The ArmarX Framework - Supporting high level robot programming through state disclosure

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

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

Abstract: In this paper we introduce the robot development environment (RDE) ArmarX. The goal of ArmarX is to ease the development of higher level capabilities for complex robotic systems. ArmarX is built upon the idea that consistent disclosure of the system state highly facilitates the development process. In order to show its applicability, we introduce a robot architecture for a complex humanoid system based on the ArmarX framework and discuss essential aspects based on an exemplary pick and place task.

1 Introduction

Robotic platforms available in the service and assistive robotics area have made tremendous progress in terms of integrating motor and sensory capabilities in the last decade. Complex motor capabilities such as locomotion, two-arm and dexterous manipulation in combination with rich sensory information from visual, auditory, and haptic systems allow us to push the limits towards more dynamical and interactive areas of application. On the other hand, the steady increase of onboard computing power as well as the availability of distributed and cloud computing facilities provide essential tools in order to increase the flexibility and adaptability of these platforms.

In order to benefit from these developments and to establish current service and assistive robots in our daily life, the 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 the 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 through the past decades accompanying the development of robotic platforms. Depending on the target platform and its intended application, a different level of abstraction is realized by each RDE. Several RDE frameworks put

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

Several preconditions have to be met by the RDEs in order to support a variety of robotic platforms. Such preconditions include *platform independence*, *distributed processing*, *interoperability* and *open source* availability. While platform independence and distributed processing are critical in order to support a variety of robotic platforms, interoperability and availability under an open source license are necessary in order to ease software integration. Most of the state-of-the-art RDEs fulfill these preconditions. Another critical aspect of RDEs consists in their ability to assist in the development of higher level skills. For this purpose, the *disclosure of the system state* is an essential requirement. Access to the current state of the system on all levels allows reducing training times significantly while enabling on line inspection and debugging facilities.

This disclosure of the system state is one of the key ideas behind the ArmarX RDE framework. ArmarX makes use of *well-defined interface definitions* and *flexible communication mechanisms* in contrast to traditional message passing protocols. We will show that based on these ideas the disclosure of the system state as well as an application programming interface (API) at a high level of abstraction can be established. To this end, we will introduce a robot architecture for a complex humanoid system based on the ArmarX framework and discuss essential aspects based on an exemplary pick and place task.

2 ArmarX Overview

2.1 Design Principles

For any RDE it is essential to fulfill a number of essential prerequisites in order to be applicable in robotics research and development. In the following, we are goint to briefly discuss design principles, which in our eyes an RDE needs to address in order to match these requirements of today's robot platforms and development processes.

- **Distributed processing**
  Typical hardware architectures in robotic systems include several embedded PCs. Distributing the processing to these different PCs is often necessary due to the integrated design of the robotic platforms. Usually, each PC interfaces with a set of subsystems of the robot and might include specialized hardware such as CAN cards, camera interfaces, or high performance computing facilities. In order to support such systems, applications developed in ArmarX are distributed. Communication is realized using either Ethernet or shared memory to allow transparently distributing the distributed program parts onto the hardware.

- **Interoperability**
  An essential requirement consists of the interoperability in heterogeneous environments. The hard- and software used in today's robotic platforms varies and is far

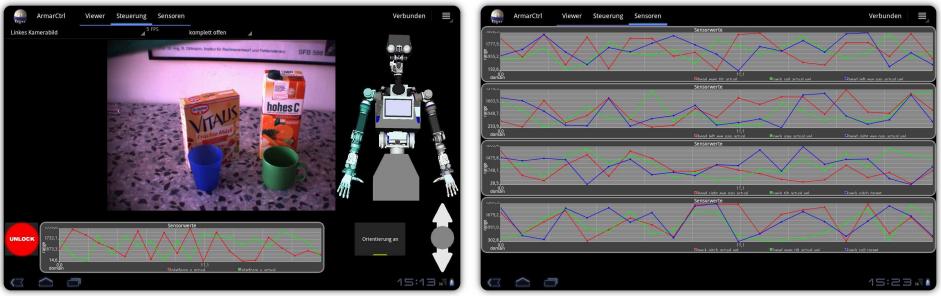

Figure 1: Armar control GUI for Android. Interfacing with ArmarX applications is based on an interface definition language (IDL) which supports a variety of hardware platforms and operating systems using different programming languages.

from being standardized. Hence, it is necessary that RDEs allow bridging these gaps by supporting a variety of different hardware platforms and operating systems. This enables easy integration of new hardware components without the necessity to adapt the RDE to a new platform. In order to account for this requirement, the ArmarX RDE can be compiled under Linux, Windows, and Mac OS X. A distributed application can be build out of the box which spans platforms running any of the mentioned operating systems.

Further, ArmarX makes use of an interface definition language (IDL) which supports a variety of platforms and programming languages. Thus, the covered hardware can be easily extended by implementing these interfaces on the target hardware using its primary programming language. Programming languages that are supported by the IDL include C++, Java, C#, Objective-C, Python, Ruby, PHP, and ActionScript. These capabilities facilitate interfacing with the robot using e.g. mobile devices as illustrated in Figure 1.

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

Besides the above, disclosure of the system state is one of the key design principles behind ArmarX. As stated in the introduction, two mechanisms support this idea on the technical level: well-defined interfaces and flexible communication mechanisms. In the subsequent chapters, we will further develop this idea and show how this 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.

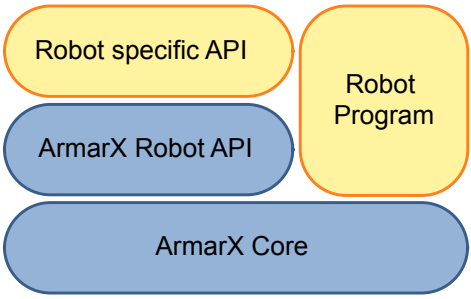

Figure 2: ArmarX is organized in two layers. The core layer implements all facilities to implement distributed applications as well as basic building blocks to establish a robot software architecture. Based on these building blocks, the robot API layer implements more complex functionalities supporting e.g. kinematics, memory, and perception. The generic robot API can be extended in order to implement a robot specific API. The robot program is implemented as distributed application making use of the generic and specific robot APIs.

2.2 System Architecture

ArmarX is organized in two layers as illustrated in Figure 2: the core layer and the robot API layer. The core layer implements all facilities to implement distributed applications. It abstracts the communication mechanisms, provides basic building blocks of the distributed application, and facilities for visualization and debugging. While the core layer is implemented in a single project, the robot API layer comprises several projects which provide access to sensori-motor capabilities on a higher level. These projects include memory, robot and world model, perception, and execution modules. The basic building blocks of the core layer are specialized in the robot API for the different functionalities provided by this layer. By means of extension and further specialization, a robot specific API level can be implemented by the developer. Usually, the robot specific layer includes interface implementations for accessing the lower sensori-motor levels and robot specific models. Depending on the application, this layer can also include specialized memory structures, instances of perceptual functions, and control algorithms. The final robot program is implemented as a distributed application, making use of the generic and specific robot APIs.

In the following sections, we will step through this architecture bottom-up, starting with the core layer in the subsequent section and the robot API layer in Section 4. An example for the robot specific API is discussed in Section 5 for the humanoid robot ARMAR-III [AAV+08]. The application of the API is demonstrated in a pick-and-place scenario.

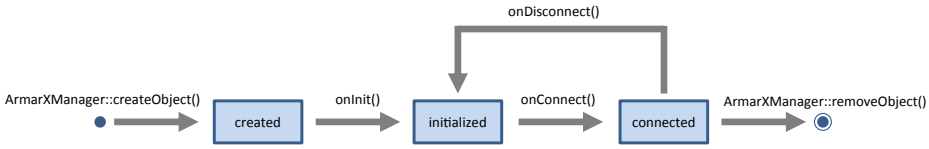

Figure 3: Each process owning an ArmarXManager can create ArmarXObjects that are able of communicating with any objects in the distributed application. Dependencies of the ArmarXObject are made explicit in order to monitor the connectivity of the object. If all dependencies are resolved, the object enters the *connected* state, if a dependency is lost, the object returns to the *initialized* state. This assures consistency among the objects in the distributed application.

3 The ArmarX Core Layer

3.1 Communication in ArmarX

Based on well established communication mechanisms, the core layer provides the basic blocks to build distributed applications that implement robot architectures. For this purpose, we make use of the ZeroC Internet Communication Engine (Ice) as distributed computing platform [Hen04]. The Ice platform implements distributed processing over Ethernet in a platform and programming language independent manner. It provides several communication modes such as synchronous and asynchronous remote procedure calls and publisher-subscriber mechanisms. Further, Ice provides the interface definition language (IDL) Slice, which allows defining interfaces and classes in a network transparent manner.

The ArmarX core layer offers two key components: the *ArmarXObject* and the *ArmarXManager*. The ArmarXObject is the basic building block of the distributed application, while the ArmarXManager handles the life cycle of ArmarXObjects that belong to the same process. The life cycle of an ArmarXObject is illustrated in Figure 3. Each ArmarXObject is reachable within the ArmarX distributed application and can communicate via remote procedure calls or a publisher-subscriber mechanism. In order to guarantee the consistency of the distributed application, dependencies between ArmarXObjects are made explicit. Only if all dependencies of an ArmarXObject are fulfilled, the *connected* state is reached. If a dependency is lost, the ArmarXObject returns to the *initialized* state and waits until all dependencies become available again.

The ArmarXManager is lightweight an can easily be integrated into existing applications. The ArmarXManager allows the creation of ArmarXObjects which then have access to the interfaces of the complete distributed application. Thus, the ArmarXManager largely supports interoperability and integration of third-party software.

The core layer also provides a number of essential tools available within ArmarX such as transparent shared-memory, Ethernet transfer of data, and thread pool based threading facilities.

3.2 Building Blocks of the Robot Application

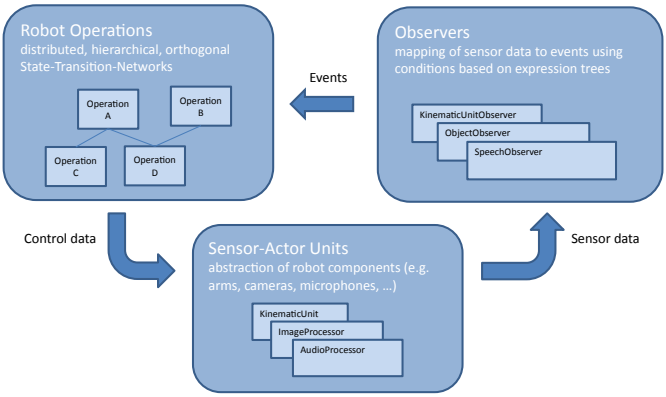

Figure 4: The application programming interface comprises three different elements. The *Sensor-Actor Units* serve as abstraction of robot components, the *Observers* generate events from the continuous sensory data stream which result in transitions between *Operations*. The operations are organized as hierarchical state-transitions networks.

An outline of the three basic elements of the application programming interface (API) is illustrated in Figure 4. The lowest level of abstraction is the *Sensor-Actor Unit* which serves as abstraction of robot components. *Observers* monitor the sensory data stream and generate application specific events which then trigger the appropriate *Operations*. The operations then actuate the robot by issuing control commands. In the following, each of the three elements is described in more detail.

Sensor-Actor Units A Sensor-Actor Unit provides the abstraction of a component of the robot such as kinematic structures or cameras. The sensory data is made available within the ArmarX distributed application using a publisher-subscriber mechanism whereas the control interface is realized using remote procedure calls. Sensor-Actor Units provide methods for handling concurrent access by making use of a request-release mechanism. Due to this unique representation of control and sensory data, other components (e.g. ArmarX observers) are enabled to communicate with these componnets in a standardized way.

Observers The ArmarX API uses an event driven approach. API events originate from desired patterns in the sensory data such as high motor temperature or the reaching of 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. A set of basic checks is implemented in the API. Further, the checks are extensible towards more advanced, application specific conditions.

Operations The robot operations are organized in state-transition networks. Each state in the network can issue control commands or start asynchronous calculations. Transitions between states are triggered by events issued from observers. The state transitions also define the data flow between states by mapping output values of one state to input values of another state, thus defining the parameters for each operation. The state-transition networks are implemented as hierarchical, distributed, and orthogonal statecharts. Each statechart is embedded into an ArmarXObject and can be used as a substate in another statechart (see Figure 8).

3.3 Graphical User Interface

The ArmarXGui serves as a plugin mechanism that allows the realization of graphical user interfaces (GUI) which can communicate with the ArmarX framework. Besides a variety of ready-to-use GUI plugins, such as LogViewer, data plotter or 3D visualization widgets, the ArmarXGui mechanism allows the programmer to realize custom plugins based on the well established Qt framework. In Figure 5 two views of an ArmarXGui instance are depicted. The left image shows a 3D visualization and plugins that are related to several sensor-actor units of the robot. These sensor-actor plugins can be used for emitting control commands as well as for visualizating of internal data, i.e. joint values. As an example, the right side of Figure 5 shows the ArmarX LogViewer plugin.

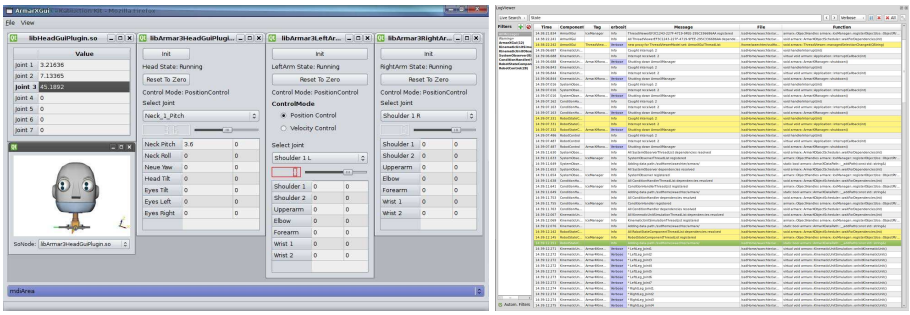

Figure 5: The ArmarXGui provides a variety of ready-to-use widgets which can be extended by a plugin mechanism in order to customize graphical user interaction.

4 The ArmarX Robot API

4.1 Overview

The ArmarX Robot API comprises robot definitions and framework components to uniformly provide an interface to perception modules and memory structures. Additionally, entry points for custom robot program implementations are defined. In the following, we will briefly discuss the generic and ready-to-use API components, which can be parameterized and customized in order to use the API for a specific robot as shown in Section 5.

4.2 Kinematics

The internal robot model consists of the kinematic structure, model files and additional parameters such as mass or inertial tensors. The data is consistently specified via XML files which are compatible to the robot simulation toolbox *Simox* [VKU+12]. This allows making use of all Simox related functionalities, such as collision detection, motion and grasp planning or defining reference frames for transforming poses to different coordinate systems. In order to make the robot model network transparent, ArmarX provides mechanisms to conveniently access a shared robot data structure in order to query for joint values or coordinate conversions.

4.3 Memory

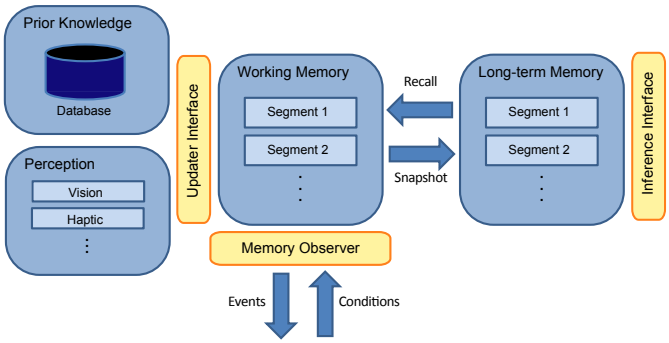

Figure 6: The robot API offers the MemoryX architecture consisting of a working memory and a long-term memory. Both memories can be accessed within the distributed application. Appropriate interfaces allow attaching processes to the memory for updating and inference.

The robot API includes the memory layer MemoryX. This layer includes basic building blocks for memory structures which can be either held in the system's memory or made

persistent in a database. Using these building blocks, the memory architecture illustrated in Figure 6 is realized. The architecture comprises a working memory (WM) and a long-term memory (LTM). Both memory types are organized in segments which can be individually addressed and can contain arbitrary types or classes. Further, they 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, features, ...). Besides the possibility of directly addressing the WM, the working memory observer allows generating events on changes of the memory content. The LTM offers an inference interface which allows attaching learning and inference processes.

4.4 Perception

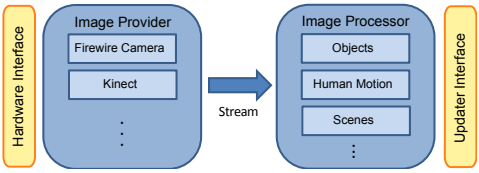

Figure 7: Image processing in the robot API. The image provider abstracts the hardware and streams the data via shared memory or Ethernet to the image processor. The processing result is written to the working memory.

The current implementation of the perception layer in the robot API provides facilities for including camera based image processing in the distributed robot application. The perception layer allows implementing image providers and image processors as illustrated in Figure 7. The image provider abstracts the imaging hardware and provides the data as a stream either 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.

4.5 Robot Program

Any robot program responsible for the overall behavior of the robot is organized as a statechart. The root state of each robot program contains administrative substates for starting, stopping, initializing, and most importantly, loading of program logic. All program logic is dynamically loaded at runtime and is implemented as operations using statecharts. Although the single applications run on different hosts, the program logic is represented as one comprehensive statechart.

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

In this section we will show the disclosure of the system state by means of realizing a pick and place task for the ARMAR-III robot using the ArmarX framework.
In the first part the robot specific extensions to the generic ArmarX robot API are discussed. These extensions are used in the second part, where a pick and place robot program is realized. Note, that only the components which are needed for accomplishing pick and place actions are discussed here. Afterwards, we present several mechanisms for disclosing the system state.

5.1 Robot API for ARMAR-III

The generic ArmarX Robot API (see Section 4) provides several components which have to be customized for usage on 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 when a robot specific implementation (e.g. hardware access) is needed.

- **Kinematics** The kinematics and all relevant coordinate frames of ARMAR-III are specified and Sensor Actor Units for the arms, both hands, torso and platform are implemented. A visualization of the robot and all coordinate frames can be seen in Figure13(a).
- **Memory** The environment, which is assumed to be known for this task, is stored in MemoryX and provides knowledge about the surrounding of the robot in the spatial memory segment. Additionally, a set of known objects together with grasping information is stored in the prior knowledge database.
- **Perception** The visual perception framework of ARMAR-III includes methods for stereo-camera based object recognition and localization for segmentable and textured objects using the methods described in [AAD06]. The resulting object locations are stored and made available in the spatial memory segment of the working memory.

5.2 The Pick and Place Robot Program

A robot architecture for the humanoid robot ARMAR-III has been implemented based on the ArmarX framework. The suitability of ArmarX for establishing higher level capabilities is demonstrated based on this architecture in a pick and place task. In order to accomplish this task, several components have been implemented based on the core and API layers (see Section 5.1).

An excerpt of the hierarchically organized operations involved in the pick and place task is illustrated in Figure 8. The highest level implements initialization and loading of different robot programs. The level of abstraction drops through the layers from a complete pick and place operation, over an object grasping operation down to the lowest level where visual servoing for grasping is implemented. All operations are realized using the hierarchical ArmarX statechart mechanism. All state transitions are initiated by suitable observers which issue the required events.

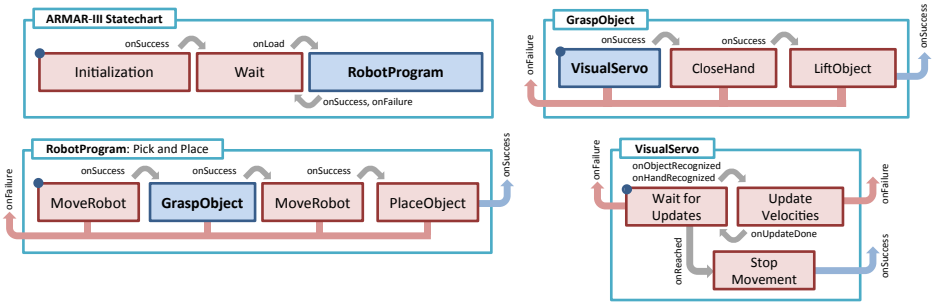

Figure 8: An excerpt of the operations involved in a pick and place task on ARMAR-III. The operations are realized using the ArmarX hierarchical statechart mechanism. On each hierarchy level of this example the state highlighted in blue is further refined.

5.3 Disclosure of the system state

ArmarX provides several mechanisms to disclose the internal state of a robot program on different levels. These mechanisms include monitoring of sensory data, events and conditions, and the execution state of the distributed application. Figure 8 illustrates one aspect of ArmarX to disclose the system state on the program logic level.
Furthermore, other internal data can be inspected during the executing of the pick and place statechart through several ArmarXGui plugins. A selection of available plugins is explained in the following paragraphs.

MemoryX Visualization The content of the robot's working memory can be visualized in a 3D view of the current environment. Figure 9 depicts the memory state before and after the execution of the pick and place task.

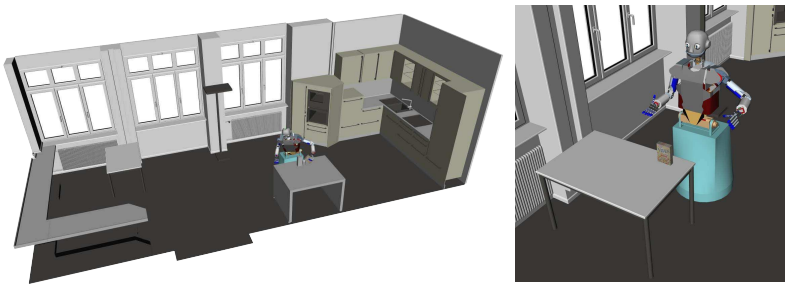

Figure 9: (a) ARMAR-III grasping an object. (b) The world state after placing the target on the table.

The object segment of the long-term memory (see Figure 6) holds information about the target object, such as visual features, 3D model or precomputed grasping data. Figure 10 shows the 3D model and the associated grasp (a) which is used as a target coordinate frame (b) during execution (c). The current state of the relevant coordinate systems (left TCP and target pose) can be plotted as shown in Figure 10(c)), allowing to inspect spatial data channels that are considered by the observers.

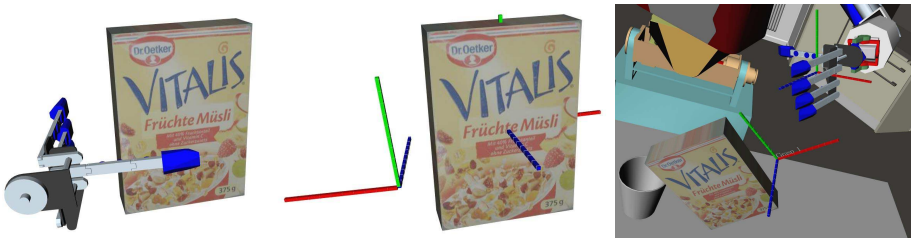

Figure 10: (a) A grasp that is associated with an object in the database. (b) The resulting grasping target, defined in the object's coordinate frame. (c) A visualization of the left TCP coordinate system and the target pose, captured while executing the VisualServo state.

Logging The ArmarXGui LogViewer plugin allows displaying, conditional grouping, sorting, and searching of all ArmarX log messages of all distributed ArmarX applications. Additionally, extended information such as component state, application IDs and back-tracing information is available (see Figure 5).

Visualization of Application Dependencies The system state of a distributed ArmarX robot program can be queried and visualized during runtime. Since all ArmarXObject instances are accessible and their dependencies are explicit (see Section 3.1) a hierarchical view of the system's connectivity can be generated. Figure 11(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. Figure 11(b) shows

the same system after the missing dependency was resolved and the remaining open connections could be established. Even though the components are running on different hosts, their dependencies can be inspected from anywhere in the network.

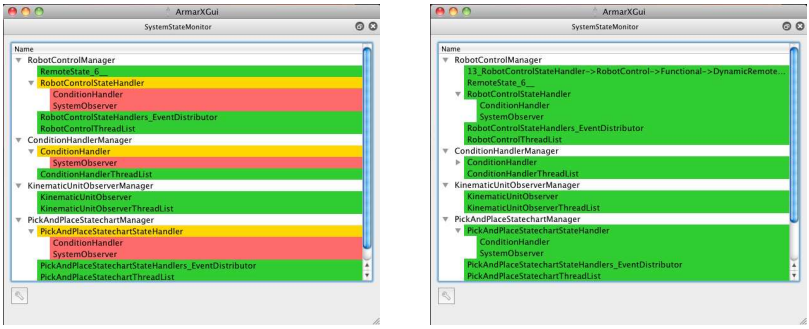

Figure 11: Distributed robot program connectivity before (a) and after (b) the SystemObserver component is started.

StatechartViewer The StatechartViewer provides a visual tool for inspecting program logic. For easy comprehension of complex robot programs each statechart hierarchy level is visualized by displaying all states, connecting transitions as well as input- and output parameters of each state (see Figure 12). The visualization of the currently active state at each hierarchy level and the data flow is continuously updated. Furthermore, the StatechartViewer enables the user to interact with the statechart. For example, transitions can be triggered manually or breakpoints in specific states can be set to halt the complete robot program for debugging purposes.

ConditionViewer The ConditionViewer is a visualization tool for analyzing the currently active and all expired conditions. These conditions are installed on the ConditionHandler, the central condition processing unit. The arbitrary large, boolean conditions are visualized with a tree-graph (see Figure 12(b)). Additionally, the current fulfillment of a complete condition or its subterms is visualized via colors easing the inspection of conditions.

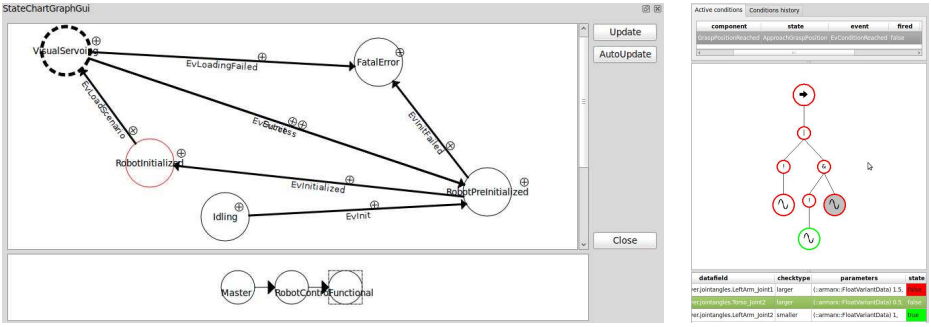

Figure 12: (a) StatechartViewer for inspecting the program logic and the current state. (b) ConditionViewer is an inspection tool for active and expired conditions on the observers.

Reference Frames and Spatial Segments of the Working Memory The current state of reference frames can be inspected through visualization tools that allow depicting a selection of coordinate systems. The coordinate system are either linked to the current joint configuration or to a cached robot state (see Figure 13(a)). Additionally, the spatial content of the robot's working memory can be inspected as shown in Figure 13(b).

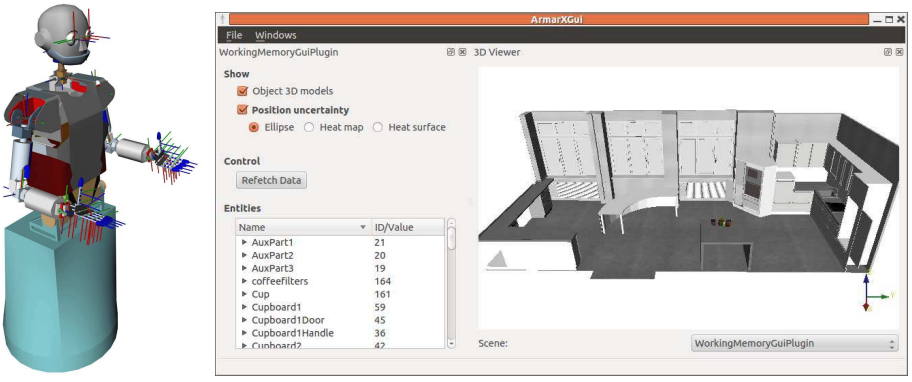

Figure 13: (a) The kinematic structure, the 3D model and all relevant coordinate systems of ARMAR-III. (b) The spatial information of the robot's working memory can be inspected by the MemoryX gui plugin.

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 applica-

tions while offering support to disclose the system state on all abstraction levels. The basic functionality is covered by a generic communication framework on top of which we developed 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.

Acknowledgment

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

[AAD06] 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)*, pages 5339–5344, 2006.

[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.

[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.

[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.

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

[MCA] MCA2. Modular Controller Architecture. `http://www.mca2.org/`.

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

[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.

[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.