DOI: 10.1109/IROS.2010.5649098

The Robotics API: An Object-Oriented Framework for Modeling Industrial Robotics Applications

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

***Abstract*— During the last two decades, software development has evolved continuously into an engineering discipline with systematic use of methods and tools to model and implement software. For example, object-oriented analysis and design is structuring software models according to real-life objects of the problem domain and their relations. However, the industrial robotics domain is still dominated by old-style, imperative robot programming languages, making software development difficult and expensive. For this reason, we introduce the object-oriented Robotics Application Programming Interface (Robotics API) for developing software for industrial robotic applications. The Robotics API offers an abstract, extensible domain model and provides common functionality, which can be easily used by application developers. The advantages of the Robotics API are illustrated with an application example.**

I. Introduction

Today, industrial robots are still programmed with textual, proprietary robot programming languages. These languages are provided by robot manufacturers for their products and are bound to their robot controllers. They are derived from early imperative languages like ALGOL or Pascal and have in common that they offer robotics-specific data types, allow the specification of motions, and process I/O operations for the communication with external systems (e.g. tools, sensors, or PLCs). Examples are the KUKA Robot Language or RAPID from ABB. Due to these low-level languages, programming an industrial robot is a difficult task requiring considerable technical expertise and time. Hence, industrial robots are usually equipped and programmed to perform only one particular task for a considerable time. This might be acceptable for mass production as in automotive industries, but for small and medium enterprises with rapidly changing products in small batches, the introduction of industrial robots is an expensive and risky decision.

Furthermore, robot programming languages are strongly limited compared to general-purpose languages. For example, there is no built-in support for graphical user interfaces, and external connectivity is limited, which makes e.g. connecting to databases or accessing web services difficult. Hence, developing software for configuring and supervising industrial robotic cells is a very complex and error-prone task. The examples from [1] and [2] illustrate the efforts necessary for developing custom applications for commercially available robot controllers. Considering the future challenges of industrial robotics [3] like flexible manufacturing systems, human-friendly task description or cooperating robots, existing approaches for developing applications for robotics are reaching their limits.

To overcome these limitations, there have been several academic approaches providing robot-specific libraries for general-purpose languages. Examples are RIPE [4], MRROC+ [5] and the Robotic Platform [6] Today, there's a trend in robotics towards component-based software engineering [7], as the robotics domain, especially for experimental robotics, is considered as too diverse and too inhomogeneous to develop one stable reference domain model [8]. Examples for component-based robotics frameworks are Player [9] and OROCOS [10]. However, these libraries and frameworks use a low level of abstraction i.e. developers still need profound knowledge in robotics and often in real-time programming.

From our point of view, the *industrial* robotics domain can greatly profit from a rich object-oriented framework. Such a framework should provide the common concepts of this domain in form of class structures. Existing approaches like SIMOO-RT [11] and the above-mentioned Robotic Platform [6] do not model concepts beyond different manipulators and rather simple movements. Therefore, we developed the *Robotics Application Programming Interface (Robotics API)* as the main element of a three-tiered software architecture [12] and present it in this paper. Its main contribution is the introduction of a comprehensive model of the industrial robotics domain that comprises modeling devices, action descriptions as well as interesting parts of the physical world.

The paper is structured as follows: Sect. II describes why and how object-orientation can be applied to industrial robotics and motivates the chosen architecture. In Sect. III, the structure of the Robotics API including its domain model is presented in detail. Subsequently, the advantages of this object-oriented framework are illustrated with an industrial application example in Sect. IV. The low-level execution of Robotics API commands with regard to real-time constraints is described in Sect. V. Finally, conclusions are drawn in Sect. VI.

II. Object-Orientation in Industrial Robotics

Nowadays, business software systems are usually constructed using techniques such as object-oriented analysis, design and programming. Elaborate software design processes like the Unified Process [13] exist, as well as

The authors are with the Institute for Software and Systems Engineering, University of Augsburg, D-86135 Augsburg, Germany. E-mail of corresponding author: angerer@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 and MRK-Systeme GmbH.

methods and guidelines for constructing object-oriented software architectures and solving recurring design problems by appropriate patterns [14]. In object-oriented design, real world items are often modeled directly as software objects, which dramatically helps understanding complex software architectures. The object-orientation paradigm has a variety of features to support reuse of existing code. By using concepts like inheritance, it is possible to create large but still generic libraries, that can simplify the development of new applications for a certain domain. In that way, cost and effort of software development can be greatly reduced.

There exists a number of object-oriented frameworks for the robotics domain (e.g. [4], [5], [6], [11]). Some of these mainly focus on communication aspects between distributed system parts, whereas others also provide a basic class model covering important robotics concepts. However, an elaborate class model cannot be found in any of those frameworks. Brugali and Scandurra [7] even argue that it is difficult to define an object-oriented framework that remains stable during its evolution, with regard to its class structure. For this reason, they propose the approach of constructing robotics software systems out of functional components with defined interfaces (defined as Component Based Software Engineering), which they argue to be suited for robotics. While this may fit for the extensive domain of *experimental robotics*, we believe that an elaborate object-oriented framework can be a valuable basis for application development in the comparatively narrow domain of *industrial robotics*. Applications in this domain are usually built upon common, stable functions like high-level motion instructions that are provided by the basic robot control system. Thus, the required abstraction level is higher than the level a coarse-grained component architecture provides. An adequate object-oriented framework architecture can provide a high abstraction level for re-using functionality that is common for today's industrial robot use cases and, furthermore, even cover future trends by reusing, extending and composing existing classes. Aside of that, a large number of existing, object-oriented class libraries exist for modern languages like C# or Java. They provide advanced functionality like image processing or complex user interface elements, which can be directly used with an object-oriented robotics framework.

A fact that complicates the design of an object-oriented programming framework for robotics – and robot programming in general – is the need for real-time hardware control. Especially in the domain of industrial robotics, deterministic execution of developed and once tested programs is of utmost importance. This special design requirement to today's industrial robot controls, in particular for meeting safety criteria and quality standards, resulted in the development of proprietary programming languages that are interpretable with certain timing guarantees. However, an analysis of a wide range of industrial tasks for which robots are employed provides an interesting result: Those actions that require hard real-time guarantees comprise a closed set of basic primitives, whereas most of the workflow inside the respective applications is not hard real-time critical. This led to the conclusion that most of the workflow of such applications can be programmed in a standard environment without regarding real-time aspects, whereas only real-time critical control flows have to be executed in a specially designed runtime environment.

We developed a novel architectural approach that allows a tight integration of high-level robot programs and real-time critical execution of low-level control flows, which is presented in detail in [12]. Following this approach, real-time critical commands can be specified using the object-oriented *Robotics API* and are then dynamically translated into commands for the *Realtime Primitives Interface (RPI)* [15]. A RPI-compatible *Robot Control Core (RCC)* executes those commands, which consist of a combination of certain, predefined (yet extendable) calculation modules, and a specification of the data flow among them. Implementing only the RCC with real-time aspects in mind is sufficient to allow the atomic execution of Robotics API commands under real-time conditions. Existing frameworks for real-time robot control can be used to implement the RCC. In [15], we used OROCOS as a basis for our prototypical RCC. In this paper, we focus mainly on the design of the Robotics API and how this programming framework supports the development of applications for robotics, but also outline the process of dynamic RPI command generation on the basis of an application example.

III. Structure of the Robotics API

Robotic applications usually model some part of the physical world. In the simplest case, they just define points in space that are necessary for robot movements in the application. Depending on the concrete application or application class, more information about the robot's environment is represented. For example, in assembly applications, the notion of different workpieces that have to be assembled can be helpful. However, such complex models of the reality are predominantly supported by offline programming tools, whereas standard robot controls usually only support the definition of points.

The scope of the Robotics API comprises both use cases, as it supports both basic robot programs as well as complex, domain specific applications. Therefore, the definition of points in space as well as arbitrary physical objects is possible. Fig. 1 shows an overview of the basic class structure inside the Robotics API, which consists of a total of about 70 classes. The lower left part of this diagram contains those classes that support modeling geometric relations:

- A *Frame* is a spatial point with a unique name. Each Frame can have an arbitrary number of relations directing to it or originating from it.
- A *Relation* connects a pair of Frames and defines a geometric relation between them, expressed mathematically by a homogeneous *TransformationMatrix*.
- A *SpatialObject* aggregates a number of Frames that logically belong together. A SpatialObject can be seen as the 'scope' of a Frame, as each Frame is assigned to

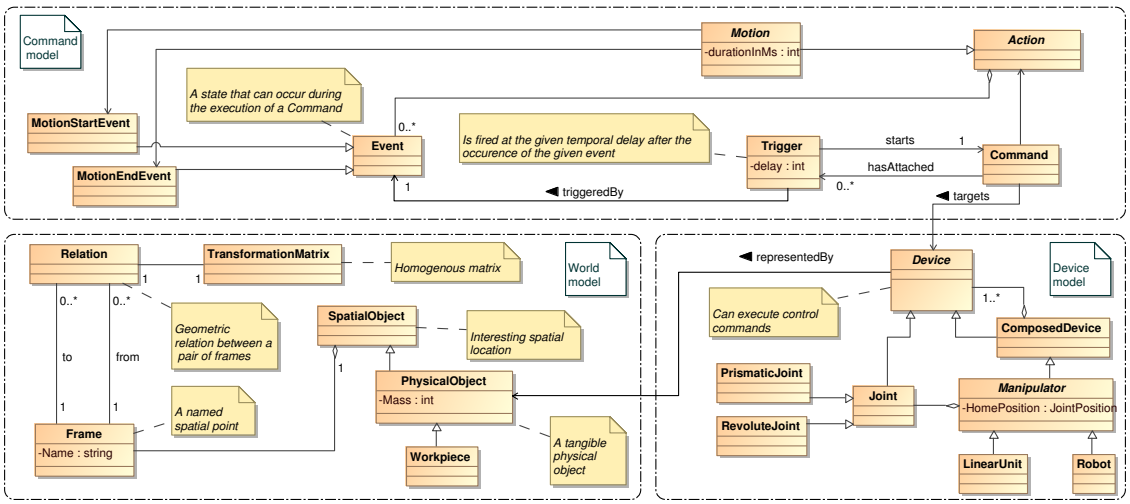

Fig. 1. Robotics API: basic class structure

exactly one SpatialObject. The *World* is a special SpatialObject defining the notion of a globally unique world concept. It knows a distinguished Frame *WorldOrigin*, which is a globally unique world frame.

- A *PhysicalObject* is a concrete object in the world and is a specialization of SpatialObject. Attributes like mass, size etc. can be defined for it.

When it comes to the type and number of devices that shall be controlled, classical robot controls can handle two kinds: (1) A single robot that can be controlled natively and (2) external (from the view of the robot control) periphery like tools, sensors and complete systems that the robot shall interact with. Periphery is usually connected to the robot control via field buses which can be directly read and written in most robot control languages. Languages like KRL provide additional commands for the control of certain periphery that abstract from the IO level by introducing control commands on a logical level.

The Robotics API provides basic classes to support all kinds of devices and to control them. The relevant part of the class structure is shown in the lower right part of Fig. 1:

- *Device* is the basic class for all kinds of mechanical devices that can be controlled, such as robots, grippers or any other kinematic structures. Any such Device must be controllable by the Robot Control Core (see Sec. II). Each Device can have a PhysicalObject attached that describes its physical properties. *ComposedDevice* is intended to be the base class for devices that are compositions of other devices, like e.g. a robot on a mobile platform. It is possible to control each single device of this composed device separately, or treat the composition as a whole, depending on the use case.
- A *Joint* represents a mechanical joint, which is the most basic element of any robot. The Robotics API supports *PrismaticJoints* and *RevoluteJoints*
- A *Manipulator* is an example of a common concrete device. It consists of several *Joints*, which are themselves Devices that can be controlled separately. This makes a Manipulator the simplest kind of ComposedDevice. One example for a concrete Manipulator is a *Robot*.

In object-oriented design and programming, objects usually carry their state with them as attributes, as well as methods operating on that state and representing the functionality of the respective object. Thus, one would expect that Robotics API devices contain several methods for performing usual actions (e.g. Robot.MoveLinear(), Gripper.Close()). However, the Robotics API models commands that shall be executed by devices as separate objects, which corresponds to the 'Command pattern' defined in [16]. This kind of modeling allows the flexible combination of multiple commands into a bigger, complex command. This realizes the core idea of encapsulating real-time critical action sequences. Sec. V explains this mechanism more in detail. Of course, convenience methods can be implemented that serve as shortcuts for commonly used functionality and rely on the Command pattern structure in their implementations, like the aforementioned movement methods in the class Robot. Below, the classes forming the Robotics APIs command model are explained:

- An *Action* represents some sort of operation that can be executed by certain devices (e.g. a *Motion* by a Manipulator).
- A *Command* specifies a Device and an Action the Device should execute.
- Actions contain a set of *Events*. An Event can occur when an Action is executed and describes that a certain state has been reached. Events are specific for each Action: E.g., a Motion has a *MotionStartEvent* and a *MotionEndEvent*.
- A *Trigger* can be attached to any Event. A Trigger starts a new Command when the respective Event has occurred, with a specified temporal delay. An arbitrary number of Triggers can be defined for each Event.

The Robotics API is an open, extendable framework for

high-level programming of industrial robot applications. By providing basic concepts for world modeling, devices and action specifications, it promotes reuse of logic common to such applications. Compared to the manipulator-level robot programming languages used in today's robot controls, the Robotics API also facilitates the notion of real-world objects like workpieces that are to be manipulated.

IV. Application Example

For evaluating the usefulness of the Robotics API as a basic framework for industrial applications, we created a draft implementation of a welding application as a proof of concept, based on the manual of the KUKA ArcTech Digital add-on technology package [17]. Welding is a typical use case for industrial robots and comprises many challenging aspects of programming robots:

- Configuring and controlling a robot and a welding tool
- Specifying points and welding lines that are, in general, specific to a type of workpiece
- Defining the welding work flow, including movements and synchronized tool actions

During implementation, we extended the Robotics API by classes that are common for welding tasks. In Fig. 2, those classes are shown in dark color, together with those classes of the Robotics API (in light color) that they relate to. The diagram is structured similar to Fig. 1.

For the welding application, the device model of the Robotics API had to be extended. Two new Device subclasses were introduced: *WeldingTorch* and *WeldingRobot*. WeldingTorch represents the tool that is used for welding and is modeled as ComposedDevice, consisting of multiple instances of *IODevice*. An IODevice is a generic representation of an input or output port of the robot control that can be read or written. In that way, the WeldingTorch class provides a high-level interface for configuring and controlling the device, while the information about the IO configuration stored in the IODevices can be used for mapping the high-level actions to input and output signals on the robot control. The class WeldingRobot is a ComposedDevice, too, which aggregates the used Robot and the WeldingTorch mounted at its flange. The WeldingRobot has the ability of moving the robot arm (with the correct center of motion, i.e. the tip of the WeldingTorch) and for executing a complex *Weld()* operation. This operation takes a *WeldingLine* as a parameter as well as specific *WeldingParameters*. Those parameters specify characteristic details of the welding operation like the timeout for the ignition of the welding arc. The WeldingLine consists of a sequence of motion definitions, which specify the welding seam that the WeldingRobot shall follow. The class *WeldedWorkpiece* is a subclass of Workpiece and knows a set of WeldingLines, so that large parts of a welding program can be reused when the WeldedWorkpiece is exchanged.

The most interesting part of our application is the implementation of the WeldingRobot.Weld() method. This operation performs a complete welding operation of a given WeldingLine. This operation consists of the following parts: (1) Perform the ignition movement to the start point of the welding seam and turn on the shielding gas flow. (2) After the defined gas preflow time, initiate the arc ignition. (3) As soon as the arc has been established, start the first motion along the seam. (4) Perform all necessary motions along the seam in a continuous manner. (5) As soon as the end of the last motion segment has been reached, turn off the arc. (6) Wait for the defined crater time and the defined gas postflow time and after that, turn off the shielding gas flow. (7) Finally, perform the final movement away from the workpiece.

Most parts of this operation have to be executed within defined timing windows. In particular, all movements have to be executed without any delay as soon as the arc has been established, otherwise the workpiece will be damaged. Though some other steps (e.g. waiting for the gas postflow) perhaps do not require hard real-time guarantees, we chose to implement all steps as one complex Robotics API Command. Listing 1 shows the first lines of code of the Weld() method.

```
// start the ignition movement
Command ignitionMovementCmd =
    new Command(Robot, line.IgnitionMovement);

// start gas depending on status of ignition movement
Command startGasCmd =
    new Command(WeldingTorch, new GasOn());

Trigger startGasTrigger = new Trigger(
    line.IgnitionMovement.OnMotionEnded,
    startGasCmd);

ignitionMovementCmd.AddTrigger(startGasTrigger);

// initialize the welding arc after the gas preflow time
Command startIgnition =
    new Command(WeldingTorch, new ArcOn());

Trigger startIgnitionTrigger = new Trigger(
    line.IgnitionMovement.OnMotionEnded,
    startIgnition,
    parameters.GasPreflowTime.Milliseconds);

ignitionMovementCmd.AddTrigger(startIgnitionTrigger);

... // code for steps 4-7 is omitted

ignitionMovementCmd.Execute();
```

Listing 1. Excerpt of the WeldingRobot.Weld() method.

The statements that are shown create a command for letting the Robot do the ignition movement, and attach triggers to the command, which start the gas preflow and initiate the ignition. The last line of the listing shows the end of the Weld() method, where the ignition movement is actually executed. This leads to the execution of all commands that are connected to the ignition movement command via events and triggers. The execution of all those commands is performed as one atomic step and with real-time guarantees considering the timing specifications.

Having defined those basic classes, every weld application can just be implemented as a series of calls of the Weld() method (corresponding to the WeldingLines defined on the WeldedWorkpiece) with adequate transfer movements in between that move the robot from one welding line to the next. Programmers of welding applications do not have to deal with any real-time aspects of their applications, as those are encapsulated inside the WeldingRobot's implementation.

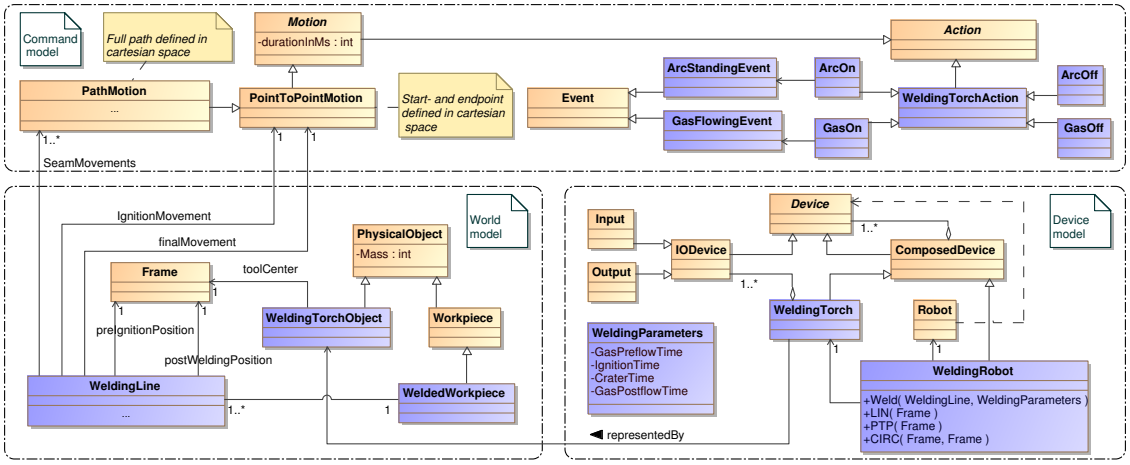

Fig. 2. Structure of a welding application on top of the Robotics API

The implementation of all the classes specific for welding functionality (shown in dark color in Fig. 2) took only about 100 lines of code, while all other parts (Robot, Command, Trigger etc.) could be re-used directly from the Robotics API.

V. Execution of Robotics API Commands

Using the Robotics API, complex and domain specific commands can be specified. However, to run these commands on real robots, they are converted into an executable form. Our approach uses dataflow specifications, which are expressed with RPI and describe the communication among various basic modules representing hardware, calculation and control algorithms.

These RPI commands can be sent to a compatible robot controller, where they are executed respecting hard real-time constraints. The controller also has to return status information about its executed commands and devices to the Robotics API layer. This way, the running application can be synchronized to the execution on the controller, and can always work on up-to-date state information about the existing devices.

The transformation of Robotics API commands into RPI commands is specific to the particular Actions, Devices and Events used, but follows a general pattern:

- *Devices* are turned into consumer modules featuring dataflow inputs specific to the type of actions the devices support. For example, a robot object in the Robotics API can be mapped into a module accepting Cartesian position values as an input. This module's implementation controls the physical robot, following the trajectory received on the input port.
- *Actions* are represented by modules producing data that will be processed by the devices. Motion actions thus become trajectory generator modules which calculate the desired position of the robot end effector at each interpolation cycle. Motion overlays or other action modifiers cause additional modules to be added which accept data produced by the primary action and calculate the corresponding overlay or modification.
- To enable control flow and conditional execution, the modules representing actions and devices have an input port *active* controlling whether the module shall be evaluated. These ports are connected to a module called *trigger* which can be controlled over its *on* and *off* inputs. This way, a trigger activating an action for a device can be transformed into a module evaluating the event condition and switching on the trigger for the action and device. Of course, this evaluation module has to be able to access status information about other running actions, provided as additional output ports by action modules.

Fig. 3 gives an example for a Robotics API command structure. It consists of the initial movement in a welding application (action *ignitionMovement*) executed by a certain device (*robot*), and enables the gas flow (action *gasOn*) of the attached *weldingTorch* (device) once the initial motion is completed (*startGasTrigger* triggered by a *motionEnded* event).

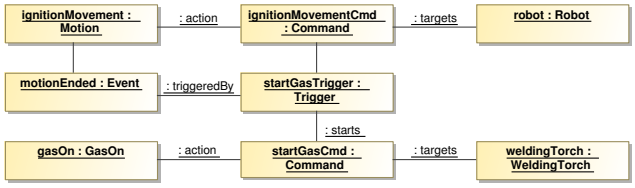

Fig. 3. Robotics API Command

The generated RPI command is given in Fig. 4. The top part shows the representation of the *ignitionMovementCmd* command, consisting of a *trajectory generator* as an implementation of the motion, and a *robot* module representing the controlled robot. The lower part implements the *startGasCmd* command by sending a binary value to the *digital output* the torch is attached to. The check and trigger modules in the lower left part check the progress of the trajectory generator and enable control of the digital output once the motion is completed.

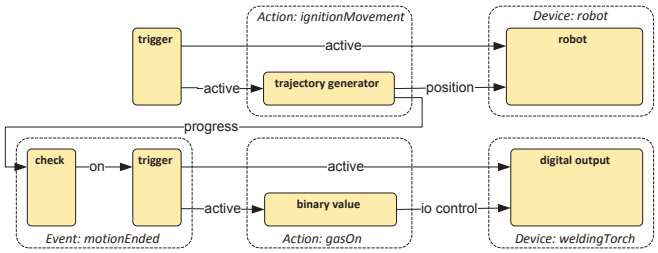

Fig. 4. Generated RPI net

VI. Conclusion & Future Work

In this paper, we have proposed the Robotics Application Programming Interface for developing software for industrial robots. It offers an object-oriented model for the (industrial) robotics domain and provides common functionality which can be easily used by developers. Main concepts are objects for robots, tools, frames or actions (e.g. motions or tool actions). The Robotics API is embedded into a software architecture and relies on a real-time capable robot control core. Actions which need precise timings (e.g. motions or switching actions) are encapsulated inside a command structure and will be executed atomically by the robot control. Developers can extend the Robotics API in order to create application-specific functionality or to add support for new devices. The welding example from Sect. IV is such an extension and introduces e.g. the composed device *WeldingRobot* with its (configuration) properties and actions.

Due to the high-level programming model and the tight, but hidden integration of low-level command execution, application developers are able to focus on solving the application-specific problems and, as far as possible, do not need profound knowledge in robot control and real-time programming. Extensions facilitate the reuse of application-specific functionality and promote a separation of concerns: Domain experts model and implement extensions while application developers use them. Furthermore, the Robotics API allows robotic applications to be developed using *standard* technologies and *non real-time* environments. The current implementation of our prototypical Robotics API is created as a class library based on Microsoft's C#, which is an object-oriented language on top of the .NET framework. With its built-in memory management, the .NET framework runtime and its languages are very robust against many common programming errors. Furthermore, the development environment Visual Studio provides extensive support for developing, modifying and testing applications. For the realization of the welding example (see IV), we also used C# and Visual Studio. This allowed a fast and clean implementation of this application. From our point of view, our proposed approach will improve productivity as well as quality in the development of robotics software [18] and can leverage the use of industrial robots in small and medium enterprises.

The approach has been successfully applied to program and control two KUKA lightweight robots, showing its advantages in software development for robotics. In order to prove its universal validity and the improvements in software quality, we are applying our approach to a set of more complex examples. Concerning the Robotics API, next steps include the introduction of sensors, extensions of the world model (e.g. including moving frames) as well as sophisticated error handling concepts. Moreover, we are currently extending our approach to program real-time cooperation tasks like load-sharing motions or rendezvous operations. The Robotics API was designed to support such advanced tasks as well, and we plan to verify that using the lightweight robots.

References

[1] J. N. Pires, G. Veiga, and R. Araújo, "Programming by demonstration in the coworker scenario for SMEs," *Industrial Robot*, vol. 36, no. 1, pp. 73–83, 2009.

[2] J. G. Ge and X. G. Yin, "An object oriented robot programming approach in robot served plastic injection molding application," in *Robotic Welding, Intelligence & Automation*, ser. Lect. Notes in Control & Information Sciences, vol. 362. Springer, 2007, pp. 91–97.

[3] M. Hägele, T. Skordas, S. Sagert, R. Bischoff, T. Brogårdh, and M. Dresselhaus, "Industrial Robot Automation," European Robotics Network, White Paper, Jul. 2005.

[4] D. J. Miller and R. C. Lennox, "An object-oriented environment for robot system architectures," in *Proc. 1990 IEEE Intl. Conf. on Robot. & Autom.*, Cincinnati, Ohio, USA, May 1990, pp. 352–361.

[5] C. Zieliński, "Object-oriented robot programming," *Robotica*, vol. 15, no. 1, pp. 41–48, 1997.

[6] M. S. Loffler, V. Chitrakaran, and D. M. Dawson, "Design and implementation of the Robotic Platform," *Journal of Intelligent and Robotic System*, vol. 39, pp. 105–129, 2004.

[7] D. Brugali and P. Scandurra, "Component-based robotic engineering (Part I)," *IEEE Robot. & Autom. Mag.*, vol. 16, no. 4, pp. 84–96, 2009.

[8] D. Brugali, A. Agah, B. MacDonald, I. A. Nesnas, and W. D. Smart, "Trends in robot software domain engineering," in *Software Engineering for Experimental Robotics*, ser. Springer Tracts in Advanced Robotics, D. Brugali, Ed. Springer, Apr. 2007, vol. 30.

[9] T. Collett, B. MacDonald, and B. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proc. 2005 Australasian Conf. on Robotics and Automation*, Sydney, Australia, Dec. 2005.

[10] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Proc. 2001 IEEE Intl. Conf. on Robot. & Autom., Seoul, Korea*. IEEE, May 2001, pp. 2523–2528.

[11] L. B. Becker and C. E. Pereira, "SIMOO-RT – An object oriented framework for the development of real-time industrial automation systems," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 4, pp. 421–430, Aug. 2002.

[12] 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 Intell. Robots and Systems, St. Louis, Missouri, USA*. IEEE, Oct. 2009, pp. 2108–2113.

[13] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Prentice Hall, 2004.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sornmerlad, and M. Stal, *Pattern-Oriented Software Architecture: A system of Patterns*. Wiley, 1996.

[15] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif, "Interfacing industrial robots using realtime primitives," in *Proc. 2010 IEEE Intl. Conf. on Autom. and Logistics, Hong Kong, China*. IEEE, Aug. 2010, pp. 468–473.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*. Addison Wesley, 1994.

[17] *KUKA.CR.ArcTech Digital 2.0*, KUKA Robot Group, 2008.

[18] 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 Inform. in Control, Autom. & Robot. (ICINCO 2010), Funchal, Madeira, Portugal*, vol. 2. INSTICC Press, Jun. 2010, pp. 437–440.