Journal of Software Engineering for Robotics 4(1), May 2013, 1-22
ISSN: 2035-3928

Robotics API: Object-Oriented Software Development for Industrial Robots

Andreas ANGERER[1,*] Alwin HOFFMANN[1] Andreas SCHIERL[1] Michael VISTEIN[1] Wolfgang REIF[1]
[1] Institute for Software & Systems Engineering, Department of Computer Science, University of Augsburg, Augsburg 86159, Germany

Abstract—Industrial robots are flexible machines that can be equipped with various sensors and tools to perform complex tasks. However, current robot programming languages are reaching their limits. They are not flexible and powerful enough to master the challenges posed by the intended future application areas. In the research project SoftRobot, a consortium of science and industry partners developed a software architecture that enables object-oriented software development for industrial robot systems using general-purpose programming languages. The requirements of current and future applications of industrial robots have been analysed and are reflected in the developed architecture. In this paper, an overview is given about this architecture as well as the goals that guided its development. A special focus is put on the design of the object-oriented Robotics API, which serves as a framework for developing complex robotic applications. It allows specifying real-time critical operations of robots and tools, including advanced concepts like sensor-based motions and multi-robot synchronization. The power and usefulness of the architecture is illustrated by several application examples. Its extensibility and reusability is evaluated and a comparison to other robotics frameworks is drawn.

Index Terms—Control Architectures and Programming, Cooperative Manipulators, Domain-specific architectures, Manipulation and Compliant Assembly, Object-oriented programming .

1 INTRODUCTION

ROBOTS are in general highly flexible machines. Most robotic devices, especially articulated arms used in industry today, have many degrees of freedom that allow them to adapt to a large variety of tasks and environments. Recent trends of integrating one or more robot arms and a mobile robot platform, like the PR2 [1] or TUM-Rosie [2] systems, increase the degree of freedom even further.

Exploiting the flexibility of robots to create systems that help humans has been subject to research for over three decades now. As of today, robots are still most frequently used in high-volume manufacturing industry, in particular the automotive industry. There, they perform repetitive and often heavy work. Besides their reliability and performance, the high quality delivered by robot manufacturing systems is one of their strong points. The *spatial flexibility* of robots is an important prerequisite in this application domain, as it enables robots to perform complex operations that would not be possible with less flexible manufacturing systems. However, the *process flexibility*, which means the adaptability of a system to changing tasks, is secondary in this context. In many cases the systems are installed and programmed with high effort to perform a particular task for a very long time.

This second dimension of flexibility, the process flexibility, can be considered the most important prerequisite for a broader use of industrial robots in the future [3]. The potential for many other applications, like small batch assembly or large-scale construction, has been recognized (c.f. Hägele et al. [4]) and much effort has been spent to embrace the use of robots in these fields. It can be stated that process flexibility of robot systems is mostly a *software issue*, compared to the spatial flexibility which can be seen predominantly as *hardware issue*. This separation is – due to the mechatronics nature of robot systems – not a sharp one. However, from the authors' point of view, the software design of robot control systems is the key to substantially improve process flexibility of robots and enable their use in many more areas.

Today, there is a variety of proprietary robot programming languages developed by manufacturers of robot systems. They have been developed some decades ago and have in general not evolved much since then. Though they offer features

Regular paper – *Manuscript received October 05, 2012; revised March 27, 2013.*

- *This work presents results of the research project* SoftRobot *which was funded by the European Union and the Bavarian government within the* High-Tech-Offensive Bayern. *The project was carried out together with KUKA Laboratories GmbH and MRK-Systeme GmbH and was kindly supported by VDI/VDE-IT GmbH.*

advantageous for industrial robot programming (e.g. hardware abstraction, real-time execution), they often suffer from other drawbacks. In the joint research project SoftRobot, the issue of software maturity in industrial robot controllers was addressed by the Institute for Software & Systems Engineering (ISSE) of the University of Augsburg, KUKA Laboratories GmbH (KUKA) and MRK-Systeme GmbH (MRK). The goal of the SoftRobot project was to develop a software architecture that enables software development for industrial robot systems with modern general-purpose programming languages. It should meet today's functional requirements to robot programming systems as well as identified requirements of future applications, while at the same time making the methods and tools of modern software engineering available in the robotics domain.

In this article, we present the software architecture that has been developed in the SoftRobot project, referred to as the *SoftRobot Architecture*. A special focus is put on the design of the object-oriented robot application framework called *Robotics API* that is part of this architecture. Sect. 2 gives a deeper insight into the current state of software development in industrial robotics and its special challenges. Based on that, the requirements and the resulting SoftRobot architecture are presented in Sect. 3. In Sect. 4 the software design of the Robotics API core is exposed in detail. Sect. 5 summarizes experiences made during the development of applications with the Robotics API and motivates the design of the Robotics API Activity Layer, which is particularly geared towards application developers. The structure of this layer is then illustrated in Sect. 6. Sect. 7 evaluates the practical usefulness as well as the extensibility, reusability and performance of the approach and draws a comparison to other work. Finally, we sum up our results and conclude the article in Sect. 8.

2 State of the Art in Industrial Robot Application Development

Industrial robots have been used in large-scale production areas for decades already. Thus, most manufacturers of industrial robot systems have a background in constructing robots and robot controllers tailored to the needs of those domains. Special programming languages and sophisticated tool chains have been developed to support application development. At some point in time, industrial robotics was even considered a solved problem (see Hägele et al. [5]). However, this situation is changing as robotics manufacturers are trying to expand to new markets. There is a general trend driven by industrial robot manufacturers as well as customers to use robots not only in large-scale manufacturing systems, but also for small batch sizes, as co-workers for humans and even for services in everyday life. This poses challenges to the existing hardware, but in particular to the existing software ecosystems of industrial robots, which are not able to sufficiently fulfil the changed requirements of those new domains. The rest of this section gives an overview about existing programming languages, tools and frameworks for industrial robots. Research approaches are omitted here, but are evaluated later in Sect. 7.4.

Industrial robot programming systems can be divided into so-called online and offline programming approaches. Online programming means creating robot programs with direct involvement of the physical robot system. Special programming languages and development tools exist for that purpose. Offline programming, in contrast, is possible without access to the robot and mostly involves CAD-based simulation environments. Both approaches and their relevance to this work are examined in this section.

Online programming is inevitable for using today's industrial robots, either for creating complete robot programs directly, or for adapting programs created with offline programming tools to environment details. For online programming, proprietary languages, varying among manufacturers, are predominantly employed. Examples are the KUKA Robot Language, ABB's robot programming language RAPID, or KAREL developed by FANUC Robotics. These languages are tailored to robot programming by using special commands for robot motion or controlling robot tools. Most robot controllers provide hand-held devices, often called teach pendants, for manually controlling the robot and creating robot programs at the same time.

KUKA's KRL is examined further as an exemplary robot language – many of its strengths and weaknesses can be found in other proprietary robot languages as well. KRL is an imperative language with a syntax similar to Basic/Pascal. It features typical control flow statements (if, for, while) as well as built-in commands for moving a robot arm and operating robot tools. Compared to general-purpose languages, it is limited in the following respects: 1) memory management, as there is e.g. no support of explicit memory allocation and memory manipulation, 2) parallelism, as KRL has no threading model, but only special statements for executing e.g. tool actions in parallel to robot motions and 3) comprehensive applications in general, as KRL has e.g. no support for file access and UI elements. KRL programs are executed by an interpreter, which provides some special features. By interpreting several statements in advance, motion planning beyond the currently executed movement is possible. Based on that, blending between successive motions is realized. The interpreter is able to start execution of a program at an arbitrary code line, execute programs step-wise (one step being either a single code line or a single motion statement) and also stopping and continuing program and motion execution at any time. Even backward execution of programs is possible, which is useful when fine-testing specific robot motions contained in a larger program. Program execution is guaranteed to be deterministic, i.e. a program that has been tested once is guaranteed to be executed in exactly the same way in the future.

Although KRL includes many features supporting basic steps in the development and parametrization of robot applications, it increasingly suffers from other shortcomings. Advanced motion control concepts required for future applications (cf. Hägele et al. [4]) like (hybrid) force/torque control or sensor-guided and sensor-guarded motion in general are not supported. Real-time cooperation among multiple robots is supported, but complex to use in practical scenarios, as it involves writing different programs for each robot and synchronizing them appropriately. For complex applications, the language lacks structuring mechanisms (e.g. object or component orientation). At the same time, connectivity to external systems is limited. The examples from Pires [6] and Ge [7] illustrate the efforts necessary for connecting high-level programs developed in common general-purpose languages to robot controllers and their languages.

Offline programming systems are offered by most manufacturers of industrial robots (e.g. KUKA.WorkVisual, the ABB RobotStudio or FANUC's Roboguide). Third party providers offer solutions that support robots from different manufacturers, e.g. RobotMotionCenter [8] by Blackbird Robotersysteme GmbH. Offline programming systems usually allow the modelling of robot operations by specifying them in virtual graphical environments, e.g. specifying motions by defining them directly on the surface of a three dimensional representation of the workpiece. These tools usually generate code for the robot programming language supported by the target system. Thus, they suffer from the same limitation in functionality as the underlying languages. Additionally, in most cases it is necessary to also use online programming for adapting generated programs to the actual physical environment.

From our point of view, the industrial robotics domain could profit from a software architecture that combines the necessary real-time aspects with a high-level interface for specifying robot programs. We propose to realize this interface based on a state-of-the-art general-purpose language. This promises various advantages: Modern general-purpose languages incorporate many of the advances of the software engineering domain from the past decade and thus provide powerful language features. The fact that such languages are supported by active communities implies that many tools and libraries are available that the robotics domain could also profit from. Additionally, there are many developers who already know those languages, compared to today's proprietary robot languages that usually have to be learned first. Finally, those languages themselves evolve driven by a strong community, thus robot manufacturers could be relieved from the need to extend and support their own language.

3 The SoftRobot Architecture

The overall goal in the development of the SoftRobot Architecture was to enable the programming of industrial robot systems with modern general-purpose languages. Additionally, the architecture should not rely on real-time capabilities of those languages for two reasons: 1) Developers should not have to care about real-time guarantees in the programs they create. 2) Languages with automatic (and therefore usually non-deterministic) memory management should be usable to relieve developers from manual memory management. However, dealing with real-time aspects is inevitable in an industrial robot control architecture, so an important research issue was to create an abstraction for real-time tasks in the SoftRobot Architecture. To find the right level for this abstraction, an analysis of a broad variety of typical industrial robot applications (e.g. gluing, welding, palletizing) was performed. As input, products of KUKA Robotics and MRK were used, as well as some applications of their customers. The results of the analysis showed that these applications embody only a small set of real-time critical tasks and a limited number of combinations of those tasks. For example, interpolation of a robot movement clearly has to be considered real-time critical. Executing a tool action during a robot motion (e.g. activating a welding torch) is an example of a real-time critical combination of tasks. Besides that, typical robot applications contain a workflow of different tasks that can itself be considered not hard real-time critical. The specification and execution of this workflow can thus be left to a standard programming language without hard real-time support.

Some additional requirements also largely influenced the resulting architecture. These requirements were again determined in discussions with all project partners. In these discussions, input from the companies' practical experiences and customer feedback as well as requirements identified in literature were taken into account. The following five requirements were identified to be of central importance to the design of the SoftRobot architecture:

1) **Support for sensor-guided and sensor-guarded operations.** The trend towards tighter integration of sensors into robot control, e.g. for force-controlled motion or monitoring safety properties, should be embraced in the architecture. The need for force-controlled operation in future automation tasks has been particularly stressed e.g. by Hägele et al. [5].
2) **Support for multi-robot applications.** Tight cooperation and synchronisation of multiple robots should be supported. KUKA shares the vision of more intensive use of multi-robot systems in future applications, stated e.g. by Hägele et al. [4]. Today's many commercial robot controllers support real-time robot cooperation (e.g. by KUKA [9] and ABB [10]). However, most controllers follow the paradigm 'one program per robot' which enforces writing multiple programs even in scenarios where it is not adequate and adds unnecessary complexity. The SoftRobot architecture should break this paradigm and allow for natural integration of multi-robot cooperation in any robotic application.
3) **Extensibility and reusability.** It should be possible to

extend the architecture to integrate new devices (sensors, robots, etc.), operations (new movements, control algorithms, etc.) or even new programming languages/interfaces. Extensibility has often been mentioned as an important aspect of reusability, e.g. in the context of the Player/Stage project by Vaughan [11] or the OROCOS project by Bruyninckx [12]. A high degree of reusability of generic concepts will decrease the effort required to extend the architecture.

4) **Different user groups.** Different kinds of users should be able to use or extend the architecture. For example, customers using robots in their factories need simple, quick programming interfaces, while system integrators need access on a lower level to integrate robots into complex multi-robot or multi-sensor systems. This requirement was driven by KUKA as hardware vendor and seller of robot systems and MRK as a system integrator.

5) **Support concepts specific to industrial robotics.** Industrial robot controllers have developed some special features that are also reflected at the programming language level. One of the most important special concepts is motion blending, which means the continuous execution of successive motions stated in a robot program. Other examples are single-step and backwards execution of robot programs as well as direct selection of motion records to be executed. The newly developed architecture should also take such features into account and should make them accessible to programmers in an easy way. This requirement was heavily influenced by the experiences of KUKA and its customers.

To achieve all the aforementioned goals, we developed a two-tier software architecture (introduced by Hoffmann et al. [13]), which is depicted in Fig. 1. Boxes inside the Robotics API and RCC parts represent important concepts. Similar to UML, simple lines indicate associations between concepts and lines with filled arrow indicate generalizations. Dotted lines between Robotics API and RCC concepts indicate that some concept of the Robotics API is transformed to some other concept of the RCC during runtime.

The architecture's most important aspect is the separation between the *Robot Control Core* tier and the *Robotics API* tier. The Robot Control Core (RCC) is responsible for real-time hardware control. It exhibits the *Realtime Primitives Interface (RPI)*, described in previous work by Vistein et al. [14], that was designed to flexibly specify fine-grained real-time critical commands. The Robotics API uses this interface for tasks that need to be executed deterministically and with real-time guarantees, like e.g. robot movements, and also provides mechanisms for the real-time critical combination of those tasks by employing the flexibility of RPI. All such tasks and task combinations are sent to the RCC for execution. From the perspective of a Robotics API application, this execution is done atomically. The Robotics API is implemented in Java.

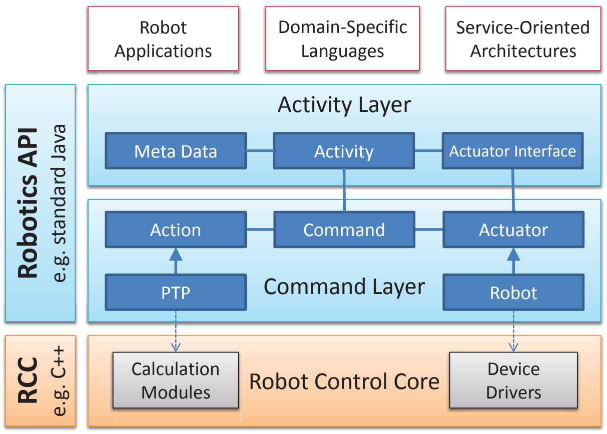

Fig. 1. The *SoftRobot* architecture.

It is divided into two layers, called the *Command Layer* and the *Activity Layer*. The following paragraphs take a closer look at both tiers (as well as the layers of the Robotics API) and their roles considering the goals stated above. The detailed design of each Robotics API layer is presented in Sect. 4 and Sect. 6.

3.1 Robot Control Core

The Robot Control Core accepts commands specified in the RPI dataflow language, consisting of calculation modules with defined input and output ports that may be interconnected by typed dataflow links. Calculation modules may use device drivers to send data to or read data from devices. Commands are specified as graphs of interconnected calculation modules, thus they are also called *RPI graphs*. Such a graph is executed cyclically at a high rate (typically 1kHz). This execution is performed deterministically by a real-time runtime environment. In each execution cycle, the RPI graph is fully evaluated. This dataflow-based approach is used to implement all functionality that is required by the higher levels. For example, the interpolation of pre-planned robot actions (like e.g. motions along defined cartesian paths), actions that are partially or completely based on sensor measurements, and reactive behaviour (e.g. stopping a motion at the occurrence of a certain event) are all encoded as RPI dataflow graphs. In this respect, the RCC does not offer a diverse set of different operations, but exposes only its capability to interpret RPI graphs (and quickly switch from one graph to the next, see below). All high-level functionality that is explained in this work is achieved by encoding it appropriately as an RPI dataflow graph.

Within the SoftRobot project, a reference implementation of a Robot Control Core was created based on the Orocos framework [12]. We call this implementation SoftRobot RCC. It makes use of some scheduling features of Orocos provided through TaskContexts as well as kinematic calculation functions provided by the KDL. However, the concept of a

Robot Control Core is not in any way bound to Orocos, thus other (hard real-time capable) frameworks or a pure C/C++ implementation may be used.

On the RCC level, sensors can be integrated into commands in arbitrary ways, as their measurement data is usually accessible via output ports of calculation modules. The real-time execution of the dataflow graphs can also be the basis for synchronisation of multiple devices. The RCC furthermore supports adding new device drivers and calculation modules (even at runtime). In sum, the Reqs. 1, 2 and 3 are considered already at the lowest architectural level. It turned out that supporting some of the special concepts of industrial robot control (Req. 5) also requires support on the RCC level. Instantaneous switching between the executed dataflow graphs (see a previous paper of Vistein et al. [15]) can be used to realize e.g. motion blending. Sensor guided motions, in particular force-based manipulation, also profit from this concept. The SoftRobot RCC was extended to support instantaneous command switching. Regarding Req. 4, the RCC tier is targeted at device manufacturers that need to develop and test new device drivers and calculation modules for their hardware.

3.2 Robotics API Command Layer

The Robotics API Command Layer defines a model for specifying operations that should be executed by devices, and for defining combinations of such operations. The most basic way of defining an operation is employing a *Command* that tells an *Actuator* to execute a certain *Action*. Commands can be composed to form more complex Command structures. Composition is based on an event mechanism, with events being provided by Actuators, Sensors and Actions, or Commands themselves (due to their life-cycle, e.g. Command has ended). This will be described in detail in Sect. 4. At this point, it is important to note that the Robotics API relies on the Robot Control Core for executing these operations with real-time guarantees. The reference implementation of the Robotics API includes a generic and extensible algorithm for transforming high-level concepts (Actions, Actuators and Commands, as well as all concepts regarding composition of Commands) to the RPI language that is accepted by the SoftRobot RCC. Details about this algorithm are out of the scope of this work, but can be found in previous work of Schierl et al. [16]. The concept of transforming complex Robotics API Commands, which contain continuous as well as discrete, event-based aspects, to an RPI dataflow graph is the key to achieve the proposed abstraction for real-time matters in the SoftRobot architecture. Besides specifying the structure of single complex Commands, the Robotics API also supports defining conditions at which it is possible to switch between separate Robotics API Commands. Based on this, the Robot Control Core performs instantaneous switching from a running RPI dataflow graph to its successor.

The requirements stated above are also reflected in this layer. Due to the Command composition mechanisms, various flavours of sensor-based operations are possible (Req. 1). Sensor guided motion is supported as well and is in general realized by concrete Action implementations. Considering Req. 2, the notion of multiple robots in one program is achieved by employing natural principles of object orientation, i.e. having multiple instances of the same class (e.g. Robot) and having different (sub-)classes for different real-world concepts (e.g. different Robot types). Coordination and synchronisation of multiple robots is supported by Command composition. By employing the Command switching mechanism described above, motion blending can be realized. Single-step execution of programs can be achieved using e.g. a debugger, as will be detailed in the next section. In this way, the Command Layer accounts for Req. 5. To support extensibility (Req. 3) the Robotics API as a whole employs a plug-in structure to support new types of Actions, Actuators and Sensors. The user groups intended to work on this layer (Req. 4) are robot manufacturers as well as system integrators. Robot manufacturers might extend this layer by introducing new Actuators and Actions as counterparts to extensions to the RCC. System integrators may implement new Actions as well, or use the Command model to implement complex operations as part of an application or to provide them to end users in a specialized API.

3.3 Robotics API Activity Layer

The Activity Layer in the Robotics API extends the Command Layer and provides an easy-to-use programming interface to developers of robotics applications. While the Command Layer is concerned about basic operations of devices (like a robot executing a linear motion) and very generic combination mechanisms, the Activity Layer adds concepts frequently used in application development, taking into consideration some robotics-specific requirements. On this level, Actuators offer a set of *ActuatorInterfaces*, each providing specific *Activities* that the Actuator may execute. For instance, robots offer a LinInterface, providing Activities for different kinds of linear motions (e.g. to absolute goals in space or to goals specified relative to the current position). Activities are characterized by a particularly designed asynchronous execution semantics, which will be described in detail later. This semantics allows for easily specifying continuous execution of real-time operations on a programming language level. In addition, there exist some predefined ways of combining Activities (e.g. for parallel or conditional execution). As all Activities and their combinations are implemented on top of the Command Layer mechanisms, the real-time properties are preserved.

Using the Activity Layer, defining sensor-based motion guards is still possible (Req. 1). However, flexibility is limited in some respect, in favor of a slimmer API. Different kinds of sensor guided motions are supported and realized by special Activity implementations. Furthermore, special implementations of ActuatorInterfaces and Activities provide support for particular multi-robot operations like synchronized motions

(Req. 2). Like Actuators and Actions, new ActuatorInterfaces and Activities may be added using the Robotics API's plug-in mechanism to provide extensibility (Req. 3). Robot manufacturers or system integrators might do this to provide new functionality to application developers (Req. 4). The Activity Layer is particularly geared towards those developers and provides them with an easy-to-use programming interface.

As mentioned above (Req. 5), developers of robotic applications are used to some unique features of robot programming languages and their execution environments. The Activity Layer therefore supports easy specification of continuous motion instructions, which is described in more detail later. Single-step execution of (motion) instructions can be achieved by Java debugging. As to not confuse developers by stepping through internal operations of the Robotics API, the debugger can be configured appropriately. For example, the debugger of the Eclipse IDE provides the concept of step filters that control which methods or classes should not be stepped into during debugging. We are already working on an Eclipse Plugin that supports application development with the Robotics API. This plugin can be easily extended to adequately pre-configure the Eclipse debugger for robotic applications. However, developers have to take care if their program uses the ability of the Robotics API to execute Commands or Activities asynchronously. This will decouple the Robotics API program flow from the robot operations that have been triggered. While the program flow may be paused by the debugger, the robot operation that has been triggered before will not, which developers have to be aware of.

Considering direct selection of motion records, we did experiments with extending Eclipse with a special program launcher. It uses Java Reflection to allow developers to select only certain methods to be executed. While this is not as convenient as e.g. KRLs ability to jump to single motion statements, it can ease development and testing when the program is adequately structured. Backwards execution of robot programs is a feature that we did not tackle yet with our approach. Modifying the Java Virtual Machine to allow true backwards execution seems hard to realize. However, for practical cases it might be helpful to extend at least some Activities by a kind of "'undo"' functionality that e.g. could be used to drive backwards. Another possible approach is to record motion interpolation and enable a reverse playback of the interpolated values. These approaches would at least ease the testing of series of subsequent motions. However, this inability to completely match a domain specific language like KRL has to be accepted as a limitation when using a general-purpose language.

The past sections illustrated the separation between the different tiers and layers in the SoftRobot architecture and defined the responsibilities of each part. The following sections will go into details about the design of the Robotics API tier with its two layers.

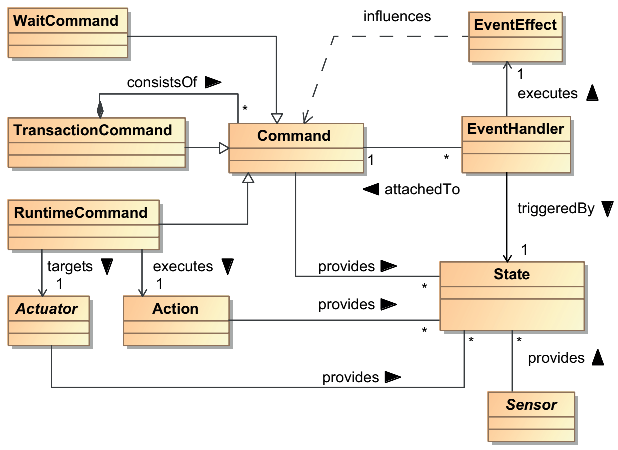

Fig. 2. Structure of the Robotics API Core, including Commands and their composition

4 Design of the Robotics API Core

The following sections will present the software design of the Robotics API core and the Command Layer in this core. The decisions that led to its design will be illustrated. The design of the Activity Layer is described separately in Sect. 6.

4.1 Actuators and Actions

Actuator (cf. Fig. 2, bottom left) can be considered one of the most basic classes in the Robotics API. An Actuator is some robotic device that is controllable by the application. Examples are single robot joints, complete robot arms, mobile platforms or just field bus outputs of some robot control system. Actuators have some properties (e.g. the number of joints of a robot) and can be configured in certain ways (e.g. defining the maximum allowed velocity of motions). Actuators in the Robotics API usually do not contain code that implements the execution of any kind of operation (e.g. for interpolating motions of a robot arm). They can rather be seen as proxy objects representing devices controllable by the Robot Control Core. Instances of Actuators should not store any state of the physical device they represent. Instead, they usually retrieve their data by directly communicating with the RCC when necessary. This ensures that the data provided is always up-to-date.

Control operations that Actuators should execute are modelled by the class *Action* (cf. Fig. 2, bottom left). Examples for concrete subclasses are LIN (linear motion of a robotic device in Cartesian space), SetValue (setting a value for a field bus output) or MoveTo (telling e.g. a robot base to move to a certain goal). Actions carry parameters that specify their execution more exactly (e.g. the velocity of a linear motion). Like Actuators, Actions as well do not contain code specifying how their execution is implemented exactly. Actions can rather be seen as a kind of token, semantically identifying a certain operation. Actions are not even bound to a specific Actuator: An Action specifying a linear Cartesian movement may be executed by a robot arm as well as a mobile platform, or even by some Actuator that represents a mobile robot with an arm (i.e. the combination of both). The different Actuators might interpret the Action slightly different, but should obey the general semantics, i.e. moving linearly.

The splitting of Actuator and Action means separating *who* should do something and *what* should be done. This separation of concerns is based on the following assumptions:

- Actions are atomic, real-time critical operations that have to be executed by a Robot Control Core. Therefore, Actions in the Robotics API can only describe *what* is to be done, whereas the Robot Control Core decides *how* it should be performed by implementing it.
- There is no defined, finite set of operations that a certain Actuator can execute (e.g. a robot arm is able to perform arbitrary application-specific operations). Thus, separating the definition of Actions from the definition of the Actuator itself provides the possibility of defining and using arbitrary Actions in a uniform way.

To actually let an Actuator execute an Action, a *Command* has to be defined. The definition and composition of Commands is explained in Sect. 4.3. As large parts of the composition mechanism are based on the concepts of *Sensors* and *States*, the next section will explain those first.

4.2 Sensors and States

Sensors (cf. Fig. 2, bottom right) play an important role in many parts of the Robotics API. Sensors are in general sources of data which is available in a Robotics API application as well as in the Robot Control Core layer. This means that Sensors are, like Actuators, proxy objects for data sources located in the RCC layer. Examples are "typical" sensor devices, such as laser rangefinders, light barriers, or field bus inputs. Sensors can also represent data sources inside Actuators, e.g. sensors measuring joint angles of a robot. A third type are derived Sensors which deliver results of calculations on values of other Sensors, e.g. a *MultipliedDoubleSensor* delivering the result of the multiplication of two *DoubleSensors*.

Sensors may be used for certain Actions. For example, an Action modelling a force-based robot motion might be parametrized with a concrete Sensor delivering force measurement values. A second very powerful use of Sensors is related to the real-time event model of the Robotics API. The basic concept of this model is the class *State* (cf. Fig. 2). A State represents some condition that may be true or false at a given point in time. If the condition is true, the State is *active*, otherwise *inactive*. Sensors provide methods to select different kinds of States. For instance, all DoubleSensors provide the methods `isGreater()` and `isLess()` which return States having the respective semantics. States can again be combined. E.g., the method `and(State)` of class State returns a new State that is active only if both of the former States are active at the same point in time. Note that this definition of a State differs from the role of states in the frequently used state machine concept. In the Robotics API, a State captures a certain interesting condition from an otherwise infinite state space of a robotic system. The approach of defining States only for those aspects of the system's state that are relevant to the task proved to fit the requirements of typical industrial robotics applications well.

Like each Sensor, also every State can be employed in real-time critical robot tasks. This means that every possible definition of a State needs to have a semantically equivalent implementation on the RCC layer. The next section explains the interrelation between Actuators, Actions, Sensors and States when defining real-time Commands.

4.3 Composing Real-Time Commands

The abstract class *Command* (cf. Fig. 2) is the central concept modelling a real-time critical operation in the Robotics API. It generalizes three kinds of operations:

- *RuntimeCommand* is the most basic kind of Command, binding an Action to an Actuator. The semantics simply is that the defined Actuator should execute the given Action.
- *TransactionCommand* is a composition of other Commands according to rules specified by a set of pre-defined mechanisms. All commands contained in a TransactionCommand are executed in the same real-time context.
- *WaitCommand* models a real-time critical timer that waits exactly for a given time.

To each Command, an arbitrary number of *EventHandlers* can be attached. An EventHandler reacts to the change in the activeness of a given State. It can be parametrized to react to the event that the State is becoming either active or inactive, and the reaction can be limited to the first occurrence of such an event. The handling logic for each EventHandler is specified by one of the following *EventEffects*:

- *Start* triggers execution of a Command.
- *Stop* forcefully aborts a Command.
- *Cancel* gracefully cancels a Command.
- *Raise* activates another State (to which further EventHandlers might be listening).
- *External* denotes an EventEffect that triggers some logic *outside* of the respective Command, thus leaving the real-time context. An example for an external EventEffect is starting some non-real-time Robotics API thread.

The difference between Stop and Cancel needs some further explanation. Stopping a Command instantaneously does not give it time to clean up and can lead to unexpected consequences. For example, stopping a robot motion instantaneously is physically impossible, as the robot always needs to decelerate. Stopping Commands in general leaves Actuators in an uncontrolled state, e.g., causes an emergency stop of motor controllers of a robot arm. Thus, it should only be used in extreme cases. Instead, Cancel is preferred to give the cancelled Command the opportunity to terminate in a controlled way. For example, in case of a robot motion, Cancel should brake the robot until halt, and then terminate the command. When a TransactionCommand is cancelled, its concrete definition decides how to handle the Cancel request (e.g., forwarding it to all or only some of its inner Commands). In contrast, stopping a TransactionCommand always stops all inner Commands immediately.

Some additional rules constrain the use of certain EventEffects: Start may only be used to start another Command inside the same TransactionCommand. So Start cannot be used at all in RuntimeCommands. Stop and Cancel may target the Command itself or one of the inner Commands of a TransactionCommand. These rules are enforced at runtime.

Sensors have been identified as possible providers of States in Sect. 4.2. However, States can also be provided by Actions, Actuators and Commands. *Action States* describe e.g. certain progress or error conditions of an action. Examples are States telling that a certain via-point of a trajectory has been passed, or that the Action has started or completed execution. *Actuator States* include certain error states of the Actuator (e.g. that a set-point commanded by the Action is invalid or that an emergency stop has occurred), and a completion state (whenever the Actuator has reached its latest set-point). *Command States* cover the life-cycle of each Commands by e.g. indicating that it has been started, stopped or cancelled.

Through the mechanisms presented above, the Robotics API's Command model allows the specification of arbitrarily complex operations. However, those operations have to be finite, i.e. can not contain loops. In particular, it is not allowed to re-start Commands that have already been executed. Thus, complex control flow involving looping has to be realized in Robotics API applications, using (non-real-time) mechanisms of the programming language. This decision led to less problems in the algorithm transforming Commands to RPI graphs. Up to now, no practical limitations arose due to this. We are, however, investigating means of relaxing or dropping this restriction.

Fig. 3 shows an example Command that has been used in the implementation of the Factory 2020 application (see Sect. 7.1). The figure is simplified a bit to improve readability. It shows the structure of a TransactionCommand named `screwing` which implements the process of inserting and tightening a screw with a Light Weight Robot arm with attached screwdriver. This TransactionCommand consists of three RuntimeCommands: `insertScrew`, `maintainForce` and `tightenScrew`. `insertScrew` drives linearly towards the screw hole until the robot's force sensor detects that a force threshold is exceeded (i.e., the hole was reached). In this case, `insertScrew` is cancelled and the linear motion is stopped. This sensor guarding is implemented by a combination of the DoubleSensor measuring the absolute force the LWR arm detects, an adequately defined SensorState and an EventHandler

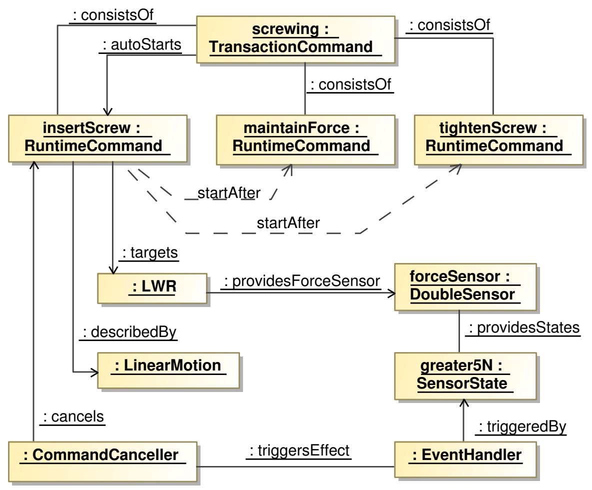

Fig. 3. Object diagram of a Command that inserts and tightens a screw.

triggering a CommandCanceller when this State is entered. After the `insertScrew` RuntimeCommand has completed, `maintainForce` and `tightenScrew` are started in parallel. The implementation of the starting process is left out for clarity and just indicated by the dotted arrows marked with `startAfter`. It is realized by an EventHandler that reacts to the CompletedState of the `insertScrew` command and triggers a CommandStarter for the respective Command. Though this example is only the last step in a series of operations needed to insert screws in the Factory 2020 application, it illustrates the flexibility of the Robotics API for defining realtime critical operations.

4.4 Executing Real-Time Commands

As noted before, execution of Commands is required to be performed with real-time guarantees. The SoftRobot architecture requires a Robot Control Core to take care of such tasks (cf. Sect. 3). The Robotics API is designed to be independent of a concrete RCC implementation. The concept *RoboticsRuntime* in the Robotics API layer serves as an adapter to the RCC which can accept Command structures and transform them in such a way that the concrete RCC implementation can execute them. Multiple implementations of a RoboticsRuntime may be employed, serving as adapters to different kinds of Robot Control Cores. The SoftRobot architecture reference implementation employs a Robotics-Runtime that transforms Robotics API Commands to realtime dataflow graphs according to the Realtime Primitives Interface specification (cf. Sect. 3). This article will not go into details about this transformation process, but rather focus on the semantics of Command execution from the view of a Robotics API application. This semantics is required to be satisfied by any RoboticsRuntime implementation.

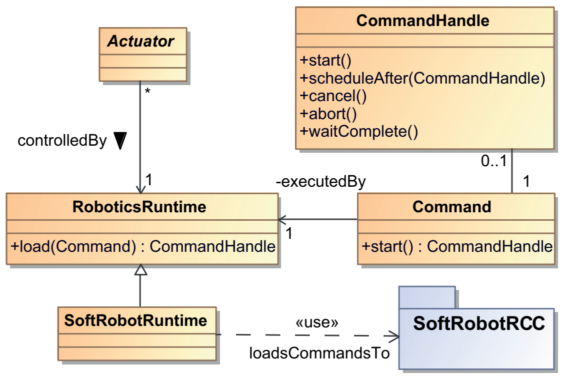

Fig. 4. A RoboticsRuntime serves as an adapter for executing Commands on a Robot Control Core

Fig. 4 depicts the classes that participate in Command execution. The class RoboticsRuntime provides a method `load(Command)`. This method is responsible for performing any step that is necessary to load a Command to a Robot Control Core. In case of the SoftRobot RCC, the corresponding RoboticsRuntime implementation transforms the Command into an RPI real-time primitive graph according to an algorithm described by Schierl et al. [16]. After transformation, the result is serialized and transmitted to the SoftRobot RCC via a network connection. There, it is de-serialized and the appropriate initialization of RPI modules is performed.

The `load()` method of the RoboticsRuntime returns a *CommandHandle* that can be used to control execution on the Robot Control Core via the following methods:

- `start()` begins execution of the Command associated with the CommandHandle.
- `scheduleAfter(CommandHandle)` schedules the Command for execution right after another Command that is already running, identified by the respective CommandHandle.
- `cancel()` sends a Cancel signal to the Command while it is running.
- `abort()` terminates the Command instantaneously while it is running.
- `waitComplete()` blocks the caller until the Command has finished execution.

The method `scheduleAfter(CommandHandle)` was introduced to realize an instantaneous transition between Commands as described in [15]. The Robot Control Core has to guarantee that this transition takes place in a defined, very short time frame (e.g., without losing an execution cycle in case of the SoftRobot RCC). Using this mechanism, continuous execution of Commands is possible, which is useful e.g. for blending motions or force based manipulation. Sect. 6 goes into details about how continuous Command execution is employed by the Activity Layer. The difference between `cancel()` and `abort()` is also worth noting. The semantics of both methods is similar to that of the EventEffects Cancel and Abort used inside composed Commands (see Sect. 4.3), but applies to the Command as a whole.

From the view of a Robotics API application, Command execution in general is an atomic process that can only be influenced by cancelling or terminating the Command. This means that all factors influencing its execution (e.g. stopping when a certain contact force is exceeded, using sensor input for visual servoing) must be known before a Command is started and be integrated in its definition (cf. Fig. 3). This assumption makes it possible to execute the Command completely in a real-time context. There are some exceptions to this rule that may be introduced mainly by concrete Action implementations. An example is the Action CartesianJogging, which is intended to be used for manual control of robots on a velocity level. This can be useful for e.g. teaching robot motions using a joystick. CartesianJogging provides a communication channel for setting the target velocity from a Robotics API application during runtime of a Command employing this Action. There are no guarantees about the time it takes to transmit control values from the Robotics API application to the running Command, so this channel cannot be used for real-time critical data. However, for Actions like CartesianJogging, this is acceptable. The Robotics API provides support for defining such communication channels if needed.

5 From Commands to Activities

The Robotics API's Command model provides a very powerful, flexible basis for defining real-time critical robot operations. It was designed to support all requirements of today's industrial robotics applications as well as tight integration of sensor based and multi-robot operations in future applications. However, the generality of this model sometimes leads to cumbersomeness when using it directly for the development of robotic applications. In particular, applications created on top of the Robotics API Command Layer looked complicated compared to applications created with KUKA's robot language KRL. This observation is not too surprising, as KRL is designed as a Domain Specific Language. It was developed solely for the purpose of creating industrial robotics applications, whereas the Robotics API wants to extend a general-purpose language for supporting development of such applications. In a sense, both architectures are approaching the problem of finding an optimal industrial robot language from opposing sides: KRL increasingly suffers from the narrowness of its design, while being tailored to easy programming of typical tasks of industrial robots. On the other side, the Robotics API has its strong points in the flexibility and power of a modern language, but lacks the minimalistic, domain-centred style of KRL (or similar DSLs).

To overcome this problem in the Robotics API, several possibilities were identified:

1) *Revising the design of the Robotics API Command Layer.* In a re-design of the Command model, a stronger focus

could be put on the interface to application developers, reducing its complexity.

2) *Introducing an application development layer on top of the Command Layer.* A separate layer on top of the Robotics API Command layer could be designed to introduce an interface more focused on the needs of robotics application developers.
3) *Creating a Domain Specific Language on top of the Robotics API.* There are several powerful tools and frameworks for creating DSLs on top of modern programming languages. Using those tools, developing a DSL for industrial robotics should be possible with moderate effort.

The first of those approaches has the advantage of staying within one single layer in the Robotics API. No separate layer on top of the Robotics API core would have to be introduced, adding no additional complexity to the design. However, the following considerations led to a decision against this solution. The current design of the Robotics API core and its Command Layer proved to be a stable and flexible platform for specifying Commands that can be used as a description of real-time critical operations to be executed by a Robot Control Core. This in itself defines a clear scope of responsibility. Following the principle of Separation of Concerns (attributed to Dijkstra [17]), it seemed logical to choose one of the latter solutions. Additionally, all stated requirements like multi-robot coordination and sensor integration could be fulfilled by the current Command Layer design. Changing this design according to completely different requirements like a minimalistic programming interface bears the danger of sacrificing some of the other requirements.

This led to the decision to establish a separate interface, being easy to use and particularly tailored to the needs of application developers. The introduction of a DSL seemed like the ultimate way of tailoring such an interface to the requirements. However, one of the strong points of the Robotics API is the Java ecosystem with its many libraries for different purposes. This was identified to be useful also for the development of robotics applications, e.g. for integrating computer vision algorithms or developing intuitive human-robot interfaces. The use of a DSL for developing robot applications would arise the question of interoperability with Java.

To stay in the Java ecosystem, we decided to introduce a separate Java-based layer on top of the Robotics API. This layer should provide an abstraction of the Robotics API Command model that reflects the mechanisms useful for robotics developers. In particular, features like motion blending should be easily usable by providing appropriate API options. The next section presents the design of the Activity Layer in the Robotics API. The Command Layer is not coupled to the Activity Layer, thus the Robotics API may still be used without the Activity Layer extension. The Activity Layer can be considered one of possibly several kinds of interfaces offered in the future towards application developers.

6 Modeling Operations by Activities

The separation of Action and Actuator in the Robotics API Command Layer is a very flexible mechanism (cf. 4.1), but at the same time a source of complexity when implementing robot operations. Instead of writing something like

```
robot.lin(goal);
```

for letting a robot execute a linear motion to a given goal, developers have to write

```
robot.getRuntime().createRuntimeCommand(robot,
    new Lin(start, goal)).execute();
```

using the Robotics API Command Layer. The additional complexity is caused on the one hand by the need to create a RuntimeCommand for defining Actuator and Action to be executed. On the other hand, Actions often need many parameters for fully specifying them. In this case, the Lin Action requires a start point of the motion to be specified in addition to its goal point. Intuitively, the current position of the robot at the time the motion command is issued could implicitly be taken as start position. This would enable a more compact specification of the Lin Action like presented in the first code snippet above. However, any RuntimeCommand can be used arbitrarily inside larger TransactionCommands. When a TransactionCommand is processed in the RoboticsRuntime, the full specification of all Actions is already required. In the Lin example, information about the start and goal position of a motion is needed to calculate a correct trajectory, which is then interpolated with real-time constraints by the Robot Control Core. Determining the robot's current position at the point in time that the RuntimeCommand containing the Lin Action is actually executed is in general not possible. It depends largely on the Commands in the TransactionCommand that are to be executed before the current RuntimeCommand, which might also move the robot.

Another frequently used feature in industrial robot programming is continuous execution of motions with blending between single motion instructions. In commercial robot languages like KRL, this feature is enabled by just adding a special keyword to the motion instruction that may be blended over to a subsequent motion. The Robotics API Command Layer supports motion blending between successive Commands as well, backed by RPI's instantaneous switching mechanism (cf. Sect. 3.1 and [15]). However, specifying appropriate Robotics API motion Commands that handle correct blending in between them is quite complex. Thus, application developers should not have to do that themselves.

A reduction of this kind of complexity on API level is achieved by two different mechanisms. The first one is the introduction of the concept *Activity*, which is presented in Sect. 6.1. The second one involves creating Activities with the help of *ActuatorInterfaces*, introduced in Sect. 6.3. By combining these two concepts, the resulting Activity Layer offers a special interface to developers, which is illustrated with several examples. The Activity Layer also provides a set of predefined mechanisms for composing Activities. Sect. 6.2 gives an overview of this.

6.1 Specification and Execution of Activities

An Activity is defined as a real-time critical operation, affecting one or more Actuators, that supplies meta data about the state of each Actuator during or after the execution of the operation. The real-time critical execution logic of an Activity is implemented by a Command. The execution of Activities is controlled by an *ActivityScheduler*. This scheduler is a singleton object, meaning that there exists exactly one global instance of it in each Robotics API application.

Activities expose among others the public methods `execute()` and `beginExecute()`. The former starts execution of the operation modelled by the Activity and blocks the calling thread until execution fully completes. The latter method also starts execution, but returns control to the calling thread once the operation is actually started (i.e. the respective Command has been started on the Robot Control Core). This asynchronous execution gives the caller the possibility to start execution of a second Activity shortly after an Activity has been started.

The execution of each Activity is controlled internally by the ActivityScheduler. Therefore, the implementations of `Activity#execute()` and `Activity#beginExecute()` submit the Activity for execution to the ActivityScheduler by calling appropriate methods. The ActivityScheduler stores, for each Actuator, the last Activity that has been started and affected this Actuator. Thus, when a new Activity is to be scheduled, the ActivityScheduler can supply all relevant preceding Activities to the new Activity. Note that there might be multiple (or no) preceding Activities, as multiple Actuators affected by the new Activity might have executed different Activities before. The new Activity can inspect the preceding Activities and their meta-data and can extract information needed for its own execution. For example, a motion Activity a is interested in the Actuator's state during the execution of a preceding motion p. If p is already completed at the time of a's start, a might take the robot's current position as its starting point. The same applies if there is no Activity preceding a (e.g. after program startup). If p is still running, a might inspect meta data provided by p about the robot's motion (position, velocity, acceleration, ...) at some future point in time where the motion executed by p should be blended into the motion executed by a. If a can extract all necessary information, it can signal the ActivityScheduler that it is able to take over the running Activity p. In this case, the ActivityScheduler will perform a scheduling of the Command provided by a. To achieve that, it calls the Command's method `scheduleAfter(CommandHandle)` to let the RoboticsRuntime perform instantaneous switching between the currently running Command (provided by p, identified by its respective CommandHandle) and the new Command (cf. Sect. 4.4).

In detail, the ActivityScheduler distinguishes three cases when a new Activity a is to be scheduled:

1) If all Actuators affected by a did not execute any Activity before, a is started.
2) Otherwise, if, for at least one Actuator affected by a, another Activity p is already running and an Activity s is already scheduled after p for the same Actuator, scheduling of a is rejected. This implies that the ActivityScheduler can only schedule one Activity at a time.
3) Otherwise, the following steps are performed:
    a) Determine the Activities P previously executed for all Actuators affected by a. Thus, P contains at most one Activity for each of these Actuators.
    b) Wait until at most one Activity p of the Activities P is still running.
    c) Let a determine all Actuators Ψ that it is able to take control of.
    d) If Ψ is a superset of all Actuators controlled by p, schedule a's Command after p's Command.
    e) Otherwise, await end of execution of p's Command, then start a's Command.

The restriction to keep only one Activity running before scheduling a new Activity (step 3b) had to be introduced due to the fact that RPI currently only allows a Command to be scheduled after exactly one predecessor. There is, however, work in progress to release this restriction.

The sequence diagram in Fig. 5 illustrates the dynamics of Activity execution and its interaction with the Command Layer. The diagram shows the most basic flow when an Activity a is started asynchronously by calling its method `beginExecute()`. During scheduling, the ActivityScheduler calls the Activity's method `prepare()`, supplying the list of Activities that were previously executed by Actuators affected by a. Based on this data, a has to decide for which Actuators it is able to take over control, and has to define an appropriate Command c that implements a's real-time behaviour. The ActivityScheduler afterwards retrieves c and starts it. The Command is then loaded in the appropriate RoboticsRuntime, which in this case transforms the Command into an RPI dataflow graph. This graph is then transmitted to the Robot Control Core and real-time execution is started. Only then the call to a's method `beginExecute()` returns. Thus, it is ensured that execution has definitely started at this time. When applications call the method `execute()` of an Activity, the workflow is basically the same. However, `execute()` calls the Command's `waitComplete()` method after scheduling.

Fig. 6 shows important timing aspects in the execution of an Activity. The depicted Activity a is assumed to be modelled by the RuntimeCommand *insertScrew* from the example shown in Fig. 3. For clarity of presentation, a is assumed to have three execution states (in reality, an Activity has more states): *New*, *Running* and *Completed*. Starting from state *New*, when

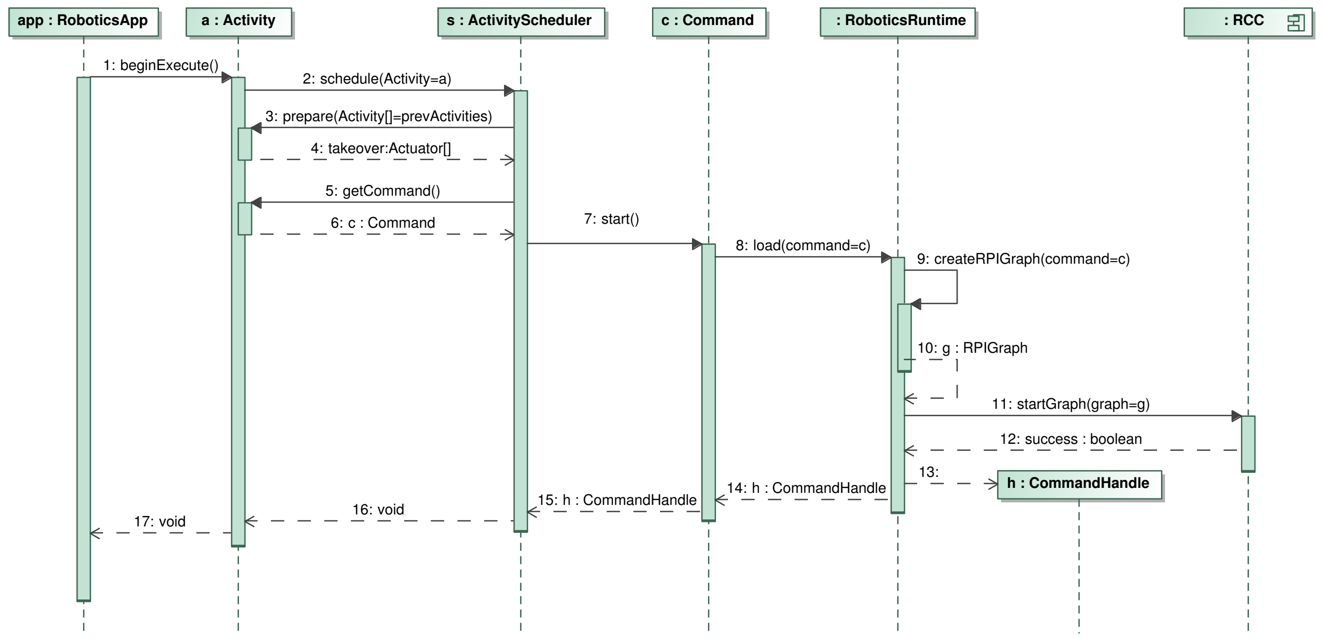

Fig. 5. Sequence diagram showing the asynchronous execution of an Activity.

a's method beginExecute() is called, a enters state *Running* as soon as its Command has entered its own *Running* state. However, there are no hard guarantees about how long this procedure will take, as the Java-based Activity scheduling and Command loading mechanism is invoked (cf. Fig. 5). This is tolerable, as starting Activities is in general not real-time critical. In contrast, all interactions between different parts of the *insertScrew* RuntimeCommand are executed with real-time constraints. Note that during runtime, all parts of each Command have been transferred to the RCC in form of a RPI real-time dataflow graph, and that all interactions take place inside this graph. In this respect, all parts of Fig. 6 below a do not show interactions between the runtime Robotics API objects themselves, but the interactions between their counterparts in the dataflow graph. The borders of all lifelines in the real-time part of the figure are marked with small tick symbols. Each tick represents one execution cycle of the RPI graph in the RCC (1ms in case of the SoftRobot RCC). Messages that do not cross cycle borders can be considered to have an effect immediately.

An important aspect is the handling of sensor guarding inside this Command. Recall that *insertScrew* should be cancelled once the LWR's force sensor detects a force greater than 5N. It is guaranteed (by construction of the dataflow graph in combination with the execution guarantees of the RCC) that after the reading of a critical sensor value, the appropriate reaction is determined within at most 2 cycles. This delay is inevitable as stated by the sampling theorem (cf. Marks [18]). The reaction in this case is triggering a CommandCanceller. Due to reasons of computability[1], the execution of CommandCancellers is always delayed by one execution cycle. Thus, cancelling of *insertScrew*, which leads to the LinearMotion action braking the robot as fast as possible, is triggered in the next cycle after the CommandCanceller has actually been started. In sum, it is guaranteed that the robot starts to brake at most 3 cycles after a critical force occurred. This same guarantee applies to all EventHandlers that can be defined as part of Commands.

The presented diagram covers many details that relate to the execution of Robotics API Commands. However, it is presented in this place as it also illustrates the transition from the soft real-time context of the Robotics API to the hard real-time RCC context. Application developers cannot rely on exact timing considering the start of an Activity. They can, however, rely on the fact that once the Activity is running and starts to control Actuators, it will be executed in exactly the same manner every time, with hard real-time guarantees in the range of milliseconds.

The particular contribution of the Activity Layer itself that was presented in this section is the introduction of a mechanism to plan execution across Activities. This is not possible using the Command Layer only, as Commands neither have knowledge about their predecessors nor about the state of Actuators during Command execution. The combination of Activity and ActivityScheduler addresses both points and introduces a pure Java-based planning mechanism for multiple

1. It has to be ensured that RPI dataflow graphs do not contain dataflow cycles that would have to be evaluated in the same execution cycle, which would lead to infinite recursion

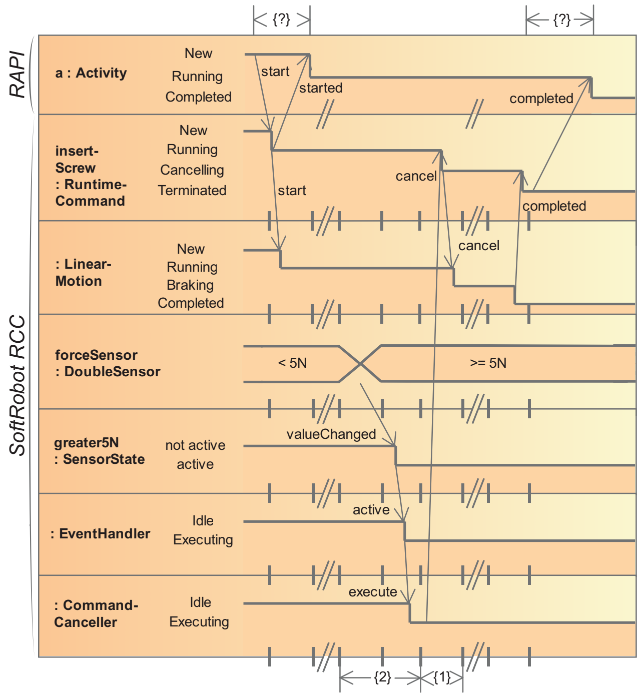

Fig. 6. UML timing diagram of Activity execution. Irrelevant parts of execution have been omitted, indicated by diagonal line pairs on lifelines. Tick symbols indicate cyclic execution of a lifeline. Important timing constraints are annotated, where '?' indicates that no timing guarantees can be given at all.

successive operations. For application developers, this results in a very easy way of activating motion blending by just executing Activities asynchronously. However, blending between subsequent motions is only one example of establishing an Activity-specific execution semantics using asynchronous execution. Other possible use cases include e.g. force-based manipulation or in general Activities that employ different kinds of control algorithms. In a scenario where Activities let Actuators apply defined forces to the environment, subsequent Activities may have to be aware of this situation and cope with it to ensure stability of the system.

Note that developers have the choice to perform single-step execution of Activities by using the synchronous method `execute()` at any point in their programs. Also note that, considering timing guarantees, this is only a best-effort approach. There is no guarantee that Activity scheduling will be fast enough to schedule the respective Commands before a preceding Command has reached a certain execution state. In the case of motion blending, the implementations of motion Activities then discard blending over the preceding Activity. Instead, the preceding motion is executed to its natural end and the following motion starts from there. With motion blending being an optimization technique in robot programs, this behaviour is acceptable if it happens only in few cases. However, application developers as well as developers of new kinds of Activities have to be aware of it.

6.2 Composing Activities

One of the strongest features of the Command Layer is the possibility to compose Commands using a flexible event-based mechanism and to execute such composed Commands on the Robot Control Core with real-time guarantees. Implementations of different Activities employ this mechanism to provide more complex functionality out-of-the-box to programmers. For instance, implementations of Light Weight Robot Actuators support performing a certain motion until a contact force is measured and then switch to a mode where the force is maintained. The respective ActuatorInterfaces for Light Weight Robots in the Robotics API realize this as a sequential composition of single Activities (motion and maintaining force, where the first one is additionally sensor-guarded). Furthermore, all implementations of motion Activities contain different execution paths to handle the cases that a motion blends over a preceding motion or that the motion is executed from its original start point. This is realized by parallel composition of several Activities that are executed in a mutually exclusive manner.

The Activity Layer also allows the composition of any Activities to more complex Activity structures with a defined execution semantics. In contrast to the Command composition mechanism, the flexibility is reduced a bit. Instead, templates for recurring composition patterns are provided. This again is in many ways inspired by features provided by today's specialized robot programming languages. Other patterns were identified during development of application prototypes in the SoftRobot project. Currently the following composition patterns of Activities are provided:

- *Sequential composition*. The specified Activities are executed one after another (optionally with blending).
- *Parallel composition*. The specified Activities are started at the same time and run in parallel.
- *Conditional execution*. Based on a given condition, either one or another Activity is started.
- *Composition of main task and subtasks*. One or more Activities are started at defined conditions during the execution of a main Activity.

Sequential composition proved to be useful e.g. for robot grippers that need a sequence of field bus outputs to be set in order to control operation. Parallel composition was employed for instance to realize synchronized motion of multiple robot arms. Conditional execution is useful e.g. for deciding which operation to execute next based on measurements of sensors. By composing sub tasks with a main task, tool actions may be executed at defined points of a robot motion.

Note that composing Activities creates new Activities, which may again be composed to arbitrarily complex structures. Each type of composed Activity creates a Robotics API

Command that combines all Commands created by the inner Activities, plus appropriate EventHandlers so that the behaviour of the resulting Command complies to the semantics of the composed Activity. When composing Activities, it is not possible to create a loop by e.g. re-starting an Activity that has already been executed. This limitation is caused by the design of the Command model (cf. Sect. 4.3), which Activities use to implement their functionality. However, the composition mechanisms for Activities combine the meta-data of inner Activities as well, such that scheduling of a composed Activity can be performed appropriately. Thus, also infinite sequences of complex Activities can be scheduled (on a best-effort basis, as mentioned above) which will result in continuous operation if feasible. For instance, blending across a (infinite) series of motion Activities of multiple synchronized robot arms is possible.

6.3 Binding Functionality to Actuators

For application developers, an important point is the creation of concrete Activities for a specific Actuator. In the Robotics API Command Layer, Actuators and Actions are not coupled at all. The reasons for this design are explained in Sect. 4.1. This missing coupling was identified to be a difficulty considering usability by developers. They cannot identify on an API level which Actions are usable for which Actuators. Even worse, the validity of an Action for use with a certain Actuator can only be checked at runtime in a late phase of Command processing and can thus lead to complicated runtime exceptions that must be interpreted by developers.

To mitigate this issue, the concept *ActuatorInterface* was introduced. ActuatorInterfaces combine a set of related functionalities of Actuators and expose them to application developers. ActuatorInterface is intended to be subclassed to form more concrete ActuatorInterfaces. For example, all Robots in the Activity Layer provide amongst others a LinInterface and a PtpInterface. The former provides methods to execute different kinds of linear Cartesian motions (e.g. to absolute or relative goals), the latter is similar but covers point-to-point motions in joint space. By that mechanism, functionality is semantically grouped. Additionally, developers can query Actuators directly about the ActuatorInterfaces they support. Thus, they get direct and clear feedback whether a given Actuator is able to perform certain kinds of operations.

The best imaginable solution for realizing the ActuatorInterface concept would be to provide developers with statically typed extension of the respective Actuators. However, due to the characteristics of Java (single inheritance, no mixin mechanism, no extension methods like in C# etc.), extensible functional composition could only be realized using an aggregation mechanism. Thus the definition of the class Actuator was revised such that it became an aggregation of ActuatorInterfaces. This actually required a modification on the Robotics API Core/Command Layer. However, as the basic ActuatorInterface is very slim and generic, this extension did not introduce further coupling, in particular not to the Activity Layer itself. The Activity Layer instead relies on the ActuatorInterface concept and uses it as a basis for grouping Activities and relating them to Actuators. Concrete ActuatorInterface implementations like the aforementioned LinInterface and PtpInterface expose methods that create and return Activity implementations.

In the following, we will demonstrate the usage of this interface with concrete examples to illustrate the programming interface to application developers. Executing a linear motion with a robot requires the statement given in Listing 1. It first selects the appropriate ActuatorInterface of the Actuator `robot`, in this case the interface of type `MotionInterface`. After that, the method `lin(goal)` of this interface is called. It creates an Activity that performs a linear motion of the robot. Finally, calling `execute()` begins execution of this motion and blocks until its end.

```
robot.use(MotionInterface.class)
    .lin(goal).execute();
```

Listing 1. Executing a linear motion, the Activity way

Although the syntax is not as compact as envisioned at the beginning of Sect. 6, it has a straightforward structure. The sacrifices considering compactness in turn enable extensibility, as arbitrary new kinds of ActuatorInterfaces may be loaded and used at runtime. Furthermore, asynchronous Activity execution can be employed to take advantage of cross-Activity functionality like motion blending. This is illustrated by Listing 2.

```
1 MotionInterface motion =
    robot.use(MotionInterface.class);
2 motion.lin(goal, new
    BlendingCondition(0.7)).beginExecute();
3 motion.lin(goal2).execute();
4 motion.lin(goal3).execute();
```

Listing 2. Motion blending with Activities

As prerequisite for blending motions, the point at which blending from one motion to another should be started has to be specified. This has to be done by supplying a parameter to the motion activity upon its construction (line 2, the value 0.7 indicates 70 percent of the motion's total time). Moreover, blendable motions have to be executed asynchronously (line 2). Motions executed synchronously will be executed completely to their natural end (lines 3 and 4).

As scheduling of Activities is performed on a per-Actuator basis, Activities affecting different Actuators can be executed in parallel. Developers can take advantage of this, but also have to be aware of this semantics to prevent unwanted effects. For example, in Listing 3, the Activity closing the gripper attached to the moving robot (line 3) will be executed in parallel to the first motion (line 2). In particular, it will not be executed after the first motion has ended or when it has reached 70 percent of its execution progress - these dependencies will only be respected by Activities affecting the same Actuator (line 4), in this case the robot.

```
1 MotionInterface motion =
    robot.use(MotionInterface.class);
2 motion.lin(goal, new
    BlendingCondition(0.7)).beginExecute();
3 gripper.use(GrippingInterface.class)
    .close().beginExecute();
4 motion.lin(goal2).execute();
```

Listing 3. Asynchronous parallel execution of Activities

7 Evaluation

The SoftRobot architecture presented in the previous sections provides robotic application developers with a convenient programming interface for industrial robots in a modern general-purpose language. A reference implementation in Java and C# is available soon[2]. Developers can profit from different state-of-the-art programming languages available on the JVM and .NET platform (e.g. JRuby, Jython, Scala, Clojure), as well as a bunch of IDEs, tools and libraries for all kinds of purposes. Implementations of the JVM are available for a large variety of operating systems (e.g. Linux, Windows, QNX, VxWorks), thus ensuring portability of applications. The main contributions of the SoftRobot architecture are a convenient *object-oriented programming model* and its integrated *abstraction for real-time critical operations* that together meet the requirements today's and future industrial robot tasks. This section will demonstrate the usefulness of the approach by means of practical application examples, evaluate the extensibility, reusability and performance of the architecture and draw a comparison to other frameworks.

7.1 Application examples

In the SoftRobot project, various robotic applications have been developed. The motivation was always twofold: On the one hand, these applications served as test-beds for evaluating the state of the SoftRobot software architecture. The goal was finding out if the developed concepts are stable considering requirements of various applications, if there are concepts missing for realizing certain functionality and if the reference implementation is stable and free of errors. On the other hand, applications were chosen to illustrate the strong points of the SoftRobot architecture and the Robotics API in particular. Some applications could be developed very quickly compared to similar solutions based on current robot controller architectures, while others expose features that are barely realizable using existing controllers. This section presents three different application prototypes that have been developed in SoftRobot.

2. The reference implementation will soon be released and will be available via http://robotics.isse.de

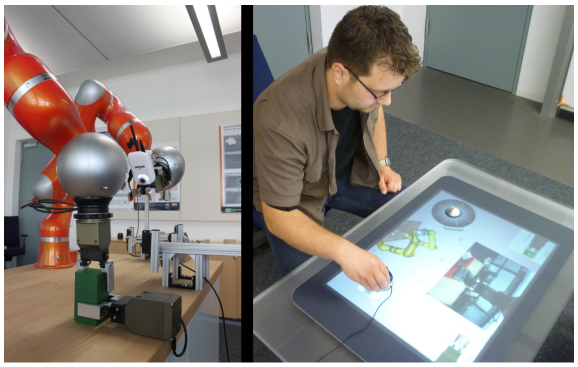

Fig. 7. Robot setup (left) and operator interface (right)

7.1.1 Tangible Teleoperation

The Tangible Teleoperation application (c.f. [19]) was designed as a case study for multi-robot applications with the Robotics API. At the same time, it should demonstrate the possibilities of employing novel user interface paradigms. In this context, a strong point of the SoftRobot architecture is the interoperability between many modern programming languages. This is a clear advantage over today's proprietary robot programming languages, where the interaction with high-level applications is often cumbersome. The Tangible Teleoperation system (cf. Fig. 7) is based on a tangible user interface device, called Microsoft PixelSense[3] (formerly known as Microsoft Surface). It is designed to tele-operate a two-arm robot system in an intuitive way by employing tangible user interface elements. Operators are able to manually control the robots and perceive the state of the system during its interaction with the environment, though they do not have eye contact to the robots. For this purpose, various feedback is given by the Tangible Teleoperation application: A video stream of a camera mounted on one of the robots, a 3D model of the current poses of the robot arms which is updated continuously, and a visualization of the end-effector forces measured by the KUKA Lightweight Robots.

Different operation modes are supported by the Tangible Teleoperation application: The robot with the camera mounted on it may be controlled directly and can be used to manipulate the environment. For a better overview of the scene, a second control mode, called observer mode, was introduced. Here the robot with attached camera takes the role of a passive observer, focusing the effector of a second robot arm. The observer perspective can be adjusted by rotating the camera robot around the observed scene and moving it closer to or farther away from the point of interest. The second robot is actually controlled by the human operator, while the observing robot automatically follows every movement of this robot and thus keeps the perspective centred on the second arm while it manipulates the environment.

3. http://www.microsoft.com/en-us/pixelsense/default.aspx

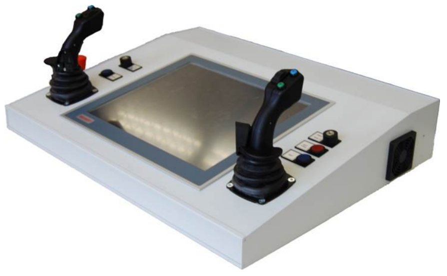

Fig. 8. Control panel of the Remote Manipulator Control

The application and UI logic was implemented using C# and the Microsoft .NET framework. The multi-touch tangible user interface was implemented based on the Microsoft Surface SDK, while the Robotics API was employed for controlling the robots. To integrate the Java-based Robotics API into a .NET application, it was automatically converted to a .NET library using the IKVM.NET compiler[4]. The Robotics API robot implementations provide special ActuatorInterfaces for manually moving the Actuators on a velocity level. Thus, the effort required for controlling the robots from this high-level application was minimal - it comes down to using just another library from developers' point of view. The complete application was realized by a student in about two months of work.

7.1.2 Remote Manipulator Control

The Remote Manipulator Control (RMC) system is a product of MRK-Systeme GmbH. It features a industry-grade control panel with a touch screen and two joysticks for tele-operating KUKA robots (cf. Fig. 8). The system is employed by MRK's customers to perform material handling with robots in hazardous environments. A computer system integrated into the control panel provides a feature-rich user interface on the touch screen. Users can move the connected robot manually with the joysticks on the panel. Additionally, they can define movement paths for the robot and let it move along those paths automatically. The touch screen employs a 3D visualization of the robot pose. For controlling robot movement, the standard KUKA Robot Controller is employed.

In the SoftRobot project, the RMC system was chosen for evaluating the migration of an existing commercial robot application to the SoftRobot architecture. The system is particularly interesting for such an evaluation, as during its development, much effort has been put into the interface to the KUKA Robot Control. MRK employs the technology KUKA RobotSensorInterface, which allows for cyclically sending goal positions to the robot. The communication has to respect defined timing delays of some milliseconds. To achieve a stable connection with sufficient performance, MRK invested about two months of work. Another two months were then spent for extending the communication protocol to allow transferring series of motion instructions to the robot controller. This was necessary for the definition of complete movement paths in the RMC application (MRK calls this feature 'Autodrive'). To achieve this, user defined fields in the packages sent via the RobotSensorInterface were used to encode different motion instructions and their parameters. These user defined fields were then evaluated by some program on the robot control, which triggered other programs appropriately.

4. http://www.ikvm.net/

MRK performed a migration of the RMC system to the SoftRobot architecture in about two weeks of time. It turned out that all functionality required by the application was already provided by the Robotics API. The parts of the application logic that triggered robot commands by connecting to the Robot Control via RobotSensorInterface just had to be substituted by calling appropriate methods of the Robotics API. In an evaluation following the migration, MRK emphasized the reduction of complexity in the Remote Manipulator Control application. Though a realistic assessment of the save in effort when re-developing the application on top of the SoftRobot architecture is hard to do, MRK estimates that the realization of the Autodrive feature alone would only have been about 10% of the effort (two months, see above) that was originally required. Furthermore, no loss in application performance compared to the original system could be observed. Finally, MRK identified some possible approaches to increase safety of the application by employing e.g. sensor based motion guards, using features of the Robotics API.

7.1.3 Factory 2020

To summarize the research results of the SoftRobot project, a prototype of a robotic manufacturing application called Factory 2020 was developed. The application was designed along the vision of a fully automated factory of the future, in which different kinds of robots cooperate to perform complex tasks. In the Factory 2020 demonstrator, two robot arms and a mobile robot platform work together in part assembling. To achieve that, force-based motion, real-time motion synchronization and different coordination patterns are applied.

Fig. 9 gives an overview about the tasks performed in Factory 2020. Two different containers with parts to be assembled are delivered by an autonomously navigating robot platform. Before assembling, both containers have to be transported onto the workbench. The robot arms are locating the exact position of the containers on the platform by touching certain prominent points. The position may vary due to inaccurate navigation of the platform and some inaccuracy of the containers on the platform. The touching operation employs motions guarded by force sensor measurements to make the robots stop upon contact. After the locating process, both arms grip the containers and cooperatively transport them onto the workbench. These motions have to be real-time synchronized to ensure proper transport without damaging the containers or robots.

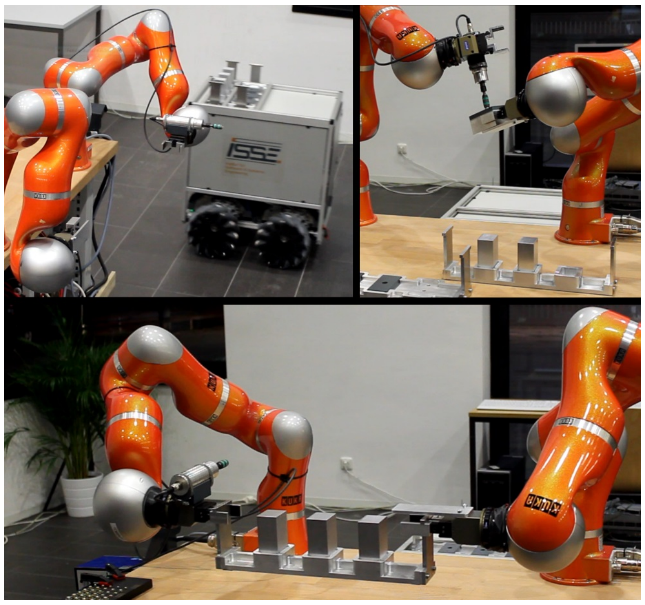

Fig. 9. In Factory 2020, an autonomous robot platform delivers parts (top left), robot arms cooperate in transporting (bottom) and assembling them (top right).

Once the containers have been placed on the table, each robot arm picks a part from one of the containers. The parts are then assembled. For this operation, the robots apply a defined force on the parts to compensate slight variations in the fitting between top and bottom part. Finally, both parts of the final workpiece have to be bolted together. For this purpose, an electrical screwdriver is attached to one of the robots as its second tool. This robot first fetches a screw from a screw magazine, then transports it to the workpiece, inserts it and tightens the screw using the screwdriver. These operations are also performed using force-based motions, which again allows for compensating variations in part quality and the process itself (e.g. slight deviations when gripping a workpiece part). The final workpiece is then put back into one of the containers. After all workpieces have been assembled, the containers are put back onto the platform, which delivers the workpieces to their destination.

The Factory 2020 application shows a number of strong points of the SoftRobot architecture. First, the real-time synchronization of both robot arms required for cooperative transport is possible by 1) planning motion for both arms exactly and deterministically the same way, using a common center of motion, and 2) starting the planned motions at exactly the same time. Both features are provided natively by the SoftRobot architecture. A simple Actuator has been introduced that aggregates two robot arms and provides an ActuatorInterface for performing synchronized linear motions with both arms. The implementation of the respective Activity first determines appropriate motion center points for both robot arms, then calculates separate linear motions for each arm and finally combines those to one Activity using parallel composition (see Sect. 6.2). This took only about 300 lines of Java code and even enabled motion blending across several two-arm motions by combining meta-data of the linear motion Activities of the single arms.

Activity composition was also used to realize force-guarded motions like e.g. for detecting the location of the workpiece container or inserting screws into workpieces. For structuring the complete automation system software, it was divided into several components by means of the Java-based module system OSGi [20]. This module structure eased deployment of components to different physical systems (robot arms, mobile base) and the coordination of those systems. For implementing the behaviour of components which coordinate the interaction with and among other components, state machines were employed. The standardized State Chart XML [21] format was used to formally model those state machines. Based on the Apache Commons SCXML implementation, a graphical editor and a runtime environment for SCXML state machines was created.

7.2 Complexity and practical reusability

Reusability was identified as one of the core requirements to the complete SoftRobot architecture. In this context, integrating new devices and operations into the architecture should be possible with small effort by reusing generic concepts of the Robotics API or existing extensions. In particular, the required extensions to the real-time Robot Control Core should be kept small. It is inevitable to integrate a real-time capable device driver in the RCC, as this layer has the responsibility for communicating with the devices and controlling them. However, the design of the Robotics API, the generic mapping procedure to RPI and a hardware abstraction layer inside the RCC reference implementation allow for integrating new Actuators with minor effort.

To get a rough idea about the complexity of the architecture and the size of reusable parts, Fig. 10 compares the Lines Of Code (LOC, without comments) of the SoftRobot core component and different extensions, including their "vertical" distribution across the architectural layers. The figure shows the following components along the horizontal axis:

- The bare SoftRobot core component, including the Robotics API Core, the generic parts of the algorithm for transforming those Core concepts to RPI-capable runtimes, and the SoftRobot RCC including a real-time capable RPI interpreter. This component does not even contain support for any kind of robot.
- The Robotarm extension, introducing the definition of a robot arm consisting of joints into the architecture.
- The LWR, Staubli TX90 and UR5 extensions, introducing vertical support for three different kinds of robots, all based on the Robotarm extension.

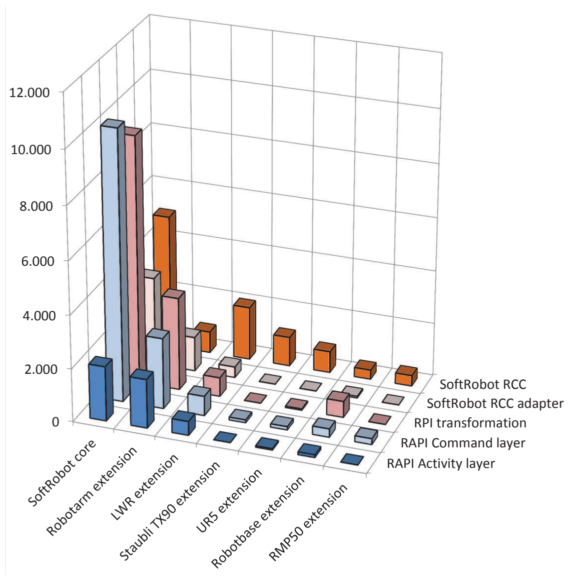

Fig. 10. Distribution of Lines Of Code in the SoftRobot architecture reference implementation.

- The Robotbase extension, introducing the definition of a mobile robot base into the architecture.
- The RMP50 extension, introducing support for the Segway RMP50 device, based on the Robotbase extension.

The depth axis of the figure shows the following vertical layers of the architecture:

- The Activity layer of the Robotics API,
- the Command layer of the Robotics API,
- the RPI transformation algorithm in the Robotics API,
- the adapter to the SoftRobot RCC in the Robotics API
- and the SoftRobot RCC itself.

When comparing the LOC (vertical axis) of the four Java-based layers and the RCC layer, we find that the biggest part of the generic core component of the architecture is implemented inside the Robotics API. When introducing the generic Robotarm extension, relatively little code was required in the RCC layer. However, implementing RCC drivers for concrete robots (LWR, Staubli, UR) takes some effort, which can primarily be attributed to the complex low-level interface offered by the robots themselves (e.g. FRI for the LWR based on an UDP protocol, uniVAL for the Staubli based on CANopen and Ethercat). The picture is similar for the generic Robotbase extension and the concrete RMP50 extension component. However, the support of robot bases is rather preliminary and we expect the numbers for the Command and Activity layer to increase, while the RPI transformation should be quite complete already.

A second result is that there is a considerable amount of code required for the transformation of Robotics API concepts to RPI. The largest part of this code is contained in the SoftRobot Core package and is thus responsible for the generic transformation of all parts of the Command model. This code is reused by all extensions to the Robotics API that rely on its Command model. However, the Robotarm extension contains a considerable amount of RPI transformation code as well. A more detailed investigation of this part of the Robotarm extension showed that the largest part of this code is responsible for implementing the transformation for motion Actions. This also includes the path planning for different kinds of motions (e.g., point-to-point motion, linear motion and spline motion). This can be considered a weakness in the current structure of the Robotarm extension: The path planning code should be separated from the transformation rules that map Actions to RPI dataflow graphs. We plan to improve this in further versions, thus shifting a considerable amount of code from the RPI transformation layer to the RAPI Command layer part of the Robotarm extension.

However, the current realization of the Robotarm extension does not cause problems with reusability when implementing packages for concrete robots. The Staubli extension only requires minimal Java code (ca. 100 lines) in the Robotics API Command layer. This code defines the link/joint structure of the Staubli TX90 robot and its default parameters (e.g., for velocities and accelerations). All other functionality is reused from the Robotarm extension. In particular, no lines of code are required for RPI transformation and the adapter to the RCC. Almost the same applies to the UR extension. Less than 150 lines of code had to be added to support the robot's "Freedrive" mode, where it can be moved by hand. As this mode is activatable by executing a special Activity, the UR extension contributes a bit of code to the Activity layer. The KUKA Lightweight Robot with its integrated torque sensors, different joint controller modes and the possibility for force-guarded and force-guided operations required additional code on all Java layers (around 2000 LOC in total).

Though the LOC measure is not a precise way to measure reusability, the clear results should indicate a high degree of reusability of the generic part of the SoftRobot architecture. In particular, the amount of code required for integrating new types of robots is pleasingly low. Keep in mind that each of the robots inherits all functionality for executing different kinds of motions (including motion blending), real-time critical synchronization with other devices and reacting to sensor events. This also means that in all of the presented examples[5], one type of robot could have been replaced by another type by literally just replacing the physical hardware.

7.3 Performance

A good performance in terms of scheduling new Activities is important to achieve high cycle times in industrial robot applications, in particular to ensure that motion blending is executed as often as possible. The performance of the

5. Excluding the force-based manipulation operations in the Factory 2020 application, admittedly.

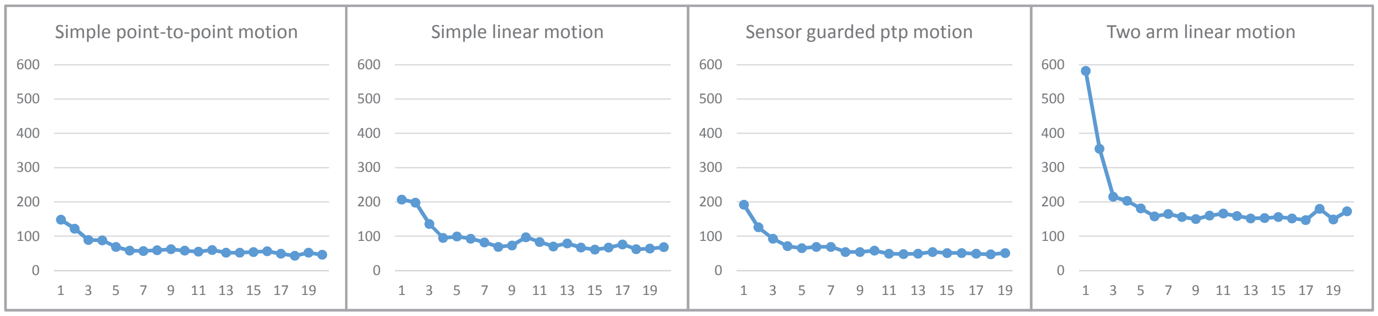

Fig. 11. Time needed to start different types of Activities. Each type of Activity was executed multiple times in a loop (horizontal axis), and the duration until it was started on the RCC was measured (vertical axis, in milliseconds).

SoftRobot reference implementation was examined using a series of benchmarks with KUKA LWR arms:

- a simple point-to-point motion of a single arm to a pre-defined goal in joint space;
- a simple linear motion of a single arm to a pre-defined goal in Cartesian space;
- a sensor-guarded ptp motion of a single arm, which was cancelled when the robot moved too close to a fixed obstacle, causing the robot to brake immediately;
- a series of linear motions of two synchronized arms to pre-defined Cartesian goals, using the two-arm Actuator implementation of Factory 2020 (see Sect. 7.1.3).

All the above-mentioned Activities were executed several times in a loop, and the time to start each motion Activity was measured. The robot arm was moved back to its original position at the end of each loop iteration in the first three tests, which was excluded from the measurements. In the fourth test, the series of motions ended at at the initial position, thus repositioning was not necessary.

In the sensor-guarded point-to-point motion, the robot's encoders and the forward kinematics function were used to measure the arm's current Cartesian position. Based on this, the distance between the robot's flange and the known location of the obstacle was calculated in real-time. Once this distance became too small, the motion was cancelled. While this is a rather artificial example, it can demonstrate the impact of sensor guards on Activity loading performance.

All tests were executed on a fast desktop PC (Intel Core i5-3470, 8GB RAM). This PC ran Ubuntu 12.04 with the Xenomai real-time framework[6] installed. The SoftRobot RCC was hosted as a real-time process on this PC and communicated with the robots via a dedicated network adapter. A Robotics API benchmark application was executed on the same PC, but with lower priority.

Fig. 11 shows the benchmark results. In all tests, the performance increased significantly after the third to fifth motion had been executed. The strong hypothesis is that this is caused by the Just-In-Time (JIT) compiler in the Java Virtual Machine. A short test without JIT compiler showed constant timings on a high level and confirmed the hypothesis. This behaviour also occurred consistently when a mixed series of point-to-point and linear motions was executed and those motions were parametrized with various goals.

6. http://www.xenomai.org/

Overall, basic motions take roughly 50–200 ms to be started. Using sensor guards does not have a notable impact on performance. This might change, however, when a large number of sensor guards is used, or when sensor data requires complex processing in order to evaluate the guarding condition. The two arm Actuator implementation used to execute synchronous linear motions takes roughly double the time to schedule a motion compared to the single arm case. A detailed evaluation indicated that only about 20–30 % of the scheduling time was consumed by planning the operation (which equals motion planning in the tests). The rest of the time was needed to transform the Command to an RPI dataflow graph specification, serialize this graph, transmit it to the RCC via a network protocol and build the appropriate RPI graph instance on the RCC. More in-depth analysis is necessary to optimize this process.

The observed performance can be considered sufficient for most practical cases. In all the tests, the time the robot actually moved was at least double the scheduling time. However, when the execution time of a robot operation gets small, e.g. in case of small motions or motion blending in early motion phases, the scheduling time may limit application performance. In particular, the current implementation of the two arm Actuator may be limiting performance in this respect. It can obviously be optimized, as it currently plans and interpolates the same motion for each robot arm separately. While this results in a simple and small implementation on the Robotics API level, it causes a performance overhead in all phases of scheduling of the resulting motions.

7.4 Comparison to other approaches

The SoftRobot approach is geared towards a more efficient software development for industrial robot systems. Large emphasis was put on the goal to introduce a widely used standard programming language to industrial robot systems, replacing proprietary, vendor-specific robot languages. This was driven by the strong belief of all project partners that programming robot applications is not fundamentally different from programming other types of applications. This section will evaluate our approach in comparison to existing robot programming approaches from academia and industry.

7.4.1 Proprietary robot programming languages

For comparing the SoftRobot architecture and its programming interface to proprietary robot languages, we take the KUKA Robot Language (KRL) as an example[7]. When just comparing the interface towards robot programmers, KRL has the advantage of a more minimalistic, imperative programming model. The Robotics API introduces a much more flexible, but also more complicated object-oriented programming model. However, when applications get more complex, developers profit from a full-fledged modern programming language supporting a large number of design patterns for structuring code. There exist powerful IDEs as well as a large number of other tools and libraries. In particular, tools for creating DSLs allow for introducing new domain-specific programming interfaces rapidly. As a proof of concept, and to evaluate backward compatibility to existing robot programs, one of our students created a KRL interpreter on top of the Robotics API in his master thesis [22].

Considering functionality, the SoftRobot architecture provides native support for tight integration of sensor feedback into robot operations. Furthermore, forming robots to teams that collaborate in real-time is easily possible. KRL provides support for connecting multiple robot controllers to achieve real-time robot cooperation as well. This implies writing one program for each robot and synchronizing with special statements. We compared the effort required for realizing a scenario similar to the cooperative container transport in the Factory 2020 application (see Sect. 7.1.3). An experienced KRL programmer invested more than 3 times the effort to let two robots perform cooperative transport of a single workpiece compared to an experienced Robotics API developer.

7.4.2 Existing object-oriented approached

There are several academic approaches providing robot-specific libraries for general-purpose languages. Early examples are RCCL [23] in C and PasRo [24] in Pascal. With the emergence of object-oriented languages, robot-specific libraries for these languages have been implemented, too. Examples are ZERO++ [25], MRROC++ [26]. RIPE [27], the Robotic Platform [28] and SIMOO-RT [29]. These frameworks employ object orientation for introducing hardware abstraction. However, in general a low level of abstraction is used so that developers need profound knowledge in robotics and often in real-time programming. Using C or C++ may result in a better runtime performance compared to Java, which is executed on a virtual machine rather than directly on the physical hardware. However, we are confident that the workflow-related parts of robot applications are executed fast enough also in Java, and our practical experiences confirm this. (Real-)Time critical operations are delegated to the RCC, as explained. This abstraction for real-time tasks is one important paradigm of the SoftRobot architecture that cannot be found in most other object-oriented robot programming frameworks. One exception is the ORCCAD architecture introduced by Borrelly et al. [30]. In fact, this project takes a similar approach in that a real-time abstraction is provided for complex robot tasks. However, the workflow parts of robotic applications that are not real-time critical have to be implemented in a proprietary language (MAESTRO) designed by the authors. Using a standard language like in the SoftRobot approach has many advantages for application development (see above) and, in the case of a language with automatic memory management, relieves developers from memory management issues they have to care about in C/++.

7.4.3 Component-based approaches in robotics

For managing the complexity of modern autonomous robots and large-scale distributed application scenarios, *decomposition of systems into components*, *composability of subsystems* and *distribution of components* have been identified as core requirements (stated e.g. by Hägele et al. [5] and addressed e.g. by Nesnas et al. [31], Finkemeyer et al. [32] and in particular most component-based systems like Player [33], ROS [34] or ORCA [35]). This led to a trend for Component-Based Software Engineering in the past years (cf. Brugali et al.'s work [36], [37]). Many of the aforementioned component frameworks do not particularly support hard real-time execution of robot tasks, which makes it harder to achieve the precision required in industrial applications. Implementing the workflow of an application in a component-based system usually requires multiple configuration and coordination components (c.f. Prassler et al. [38]). Coordination components are commonly implemented using state machine mechanisms (e.g. rFSM [39] or ROS smach [40]). While this is certainly well suited for a certain class of systems, we believe that for implementing complex, partly real-time critical workflows in industrial application scenarios, an object-oriented API like provided in the SoftRobot approach is better suited. For composing larger automation systems, we are however following the component-based approach as well. As previously mentioned (Sect. 7.1.3), the Factory 2020 application is built of OSGi components. In this case, using components to structure the systems and realizing the workflow of complex process steps (e.g. cooperative transport, workpiece assembling) based on the Robotics API proved to be a good combination. We are continuing research about the distribution of automation processes to component architectures.

7. Other vendor-specific languages may slightly differ, but the main evaluation points should yield the same results.

8 Conclusion

This article presented the SoftRobot architecture, a layered, vendor-independent software architecture for industrial robotics application development. A special focus was put on presenting the design of the object-oriented Robotics API, which provides a powerful and flexible programming interface. It is designed to realize complex robotic applications including multi-robot cooperation and integration of sensors in guiding and guarding robot operations. The particular contributions of the SoftRobot approach are:

1) An abstraction for real-time critical robot operations, which can be specified using dataflow graphs defined in the RPI language.
2) An object-oriented framework, designed according to the requirements of the industrial robotics domain, and integrating a model for robot operations which employs the aforementioned real-time abstraction.

We have evaluated the practical usefulness of the SoftRobot approach by creating a reference implementation of a Robot Control Core and the Robotics API, and using this implementation to realize complex applications. Some examples have been presented in this work, including real-time synchronized, blended motion of two robot arms. The overall performance of the reference architecture can be considered adequate, but leaves room for improvements.

An increasing number of undergraduate students are working with the reference implementation. We also started to arrange hackathons, e.g. with the KUKA youBot. For this purpose, we provide a youBot extension for the SoftRobot architecture. The results of these hackathons as well as the feedback from the students that attended them are very promising: The majority of participants were computer science students that have little or no experience with any kind of robots. Though, they managed to create complex tech-demos with the youBots, e.g. writing user-defined words on a whiteboard using a combination of robot arm and base movement.

An evaluation of the distribution of code throughout the framework structure showed clear indication that the design of the architecture supports extensibility and reusability. The integration of new robotics devices as well as new functionality for those devices (or already existing devices) requires minor effort and can re-use many concepts that are already implemented. We are working towards integrating a broader variety of devices. Robot arms by different manufacturers (KUKA, Staubli, Universal Robots) are supported already, which illustrates the vendor-independent approach. Currently, mainly serial robot structures have been considered. Integration of parallel or hybrid structures (like in many humanoid robots) will be considered in the future, which might raise the need for other kinds of synchronization paradigms for Actuators in the architecture. Furthermore, we will continue to investigate the integration of this architecture with component-based systems like OSGi to find the right level of use for each paradigm. Finally, we are developing an ecosystem of tools around the SoftRobot architecture to better support users, for instance to allow teaching robot motions with the help of tablet devices.

Acknowledgments

The authors would like to thank the project partners from KUKA Laboratories GmbH and MRK-Systeme GmbH, Jun.-Prof. Frank Ortmeier from Magdeburg University as well as VDI/VDE-IT for their support and cooperation in setting up and successfully conducting the SoftRobot project.

References

[1] Willow Garage, Inc. PR2 - Overview. [Online]. Available: http://www.willowgarage.com/pages/pr2/overview 1
[2] Cotesys Central Robotics Laboratory II. TUM-Rosie. TU Munich. [Online]. Available: http://ias.cs.tum.edu/robots/tum-rosie 1
[3] J. N. Pires, "Robotics for small and medium enterprises: Control and programming challenges," *Industrial Robot*, vol. 33, no. 6, 2006. 1
[4] M. Hägele, T. Skordas, S. Sagert, R. Bischoff, T. Brogårdh, and M. Dresselhaus, "Industrial robot automation," European Robotics Network, White Paper, Jul. 2005. 1, 2, 2
[5] M. Hägele, K. Nilsson, and J. N. Pires, "Industrial robotics," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Berlin, Heidelberg: Springer, 2008, ch. 42, pp. 963–986. 2, 1, 7.4.3
[6] 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
[7] 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. 2
[8] Blackbird Robotersysteme GmbH. Blackbird Robotersysteme GmbH. [Online]. Available: http://www.blackbird-robotics.de 2
[9] Hub Technologies. KUKA Robotics. [Online]. Available: http://www.kuka-robotics.com/germany/en/products/software/hub_technologies 2
[10] C. Bredin, "ABB MultiMove functionality heralds a new era in robot applications," *ABB Review*, vol. 1, pp. 26–29, 2005. 2
[11] R. T. Vaughan and B. Gerkey, "Really reusable robot code and the player/stage project," in *Software Engineering for Experimental Robotics*, ser. Springer Tracts in Adv. Robotics, D. Brugali, Ed. Springer, April 2007, vol. 30. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68951-5_1610.1007/978-3-540-68951-5_16 3
[12] 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. 3, 3.1
[13] 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, MO, USA*, 2009. 3
[14] M. Vistein, A. Angerer, A. Hoffmann, A. Schierl, and W. Reif, "Interfacing industrial robots using realtime primitives," in *Proc. 2010 IEEE Intl. Conf. on Automation and Logistics (ICAL 2010), Hong Kong, China*. IEEE, Aug. 2010, pp. 468–473. 3
[15] ——, "Instantaneous switching between real-time commands for continuous execution of complex robotic tasks," in *Proc. 2012 Intl. Conf. on Mechatronics and Automation, Chengdu, China*, Aug. 2012, pp. 1329 –1334. 3.1, 4.4, 6
[16] A. Schierl, A. Angerer, A. Hoffmann, M. Vistein, and W. Reif, "From robot commands to real-time robot control - transforming high-level robot commands into real-time dataflow graphs," in *Proc. 2012 Intl. Conf. on Informatics in Control, Automation and Robotics, Rome, Italy*, 2012. 3.2, 4.4
[17] E. W. Dijkstra, "EWD 447: On the role of scientific thought," *Selected Writings on Computing: A Personal Perspective*, pp. 60–66, 1982. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF 5

[18] R. J. Marks II, *Introduction to Shannon Sampling and Interpolation Theory*. Springer, 1991. 6.1

[19] A. Angerer, A. Bareth, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif, "Two-arm robot teleoperation using a multi-touch tangible user interface," in *Proc. 2012 Intl. Conf. on Informatics in Control, Automation and Robotics, Rome, Italy*, 2012. 7.1.1

[20] *OSGi Core Release 5*, OSGi Alliance Spec., Mar. 2012. [Online]. Available: http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf 7.1.3

[21] *State Chart XML (SCXML): State Machine Notation for Control Abstraction*, W3C Working Draft 6, Dec. 2012. [Online]. Available: http://www.w3.org/TR/scxml/ 7.1.3

[22] H. Mühe, A. Angerer, A. Hoffmann, and W. Reif, "On reverse-engineering the KUKA Robot Language," *Workshop on Domain-Specific Languages and models for ROBotic systems, 2010 IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems, Taipeh, Taiwan*, 2010. 7.4.1

[23] V. Hayward and R. P. Paul, "Robot manipulator control under unix RCCL: A robot control C library," *International Journal of Robotics Research*, vol. 5, no. 4, pp. 94–111, 1986. 7.4.2

[24] C. Blume and W. Jakob, *Programming Languages for Industrial Robots*. Springer-Verlag, 1986. 7.4.2

[25] C. Pelich and F. M. Wahl, "ZERO++: An OOP environment for multiprocessor robot control," *International Journal of Robotics and Automation*, vol. 12, no. 2, pp. 49–57, 1997. 7.4.2

[26] C. Zieliński, "Object-oriented robot programming," *Robotica*, vol. 15, no. 1, pp. 41–48, 1997. 7.4.2

[27] D. J. Miller and R. C. Lennox, "An object-oriented environment for robot system architectures," in *Proc. 1990 IEEE Intl. Conf. on Robotics and Automation*, Cincinnati, Ohio, USA, May 1990, pp. 352–361. 7.4.2

[28] 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. 7.4.2

[29] L. B. Becker and C. E. Pereira, "SIMOO-RT – An object oriented framework for the development of real-time industrial automation systems," *IEEE Trans. Robot. Autom.*, vol. 18, no. 4, pp. 421–430, Aug. 2002. 7.4.2

[30] J.-J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro, "The ORCCAD architecture," *Intl. Journal of Robotics Research*, vol. 17, no. 4, pp. 338–359, Apr. 1998. 7.4.2

[31] I. A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and T. Estlin, "CLARAty and challenges of developing interoperable robotic software," in *Proc. 2003 IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, Las Vegas, USA, Oct. 2003, pp. 2428–2435. 7.4.3

[32] B. Finkemeyer, T. Kröger, D. Kubus, M. Olschewski, and F. M. Wahl, "MiRPA: Middleware for robotic and process control applications," in *Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, San Diego, USA, Oct. 2007, pp. 76–90. 7.4.3

[33] 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. 7.4.3

[34] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009. 7.4.3

[35] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: Components for robotics," in *Workshop on Robotic Standardization. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, Beijing, China, Oct. 2006. 7.4.3

[36] D. Brugali and P. Scandurra, "Component-based robotic engineering (Part I)," *IEEE Robotics & Automation Magazine*, vol. 16, no. 4, pp. 84–96, Dec. 2009. 7.4.3

[37] D. Brugali and A. Shakhimardanov, "Component-based robotic engineering (Part II)," *IEEE Robotics & Automation Magazine*, vol. 20, no. 1, Mar. 2010. 7.4.3

[38] E. Prassler, H. Bruyninckx, K. Nilsson, and A. Shakhimardanov, "The use of reuse for designing and manufacturing robots," White Paper, June 2009. [Online]. Available: http://www.robot-standards.eu/Documents_RoSta_wiki/whitepaper_reuse.pdf 7.4.3

[39] M. Klotzbucher and H. Bruyninckx, "Coordinating robotic tasks and systems with rFSM statecharts," *J. Software Engineering for Robotics*, vol. 3, no 1, pp. 28–56, 2012. 7.4.3

[40] ROS SMACH. [Online]. Available: http://ros.org/wiki/smach 7.4.3

Andreas Angerer received his B. Sc. degree in computer science from the University of Augsburg in 2007 and his M. Sc. degree in software engineering from the University of Augsburg, the TU München and the Ludwig-Maximilians-Universität München in 2008. Currently, he is a member of the Institute for Software & Systems Engineering (ISSE) at the University of Augsburg. He was involved in several research projects in robotics and has published about 15 refereed journal, conference and workshop papers. His research interest covers software engineering for robotics, in particular applying modern software engineering paradigms to robotics.

Alwin Hoffmann received his B. Sc. and M. Sc. degrees in information management from the TU München and the University of Augsburg, in 2005 and 2007, respectively, and his Diploma in Computer Science from the University of Augsburg, in 2008. Currently, he is a member of the ISSE and leads the robotics research group there. He has published about 20 refereed scientific papers. His research interest covers robotics, software engineering, and service-oriented architectures.

Andreas Schierl received his B. Sc. degree in computer science from the University of Augsburg in 2007, and the M. Sc. degree in software engineering from the University of Augsburg, the TU München and the Ludwig-Maximilians-Universität in 2009. Currently, he is a member of the ISSE and has published about 15 refereed journal and conference papers. His research interest covers mobile robotics, software architectures and software engineering in robotics.

Michael Vistein received his B. Sc. degree in computer science from the University of Augsburg in 2007 and his M. Sc. degree in software engineering from the University of Augsburg, the TU München and the Ludwig-Maximilians-Universität München in 2008. Currently, he is a member of the ISEE and has published about 15 refereed scientific papers. His research interest covers software engineering for robotics, in particular real-time task execution for robots.

Wolfgang Reif is full professor for software engineering at University of Augsburg where he is also vice-president and director of the Institute for Software & Systems Engineering (ISSE). He received his doctoral degree in computer science from the University of Karlsruhe where he worked mainly in safety, verification, and automatic theorem proving. He held a professorship in software engineering at the University of Ulm from 1995 to 2000. His current research interests are software and systems engineering, safety, reliability and security, organic computing, as well as software-driven mechatronics & robotics. In these areas, he is involved in numerous research projects both in fundamental as well as application-oriented research. Wolfgang Reif is author of a large number of scientific publications, referee for numerous national funding agencies in Europe and the US, and is consultant to leading technology companies.