DOI: 10.1109/ICAL.2010.5585330

Interfacing Industrial Robots using Realtime Primitives

Michael Vistein, Andreas Angerer, Alwin Hoffmann, Andreas Schierl, and Wolfgang Reif

***Abstract*— Today, most industrial robots are interfaced using text-based programming languages. These languages offer the possibility to declare robotic-specific data types, to specify simple motions, and to interact with tools and sensors via I/O operations. While tailored to the underlying robot controller, they usually only offer a fixed and controller-specific set of possible instructions. The specification of complex motions, the synchronization of cooperating robots and the advanced use of sensors is often very difficult or not even feasible. To overcome these limitations, this paper presents a generic and extensible interface for industrial robots, the Realtime Primitives Interface, as part of a larger software architecture. It allows a flexible specification of complex control instructions and can facilitate the development of sustainable robot controllers. The advantages of this approach are illustrated with several examples.**

I. Introduction

Industrial robots are very flexible machines that can be adapted to a large variety of tasks. For example, robots are able to perform industrial processes like welding or painting, assist humans in manufacturing and even adapt their behavior according to external sensing. To adapt robots to different tasks, the underlying controller must provide an interface that allows the execution of custom programs. However, these programs must be interpreted with certain real-time guarantees in order to achieve high precision and determinism when controlling the mechanical robot devices.

A straightforward idea for programming robots is to write programs running directly on the real-time capable robot controller (usually on top of a real-time operating system), e.g. by using robot control frameworks such as OROCOS [1], which provides C++ libraries for robot control systems. In general, this approach enables the largest possible expressiveness, as the robot programmer can directly interface the robot using a general-purpose programming language. However, writing programs directly for execution on the robot controller has the major disadvantage that the application developer is responsible for writing code which does not violate any real-time constraints of the control system. Any errors in the code may lead to failure not only of the program itself, but also to the entire robot system.

To mitigate this shortcoming, most commercial manufacturers provide special robot programming languages (e.g. the KUKA Robot Language or RAPID from ABB). Those languages are usually rather simple considering their syntax, and the systems provide a support for developing programs and transmitting them to the real-time controller where they are executed obeying real-time constraints. However, a major drawback of these robot programming languages is the lack of extensibility and interoperability [2]. Usually, these languages only offer a fixed and controller-specific set of possible instructions (e.g. motions). The interaction with external devices (e.g. tools and sensors) is often limited to I/O operations. With new requirements emerging in robotics like the synchronization of cooperating robots and tight integration of sensor feedback, the robot programming languages have reached their limits.

To overcome these limitations, a new software architecture for programming industrial robots was developed in the research project *SoftRobot* and introduced in [3]. The idea behind this software architecture is the use of modern standard programming languages and environments on top of a lean and flexible robot control layer. In this way, real-time critical robot control is abstracted from application development and common functionality is provided by a generic and extensible application programming interface (API) which can be directly used to implement robotics software. Real-time critical control actions are encapsulated and executed on the real-time control layer. This strict separation is feasible because applications for industrial robots are not necessarily real-time critical over the whole duration of their execution, but can be split into small, real-time critical actions. Therefore, the API has to provide means for specifying and executing real-time critical actions as a kind of atomic transactions. The main application controls the proper execution of these actions, depending on a high-level program workflow.

For the specification of such real-time control actions, we developed the highly expressive *Realtime Primitives Interface (RPI)*, which will be presented in detail in this work. The main contribution of RPI to robotics is its flexiblity and modularity in specifying complex control tasks with a defined execution semantics. RPI is not intended to be used directly, but is the basis for introducing high-level specification of robot tasks without the need to care about real-time issues.

The paper is strucutured as follows: Sect. II explains the architectural context the Realtime Primitives Interface is embedded into and describes the requirements that influenced its definition. Subsequently, Sect. III defines the declarative language for specifying control actions, and Sect. IV explains the execution model for such control actions which is tailored to the robotics field. The benefits of RPI are shown by example commands in Sect. V. Implementation of these

Michael Vistein, Andreas Angerer, Alwin Hoffmann, Andreas Schierl, and Wolfgang Reif are with the Institute for Software and Systems Engineering, University of Augsburg, D-86135 Augsburg, Germany; vistein@informatik.uni-augsburg.de

This work presents results of the research project *SoftRobot* which is funded by the European Union and the Bavarian government within the *High-Tech-Offensive Bayern*. The project is carried out together with KUKA Roboter GmbH, Augsburg, and MRK Systeme GmbH, Augsburg.

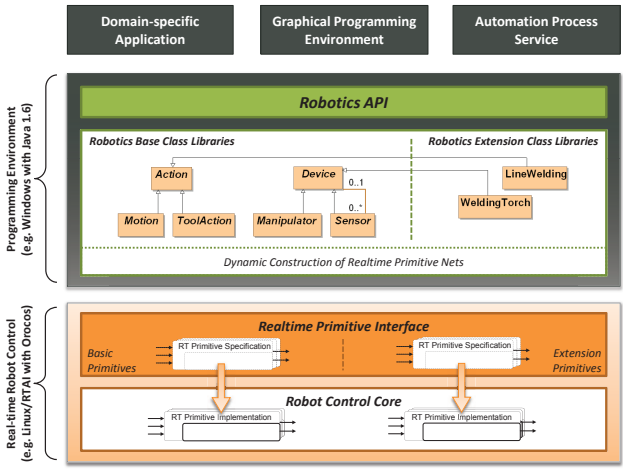

Fig. 1. Three-tier architecture for programming industrial robots using standard, non real-time capable environments.

examples using a KUKA lightweight robot are presented in Sect. VI. Finally, Sect. VII draws a conclusion and provides some outlook.

II. Architectural Context

The software architecture we developed in the SoftRobot project consists of two main layers as depicted in Fig. 1. Application developers are provided with the *Robotics Application Programming Interface*, a rich object oriented framework for building robotics applications. The *Robotics Base Class Libraries (RBCL)* provide basic implementations for the interfaces defined in the Robotics API and therefore contain classes for common concepts of the robotics domain like robots, tools, frames and actions that controllable devices shall perform. The RBCL are designed to be expandable with new functionality. Such extensions are called *Robotics Extension Class Libraries*.

This work does not go into detail about the Robotics API and its implementation, but rather focuses on the Real-time Robot Control layer (cf. Fig. 1). This layer is essential for allowing real-time control of hardware devices. Our approach follows the idea of separating high-level application logic and low-level hardware control, as described above. Consequently, hardware actions like movements of robot arms on certain trajectories can be specified as part of a high-level application workflow, but subsequently have to be translated to low-level control schemes that are executed by the real-time capable control layer. Future trends in industrial and service robotics, like tight integration of sensor feedback in motion control or cooperation of multiple robots, require a very flexible specification of such control schemes. Ideally, those schemes should not be fixed, monolithic algorithms that can be parametrized at best, but should be dynamically composable out of small, reusable control blocks (like e.g. a sensor that can trigger a stop of movement as soon as a certain value is measured). Such a fine-grained structure allows for an expressive specification of low-level tasks, and in turn, is a necessary basis for a powerful and extensible high-level robot application framework.

We thus identified the following requirements to the interface of the real-time control layer:

- It must be adequately expressive, i.e. it must be possible to specify control actions representing the current and future functionality required by industrial applications.
- It must be extensible, i.e. it must be possible to integrate new control components without redesigning the interface.
- It must be dynamic, i.e. control actions are executed at runtime without prior compilation.
- Execution must be guaranteed to be deterministic, i.e. control actions must be precisely repeatable.

As a result, the *Realtime Primitives Interface* was developed. It is a manufacturer-independent interface consisting of a declarative language for specifying control actions for industrial robots and an execution model with defined semantics. Any *Robot Control Core* (RCC) implementing the Realtime Primitives Interface must have the ability to interpret control actions specified in the RPI language under real-time conditions. The RCC is responsible for direct hardware control and must provide all real-time critical functionality. It is the only part of the architecture which has to run within a real-time capable environment.

Some other approaches have introduced concepts based on similar ideas concerning separation of high-level workflow logic and low-level control algorithms. ORCCAD [4] uses the ESTEREL language [5] to describe robot programs that have been specified graphically. The resulting programs are compiled and later executed on the robot controller. Thus, the generation process is not as dynamic and flexible compared to what is possible with RPI. An approach developed by Finkemeyer et al. [6] also decomposes complex robot tasks into smaller parts and uses so called Manipulation Primitives as interface between task-level programming and manipulator control. Similar approaches can be found in [7] and [8]. However, the smallest units of control that these approaches provide for specifying control commands describe rather complex, parametrizable operations. As shown in this paper, RPI is intended to work on a much more fine-grained level to be flexible and generic enough for meeting today's and future requirements to an industrial robot control framework.

III. Language

RPI uses a dataflow language to describe real-time critical control actions. The main concepts are *primitives* which can be interconnected with *links* to form so called *primitive nets*. The language is similar to LUSTRE [9] which is used e.g. by the commercial development tool SCADE, but integrates some specialties necessary for robot control.

A. Primitives

Primitives are functions that map n input values to m output values where $n,m \in \mathbb{N}_0$. In order to execute a primitive, i.e. to evaluate the function it represents, all input values must be available and, after execution, all output values must be assigned. Input values are read from the *input ports* and

output values are written to the *output ports* of a primitive. Primitives with no input or no output ports are legitimate.

Formally, a primitive can be expressed as a 5-tuple *(Type, Id, Input, Output, Parameters)*. Its type, i.e. what function it performs, is described by *Type*. *Id* is a unique identifier, allowing to distinguish among different primitive instances of the same type in a single net. *Input* and *Output* are each a set of typed ports. *Parameters* is a set of values that can be used within the primitive exactly like input values, with the difference that the value of a parameter can only be set once at the instantiation of the primitive. Particularly, parameters are useful for the configuration of devices (e.g. robots, tools, or sensors).

Primitives are the smallest independent functions that must be available at the RCC. They can represent devices, but also functions for planning and interpolating trajectories, calculating kinematic transformations, or different functional parts required to incorporate sensor values into a trajectory. The set of primitives provided by an RCC can vary. For example, an industrial 6-DOF manipulator might need different control primitives than a SCARA or a mobile robot. For new hardware devices, specific *driver* primitives need to be integrated into the RCC, and can subsequently be used to control that device.

B. Links

A *link* connects an output port of one primitive with an input port of another primitive. A link is always connected with exactly one output port and one input port. An output port may have multiple links attached, whereas an input port may only be connected to a single link.

C. Data Types

Input and output ports are statically typed. Four basic data types are defined, which are Boolean, String, (double precision) Real and Integer. Furthermore, composed data types can be defined. For example, matrices consisting of real values can be used to describe homogeneous coordinates. Every data type utilizes a special *null* value, which is different from every other value. Every primitive must expect *null* as an input at any time and may produce *null* as an output (e.g. if no meaningful output can be generated).

D. Primitive Nets

Several primitives interconnected with links form a *primitive net*. Usually these nets need to be acyclic. Under special circumstances (e.g. if a closed loop control must be implemented) links may form cycles, but at least one link must be marked as "delayable", e.g. the value is only guaranteed to be provided to the target primitive in the next execution cycle.

IV. Execution

Unlike a general-purpose dataflow language, RPI is tailored to industrial robotics to support the splitting of real-time control and high-level application logic, as stated in Sect. II. These requirements influenced the definition of the execution semantics, the life cycle of commands and the set of operations that an execution environment has to provide.

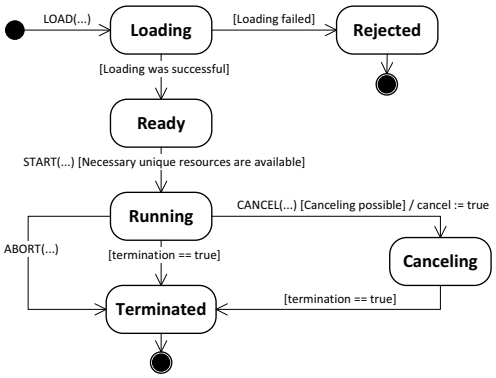

Fig. 2. Statechart representing the command life cycle in RPI

A. Execution semantics

Primitive nets with no unconnected input or output ports can be seen as *commands* which can be executed. Such a command can be issued to the robot control, and can subsequently be interpreted by a suitable execution environment running under real-time constraints on the RCC. The RCC has to provide an implementation for the primitives specified in submitted commands. Otherwise, it is not able to execute that kind of command.

The execution of primitive nets is performed periodically. In each period, the whole net is evaluated, thus all hardware devices are provided with new target values. Because links may not form cycles (with the exception of links where the data flow may be intentionally delayed), it is possible to determine an order of execution of the net which guarantees – if primitives are evaluated sequentially in this order – that the latest input data will always be available for all primitives. This ensures a fast propagation of data values throughout the whole primitive net.

In general, it is possible to execute several nets in parallel on the same RCC. However, not all primitive nets can be safely executed simultaneously. For example, if two distinct primitive nets are trying to control the same robot, they might produce contradictory results with severe consequences. For that reason, some primitives (e.g. robots and tools) carry a special flag marking them as unique resources which are unsuitable for concurrent use. The execution environment must not execute two primitive nets at the same time that both contain the same unique resource.

B. Command Life Cycle

Commands have a life cycle with several possible states of execution (c.f. Fig. 2):

- **Loading** The command has been successfully transmitted to the RCC, which is currently loading and preparing all necessary modules. In this state, no device may be controlled by that command.
- **Rejected** If the RCC cannot successfully load all necessary modules, the command must be rejected. This

occurs if no matching module can be found or instantiated for a primitive specified in the command.

- **Ready** The execution environment has successfully instantiated all necessary modules, and thus, the command is now ready for execution.
- **Running** The RCC is currently executing the command, i.e. evaluating the primitive net and controlling the devices. Unique resources are assigned to this command and cannot be used by any other command.
- **Canceling** The execution environment initiated the canceling of the command, and the command has the possibility to gracefully stop its execution.
- **Terminated** The execution of the command has finished. All devices are inactive and unique resources must be available again.

Usually, a command is automatically created within the RBCL and subsequently loaded into the RCC using the operation `LOAD(command)`. First, the robot control inspects and validates the command (checks for syntactical validity, no unconnected links or ports) and then starts to initialize all necessary modules. Hence, the command is in state *Loading*. The operation is asynchronous, i.e. the command does not necessarily have to have finished loading when the operation `LOAD` returns. The return value is a unique identifier for the command. After the execution environment has finished loading and preparing all necessary modules, the command state changes to *Ready*.

As soon as this change has happened, the operation `START(command_id)` can be called to trigger the execution of the command. If all unique resources are available, the robot control must start the execution of the command and change the net's state to *Running*. Now, the primitive net is evaluated in its determined order, i.e. the execution environment must call the modules and propagate the data from the output ports to the input ports of the next module. After all modules have been executed, devices like robots or tools must have been provided with new values for direct hardware control.

Besides, there are three more operations. The operation `CANCEL(command_id)` tries to gracefully stop the specified command, whereas `ABORT(command_id)` immediately stops the execution of the command (e.g. for debugging purposes, see also Sect. IV-C). Finally, the operation `STATE(command_id)` returns the current state of the command.

C. Robotics-specific Features

To meet the practical demands of robot control, some additional mechanisms had to be included in the definition of RPI. With the previously introduced operations `CANCEL` and `ABORT`, there exist two distinct ways of forcing a command to end its execution. Because the communication between any application using the RBCL and the RCC may be not real-time capable, no timing guarantees can be given. I.e. neither calling the abort nor the cancel operation from an high-level application is appropriate for any safety measures. Calling the `ABORT` operation will cause the net to terminate, i.e. the devices will stop receiving valid values. This operation is intended only for debugging purposes, e.g. if a net does not terminate properly itself. By contrast, the operation `CANCEL` allows e.g. a recalculation of the trajectory to stop the motion on a defined path. To achieve this, a special *Cancel* primitive exists, which provides a Boolean output port indicating if a cancel operation was requested. This primitive can be used in commands, requiring the definition of an adequate, command-specific handling of the cancel operation. If a command does not contain the primitive, it can only be aborted.

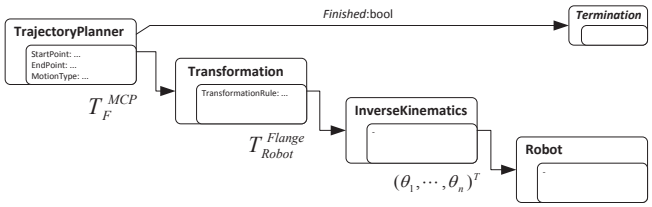

Fig. 3. Simple RPI command controlling a robot to perform a motion in Cartesian space.

Another challenge exists concerning the definite end of commands. For example, a command controlling a single movement of a robot has its obvious end with arriving at the given destination. However, if the same command contains some other movement or a tool action, it is difficult or even impossible to recognize the end of the entire command. Finally, there are even commands which do not have an inherent end and run for an indefinite time (e.g. holding a specified force at the end effector until a command-specific condition becomes true). To handle these cases, a command is supposed to indicate its termination by setting the Boolean input port of a special *Termination* primitive that has to be part of *every* command. If this input port is set to true, the execution environment must treat a command as terminated. If a command uses the Cancel primitive explained above, it is expected to set the termination signal after performing a safe ending of actions.

Sometimes it is useful to provide information about a running command, e.g. to display the current robot position to the user. For that purpose, there are monitoring primitives, which take any data value on their input port and provide this value to software outside the RCC. The frequency that is used to update the data value can be adjusted.

V. Examples

In this section, we will present some basic examples which demonstrate the usefulness and flexibility of RPI. Usually, primitive nets are automatically generated by a generic mapping algorithm implemented in the RBCL. For the sake of clarity, the nets presented in this section are simplified and lack some of the more complex constructs needed to support a generic mapping algorithm. Practical results of these examples are presented in Section VI.

Fig. 3 shows a simple command that controls a single robot to perform a motion in Cartesian space. The graphical

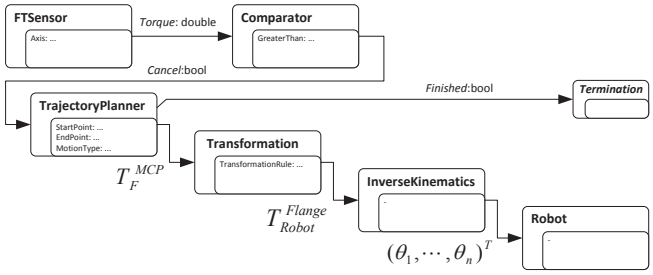

Fig. 4. RPI command controlling a robot including a force/torque sensor to stop motion upon contact.

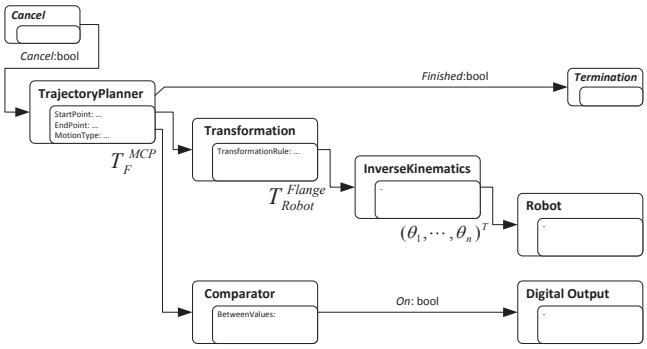

Fig. 5. RPI command controlling a robot and a welding torch synchronously.

notation used here denotes primitives as rounded boxes, and links as arrows among the boxes. Parameters are written within a smaller rectangle inside a primitive. The example contains:

- a primitive *TrajectoryPlanner* for the generation and interpolation of the trajectory,
- primitives representing the robot together with an adequate inverse kinematics calculation,
- a primitive performing a coordinate transformation,
- and finally the *Termination* primitive.

When this command is loaded, the trajectory planner calculates and interpolates a trajectory with the specified points. During the execution of the command, this primitive has to provide a new position value in each cycle. The values produced by the trajectory planner describe the position of a crucial point for the operation, the *Motion Center Point (MCP)* in some convenient frame F (e.g. relative to a workpiece). The transformation primitive converts these coordinates to the coordinates of the flange in the coordinate system of the robot. The resulting value is transferred to the primitive calculating the inverse kinematics solution, which generates joint values for the robot primitive. This primitive finally executes the motion by passing the target axis values to the robot hardware. The (closed-loop) trajectory-following control is performed inside the robot primitive. Once the trajectory planner has issued the last interpolation point, and as a consequence the robot is no longer moving, it issues `true` to the termination primitive to signal that the command is finished.

Dividing a robot command into small parts as shown in Fig. 3 results in great flexibility when it comes to modifying or extending a command. In Fig. 4, an additional force/torque sensor has been integrated. With such a sensor, a robot can measure external forces or torques that are applied for example to the tool. The command depicted above compares the torque value measured by the primitive FTSensor to a predefined maximum value and issues a cancel signal to the Trajectory Planner if the maximum torque has been breached. Of course it is also possible to connect the Comparator directly to the Termination primitive to cause an immediate emergency stop and not only a soft stop. Because the sensor integration is completely modeled using RPI, it is guaranteed that an event measured by the sensor will cause an effect (i.e. starting to brake the robot) within one execution cycle.

In Fig. 5, a more complex command is shown. The command additionally employs a Cancel primitive, which allows to trigger a soft stop from outside the RCC. Apart from the robot, this command also controls a welding torch using a digital output. The calculated trajectory is analyzed by a Comparator primitive (which usually will be automatically created by the mapping algorithm out of several simple comparison primitives and boolean operator primitives) which can trigger welding torch operations at the proper positions of the trajectory. This guarantees that the welding torch is always turned on and off at exactly the right positions.

All previously described commands can be implemented using only a few primitives. This allows a rather implementation of the robot control core which offers a large variety of possible commands.

VI. Experimental Implementation

Within the research project *SoftRobot*, we are developing a prototypical implementation of our new software architecture, including an interpreter for primitive nets running under real-time Linux. Currently, this interpreter is able to control a KUKA lightweight robot (LWR). The robot is interfaced with the *Fast Research Interface* [10] (FRI) option running on top of the standard KUKA Robot Control (KRC) hard- and software. Using this option, it is possible to control the position of the robot with a cycle time of 1ms over an ethernet link, and even employ the advanced features of this robot, such as Cartesian impedance control [11] or the force/torque sensors integrated in each axis. The data being received from the robot using FRI are used for example for the force/torque sensor primitive in Fig. 4.

The RPI interpreter itself has been implemented with C++ using the Orocos Framework [1] and is running under Linux/RTAI. Additionally, a basic set of primitives has been implemented to support all examples described in Section V. The Robotics Base Class Libraries are implemented in standard Java 1.6 and therefore can run on many different platforms. The communication between the RBCL and the RCC is performed using HTTP/REST over TCP. Using a rather high level, complex protocol on top of TCP is possible, because this communciation link does not require real-time.

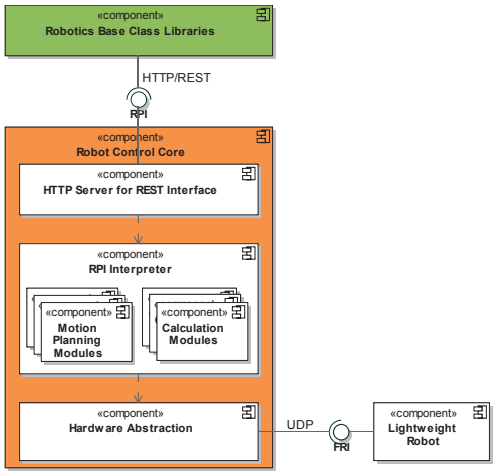

Fig. 6. Architecture of the SoftRobot implementation

The communication of the RCC with the robot using FRI is real-time critical, and therefore performed using a (KUKA proprietary) low level protocol on top of UDP. The overall architecture of the RCC is depicted in Fig. 6.

Besides primitives for controlling a lightweight robot, we also implemented basic primitives for digital and analog I/O ports. With these primitives, peripheral devices such as a gripper or sensors like light beams can be controlled.

Using the aforementioned RCC implementation, we were able to test all examples described in Section V. The primitive nets have not only been hand crafted, but also automatically generated from applications written in Java and using the object-oriented Robotics API. First experiments are looking very promising: The robot moves very smoothly, and even more complicated, automatically generated primitive nets containing about 50 primitives can be executed in under 0.1ms time on standard single core PC hardware (AMD Athlon64 3500+ with 2 GB memory).

VII. Conclusion & Future Work

In this paper we described a generic interface for programming industrial robots, the *Realtime Primitives Interface*. The language is designed to be automatically generated, and therefore integrates into the larger software architecture developed in the research project *SoftRobot*, which enables the developer to use standard programming languages and tools for industrial robots. The main contribution of RPI is that it allows to describe flexible, encapsulated real-time critical actions and can therefore abstract from real-time control in high-level programming.

In *SoftRobot* we created a prototypical RCC, which is able to interpret commands specified in RPI, and some basic RBCL classes to support the development of object oriented robot programs. Being able to use such a high-level programming interface also makes the integration of additional sensors like standard cameras and corresponding image recognition software a lot easier, for example the recognition of different workpieces, where the image processing does not need to be done in real-time.

Currently we are still examining some advanced aspects around RPI. One of those concerns more detailed semantics of the concurrent execution of commands, as well as the process of switching between consecutive commands. Furthermore, another open point concerns fault handling routines. RPI already allows the specification of fault handling routines by using primitives that detect failures (e.g. of tools) and trigger alternative motions. But there are also strategies required that can cope with failing modules on the real-time side, e.g. due to programming errors. Finally, we consider mechanisms for validating or even verifying correctness aspects of RPI commands. Besides these theoretical enhancements to RPI, we are engaged in gaining more experimental results with complex commands and multiple robots. Much effort is also put in the development of the Robotics API as a comprehensive interface for application developers.

Acknowledgements

The authors gratefully acknowledge Dr. Frank Ortmeier for his ideas and comments. The authors would also like to thank Markus Bischof and Alexander Chekler from KUKA Roboter GmbH for their support.

References

[1] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Proc. 2001 IEEE Intl. Conf. on Robotics and Automation*, Seoul, Korea, May 2001, pp. 2523–2528.

[2] A. Hoffmann, A. Angerer, A. Schierl, M. Vistein, and W. Reif, "Towards object-oriented software development for industrial robots," in *Proc. 7th Intl. Conf. on Informatics in Control, Automation and Robotics*, Funchal, Madeira, Portugal, June 2010.

[3] A. Hoffmann, A. Angerer, F. Ortmeier, M. Vistein, and W. Reif, "Hiding real-time: A new approach for the software development of industrial robots," in *Proc. 2009 IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, St. Louis, USA, Oct. 2009, pp. 2108–2113.

[4] D. Simon, B. Espiau, E. Castillo, and K. Kapellos, "Computer-aided design of a generic robot controller handling reactivity and real-time control issues," *IEEE Transactions on Control Systems Technology*, vol. 1, no. 4, pp. 213–229, Dec. 1993.

[5] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[6] B. Finkemeyer, T. Kröger, and F. M. Wahl, "Executing assembly tasks specified by manipulation primitive nets," *Advanced Robotics*, vol. 19, no. 5, pp. 591–611, 2005.

[7] G. Milighetti, H.-B. Kuntze, C. W. Frey, B. Diestel-Feddersen, and J. Balzer, "On a primitive skill-based supervisory robot control architecture," in *Proc. 12th IEEE Intl. Conf. on Advanced Robotics*, Seattle, USA, July 2005, pp. 141–147.

[8] K. Kobayashi, A. Nakatani, H. Takahashi, and T. Ushio, "Motion planning for humanoid robots using timed petri net and modular state net," in *Proc. 2002 IEEE Intl. Conf. on Systems, Man and Cybernetics*, vol. 6, 2002.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sept. 1991.

[10] G. Schreiber, A. Stemmer, and R. Bischoff, "The fast research interface for the KUKA lightweight robot," in *Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications. IEEE Intl. Conf. on Robotics and Automation*, Anchorage, Alaska, USA, May 2010.

[11] A. Albu-Schäffer, C. Ott, and G. Hirzinger, "A unified passivity-based control framework for position, torque and impedance control of flexible joint robots," *Int. Journal Robotics Research*, vol. 26, no. 1, pp. 23–39, 2007.