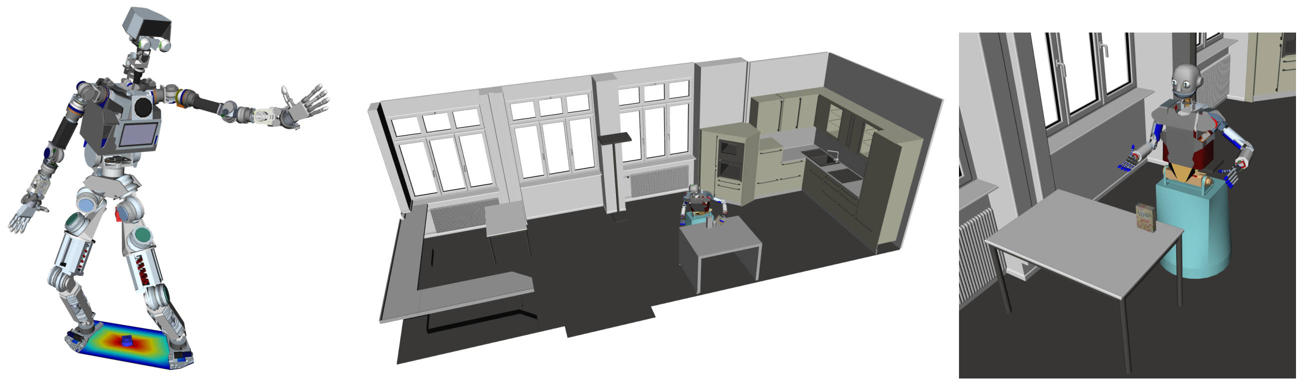


Figure 14: The humanoid robot ARMAR-4 [2] and ARMAR-III [16] in the ArmarX Simulator environment. The left image shows ARMAR-4 with a visualization of the robot's support polygon and its center of mass projected to the ground. In the middle and on the right, the KIT kitchen environment with the humanoid robot ARMAR-III is shown while a pick-and-place operation is performed.

plugins are available to visualize different aspects of the memory structure. The current environment stored in the robot's working memory can be visualized in a 3D view. Figure 10(b) depicts the memory state of ARMAR-III for learned common object locations. The object segment of the long-term memory (see Figure 6) holds information about known objects, such as visual features, 3D models or precomputed grasping data. Figure 10(c) shows a visualization of the 3D model of a cereal box and an associated grasp which can be used as a target coordinate frame for grasping. The current state of relevant coordinate systems (left TCP and target pose) can be plotted as shown in Figure 10(c), which allows inspecting spatial data channels that are considered by the observers. In addition, ArmarX supports a generic concept of reference frames. Any coordinate system can be used as reference frame, including the ones defined in the robot model (see Figure 10(a)). The long term memory of the robot for the common object locations can be inspected as shown in Figure 10(b).

- **ArmarX Simulator**
  The execution of the robot program can be analyzed within the ArmarX Simulator environment which supports physical interaction with the environment. Since both motor commands and sensor readings are transparently connected with the physics simulation, the execution of the robot program can be inspected under realistic conditions (see Figure 14). Hence, ArmarX offers the possibility to test the behavior of implemented robot programs in simulation before executing them on the real robot hardware. No changes to application level code are required for this step.

6 Conclusion

We presented the robot development environment ArmarX and showed how higher level capabilities of complex robots can be realized in heterogeneous computing environments. The ArmarX framework provides mechanisms for developing distributed robot applications while offering support to disclose the system state on all abstraction levels. Basic functionality is covered by a generic communication framework (the ArmarX Middleware Layer) on top of which the Robot Framework Layer provides a generic robot API, which can be further specialized and adapted to the demands of a specific robot. Based on an exemplary pick and place task, we showed how this specialization can be performed for a humanoid robot. Further, we presented several tools, such as graphical user interfaces and inspection plugins. These tools enable users and developers to visualize and monitor the current system state through the available disclosure mechanisms of ArmarX.

Acknowledgement

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement no. 270273 (Xperience) and grant agreement no 611832 (WALK-MAN), and from the German Research Foundation (DFG: Deutsche Forschungsgemeinschaft) 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.

Literature

[1] 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*, vol. 56, pp. 54–65, January 2008.

[2] 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, pp. 390–396.

[3] N. Ando, T. Suehiro, and T. 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*, ser. SIMPAR '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 87–98.

[4] MCA2, “Modular controller architecture,” http://www.mca2.org/.

[5] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “ROS: an open-source Robot Operating System,” in *ICRA Workshop on Open Source Software*, vol. 3, no. 3.2, 2009, p. 5.

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

[7] G. Metta, P. Fitzpatrick, and L. Natale, “YARP: Yet another robot platform,” *International Journal on Advanced Robotics Systems*, 2006, 43–48.

[8] M. Henning, “The rise and fall of corba,” *Queue*, vol. 4, no. 5, pp. 28–34, Jun. 2006. [Online]. Available: http://doi.acm.org/10.1145/1142031.1142044

[9] ——, “A new approach to object-oriented middleware,” *Internet Computing, IEEE*, vol. 8, no. 1, pp. 66–75, 2004.

[10] D. Harel, “Statecharts: A visual formalism for complex systems,” *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.

[11] J. Bohren and S. Cousins, “The SMACH High-Level Executive [ROS News],” *Robotics Automation Magazine, IEEE*, vol. 17, pp. 18–20, 2010.

[12] M. Klotzbücher and H. Bruyninckx, “Coordinating Robotic Tasks and Systems with rFSM Statecharts,” *JOSER: Journal of Software Engineering for Robotics*, pp. 28–56, 2012.

[13] 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)*, 2012, pp. 585–594.

[14] 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)*, 2013, pp. 474–491.

[15] P. Azad, T. Asfour, and R. Dillmann, “Combining appearance-based and model-based methods for real-time object recognition and 6d localization,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2006, pp. 5339–5344.

[16] T. Asfour, K. Regenstein, P. Azad, J. Schröder, A. Bierbaum, N. Vahrenkamp, and R. Dillmann, “ARMAR-III: An integrated humanoid platform for sensory-motor control.” in *IEEE-RAS International Conference on Humanoid Robots*, 2006, pp. 169–175.

Dr. Nikolaus Vahrenkamp received his Diploma and Ph.D. degrees from the Karlsruhe Institute of Technology (KIT), in 2005 and 2011, respectively. Currently, he is a postdoctoral researcher at the Institute for Anthropomatics, Karlsruhe Institute of Technology (KIT) and works on software development, grasping and mobile manipulation for the humanoid robots of the ARMAR family. From 2011 to 2012, he was postdocoral researcher at the Cognitive Humanoids Lab of the Robotics, Brain and Cognitive Sciences Department, Italian Institute of Technology (IIT), where he worked on grasp and motion planning for the humanoid robot iCub. Nikolaus Vahrenkamp is the author of over 40 technical publications, proceedings and book chapters. His research interests include humanoid robots, motion planning, grasping and sensor-based motion execution.

Address: Karlsruhe Institute of Technology (KIT), Institute for Anthropomatics and Robotics, High Performance Humanoid Technologies (H2T), Adenauerring 2, 76131 Karlsruhe, Germany E-Mail: vahrenkamp@kit.edu

Mirko Wächter received the diploma degree in computer science from the Karlsruhe Institute of Technology (KIT), Germany in 2011. He is currently a doctoral researcher at the Karlsruhe Institute of Technology (KIT) where he is a member of the High Performance Humanoid Technologies lab. His major research interests are physical robot interaction, robot cooperation with humans or other robots, motion synthesis and robotic software development.

Address: Karlsruhe Institute of Technology (KIT), Institute for Anthropomatics and Robotics, High Performance Humanoid Technologies (H2T), Adenauerring 2, 76131 Karlsruhe, Germany E-Mail: waechter@kit.edu

Manfred Kröhnert received the diploma degree in computer science from the Karlsruhe Institute of Technology (KIT) in 2010. He is currently a doctoral researcher at the Karlsruhe Institute of Technology (KIT) where he is a member of the High Performance Humanoid Technologies lab. His research interests include hardware/software architectures for humanoid robots, especially profiling and modeling of behavior and resource demands of robot programs.

Address: Karlsruhe Institute of Technology (KIT), Institute for Anthropomatics and Robotics, High Performance Humanoid Technologies (H2T), Adenauerring 2, 76131 Karlsruhe, Germany E-Mail: kroehnert@kit.edu

Dr. Kai Welke received the diploma degree in computer science from the University of Karlsruhe (TH), Germany in 2005 and his PhD from the Karlsruhe Institute of Technology (KIT) in 2011. His research interests include active vision and scene exploration for humanoid robots.

Address: Karlsruhe Institute of Technology (KIT), Institute for Anthropomatics and Robotics, High Performance Humanoid Technologies (H2T), Adenauerring 2, 76131 Karlsruhe, Germany E-Mail: kai.welke@kit.edu

Prof. Dr. Tamim Asfour is full Professor at the Institute for Anthropomatics, Karlsruhe Institute of Technology (KIT). He is chair of Humanoid Robotics Systems and head of the High Performance Humanoid Technologies Lab (H^2T). His current research interest is high performance humanoid robotics. He is developer and leader of the development team of the ARMAR humanoid robot family. He has been active in the field of Humanoid Robotics for the last 14 years resulting in about 150 peer-reviewed publications with focus on engineering complete humanoid robot systems including humanoid mechatronics and mechano-informatics, grasping and dexterous manipulation, action learning from human observation, goal-directed imitation learning, active vision and active touch, whole-body motion planning, system integration, robot software and hardware control architecture. He received his diploma degree in Electrical Engineering in 1994 and his PhD in Computer Science in 2003 from the University of Karlsruhe.

Address: Karlsruhe Institute of Technology (KIT), Institute for Anthropomatics and Robotics, High Performance Humanoid Technologies (H2T), Adenauerring 2, 76131 Karlsruhe, Germany E-Mail: asfour@kit.edu