When Hard Realtime Matters: Software for Complex Mechatronic Systems

Berthold Bäuml and Gerd Hirzinger
Institute of Robotics and Mechatronics
DLR – German Aerospace Center
82234 Wessling, Germany
Email: berthold.baeuml@dlr.de

***Abstract*— A still growing number of software concepts and framworks have been proposed to meet the challenges in the development of more and more complex robotic systems, like humanoids or networked robotics. The issue of hard realtime, however, has not been the main focus of such concepts, but is essential for building and controlling mechatronic systems. Here we discuss the specific demands of complex mechatronic systems and present a software concept, the "agile Robot Development" (aRD) concept, we developed at our institute. We show that the performance of current computing and communication hardware allows for a flexible component based concept with distributed execution even in hard realtime with rates in the kHz range.**

I. Introduction

Over the last years robotic systems reached a new level of complexity. Unlike systems with a six degrees of freedom (DOF) arm with a gripper or simple mobile robots, we see now torque controlled redundant arms, articulated hands, humanoid walking robots, cooperating swarms of mobile robots or service robots communicating with sensor networks installed in their environments. To address the challenges of developing software for such systems a number of software concepts and frameworks have been proposed. This number is still growing as until now no general abstraction has been found that fits well with all the specific demands of the diverse robotic applications and hardware.

Prominent representatives are ORCA [1], MARIE [2], MIRO [3], Player [4], OROCOS [5] MCA [6], OpenHRP [7], YARP [8] and Microsoft Robotics Studio [9]. They are all based on the idea, that a complex robotic system should be composed from interacting modules or components in the sense of the component based software engineering approach [10] with all its benefits as flexibility, code reuse or decoupling of the development flow in a team. To allow the components to be distributed on a network of heterogeneous computers all approaches also provide tools for simplifying and standardizing communication.

A. Demands of Mechatronical Systems

In our institute the specific demands for a software concept arise from building and controlling highly complex mechatronic systems, e.g. the DLR Light-Weight-Robot arms (LBR), DLR Hands [12] or the recently built upper humanoid body "Justin" with 41 DOF [11](see Fig. 1). The two main demands are: first, to provide scalable computing resources in hard realtime to allow for computationally demanding control loops in the kHz range (up to 10kHz in the near future) running over all DOF and second, to support an "agile development flow" of a small, tightly interacting team of experts in the spirit of the 'Agile Software Development' methodology [13], [14]. Such a flexible, iterative and rapid development flow is essential for building complex systems, especially when working on research protoypes. It should be easy to connect new hardware components like sensors and actuators, to scale computing resources by simply adding more CPUs, to integrate software components from different developers and to flexibly reconfigure the physical as well as the functional communication structure of the system, which is essential for iterative rapid prototyping. Special to mechatronical systems is the high complexity of the part running in hard realtime, consisting of e.g. device drivers, sensor processing, controllers, inverse kinematics, collision avoidance, state machines, ...

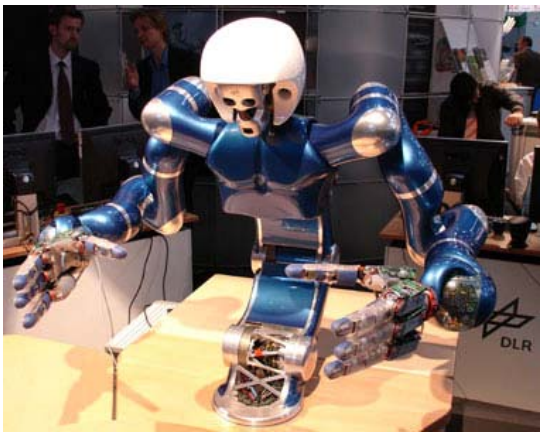


Fig. 1. An example for a complex mechatronic system: the DLR upper humanoid body Justin [11] with 41 DOF. This system is built from two DLR-LBR-III arms with 7 DOF each, two DLR-Hand-II with 12 DOF each [12] and a torso with 3 DOF.

To summarize: for complex mechatronic systmes an easy to use and flexible component based software concept allowing for distributed execution while guaranteing hard realtime is desirable.

On the other hand, the performance of standard hardware has reached a level, where such a component based, more abstract view on the system is possible. Most important for

that are the fast digital buses of up to 1GBit/s inside and to the robot components and the high computing power of commodity systems in combination with flexible communication infrastructure built from cheap components like Gigabit-Ethernet.

B. Robotic Software Concepts

The software concepts mentioned before have been successfully used in different fields of robotic applications. In what follows we briefly discuss to what extent they meet the desired requirements for mechatronical systems.

ORCA, MARIE, MIRO and Player are used in mobile robotics, where the realtime constraints are rather soft with rates in the 100 Hz range. Also Microsoft Robotics Studio is targeted on soft realtime applications so far, as the underlying operating system (OS) is Windows XP.

OROCOS provides hard realtime and has been successfully used in control applications with up to two robot arms running at more than 500 Hz. How the performance scales when going to the complexity of humanoid robots and distributed computation is needed has to our knowledge not been reported yet.

OpenHRP is designed for the development of humanoid robotics applications. It is based on RT-Middleware [15] for inter-component communication which uses CORBA and so allows for distributed execution. In all applications reported so far, however, the hard realtime parts running the low level controllers with rates in the kHz range have been implemented in monolithic modules using proprietary communication to reach the desired performance.

MCA is used for complex robotic systems like the humanoid robot ARMAR [16]. It allows for a hierarchical composition of e.g. controller components while providing hard realtime. But as it uses TCP/IP for network communication, the concept does not natively provide hard realtime for distributed execution.

YARP is another concept used in a number of complex robots like Domo [17]. It is lightweight, allows for the configuration of the quality of service (QoS) of the inter-component communication and is portable by using ACE [18]. In all the reported robotic applications the low level high rate controllers run on dedicated DSP boards. But as YARP also supports the realtime OS QNX [25], it would be interesting to see how it performs in a complex mechatronic system like Justin with high control rates and distributed computing on networked PCs.

In the rest of the paper we first discuss in more detail the demands in developing software for complex mechatronic systems by taking a closer look on our humanoid upper body system Justin. Then we introduce a simple software concept, the "agile Robot Development" (aRD) concept, we developed at our institute to pragmatically address this demands. The key points of the aRD concept are first to add only a thin layer above the realtime operating system to get the full hardware performance and second to have control over the quality of service (QoS) of the connection between components to meet the different hard, soft and non realtime constraints. Finally we present some performance examples and give an brief overview of other robotics applications we have realized with the aRD concept.

II. Analysis of a Complex Mechatronic System

As a concrete example for a complex mechatronic system we analyze here the humanoid upper body system Justin we developed at DLR for performing experiments in the control of two handed manipulation.

A. System Overview

A system oveview is given in (Fig. 2). Justin is built from five robot components: two LBR-III arms (7 DOF each), a torso (3 actuated DOF) and two DLR-Hand-II (12 DOF). Each joint is equipped with position and torque sensors. The single components are connected to the computing resources by fast digital buses with a bandwith of $>$ 8MBit/s and a clock rate of 1kHz. The system allows to run a single control loop over all 41 DOF at a rate of 1kHz. Typical examples for computationally demanding controllers are gravity compensation, impedance control [19], where e.g. touching the finger tip can sensitively move the whole upper body, or object impedance, where a virtual imepdance can be assigned to an object, which is grasped with both hands by cooperatively controlling all finger, arm and torso joints [20].

B. Functional View

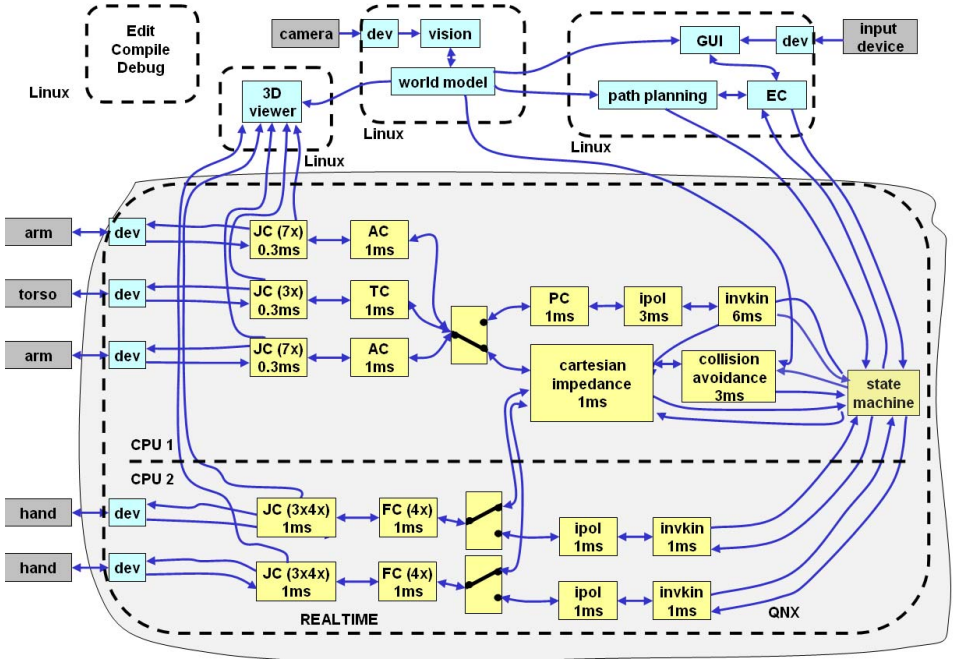
Taking a coarse view on the system, it consists of the robot hardware connected to realtime targets and in addition of non realtime computers running applications for user interaction, perception and planning. Additional hosts run tools for development and tools that allow for monitoring and profiling of the different parts of the system during runtime.

All of the functionality of the system is represented by *blocks* running in realtime or non-realtime. They perform calculations and *communicate* with each other. Typically the granularity of the realtime part is finer, as each block usually performs only a small amount of deterministic calculation. On the other hand, blocks in the non-realtime part represent more monolithic applications and can perform elaborate algorithms on complex internal representations. Here we focus mainly on the realtime part and its connections to the non realtime part.

From this it is straightforward to see a robot system as a *decentral net of calculation blocks and communication links*, in this way defining a functional view on the system. This abstraction not only helps in designing the architecture of a robot system, but also paves the way for a component-based software concept.

C. Essential Requirements

In the following we analyse in more detail the requirements for realizing such a net of blocks.



2. System overview of the complex robot system "Justin" resembling a *net of calculation blocks and communication links*. There is a realtime and a non-realtime part. In the former the granularity of the blocks is usually finer and the blocks represent a hierarchy of different controllers for each robot component (JC=joint control, AC=arm control, TC=torso control, FC=finger control) which are connected via device-driver blocks (dev) to the hardware and run with different rates (0.3ms up to 3ms). In addition there are blocks for computing the inverse kinematics (invkin) or interpolation (ipol) or blocks that control the sequence of execution (state machine) getting commands from higher-level blocks. In the non-realtime part the blocks are typically more monolithic applications like a 3D-viewer or GUI for user interaction or a vision system and path planner in combination with an execution control block (EC) for higher-level intelligence. As main operating systems QNX [25] is used in the realtime and Linux in the non realtime part and for the developments hosts.

1) Concurrent Execution: Each block is an *execution entity*, e.g. a process or thread, and can have its own *priority*. This allows to schedule the available processing time between the blocks and implicitly defines the execution order. In practice it is often more efficient and simpler to aggregate blocks into groups, where each group is an execution entity and iterates through its blocks.

2) Standardized Communication: All blocks are equal in the sense, that they all can be sources and sinks for data. Also the *connection scheme is arbitrary*. A block's output port can be connected to any other block's input ports, as long as the data formats match.

The *data format* of each port of each block can be different, but is *static* during runtime. This simplifies the implementation of a block, but also means, that a block is less than an object or a component, where different methods with different parameters and return values can be called. Nevertheless, this restriction is convenient for robotic system as the static robot hardware (e.g. always the same number of sensor values) implies the static data format for most of the blocks.

Also the block's *connection scheme* is *static* at runtime. This design decision dramatically simplifies the implementation, not only of the single block but of the mechanisms for configuring the overall system. Instead of allowing to dynamically change the connection scheme it is possible to disable and enable blocks through input ports, where a disabled block does not consume computation time.

3) Data Flow and Model of Computation: A block sends *data* by *pushing* it through its output port to the input ports it is connected to. To keep the design simple, there are no pull or send-with-reply operations, which can nevertheless be easily built on top of the push operation.

For blocks like controllers having robot hardware connected to them the "synchronous data flow" model of computation (MoC) [22] has to be provided. In this MoC the execution order in a group of synchronized blocks is deterministic, but the blocks in such a synchronization group still can run at different rates (*multirate*). In addition it is important to allow more than one synchronization groups, where between those groups a asynchronous transition must be possible. A typical example is a robot system consisting of more than one robot component, each having its own hardware clock and each of the low level controllers is synchronized with 'its' component, whereas at higher levels the controllers run over the full robot.

Both models of computation can be realized by providing non blocking write and blocking and non blocking read operations in combination with FIFO buffers at the block's input ports [21], [22].

4) System Handling: The description of the *configuration* of a system consists of two parts. First, the structure of the net of blocks has to be described. To be able to handle big systems the description must allow for a hierarchical aggregation of blocks to meta-blocks and so on. Second, the mapping of the net to the actual computing hardware has to be specified, to describe which block runs on which computer and communicates over which network links. This explicit specification of the hardware used for the communication between two blocks is essential for guarantying the *quality of service* (QoS) with regard to bandwith, latency and determinism. Only this way the developer has control over the realtime behaviour of the overall system.

At runtime the system is a decentral net of communicating blocks distributed over a network of computers. The software concept has to provide mechanisms to allow for a *central*

startup from one console and a coordinated shutdown.

5) Openness and Ease-Of-Use: The functionality of the system is implemented *in* the blocks. Therefore it is very important that the *interface* for writing a block and integrating it in the net's communication structure should be *open* to arbitrary programming languages. This is especially important as in robotics the blocks are contributed by a team of expert from different fields each requiring its specific tools and languages.

The interface for writing a block should also be *simple*, as researchers are experts in their field but not necessarily software experts and are not willing to invest much time to understand sophisticated software frameworks.

III. Implementation

In this section we give an overview of the software concept, the "agile Robot Development" (aRD) concept, we developed at our institute to realize the above described decentral net of calculation blocks and communication links. The two main guidelines for the implementation were to meet the hard realtime constraints in the kHz range and to keep it easy to use.

The current implementation of the aRD concept consists of *aRDnet*, a simple software suite developed at our institute. In addition we provide a seamless integration of a *toolchain* based on Matlab/Simulink/RTW [23] and RTLab [24], as Matlab/Simulink is the quasi-standard tool for simulation of robot dynamics and controller design. As main operating systems we use QNX Neutrino [25], a POSIX-compliant microkernel realtime OS, for the realtime target and Linux for the non realtime computers. Reduced support is also given for VxWorks and Windows XP.

A. aRDnet

In the aRD concept each block is an individual process running an arbitrary executable which, as part of the net, sends and receives data packets.

Typical examples for such blocks running in the realtime part are I/O-blocks that implement the device drivers for communication with the connected robots or other hardware. Another example are non-deterministic calculation blocks which are asynchronously coupled, for instance an inverse kinematics wich uses some kind of iterative minimization algorithm.

Each block can have multiple input and output ports, but each output port is connected to exactly one input port of an arbitrary block with matching data formats.

aRDnet is laid out as a simple software suite that supports and standardizes the communication between blocks. The suite consists of four parts: First, a library for easy implementation of a block's input and output ports. Second, the `ardnet` executable realizing communication between blocks running on different computers. Third, a template for writing Simulink stub blocks, which allows for easy communication between aRDnet blocks and Simulink models. Finally, tools for a coordinated startup and shutdown of the system.

1) aRDnet Library: The aRDnet library provides a native C/C++-interface. Based on this interfaces to other programming languages, like Python or Matlab, can also be easily built. The simple interface consists of only five functions:

- `create` and `init` for creating and initializing the input and output ports of a block. The properties of a port can be configured by special command line arguments provided at startup to the block's process.
- `send` for non-blocking sending of a data packet through an output port.
- `rec` and `tryrec` for blocking and non-blocking receiving of data from an input port.

The size and format of a data packet can be different for each port of each block, but is static and defined at compile-time.

The connection scheme of the block's ports is determined by providing command line options at startup of each block's executable. For each port of a block a separate name is specified. Connections are simply determined by matching port names for input and output.

The current implementation of the aRDnet library achieves all of the above by only a thin layer of abstraction over the functionality of the underlying operating systems. Basically, only the POSIX "named shared memory", semaphores and mutexes are used.

2) `ardnet` executable: The `ardnet` executable serves two important purposes with regard to the communication abilities in the net of blocks. Being also based on the aRDnet library it can be seen as a block, however, with special features.

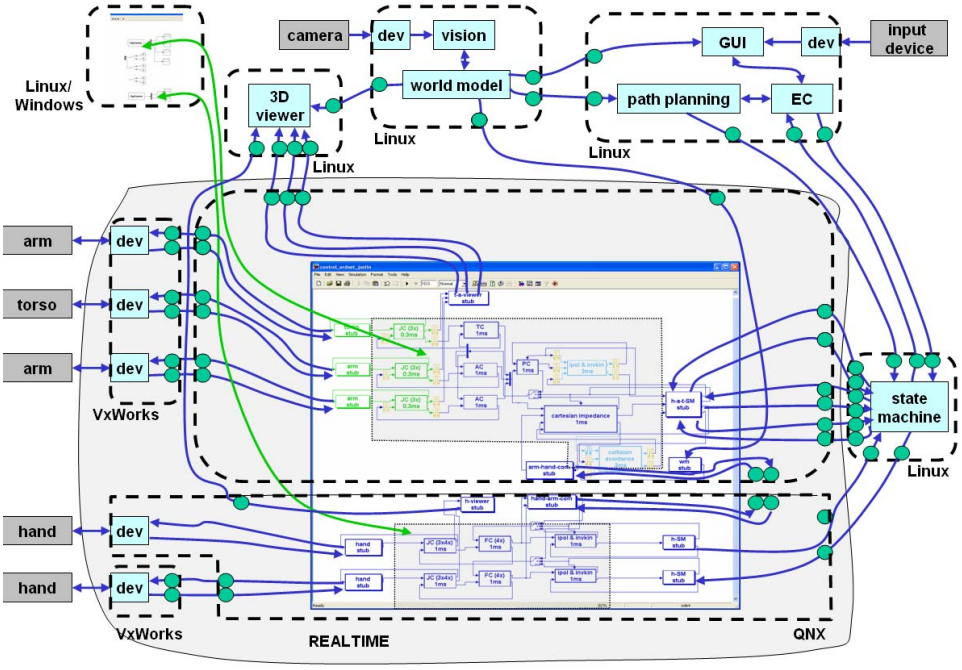
First, `ardnet` realizes the communication between two blocks on different computers by running a corresponding pair of `ardnet` processes as a network bridge. For this purpose ardnet has built-in functionality for transmission of data over the network providing a "virtual wire" between the two communicating blocks.

In the current implementation `ardnet` uses bare UDP sockets but it can easily and transparently be extended to any other transportation protocol, e.g. EtherCAT or Qnet, or even media, e.g. Firewire or InfiniBand.

To address the problem of blocks running distributed even on heterogenous computers (with differing CPU families, operating systems and compiler versions) the aRDnet suite defines compatible basic data types. In this way, the aRDnet library assures the right representation on both sides when the data packets are composed from these basic types.

Furthermore, a detailed control of quality of service (QoS) is possible by choosing different network connections, e.g. point-to-point or switched ethernet, using separate network stacks (a particular feature of the QNX microkernel architecture) and finally by adjusting process priorities. This way we have been able to achieve realtime communication over four ports with a rate of 1kHz on each line (for details see section IV).

The second purpose `ardnet` serves is to provide a *port multiplier block*. Therefore `ardnet` can be configured to have one input but multiple output ports. Each data packet arriving at the input is distributed onto all output ports.



3. Final realization of the complex robot application of Fig. 2 with the aRDnet concept. The controllers of the realtime part are generated from Simulink models (indicated by the screenshot) running distributed on two QNX PCs. To connect aRDnet standalone blocks, like the device-driver blocks, and blocks in the Simulink model the aRDnet suite provides Simulink stub blocks (blocks inside the grey area in the Simulink screenshot correspond to the controller structure from Fig. 2, whereas the blocks outside this area are the additional stub blocks).
Communication between two blocks running on different computers is realized by an `ardnet`-bridge consisting of an `ardnet` block (circles with "an") on each side.
During the final integration phase of all the parts of the robot application the originally planned computer setup (see Fig. 2) had to be massively changed because of problems with the availability of drivers and libraries. The flexibility of the aRDnet concept, however, allowed for this changes to take place with only little adaption in the functional blocks.

3) Simulink stub: To connect aRDnet blocks to blocks implemented in a Simulink model the aRDnet suite provides a template S-function code. This easily allows to generate a stub block for Simulink representing the actual block. The developer has only to implement three functions. One specifies the stub block's layout (number and dimensions of in- and outports). The other two functions translate the data packets being sent by the aRDnet block through its output ports to the outport lines of the stub block and, in the other direction, translate the data at the stub's inport lines to the data packets received at the aRDnet block's input ports.

The connection between the aRDnet block and its stub is implemented with the help of the standard aRDnet library mechanisms. This allows for a seamless integration of a Simulink models in the net.

4) Startup and Shutdown: Starting the decentral net of blocks distributed over a network of computers is done by using a hierarchy of shell scripts very similar to the way a Unix system starts up. A master script calls subscripts for setting up particular system parts (e.g. the driver blocks for the robot hardware).

To allow for the startup of a distributed system from a single central command station the aRDnet suite provides the `ardstart` command for starting up programs and scripts on a remote computer. In addition `ardstart` does bookkeeping of what has been started where. This information is needed for a coordinated shutdown of the whole system or only specific subsystems and is exploited by the `ardkill` command of the aRDnet suite.

IV. Performance and Applications

In this section we present some performance measurements based on the above implementation of the aRD concept. If not otherwise mentioned all measurements were performed on PCs with Pentium 4, 3GHz.

- aRDnet Performance: The worst case minimal round-trip time for a data packet of 1kByte size between two blocks running on QNX realtime targets is measured. 'Minimal' means here, that the second block sends the packet immediately back after receiving it. For two blocks on the same computer the round-trip time is 20μs. For two blocks on different computers connected by an `ardnet` bridge over a 1Gbit ethernet point-to-point connection the time is 200μs (and 160μs on average).
- High Rate: In a HIL setup a simple Simulink model reads analog values from an I/O-card and records them to the harddisk. At a rate of 30kHz the system introduces only little overhead (e.g. due to scheduling) of less then 10% of cpu-time.
- Multirate: A robot arm is connected to a VxWorks computer, which sends the sensor and actuator data at a rate of 1kHz via an `ardnet` bridge to a QNX realtime target running a Simulink model with a controller also at a rate of 1kHz. The very same Simulink model contains a subsystem running at a rate of 10kHz for reading in analog values from a sensor via an I/O-card.
- Deterministic Execution and Jitter: In the very same system as above, we could increase the average cpu load of the Simulink model up to a level of 90% before loosing simulation steps, even while running debug and profiling tools over a second network connection. This implies

that system jitter, even in case of the 1kHz network communication rate via `ardnet`, is smaller than 100μs.
- Justin, a complex application (see Fig. 1): The five robot components are heterogenously connected as depicted in Fig. 3 to two VxWorks and two QNX computers running the realtime part of the application. All communication between the realtime computers runs at a rate of 1kHz via point-to-point `ardnet` bridges. All 41 DOF can be controlled at a rate of 1kHz in a common control loop distributed on the two QNX PCs. In addition a number of non realimte computers running Linux are connected via the institutes LAN for running blocks for user interaction, perception and planning.

In our institute a wide range of other projects are now based on the aRD concept. These projects with about 15 development seats range from teststands of new robot joints over medical robotics, telepresence, haptic devices or brain computer interfaces.

V. Conclusion

The successful application of the aRD concept in building complex mechatronic systems is because it meets the two main demands of such systems.

First, it provides scalable computing resources in hard realtime by

- introducing only a thin layer of abstraction above the realtime OS, e.g. the blocks communicate directly without an additional server between them and
- keeping full control of the QoS of the communication in the developers hand, e.g. not hiding the details of different link types (like media or protocols).

Second, by its ease of use and flexibility the software concept supports a development flow in the spirit of the 'Agile Software Development Methodology' that is especially suited to small teams of experts. This is achieved by

- not needing a runtime environment, but each block being an standalone executable with a small C++ library linked to it,
- having simple and open interfaces with only five C++ functions and
- providing a clear separation of
  - functionality, which is implemented in the blocks,
  - composition, by specifying the connection scheme in a hierarchy of startup scripts by means of the block's command line options and
  - quality of service, which is also configured in the startup scripts by adding `ardnet` bridges between block's ports using different communication protocols and media.

As also in the future advances in robotics will be significantly based on the rising complexity of mechatronic systems it is important that software concepts specifically designed for the field of robotics address this demands.

References

[1] A. Brooks, T. Kaupp, A. Makarenko, A. Orebäck, and S. Williams, "Towards component-based robotics," in *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2005)*, 2005.

[2] C. Cote *et al.*, "Code reusability tools for programming mobile robots," in *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2004, pp. 1820–1825.

[3] H. Utz, S. Sablatng, S. Enderle, and G. K. Kraetzschmar, "Miro – middleware for mobile robot applications," in *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures*, 2002.

[4] R. T. Vaughan, B. Gerkey, and A. Howard, "On device abstractions for portable, resuable robot code," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robot Systems (IROS 2003)*, 2003, pp. 2121–2427.

[5] Orocos. [Online]. Available: http://www.orocos.org

[6] Mca2. [Online]. Available: http://www.mca2.org

[7] F. Kanehiro, H. Hirukawa, and S. Kajita, "OpenHRP: Open Architecture Humanoid Robotics Platform," *International Journall of Robotics Research*, vol. 23, no. 2, pp. 155–165, 2004.

[8] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet Another Robot Plattform," *International Journall of Advanced Robotics*, vol. 3, no. 1, pp. 43–48, 2006.

[9] Microsoft Robotics Studio. [Online]. Available: http://www.msdn.microsoft.com/robotics

[10] G. T. Heineman and W. T. Council, *Component-based Software Engineering. Putting the Pieces Together*. Reading, MA: Addison-Wesley, 2001.

[11] C. Ott, O. Eiberger, W. Friedl, B. Bäuml, U. Hillenbrand, and other, "A Humanoid Two-Arm System for Dexterous Manipulation," in *Proc. IEEE/RAS International Conference on Humanoid Robots (HUMANOIDS) 2006*, 2006.

[12] G. Hirzinger, N. Sporer, M. Schedl, J. Butterfass, and M. Grebenstein, "Torque-controlled lightweight arms and articulated hands: Do we reach technological limits now?" *The International Journal of Robotics and Research*, vol. 23, no. 4–5, 2004.

[13] (2001) The Manifesto for Agile Software Development. [Online]. Available: http://agilemanifesto.org/

[14] A. Cockburn, *Agile Software Development*. Reading, MA: Addison-Wesley, 2001.

[15] N. Ando, T. Suehiro, K. Kitagaki, and other, "RT-Middleware: Distributed Component Middleware for RT (Robot Technology)," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robot Systems (IROS 2003)*, 2005, pp. 3555–3560.

[16] T. Asfour, K. Regenstein, P. Azad, A. Schrö"der, and other, "ARMAR-III: An Integrated Humanoid Plattform for Sensory-Motor Control," in *Proc. IEEE/RAS International Conference on Humanoid Robots (HUMANOIDS) 2006*, 2006.

[17] A. Edsinger-Gonzales and J. Weber, "Domo: A Force Sensing Humanoid Robot for Manipulation Research," in *Proc. IEEE/RSJ International Conference on Humanoid Robots (HUMANOIDS) 2004*, 2004.

[18] S. Huston and C. Johnson, *The ACE Programmer's Guide, The Practical Design Patterns for Network and Systems Programming*. Boston, MA: Addison-Wesley, 2003.

[19] A. Albu-Schäffer, C. Ott, and G. Hirzinger, "A unified passivity based control framework for position, torque and impedance control of flexible joint robots," in *Int. Symposium on Robotics Research 2005*, 2005.

[20] T. Wimböck, C. Ott, and G. Hirzinger, "Impedance Behaviors for Two-handed Manipulation: Design and Experiments," in *Proceedings of the IEEE/RSJ International Conference Robotics and Automation (ICRA 2007)*, 2007.

[21] E. Lee and T. Parks, "Dataflow Process Networks," *Proceedings of then IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[22] E. Lee and S. Neuendorfer, "Actor Oriented Design of Embedded Hardware and Software Systems," *Journal of Circuits, Systems and Computers*, vol. 12, no. 3, pp. 231–260, 2003.

[23] The MathWorks. [Online]. Available: http://www.mathworks.com/

[24] OpalRT. [Online]. Available: http://www.opal-rt.com/

[25] QNX Software Systems. [Online]. Available: http://www.qnx.com/